

8-2009

A Study of Client-based Caching for Parallel I/O

Bradley Settlemyer

Clemson University, bradles@clemson.edu

Follow this and additional works at: https://tigerprints.clemson.edu/all_dissertations



Part of the [Computer Sciences Commons](#)

Recommended Citation

Settlemyer, Bradley, "A Study of Client-based Caching for Parallel I/O" (2009). *All Dissertations*. 396.
https://tigerprints.clemson.edu/all_dissertations/396

This Dissertation is brought to you for free and open access by the Dissertations at TigerPrints. It has been accepted for inclusion in All Dissertations by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.

A STUDY OF CLIENT-BASED CACHING FOR PARALLEL I/O

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Engineering

by
Bradley W. Settlemyer
August 2009

Accepted by:
Walter B. Ligon, III, Committee Chair
Adam Hoover
Melissa C. Smith
Pradip K. Srimani

Abstract

The trend in parallel computing toward large-scale cluster computers running thousands of cooperating processes per application has led to an I/O bottleneck that has only gotten more severe as the the number of processing cores per CPU has increased. Current parallel file systems are able to provide high bandwidth file access for large contiguous file region accesses; however, applications repeatedly accessing small file regions on unaligned file region boundaries continue to experience poor I/O throughput due to the high overhead associated with accessing parallel file system data.

In this dissertation we demonstrate how client-side file data caching can improve parallel file system throughput for applications performing frequent small and unaligned file I/O. We explore the impacts of cache page size and cache capacity using the popular FLASH I/O benchmark and explore a novel cache sharing approach that leverages the trend toward multi-core processors. We also explore a technique we call progressive page caching that represents cache data using dynamic data structures rather than fixed-size pages of file data. Finally, we explore a cache aggregation scheme that leverages the high-level file I/O interfaces provided by the PVFS file system to provide further performance enhancements.

In summary, our results indicate that a correctly configured middleware-based file data cache can dramatically improve the performance of I/O workloads dominated by small unaligned file accesses. Further, we demonstrate that a well designed cache can offer stable performance even when the selected cache page granularity is not well matched to the provided workload. Finally, we have shown that high-level file system interfaces can significantly accelerate application performance, and interfaces beyond those currently envisioned by the MPI-IO standard could provide further performance benefits.

Acknowledgments

I would like to thank my dissertation committee: Dr. Adam Hoover, Dr. Walt Ligon, Dr. Melissa Smith, and Dr. Pradip Srimani, for their time spent reviewing my proposal and dissertation and their insight into how to improve my results and conclusions. I would like to expressly thank Walt for his support and assistance in developing the research results presented in this dissertation. One of the primary benefits of working for Walt is the freedom he has given me to construct my own research objectives and execute my vision of the research project to explore those objectives.

I also would like to acknowledge the support I received from my friends and family. The development of this dissertation was a long, stressful, and sometimes grueling affair. I could not have finished without the support of my parents, Steve and Marilyn Settlemyer, and my brother and sister, Matt Settlemyer and Tara Francis. Their continued faith that I would eventually complete my doctorate was critical in keeping my stress and self-doubt at manageable levels during the times my experiments were failing and I could not determine a concrete reason why. I would also like to thank the other students I worked with here at Clemson. Phil Carns was a great office-mate, and taught me how to work with PVFS. Wu Yang has also been a great office-mate and a collaborator on the simulation software. Nick Mills and David Bonnie have been great friends with whom I have enjoyed working. I especially thank Will Jones for being a friend and colleague while at Clemson, and providing invaluable advice and guidance after his graduation. There is no doubt in my mind that I could not have finished this dissertation without Will's support. Will has helped me to gain perspective on the experiments that did not work out (the bad times), and helped celebrate the successful results (the good times); and for that, I am extremely grateful.

Finally, I wish to acknowledge the computing resources and assistance provided by the support staff at Clemson University and Argonne National Laboratory. This work was

made possible in part by advanced computational resources deployed and maintained by Clemson Computing and Information Technology. I would like to acknowledge the support of the staff from the Cyber infrastructure Technology Integration group. In particular, I would like to thank Randy Martin, who provided assistance to me in running complicated I/O jobs on “Palmetto”, Clemson University’s 772-node computing cluster. The assistance he provided, often well after the hours reasonable people leave work, was critical to the completion of this dissertation. I also gratefully acknowledge the Mathematics and Computer Science Division of Argonne National Laboratory for the use of “Jazz”, the 350-node computing cluster operated as part of its Laboratory Computing Resource Center.

Table of Contents

Title Page	i
Abstract	ii
Acknowledgments	iii
List of Tables	vii
List of Figures	viii
1 Introduction	1
1.1 Parallel File Systems	3
1.2 Middleware Caching in Parallel File Systems	11
1.3 Proposed Research	14
1.4 Method of Study	19
1.5 Dissertation Organization	20
2 Related Work	22
2.1 Cache Coherence	23
2.2 Data Consistency Models	23
2.3 File System Client Caching	26
2.4 Middleware-based Caching	29
2.5 Summary	31
3 Methods	33
3.1 HECIOS: The High-End Computing I/O Simulator	35
3.2 Simulator Validation	46
3.3 A Case Study: Using Server-to-Server Communication in Parallel File Systems	69
3.4 Summary	81
4 Fixed-Size Page Caching	83
4.1 File Data Cache Architecture	85
4.2 Page Delta Cache	92
4.3 Summary	111

5	Progressive Page Caching	113
5.1	Architecture Overview	113
5.2	Implementation Details	117
5.3	Performance Analysis	128
5.4	Summary	142
6	MPI View Aware Aggregation	143
6.1	Collective I/O Aggregation	143
6.2	Implementation Details	147
6.3	Performance Analysis	153
6.4	Summary	157
7	Conclusions	159
7.1	Fixed-size Page Caching	160
7.2	Progressive Page Caching	160
7.3	File View Aware Aggregation	161
7.4	Future Work	161
7.5	Contributions	166
7.6	Closing	168
	Bibliography	169

List of Tables

3.1	Palmetto Compute Node Architecture	48
3.2	Palmetto Ethernet Ping Times	52
3.3	Randomized Palmetto Ethernet Ping Times	54
3.4	Bootstrapped Ethernet Ping Statistics	55
3.5	Palmetto Myrinet Ping Times	63
3.6	Randomized Palmetto Myrinet Ping Times	64
4.1	Best cache configurations on 8CPU, 1PPN, 4ION, GigE	96
4.2	Best cache configurations on 8CPU, 1PPN, 4ION, Myrinet	96
4.3	Best optimized cache configurations on 8CPU, 1PPN, 4ION, GigE	98
4.4	Best optimized cache configurations on 8CPU, 1PPN, 4ION, Myrinet	99
4.5	Best cache configurations on 1CPU, 8PPN, 4ION, GigE	100
4.6	Best cache configurations on 1CPU, 8PPN, 4ION, Myrinet	101
4.7	Best shared cache configurations on 1CPU, 8PPN, 4ION, GigE	102
4.8	Best shared cache configurations on 1CPU, 8PPN, 4ION, Myrinet	103
5.1	Progressive Page Runtime Analysis	126
5.2	Progressive Page Storage Overhead Analysis	127
5.3	Best progressive cache configurations on 8CPU, 1PPN, 4ION, GigE	130
5.4	Best progressive cache configurations on 8CPU, 1PPN, 4ION, Myrinet	131
5.5	Best progressive cache configurations on 1CPU, 8PPN, 4ION, GigE	132
5.6	Best progressive cache configurations on 1CPU, 8PPN, 4ION, Myrinet	133
5.7	Best shared prog. cache configurations on 1CPU, 8PPN, 4ION, GigE	135
5.8	Best shared prog. cache configurations on 1CPU, 8PPN, 4ION, Myrinet	135
6.1	Aggregator Performance on 1CPU, 8PPN, 4ION	154
6.2	Aggregator Disk Performance on 1CPU, 8PPN, 4ION	154

List of Figures

1.1	Network File System Configuration	3
1.2	Parallel File System Configuration	4
1.3	PFS File Data Distribution	5
1.4	Sample Scientific Application Workflow	6
1.5	Parallel Application Software Stack	7
1.6	Middleware Interactions for a Data Cache	13
1.7	Vector Data Type MPI File View	17
3.1	Simulator Architecture	37
3.2	HECIOS Package Dependency Layering	40
3.3	HECIOS Screen Capture 1	44
3.4	HECIOS Screen Capture 2	45
3.5	HECIOS Screen Capture 3	45
3.6	Aggregate bandwidth for 64 processes	56
3.7	Aggregate bandwidth for 128 processes	57
3.8	Aggregate bandwidth for 256 processes	58
3.9	FLASH I/O Benchmark Performance on Palmetto	60
3.10	Aggregate Bandwidth on 64 Processes with MX	66
3.11	Aggregate Bandwidth on 128 Processes with MX	67
3.12	Aggregate Bandwidth on 256 Processes with MX	68
3.13	FLASH I/O Benchmark Performance on Palmetto	69
3.14	Client-initiated file create algorithm	73
3.15	Binary tree collective communication	74
3.16	Collective file create algorithm	74
3.17	File Create Performance on Jazz (Simulation)	75
3.18	Client-initiated file remove algorithm	76
3.19	Collective file remove algorithm	76
3.20	File Remove Performance	77
3.21	Client-initiated file stat algorithm	78
3.22	Collective file stat algorithm	79
3.23	File stat Performance	79
3.24	Kernel Manipulation Performance on Adenine	81
4.1	Flash I/O on 8CPU, 1PPN, 4ION, GigE	93

4.2	Flash I/O on 8CPU, 1PPN, 4ION, Myrinet	94
4.3	FLASH I/O benchmark cache capacity effects	95
4.4	Flash I/O on 8CPU, 1PPN, 4ION, GigE	97
4.5	Flash I/O on 8CPU, 1PPN, 4ION, Myrinet	98
4.6	Flash I/O on 1CPU, 8PPN, 4ION, GigE	99
4.7	Flash I/O on 1CPU, 8PPN, 4ION, Myrinet	100
4.8	Flash I/O with shared cache 1CPU, 8PPN, 4ION, GigE	102
4.9	Flash I/O with shared cache 1CPU, 8PPN, 4ION, Myrinet	103
4.10	Flash I/O with cache 64CPU, 8PPN, GigE	105
4.11	Flash I/O with shared cache 64CPU, 8PPN, GigE	106
4.12	Flash I/O with cache 128CPU, 8PPN, GigE	106
4.13	Flash I/O with shared cache 128CPU, 8PPN, GigE	107
4.14	Flash I/O with cache 64CPU, 8PPN, Myrinet limited	108
4.15	Flash I/O with cache 64CPU, 8PPN, Myrinet unlimited	109
4.16	Flash I/O with shared cache 64CPU, 8PPN, Myrinet unlimited	109
4.17	Flash I/O with cache 128CPU, 8PPN, Myrinet unlimited	110
4.18	Flash I/O with shared cache 128CPU, 8PPN, Myrinet unlimited	110
5.1	Sample Progressive Page	115
5.2	Example of dirty mask region insertion code in C	120
5.3	Progressive Page File Region Tree	122
5.4	Flash I/O with prog. cache on 8CPU, 1PPN, 4ION, GigE	129
5.5	Flash I/O with prog. cache on 8CPU, 1PPN, 4ION, Myrinet	130
5.6	Flash I/O with prog. cache on 1CPU, 8PPN, 4ION, GigE	131
5.7	Flash I/O with prog. cache on 1CPU, 8PPN, 4ION, Myrinet	132
5.8	Flash I/O with shared prog. cache on 1CPU, 8PPN, 4ION, GigE	134
5.9	Flash I/O with shared prog. cache on 1CPU, 8PPN, 4ION, Myrinet	134
5.10	Flash I/O with prog. cache on 64CPU, 8PPN, GigE	137
5.11	Flash I/O with shared prog. cache on 64CPU, 8PPN, GigE	137
5.12	Flash I/O with prog. cache on 128CPU, 8PPN, GigE	138
5.13	Flash I/O with shared prog. cache on 128CPU, 8PPN, GigE	139
5.14	Flash I/O with prog. cache on 64CPU, 8PPN, Myrinet	140
5.15	Flash I/O with shared prog. cache on 64CPU, 8PPN, Myrinet	140
5.16	Flash I/O with prog. cache on 128CPU, 8PPN, Myrinet	141
5.17	Flash I/O with shared prog. cache on 128CPU, 8PPN, Myrinet	141
6.1	MPI File View Vector Example	144
6.2	Vector Data Type Example	148
6.3	Subarray Data Type Example	149
6.4	MPI Tile I/O File Partitions	151
6.5	Aggregate bandwidth for 64 processes	152
6.6	Aggregate bandwidth for 64 processes	153
6.7	Tile I/O with aggregation on 64CPU, 8PPN, GigE	156

6.8	Tile I/O with aggregation on 64CPU, 8PPN, GigE	157
6.9	Aggregate bandwidth for 64 processes	158

Chapter 1

Introduction

The continuous improvement in microprocessor performance is a critical factor in increasing the performance of high-end cluster computers. Recently, the performance advancements of modern commodity processors has shifted from further improvement in single-threaded execution performance to the addition of multiple high-performance execution threads on a single microchip via multi-core processors. The current fastest computer system in the world, IBM's Roadrunner, leverages multi-core chip designs to achieve more than 1 Petaflop of sustained performance running the popular Linpack benchmarking program [57]. Individual Roadrunner nodes are composed of 2 dual-core Opteron processors with 16 Gigabytes of RAM and 4 eight-core Cell Processors with 16 Gigabytes of RAM. At 36 processing cores per node, Roadrunner is one example of utilizing greater intranode parallelism to achieve higher supercomputer performance.

Increasing the number of processing cores per CPU appears to be the next significant trend in microprocessor design, and high-performance computing systems are rapidly preparing for increased intraprocessor parallelism. The number of cores available per processor is likely to increase in the coming years as high-performance computers begin to achieve multiple Petaflop performance ratings. Intel has recently developed a prototype

processor with 80 cores integrated into a single microchip in hopes of building a 1 Teraflop processor designed for use in a future machine projected to achieve 100 Petaflops of sustained computing performance [31].

Increased intranode parallelism improves the processing density of supercomputers and leads to staggering performance levels for applications and benchmarks that do not perform significant amounts of I/O; however, parallel applications that process large data sets are often unable to fully utilize the massive available processing power because the processors idle while retrieving data from storage and writing results data to storage. A typical workflow for a large scientific application may require multiple data intensive steps [58]:

1. Acquiring the data,
2. Staging and reorganizing the data onto a fast file system,
3. Analyzing the data,
4. Outputting results data,
5. Reorganizing the data for visualization, and
6. Processing the data for visualization.

Each workflow stage in the scientific application requires the manipulation of research domain data, and as the amount and fidelity of data required for scientific inquiry increases, the time spent manipulating the data in secondary storage increases. The enormous size of research data sets can result in an I/O bottleneck for many data intensive scientific applications: although the available processing power is enormous, processors are forced to remain idle while large working sets are distributed to the available processing cores.

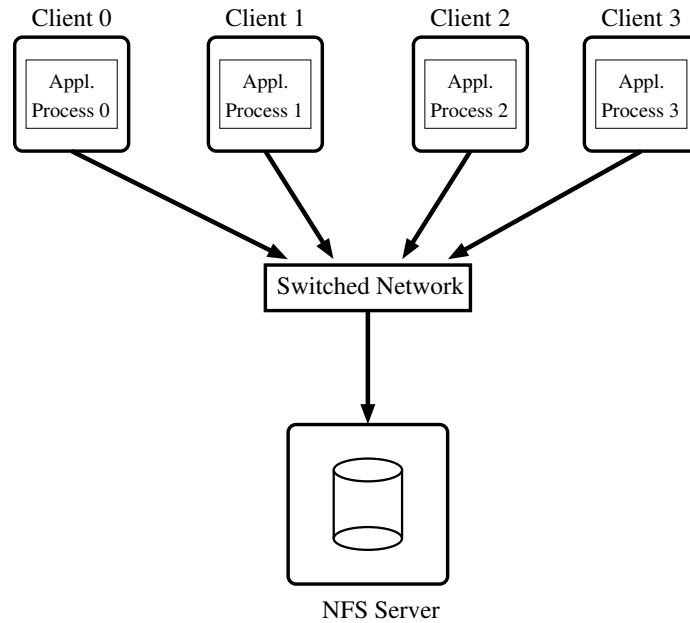


Figure 1.1: Network File System Configuration

1.1 Parallel File Systems

Parallel file systems are one popular approach to alleviating the I/O bottleneck that exists for some parallel applications. Parallel file systems are similar to traditional network file systems in that they allow many cluster nodes to mount a single consistent file system over the cluster's interconnection network. Applications are able to transparently access file data over the network connection via the same interfaces used for access to local file data. However, as shown in figure 1.1, network file systems traditionally do not provide support for distributing data over multiple I/O nodes. A scientific application running on multiple computation nodes in parallel is likely to be bandwidth constrained by the file system servers' disk and network bandwidths and unable to take advantage of the full network bandwidth and computational power available at the application nodes.

Parallel file systems attempt to provide additional file system bandwidth by extending the concept of a network file system to allow file data and metadata to reside on multiple

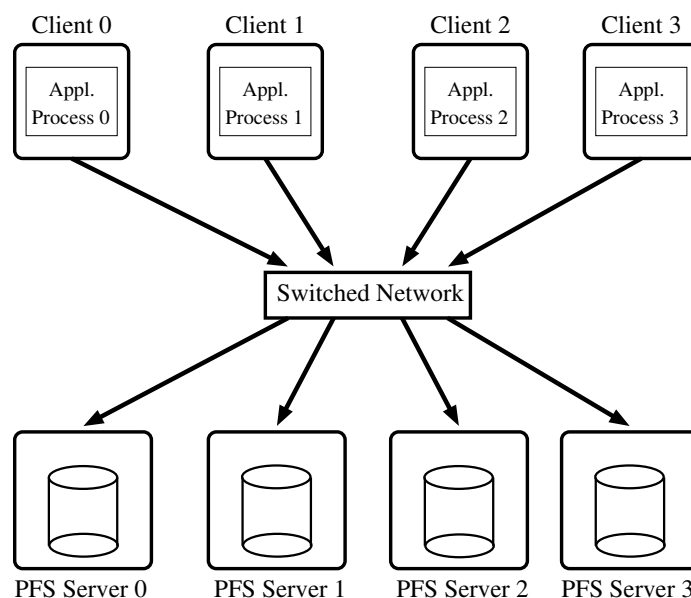


Figure 1.2: Parallel File System Configuration

remote servers. The cluster is typically partitioned into two types of nodes: compute nodes and I/O nodes. Client applications run on the compute nodes and the individual storage servers run on the I/O nodes. By increasing the number of independent network connections and disks participating in the file system, the client application is able to achieve greater aggregate throughput when accessing the file system. A typical parallel file system configuration is shown in figure 1.2.

A file’s metadata is stored on a single I/O server; however, file data is typically distributed among some or all of the I/O nodes to provide applications fast access to a single file in the parallel file system. Figure 1.3 shows a common file data distribution scheme that partitions the file data into fixed sized blocks called *strips*. The file data strips are distributed among the I/O servers in a round-robin fashion that allows parallel access to file data and load balances access to the entire file contents. A scientific application running on multiple computation nodes in parallel is able to see a single consistent view of the file while still achieving the high aggregate bandwidth available by distributing the file

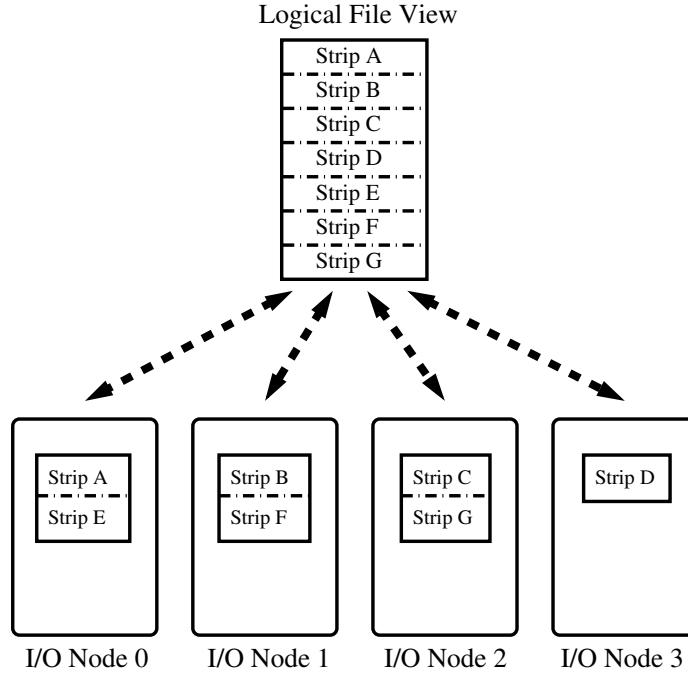


Figure 1.3: PFS File Data Distribution

contents over independently accessible I/O nodes.

1.1.1 Extant I/O Bottlenecks

Parallel file systems are well suited to providing scalable I/O for large contiguous file accesses. Network startup costs and request overhead are easily amortized over the time spent transferring large amounts of data from multiple disks in parallel. In the case where an application's data workload is dominated by metadata operations or the typical file access size is small or occurs on unaligned boundaries, an I/O bottleneck that limits application performance may still occur. One approach to avoid processors idling while waiting for file system data is captured in steps two and four of the scientific application workflow in figure 1.4. In this workflow the data is reorganized, or staged, between the computationally demanding phases of the workflow so that large contiguous file accesses

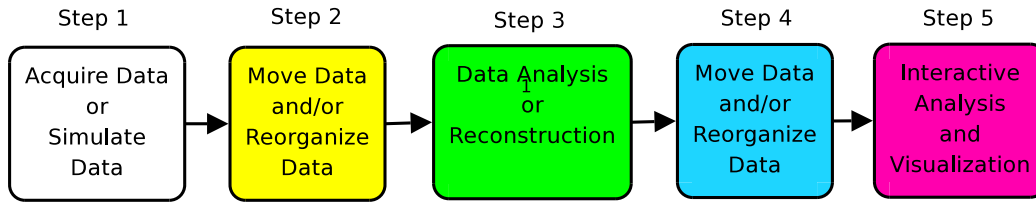


Figure 1.4: Sample Scientific Application Workflow

compose the majority of the application’s I/O requests during computation. Alternatively, the application software can be modified so that file accesses occur on large, aligned boundaries, and then the application itself reorganizes the data among the processors via message passing. Both of these approaches pose difficulties for application developers in the general case.

First, the increase in the number of processing cores per node has led to less main memory available per process despite the rapid growth in node memory sizes. The smaller memory size per process has reduced the size of each individual process’ communication buffer, even though the size of the aggregate communication has increased. Further, the increased intra-node parallelism and reduced buffer sizes makes it necessary for parallel programs to focus on intelligently overlapping processing and I/O phases to reduce contention for the smaller portions of main memory available to the process. These techniques work well for leveraging the performance in modern compute node hardware, however, they are not well suited for accessing parallel file systems. Parallel file systems favor large, contiguous accesses that allow the file system to amortize the disk and network access latencies over the much higher performance available when streaming a large buffer over a high bandwidth network. Unfortunately, large file accesses, regardless of alignment, require the application to use an approach based on large memory buffers, and force applications into alternating periods of idle processes or idle network resources to avoid contention for main memory resources.

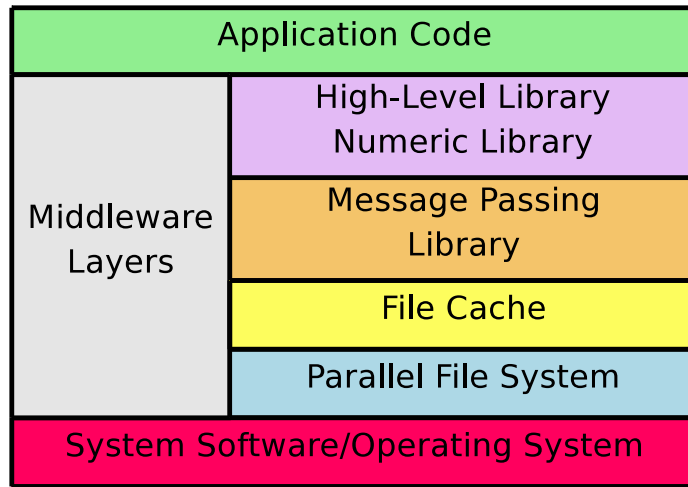


Figure 1.5: Parallel Application Software Stack

Second, the use of high-level numeric and I/O libraries makes it difficult for application writers to reorganize file data or restructure file accesses to achieve higher file system throughput. Modern numeric solvers have become so complicated that the time and expense required to develop the basic numeric routines and accompanying file formats would make it impossible to build scientific applications within reasonable time frames and budgets. Instead, application writer's leverage third-party libraries called *middleware* to more easily perform parallel numeric processing. Figure 1.5 shows a software stack for a typical parallel application. The application code calls a high-level library that provides interfaces for performing parallel computation and parallel I/O [25, 40, 42, 49]. The high-level library in turn interfaces with a message passing library in order to send and receive data between cooperating processes, and modern message passing libraries also expose an interface for performing parallel I/O using a parallel file system client. The high-level library, message passing library, and parallel file system client each constitute middleware; that is, each of these libraries provide a simple interface that abstracts the process of communicating with cooperating processes to perform parallel calculations and interact with file data.

Middleware libraries allow the researchers to focus their software efforts on their immediate scientific domain rather than on the mechanics of writing parallel codes; however, the use of multi-purpose high-level libraries disconnects the developer from the underlying I/O system making it difficult or impossible for the developer to modify the underlying routines to operate at an I/O granularity appropriate for targeted machines. Ideally, middleware libraries provide optimized support for the target platforms; realistically, high-level application interfaces are not always easily mapped onto emerging computer architectures and must balance backwards compatibility with possible performance improvements.

In the case where the application writer develops his own I/O routines, the requirement to perform I/O on evenly aligned boundaries may still impose substantial hardships. For example, in the fluid dynamics benchmarking code we use in this dissertation file I/O is more naturally expressed using smaller, non-uniformly sized chunks of data and accompanying unaligned file accesses. By requiring scientists to access the file system using artificially aligned data structures we increase the complexity and fragility of scientific codes, and tightly couple the code to the current storage system interfaces making it more difficult to leverage emerging storage architectures that offer improved performance. The notion of the I/O interface making data access more difficult rather than easier seems particularly inappropriate.

Finally, even where it is possible to tune parallel codes and libraries to use the I/O system more efficiently, system tuning can be very difficult and may require significant amounts of development time. The time and computing resources spent performing application benchmarking and tuning could be better spent accomplishing scientific research. Similarly, the time spent reorganizing file data for efficient data access is time not spent researching the problem domain. By requiring applications to reorganize the data during multiple workflow steps, the I/O bottleneck increases the amount of work performed during the workflow without improving the basic result.

1.1.2 Performance Optimizations

The performance problems associated with metadata operations and small I/O access patterns on parallel file systems are well known. The Parallel Virtual File System, or PVFS, employs two well known approaches to optimize the performance of small file system access patterns: client-side caching and compound operations. Both optimizations exist in other file systems, and have been shown to be effective at improving performance for many small access dominated workloads.

1.1.2.1 Client-Side Caching

PVFS uses a file name cache called the *ncache* to associate file system names to the unique handle used to describe files within the file system. As described earlier, a PVFS file is composed of a metadata object and a set of data objects, usually with one data object per server. PVFS directories are similar, however there is only a single data object to store the directory entries and it uses a key-value pair organization to associate directory entries to the entry's metadata object. Each of these file system objects is uniquely identified by a system-wide 64-bit value called a handle. Each server stores the data for a unique range of handles ensuring that a given handle can be located on only one server.

The client-side *ncache* associates file names to the unique handle describing the file data. The *ncache* is critical to performance because it allows the client to cache parent directory elements when opening or creating files. For example, when a client attempts to open the file name `"/base/foo"`, the client must first retrieve the metadata for the root directory, `"/"`. If the permissions allow directory searching, the client may then locate the metadata for the directory `"foo"` by searching the directory entries stored for `"/"`. Once the handle for `"/base"` is located and the metadata resolved, the client may search the directory entries for `"/base"` and determine if an entry exists for `"foo"`. By caching the handles for

each of the path elements, a client's subsequent file open request for `"/base/bar"` can avoid looking up the metadata for `"/"` and `"/base"`, and proceed to retrieve the metadata for `"/base"` and begin searching the directory entries for the name `"bar"`.

PVFS also caches file attributes (metadata), in a structure called the *acache*. Similar to the *ncache*, the *acache* exists in PVFS to avoid retrieving the same file or directory metadata attributes repeatedly. Without the *acache*, every file read or write would require querying the file's metadata to determine the data object handles. By caching the metadata attributes during the initial file open, most file I/O operations are able to proceed without any additional interactions with the metadata server.

1.1.2.2 Compound Operations

PVFS employs compound operations to improve the performance of small file accesses. A compound operation is an optimization that allows the file system to recognize that several small messages will be sent to perform a requested client action, and rather than perform each step with separate messages, the individual messages are combined into a single, compound file system request. For example, in file read or file write without compound operations, the PVFS client determines which data objects host the data being read or written. The client then sends a message to the identified servers so that both the client and server can establish a dedicated and buffered connection for file data transfer called a *flow*. The flow uses fixed-size buffers to send the data between the network and local file system at speeds approaching the maximum available bandwidth (network or disk depending on which I/O resource is limiting performance). For large file accesses, the latency of the additional message to initiate the flow connection is quite small compared to the time spent transferring data from the disk or network. For file reads and writes that are not large enough to fill the flow transfer buffer the cost of sending the additional message to establish the data flow may require as much time as the actual file I/O. In the case where the file I/O

is smaller than the available message buffer size, PVFS packs the read or written data into a compound file I/O operation that includes the data along with the described file region avoiding the overhead that would be required to establish and send the data separately as part of a flow.

1.1.3 Problem Summary

Even with optimizations such as attribute caching and compound operations, parallel file systems face significant performance problems with small file accesses, and file reads and writes occurring on non-aligned boundaries. Compound operations are able to reduce the additional network overhead associated with using PVFS; however, small reads and writes are still not well matched to the currently available storage technology. Network bandwidth has increased steadily in the last ten years, with 10Gb networks now readily available in high-end computer systems; network latencies, even with low-latency interconnects such as Myrinet, have improved very little over the same time frame [48]. Similarly, the storage capacity of modern disks continues to improve steadily; however, magnetic disks still exhibit poor random access performance and the fast random access times promised for solid state disks have not yet materialized at price points competitive with spinning disks. The performance of small I/O access to files in parallel file systems continues to be dominated by the initial latency costs of both networking and disk technologies.

1.2 Middleware Caching in Parallel File Systems

The primary impediment to improving parallel file system performance for small file I/O operations is the high latency of network and disk accesses. For that reason, our search for performance improvements should obviously include optimization techniques

that remove or hide system latency. One of the most natural extensions for improving the latency of a file system is a client-side data cache. Data caching allows us to completely eliminate file system latency in the case where the client's local data cache can satisfy the application's I/O request. In the general application of caching to system latency reduction, we envision the performance benefits to occur primarily during repeated reads to the same memory locations. However, with the exception of some out-of-core solvers, the typical parallel code is unlikely to read the same file locations repeatedly. Fortunately, data caching offers the promise of improving the performance of more than just read-after-read and read-after-write access patterns.

One of the most important ways a file data cache can improve the performance of small I/O access patterns is by completing small file writes immediately in the local cache, and then performing a *write back* on a large data block later when a cache eviction is triggered. By taking advantage of write back caching, parallel file system caches have been able to achieve improvements in write bandwidth as seen by the application; however, overall performance improvements have been difficult to observe when combined with a complex parallel file system [44].

Pre-fetching heuristics are another way that a middleware cache can improve the performance of parallel codes. Intelligent pre-fetching code can perform an *in situ* analysis of the alignment of file access boundaries and begin pre-fetching data during the computational phases of a parallel code to improve the performance of later file read requests. The latency of accessing pre-fetched data is hidden from the user as the application is able to read the pre-fetch data from the local cache and continue performing calculations immediately. Kotz and Ellis described and classified the most practical pre-fetching schemes for the common data alignment patterns in use for current parallel software [35]. Tran and Reed extended basic data pre-fetching schemes by using ARIMA time series modeling to include temporal heuristics for determining *when* to begin pre-fetching the data so that the

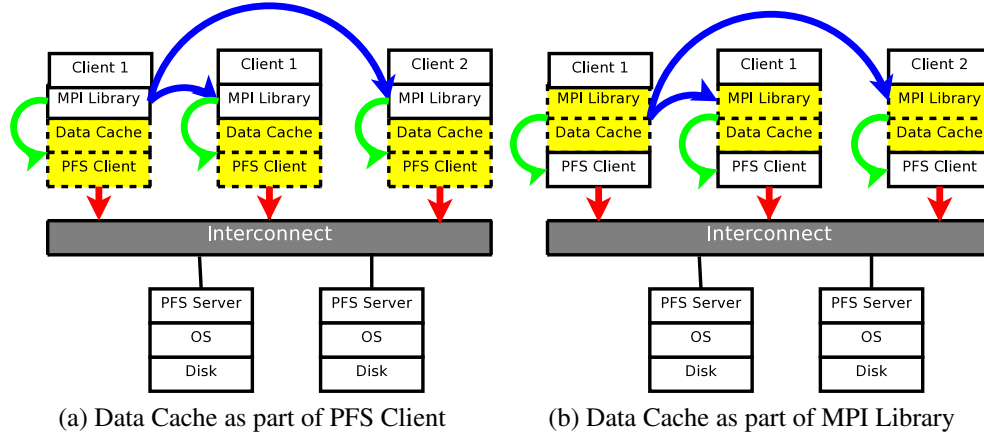


Figure 1.6: Middleware Interactions for a Data Cache

data would arrive just before each file I/O phase begins [56].

1.2.1 Benefits of Middleware

Earlier we defined middleware as a library that provides an abstraction layer that allows client code to seamlessly contact remote processes to perform distributed algorithms. Using this definition, both the MPI communication library and the PVFS client interface are middleware components. Each library allows the user to transparently contact remote processes; in the case of the MPI library the communication occurs in order to change local memory contents, and in the case of the file system client the communication occurs to update file system contents. One major difference between the libraries then, is the set of remote machines they contact to perform the service they provide. In the case of MPI communication calls, the remote nodes contacted are limited to the other application processes, whereas in the case of the parallel file system calls, the remote processes are only the file system's I/O servers.

Figure 1.6 shows the two realistic middleware configurations for the cache: within the parallel file system client or within the MPI-IO middleware driver. In terms of perfor-

mance (and our study is primarily concerned with improving performance), it appears that the former configuration may exhibit better performance by allowing the cache access to the internal file system data structures and removing a layer of overhead. The latter configuration shown in figure 1.6(b) is required to use the parallel file system’s client library in order to access the I/O nodes, meaning that all file I/O must conform to the available interface and exhibit any accompanying overhead. We do not believe that performance improvements related to the reduction of overhead are likely to be significant; however, a study of such overhead may be worthwhile (perhaps as part of a larger study of how high-level I/O interfaces affect file system performance).

On the other hand, figure 1.6(b) provides at least one significant optimization opportunity not available in the former configuration: it allows the client data caches to behave cooperatively. That is, when the data cache is implemented within the MPI library, the available communication infrastructure can be leveraged such that file data is sent directly between caches, rather than requiring an additional step of sending data to the I/O node in order to forward it to another application process’ data cache. In one of our later cache configurations we will leverage this technique to provide scalable performance in one of our cache designs.

1.3 Proposed Research

Although prior experimentation with data caching has generated promising performance improvements, we believe that a detailed study of caching middleware for parallel file systems is warranted. Past prototype-based studies have been limited by the availability of hardware resources and in the high degree of variance exhibited in the performance of systems performing hundreds of intensive jobs at the same time the researchers are attempting to measure a meaningful benchmark. Our study of caching attempts to address

these concerns by building an accurate simulation environment to use for measuring cache performance and that allows us to easily isolate the performance of the I/O subsystem and vary the simulated hardware components to provide results over a wide variety of emerging computer architectures. We also believe that we can further improve the performance of caching middleware by utilizing novel cache organizations. *In this dissertation, we perform a rigorous study to demonstrate that the performance of small and unaligned file accesses can be significantly improved with middleware-based file cache designs that reduce latency costs by using novel cache organizations to increase file system access granularity:*

- *Shared, Concurrent Access Caching,*
- *Progressive Page Granularity Caching, and*
- *MPI File View Caching.*

1.3.1 Shared, Concurrent Access Caching

Cache sharing leverages the trend toward large numbers of processing cores in emerging computing architectures by configuring a middleware cache capable of sharing data between the application processes. With a node local shared data cache we may be able to greatly improve the cache hit ratio versus a cache that is local to only a single execution thread. Cache sharing may also allow the use of MPI view-based I/O request bundling. MPI provides a mechanism for describing non-contiguous file regions called file views. Parallel applications often use file views to more easily partition non-contiguous file regions among a large number of processes. Collective I/O requests performed by each process may then be bundled together in a middleware cache, reducing the size and number of requests dispatched to the file system. By reducing the amount of request data sent to the file system, the middleware cache will be able to reduce the performance costs associated

with small I/O requests. Additionally, request bundling can reduce the requested data's fragmentation, resulting in more efficient overall access to the file system.

In studying the performance improvements available through cache sharing, we have learned that it is also important to study the page granularity used to access the file system. While large page granularity offers benefits in amortizing network startup costs over each individual cache page access, on low latency networks, small page sizes can also provide high performance by reducing the overhead related to coherence protocols. However, in order to leverage the reduced coherence costs of small pages we have found that it is necessary to use high-level file system interfaces that allow multiple pages of file data to be described in a single request. In our studies we have included traditional page caching schemes that demonstrate the effects of cache page granularity on the chosen coherence mechanism, as well as sophisticated approaches that allow a close-to-open consistency model in conjunction with cache sharing.

1.3.2 Progressive Page Granularity Caching

While the benefits of file data caching can be substantial, there are many cache configurations that can lead to performance degradation rather than performance improvement. In general, the page-based coherence protocols can lead to unnecessary performance reduction due to false sharing. False sharing is the performance degrading situation that occurs when two processes are not logically in contention for the same range of data but are forced to perform expensive synchronization operations because the data ranges reside on the same cache page, and the page can only be modified by a single process at a time.

Our scheme, progressive page granularity caching, is able to prevent resource contention under false sharing by describing only the updated data regions within each cache page. Applications performing small, unaligned file accesses are able to utilize larger page

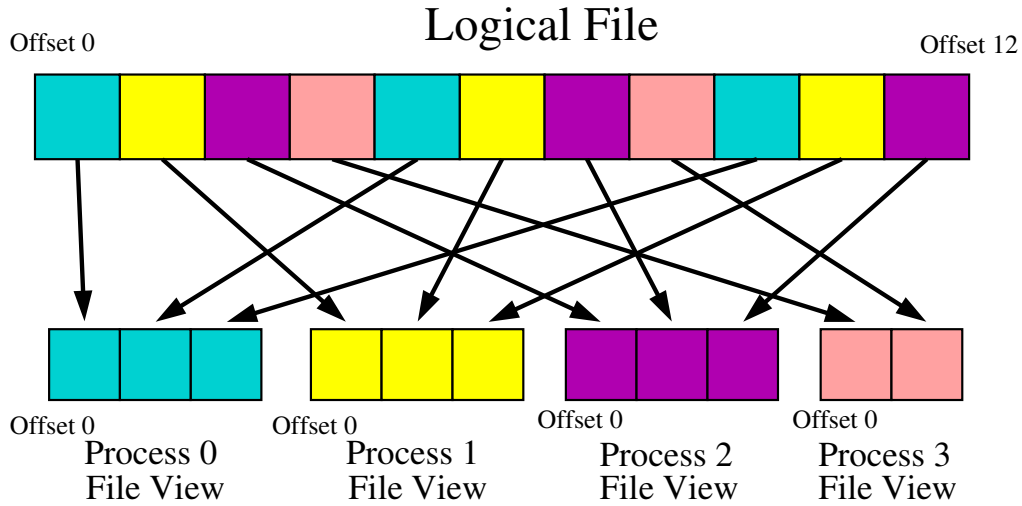


Figure 1.7: Vector Data Type MPI File View

sizes without increases in false sharing and coherence overhead. Within the design space of a progressive page cache architecture we evaluate the difficulties in constructing a cache infrastructure without fixed-size pages and the relevant performance characteristics of the algorithms and data structures used to implement a progressive page file data cache. We also perform a thorough examination of the performance of a standalone cache design and a shared cache design similar to the optimization technique we applied to the fixed-size page cache.

1.3.3 MPI File View-Aware Aggregation

Finally, we have studied cache organizations that optimize the high-level collective I/O calls popular in scientific applications. MPI File Views are used to describe regularly spaced regions within a file using an MPI Data Type. For example, a vector-based file view can be applied to a file that will cause a single read or write interaction to only access every n -th byte of a file. Data types can be combined together to allow a process to easily access complicated, but regular, non-contiguous file regions.

One of the most common uses of MPI File Views is to partition file data among many cooperating processes in a single I/O operation. The processes each apply complementary file views to the file before performing a collective file read or write operation. A collective I/O operation requires each process to simultaneously participate in the same I/O call, with each process receiving different data based on the file view applied by the current process. Typically, the collective I/O operation is used to perform *structured I/O*, where each individual process applies a file view describing dis-contiguous regions within the file; however, the aggregate file access describes a single contiguous file region. Figure 1.7 shows an example of a vectorized data type applied as a file view to partition non-contiguous file regions into a four contiguous regions when accessed through collective I/O operations by the individual processes. Although the non-contiguous file regions appear contiguous to the individual processes, the MPI library will need to translate a single collective read or write call into many smaller file access operations to perform the processes single I/O call.

We have constructed a request aggregation scheme that has knowledge of the MPI File Views in use and will allow us to aggregate the small I/O calls into larger, contiguous file I/O operations that are matched to the high-level data type interfaces provided by the parallel file system. The parallel file system itself cannot easily perform this composition operation because the file system clients have no mechanism for sharing data between one another and by the time the non-contiguous requests reach the file system servers, the opportunity for optimization has already passed. Instead, we propose enhancing our shared cache architecture to perform MPI File View combining for collective I/O operations. At present, no algebraic methods exist for finding the union of MPI File Views, which may be composed of arbitrarily nested collections of MPI data types. Rather, we rely on heuristics-based algorithms that detect interactions based on structured I/O, an easier case to handle for view unification and a more likely candidate for performance improvement versus

purely non-contiguous unstructured I/O operations.

1.4 Method of Study

We performed our studies of parallel file system caching using a simulated model of a parallel file system. Our simulator, the High-End Computing I/O simulator (HECIOS), is a trace-driven parallel file system simulator using the discrete event simulation libraries provided by OMNeT++ . The simulator is built to closely simulate the network messages and system calls used by the Parallel Virtual File System (PVFS) to perform file system tasks. The PVFS network protocol is simulated with correct details resulting in the realistic message sizes being sent over the network to the correct PVFS server processes. The majority of the work performed by the simulator consists of parsing the traces, processing the application requests into parallel file system requests, and distributing file data to the correct server. Detailed simulation models are responsible for correctly deconstructing high-level MPI collective file I/O calls into the correct parallel file system interactions. The parallel file system models then translate those interactions into the required network transactions or operating system calls.

While most of the work is done in the parallel file system models, most of the simulated time accounting is performed by the physical device simulation: the network models, operating system models, and disk models. The cluster interconnection network connecting the compute nodes and I/O nodes is simulated using an OMNeT++ compatible simulation framework called INET. INET provides a set of detailed network simulation models including a complete TCP/IP and Ethernet environment. The availability of the INET network simulation framework is one of the chief benefits of using the OMNeT++ simulation packages. PVFS uses the local file system to store node-local file data, and the requisite operating system calls (e.g. open, read, write) are simulated correctly for file

I/O; however, file metadata in PVFS is stored using Oracle’s Berkeley DB libraries and the simulator uses a simpler flat-file organization for storing file metadata. The hard disk model uses a simplified drive geometry model that accounts for typical hard disk parameters such as head movement speeds, spindle speeds, and physical inertia times, but does not take into account more complicated parameters such as disk pre-fetching and speculative block accesses.

We also developed a set of tools for converting the output from readily available parallel and serial application traces into more concise trace formats useful for our simulation software. In addition to using traces available in public repositories we also traced useful benchmarks at large scale. The collection of new I/O traces has only been possible because of the publicly available tracing tools from Los Alamos National Laboratory and large number of processing nodes available on Palmetto, Clemson University’s large cluster computer available for research use. We plan to contribute all traces collected on the Palmetto compute cluster back to the community for use by other I/O and file system research teams.

1.5 Dissertation Organization

In Chapter 2 we present an overview of related research projects in the field of data caching. Chapter 3 is a detailed description of our simulator’s software component model, and the methods we used to verify and validate the simulation model. In Chapter 4 we present the performance results of our fixed page designs including the performance data for our shared cache designs. Chapter 5 describes the algorithms and data structures used for building a progressive paged file data cache and the resulting performance data. Chapter 6 describes the heuristics used to construct our high-level data type cache and the accompanying performance results. Finally, in Chapter 7 we briefly summarize our

experimental results, describe some of the conclusions and implications of this study, and provide a set of suggestions for further work in this field based on our own findings.

Chapter 2

Related Work

The existence of an I/O bottleneck for many applications is not a novel observation. Hennessy and Patterson noted that many applications are unable to realize the performance gains available from fast CPUs due to time spent waiting for storage processing to complete [26]. One popular approach to address the performance disparity between processors and disks has been caching. Modern operating systems, such as Linux, implement a file system-based buffer cache that stores individual disk blocks as the user performs file reads and writes [6]. The benefits of disk caching are obvious; the number of individual disk accesses can be minimized and disk writes can be bundled together to improve the efficiency of disk access. Because all file accesses must proceed through a single buffer cache local to the file system, there are no cache coherence or file consistency issues associated with local file system caches. The only drawback of local file system caching is the possibility that data successfully written to the file system may not be fully committed to secondary storage, and, in the presence of a file server crash, data loss may occur. Cache aging and data flushing policies are used to mitigate this problem, though an ill-timed server failure will still result in lost data.

2.1 Cache Coherence

Parallel file systems typically deploy large caches on the individual I/O servers; however, since these caches store data for all file system clients without preference for any application I/O patterns the file system performance is still primarily governed by how the application accesses the files. Client-based file caching has the potential to provide higher performance to applications that perform frequent, small file accesses; however, the programming model for an effective caching system may need to be altered. When a process reads a region of file data, the programmer typically expects to receive the data most recently written to that file region. In the event of multiple writers, determining which write is most recent is a complicated problem requiring a global clock that synchronizes all reads and writes. Instead, we may choose to use a model that does not require that a read sees only the most recently written data. Rather, the read returns data that could have come from some valid program ordering. This expected result is the notion of coherence: a system is coherent if the values available to be read can be reconstructed from a valid ordering of the writes performed to that file region [15]. Coherence ensures that when a write occurs, it will eventually become available to all readers; coherence does not provide for when the written data will become available.

2.2 Data Consistency Models

The issue of when written data becomes available for later reads is governed by the data consistency model. The consistency model defines *which* total program orderings are valid, and thus, which written data may be returned to the application during a file read. In general, stronger consistency semantics define fewer valid orderings which provides an easier programming model, while weaker consistency models allow more valid orderings

allowing better cache performance in many cases. Below we describe several different consistency models.

2.2.1 Strict Consistency

In a strict consistency scheme, a file region read must include the contents of the most recent write to that region. That is, read and write operations are seen at all clients in the order they are dispatched into the file system. While strict consistency schemes are very natural from an application writer's point of view, the performance impacts of having a single serialization point for all file system access are likely to be too great for applications requiring high levels of I/O throughput.

2.2.2 Sequential Consistency

Another strong semantic for file access is sequential consistency [16]. Sequential consistency requires that all processes must see the same read and write ordering for file locations, and that the ordering must be a valid interleaving of the reads and writes based on the executing processes. For example, if two processes write differing data to the same file location, call it Location 1, and then perform a synchronization operation, all following reads of Location 1 must return the same data for all processes.

2.2.3 Weak Consistency

The weak consistency model takes a different approach to determining the ordering of file reads and writes. Weak consistency ensures that file reads and writes are seen in a valid total program ordering for all processes, but the same ordering does not have to be presented to all processes. This scheme is workable because weak consistency ensures that synchronization variables (e.g. mutexes) are always presented in a sequentially consistent

fashion. Thus, the programmer can define critical sections using synchronization variables to ensure that designated reads see the result from the most recent write.

2.2.4 Release Consistency

Release consistency further relaxes the restrictions of the weak consistency model [22]. Again, the read and write ordering must be from a valid program interleaving, however they need not be the same for each process. In release consistency the synchronization primitive is separated into acquires and releases. An acquire operation performs a read to gain sole access to a lock and a release operation relinquishes sole control over the lock. All critical sections are initiated with an acquire and terminated with a release. In release consistency acquire operations can be performed in any valid order on each process; however, release operations must appear in the same valid order for all processes.

2.2.5 Scope Consistency

Scope consistency further relaxes the approach described in release consistency by adding context to the synchronization variable accessed during acquire operations [28]. In scope consistency, each synchronization variable describes a *consistency scope*. Release operations only need to be delivered in-order to processes that also release that same consistency scope. So all processes do not see the same order for all release operations; but instead see the same order only for the consistency scopes released by the individual process. Scope consistency generally results in the same consistency semantic as release consistency; however, in the case where two different variables are used to synchronize the same file region, scope consistency may return different results than release consistency.

2.2.6 Entry Consistency

Entry consistency requires that a dedicated synchronization variable be attached to each shared memory location [28]. The consistency scheme then is the same as scope consistency with the modification that a separate scope exists for each location. This model is not easily supported by current programming languages and compilers, and does not map easily onto the variable-sized file accesses that characterize most file I/O.

2.3 File System Client Caching

In Chapter 1 we discussed the popularity of metadata and name caching in traditional network file systems and high-performance parallel caching. Client-based data caching has long been a recognized technique for improving file system responsiveness and I/O throughput. Here we describe the client-side caching schemes used in three important distributed file systems and a fourth approach, cooperative caching, evaluated primarily in small scale simulations. One problem with adding a client-side data cache into the underlying file system is the difficulty in tuning the cache to perform well with individual applications. None of the caches described in this section have the ability to tune file system caching parameters for individual applications.

2.3.1 The Network File System

Version 3 of the Network File System (NFSv3) implements a simple client-side data cache [23] using a relaxed consistency model. The consistency model, called close-to-open consistency, implements a weaker scheme than any we have described thus far. The basic mechanism resembles weak consistency; however, each individual file open introduces its own consistency scope. When a file is opened, any data for the file in the cache is marked

invalid and must be re-read from the server. All file writes update only the local cache until the file is closed, at which point the file data is flushed to disk. Because file data is cached as 4KB pages on the client, data caching may result in two competing processes reverting the state of a file if both are writing to the same 4KB page. In the case where the data cache does not provide a usable programming semantic, NFSv3 can support POSIX-style file locking to ensure that access to the file is serialized.

2.3.2 The Network File System, Version 4

Version 4 of the Network File System (NFSv4) uses a weak consistency scheme similar to the consistency model used in NFS version 3. When a client opens a file, the client's cached data for the file must be re-validated. And when the file is closed, the cached data must be written to the server, just as in version 3 of the protocol. File opens in NFSv4 also allow the user to request a *delegation* that grants exclusive access to the file. Any acquired read exclusive or write exclusive delegation can later be revoked by the server, and further reads and writes need to reacquire the appropriate delegation in order to proceed. Other than setting the desired delegation during the file open call, the delegation transaction protocol is entirely transparent to the user, but is one mechanism for ensuring a file is not concurrently read or written. Along with the delegation scheme, NFSv4 has added a cache coherence protocol to the POSIX advisory locking calls. When a client acquires a file region lock the locally cached data for that file region must be re-validated. Naturally, when the file region lock is released, the modified data for that file region is written to the server. By allowing cache coherence for individual file regions rather than only for the entire file, NFSv4 should provide a better model for concurrent file access than the approach used in NFSv3.

2.3.3 PPFS: The Portable Parallel File System

The Portable Parallel File System, PPFS, implemented several layers of caching within the file system [27]. In particular, PPFS supported global caching, client-side caching, and server-side caching with the ability to specify writeback policies for each component. Although the PPFS developers recognized that caching was a critical factor for improving the performance of many application workloads, the client-side caches did not provide any coherence control, and thus only produced well defined results for workloads without overlapping file accesses. One of the primary research efforts associated with the PPFS client-side caching was the development of sophisticated cache pre-fetching policies. One of the most innovative approaches is an ARIMA-based temporal data pre-fetching scheme that was shown to provide significant performance improvements for ESCAT, an electron scattering that alternates between intensive computation and intensive file data retrieval [56].

2.3.4 Cooperative Caching

Cooperative caching seeks to improve network file system performance by mutually sharing the contents of client data caches [19]. In cluster environments where high-performance, low latency message passing networks are frequently available, accessing remote clients to retrieve cached data may result in improved file system throughput. Cooperative caching offers the most opportunity for performance improvement when the clients exhibit a large degree of inter-client sharing [4]. Many projects have explored the use of cooperative caching within the file system as an effective means for improving file system performance [2, 29, 62]. Bagrodia, et al. performed simulation studies of cooperative caching for MPI-IO benchmarks on four and eight client processes. Their simulator, MP-ISIM, was used to study the number of disk accesses performed using several varieties of

cooperative caching. The study results are difficult to extrapolate to modern machines due to the small number of client processes used and the dissimilarity of their simulator to the latest parallel I/O systems (e.g. all file systems interactions use list I/O) [3].

2.4 Middleware-based Caching

The emergence of middleware as a critical component for lowering the cost and complexity of parallel code has been an important change in the development cycle of scientific applications. By using an abstract parallel machine model, applications are able to interface with middleware specifically tuned for the high-performance computer in use. Middleware caching offers the likely benefit of tuning cache parameters specifically for application workloads, although tuning options also allow the possibility of a poorly chosen parameter to degrade, rather than improve, performance. Additionally, middleware-based data caches can leverage other middleware components to take advantage of the collective nature of parallel I/O. In our study of various cache designs we will attempt to leverage the capabilities of the messaging middleware layer to improve performance whenever possible.

2.4.1 Active Buffering

Active buffering is one middleware-based approach for improving the performance of synchronous collective file write operations using MPI-IO [45]. Active buffering uses additional memory available at the compute node to copy the users output buffer into a managed output buffer called the active buffer. A dedicated thread then writes data from the active buffer to the file system. Provided the user has available unused memory, active buffering may provide higher performance than the conventional asynchronous MPI-IO bindings due to its more efficient interaction with the file system. In traditional asynchronous file writes the user cannot safely access any output buffer in use by an asynchronous collective

operation. If the time spent filling the output buffers is relatively short, the user must employ double or triple buffering techniques to ensure that the processor does not idle while waiting for the next file write to complete. Active buffering may be able to bundle multiple outstanding requests in order to interact with the file system more efficiently; however, it is not clear whether this performance advantage exists for real applications rather than specially designed benchmarks.

2.4.2 Discretionary Caching for PVFS

Vilayannur, et. al explored the performance benefits and penalties of a multi-level file cache implemented for the original PVFS [60]. Their caching scheme provided a Linux kernel module that provided file page caching for each compute node. Additionally, a single dedicated node with a larger amount of main memory acted as a global file data cache. The study found that file reads that missed in both the local and global data caches performed worse than simply reading data directly from the file system. Secondly, the cost of updating the local and global caches with data that was not later accessed also decreased performance. The authors suggest implementing a discretionary caching scheme, such that only data that will be accessed repeatedly is stored in the caches. The authors instrumented a compiler with additional directives to enable and disable data caching, and implemented a set of heuristics for determining which data blocks should be locally cached, globally cached, or bypass the cache entirely.

2.4.3 ROMIO Data Caching

The Center for Ultra-Scale Computing and Information Security at Northwestern University has prototyped several file cache designs [44, 32] with ROMIO, an open source implementation of the MPI-IO standard [51]. The basic approach involves partitioning the

file into a set of fixed-size pages. The pages are then assigned to a single computation node by taking the modulo of the page number. Job processes access file data by requesting it from the client responsible for the cache page rather than by accessing the file system directly, a cooperative caching approach. In one scheme, the file data may only be cached at the node responsible for the cached page. Another scheme implements directories at the responsible node so that another process may cache the page. All of these schemes require that file data is cached at only one node and that all file accesses occur on page aligned boundaries. Our studies intend to explore the effects of relaxing the requirement to cache data in only one location and measure the benefits of allowing file access on non-aligned boundaries.

The caching middleware developed at Northwestern University has been extended to support file data pre-fetching using a scheme called I/O signatures [32, 8]. An I/O signature is constructed by running a target application with a tracing library to store the set of all I/O accesses to a text file. An offline analysis tool is then used to parse the trace information in hopes of finding I/O access patterns. Any identified access patterns are written into the I/O signature file for the application and that file is loaded into a pre-fetching module supplied in a modified version of MPI. The measured performance improvements using I/O signatures were moderate.

2.5 Summary

Our study of middleware-based file data caching is a natural extension of many of the prior studies of file data caching for parallel codes. Although NFS versions 3 and 4 provide a mechanism for coherent file caching, data is only guaranteed to remain consistent at file page size granularity (typically 4KB). Our caching schemes use the same basic consistency model as NFS, close-to-open consistency; however, our coherence pro-

protocols provide consistent data irrespective of the cache page size. Secondly, unlike NFS and PPFS, we have elected to implement our caching schemes in middleware rather than in the actual file system client. Other middleware caching approaches have demonstrated the utility of a library-based implementation to accelerate file I/O. Although approaches such as active buffering are designed to accelerate access to small file regions, the reported results could not demonstrate significant performance improvements beyond those provided by the asynchronous file access routines included with MPI-IO. Similarly, the caching approaches developed by the team at Northwestern University appeared to provide volatile performance results, with little explanation of why some configurations successfully improved performance while other configurations degraded performance. In this dissertation, in addition to the novel cache designs we present, we are interested in performing a thorough analysis of the performance effects of cache page sizes and cache capacities. Finally, as part of our analysis, we also examine the effects of hardware related bottlenecks on the performance improvements available with file data caching.

Chapter 3

Methods

We have chosen to use simulation to perform our studies for three important reasons:

1. A simulator allows us to experiment with emerging supercomputer architectures not generally available,
2. A simulator allows us to rapidly evaluate candidate cache designs,
3. And a simulator can produce results more quickly than a prototype.

The first advantage of performing a simulation-based study rather than prototyping is the increased number of platforms we can use for experiments. One trend in emerging architectures is the use of custom hardware to improve processing performance. The Department of Energy's two fastest clusters, Roadrunner and BlueGene/P, both leverage custom hardware that does not run the standard Linux environment expected by PVFS. Porting PVFS to run on custom hardware is a complicated task, and in the case of BlueGene environments, has only been accomplished by installing Linux on the cluster and running the computer system without the use of some specialized hardware. Additionally, it is not clear

that we can even acquire sufficient access to these high-end clusters to perform any benchmarking experiments. Cutting edge hardware is highly sought after for performing basic science; receiving compute time to execute I/O benchmarks is often difficult or impossible. In simulation, we have the ability to modify the computer architectural parameters with ease. Fabricating a microprocessor with 120 cores may still be a difficult task for Intel, but simulating a processing node with 120 independent cores is simply a matter of modifying a simulator execution profile. The ability to perform simulations at emerging scales, rather than only at the scales currently available, also allows us to produce results that may more easily guide the designs of future high-end computing storage systems.

The second motivation for using a simulator to perform our study is the ease with which we can implement candidate cache architectures. Parallel file systems, like most complex distributed systems, are difficult to program and difficult to debug. Interactions between older code and new data structures often leads to difficult to understand behavior and race conditions. Two recent research prototypes based on the PVFS source code found that in addition to their proposed modifications, PVFS needed substantial modifications in existing code for buffer management [52] and socket management [9] in order to take meaningful performance measurements. By implementing a simulator that processes the traces of successful application executions we are able to avoid several of these problems. Because the simulator runs all the processes within a single thread in a single address space, race conditions and distributed debugging cease to be issues entirely. Sharing data between processes is trivial, and the use of an application trace allows us to know the outcome of each I/O event before simulating it, allowing us to avoid the problem of servicing the request and instead focusing on producing an accurate timing model and the collection of meaningful statistics.

Finally, our simulation model will allow us to generate experimental results more quickly than repeated trials using a PVFS-based prototype. Although parallel file systems

are in general complicated systems that require detailed simulation in order to generate an accurate timing model, some of the system components can be abstracted and the timing approximated without affecting the simulation results. The intelligent application of abstraction in the simulation models enables the simulator to return sufficiently accurate timing results without requiring a significant amount of simulation time. Parallel storage systems are particularly well suited to this type of abstraction because costly network and disk operations usually dwarf the amount of time spent computing file offsets and extents. We plan to simply measure the average processor time over several trials for each file system operation, and use that value as a reasonable abstraction for each operation processing time. Additionally, rather than actually spending the time to store the data to disk, we need to only calculate the amount of time it would take to store the data to disk.

3.1 HECIOS: The High-End Computing I/O Simulator

In order to study the performance effects of various file system client caching schemes we have built HECIOS, the High-End Computing I/O Simulator. HECIOS is a trace-based simulation platform for studying the behavior of parallel I/O systems on high-end cluster computers. Simulation provides several advantages over prototyping in terms of evaluation I/O system modifications for emerging computer architectures. The use of simulation allows us to measure caching effects for large scale computer architectures that are not widely available, or even at scales that are not yet in production (e.g. 48 processing cores per node). Secondly, novel caching organizations are often more easily implemented in simulation rather than via prototype. The use of global bookkeeping structures in a single address space removes many of the difficulties of keeping data synchronized in distributed systems. Finally, the use of a simulator allows us to develop abstract models of some components in the parallel I/O system, thus allowing time spent performing experimentation to

be reduced (at the expense of some result precision).

Many parallel I/O research efforts have used simulation to evaluate novel performance enhancements. Weil, et al. used a simulator to evaluate the effectiveness of various parallel file system metadata load balancing schemes [61]. Bragodia, et al. used simulation studies to evaluate the benefits of cooperative caching for parallel file system using four and eight client processes [3]. Bosch and Mullender used the Patsy simulator to measure the impacts of server-based caching policies in the traditional file system, Pegasus. Each of these projects were able to leverage detailed simulators in evaluating the performance impacts of the criterion under study; similarly, HECIOS is designed to allow us to study the performance impacts of modifying the parallel I/O software stack and the underlying parallel file system.

3.1.1 HECIOS System Architecture

HECIOS is designed to provide an accurate simulation environment for modeling the I/O performance of parallel applications accessing parallel file systems. Before running the simulator, an application that accesses the file system is executed, and a trace of the application's processing and I/O requests is recorded. HECIOS includes a set of tools for translating the output of two available tracing tools into a trace file format supported by HECIOS. The execution trace is then loaded into HECIOS and the time spent processing and performing I/O is accounted for with a detailed simulation. HECIOS produces two output files: a scalar output file that outputs values determined during simulation, and a vectorized output file that tracks the change in recorded values over the simulated time frame. Values such as cache hit rate are recorded as scalars, whereas the round trip time for each file creation is recorded as a vector.

One of the major design goals of HECIOS was allowing the easy reconfiguration

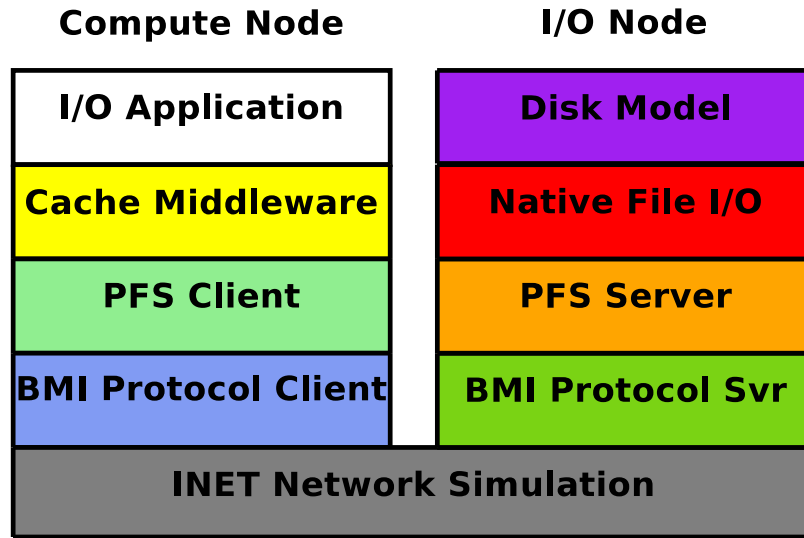


Figure 3.1: Simulator Architecture

of the simulator to model diverse cluster configurations. HECIOS provides the user the ability to select the number of compute nodes, the number of I/O nodes, the number of simultaneous processes per compute node, and the network architecture including the switch hierarchy, network component bandwidth, and network startup costs. HECIOS also supports easy configuration of system call overheads, the size of the buffer cache on the I/O nodes, the disk scheduling algorithm, and the physical disk performance characteristics. One problem with the number of available configuration parameters is the difficulty in determining how each configuration parameter affects any particular simulation run. In order to lessen the burden of measuring each of the individual parameters, we have tried to tune the simulator to an existing cluster to develop a performance baseline from which we can then designate the independent variables to study in our experiments.

Figure 3.1 shows the system architecture for HECIOS. File system requests originate from the I/O application and are dispatched into the middleware cache (which can be configured to forward messages without adding delay). If the cache cannot satisfy the request (or is configured without data caching) the request is forwarded to the file system

client. The file system client translates the application request into one or more file system requests, and dispatches the request to the BMI client model. The bulk message interface (BMI) protocol is used in PVFS to send data efficiently over many different types of inter-connection networks [10]. In our simulation, the only available network type is a TCP/IP network built using the INET network simulation components. INET simulates the delay associated with transmitting the client requests to the appropriate I/O node and delivers the message to BMI server on the I/O node. The BMI server sends the message to the parallel file system server, which constructs local file system requests based on the client request. The local file system requests are dispatched into the OS and committed to disk as necessary. Once the OS finishes processing the local requests, the OS response or responses are collected at the parallel file system server, which then constructs the correct response type and sends it over the network via BMI to the file system client. When all of the responses are received at the client, the client signals the operation completion to the application via the middleware cache.

3.1.2 HECIOS Implementation

HECIOS is written in the C++ programming language. Using C++ as our main programming language provides several tangible benefits to the development process. First, C++ includes the Standard Template Library (STL), a high quality implementation of the most commonly needed container classes (e.g. vectors, maps, and queues) alleviating the need to develop common data structures and iteration algorithms. Second, C++ is an object-oriented programming language allowing us to easily develop high-level abstractions that are then reused throughout the software. Another advantage of C++ is that we are able to easily perform black box testing by leveraging an existing C++ unit testing framework [14]. Finally, there are many freely available libraries written in C and C++ that provide capabil-

ities useful to a parallel I/O simulator.

3.1.2.1 OMNeT++ Simulation Framework

The simulation infrastructure of HECIOS is provided by the OMNeT++ discrete event simulation framework. OMNeT++ provides a complete simulation environment for scheduling events, building a network of connected simulation models, and scheduling messages for later arrival at a specified model. Simulation models are derived from the provided SimpleModule class which requires concrete implementations of methods for initialization, finalization, and message arrival. The NED scripting language is then used to construct a network of simulation models via composition into a CompoundModule or using a direct connection called a gate. One of the major advantages for HECIOS in using OMNeT++ and the NED language to construct the network of simulation components is that simulation parameters such as the number of compute nodes, or the network architecture can be easily reconfigured without recompiling the simulator.

3.1.2.2 INET Simulation Framework

The INET Simulation Framework is a network simulation package developed using OMNeT++ . INET provides detailed models for simulating TCP/IP connections over an Ethernet network. The INET simulation includes models for Ethernet media access control, collision avoidance and detection, TCP congestion control, TCP startup costs, and various queuing policies. INET has been successfully used in several research efforts [50, 34].

In developing the source code for HECIOS we have used an iterative development model that attempts to keep the simulator in a working state at all times. Every new feature required for an experiment requires a process iteration that adds the new features and refactors the existing code as necessary. One frequent problem when using iterative methodologies that require redesigning existing components to perform new tasks is the explosion

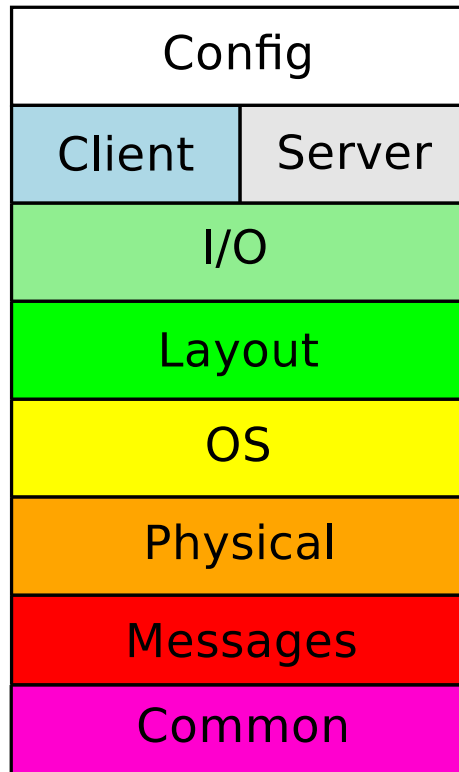


Figure 3.2: HECIOS Package Dependency Layering

in the number of dependencies in the software system. To accomplish the addition of new functionality for an iteration, it may be appropriate to simply add new relationships to existing classes, or it may be better to add new classes that abstract the new dependencies. Every connection between two classes results in a new edge in the dependency graph leading to a more complex, difficult to modify software system.

3.1.3 Software Component Architecture

In order to guide our decisions in adding new dependencies and managing the dependencies between existing classes we have divided our software project into the nine packages shown in figure 3.2. Within a single package there is no restriction on how the classes may be interrelated. However, dependencies between classes in different pack-

ages must satisfy the vertical ordering shown in the diagram. For example, classes in the Common package may not rely on classes from any other package while classes in the OS package may aggregate, inherit from, or use any classes in the Physical, Message, and Common packages. One benefit of our approach to package dependency ordering is that all dependencies between classes in different packages are unidirectional, and that bidirectional dependencies can only exist within a single package. Additionally, by striving to place new classes in the correct package, we only increase the complexity of the modified package rather than the system as a whole. And as one package becomes too complicated it can be subdivided into multiple packages, and re-factored to lessen the complexity of developing software within the package.

3.1.3.1 Config Package

The Config package is responsible for interconnecting the simulator models in the correct configuration and initializes the components according to the user specified trace file and settings. The configuration package contains the code that configures how middleware caches are shared among processes, how the compute nodes and I/O nodes are connected via switching, and constructs the mechanism for assigning file contents to individual I/O servers.

3.1.3.2 Client Package

The Client package implements the code responsible for executing the instructions in the trace file, performing middleware caching, and constructing the client-side parallel file system messages. The majority of client processing is performed using state machines built from the OMNeT++ state machine facility.

3.1.3.3 Server Package

The Server package contains the code implementing the parallel file system server-side operations. Every basic parallel file system operation is implemented as a state machine using the OMNeT++ state machine infrastructure. The server determines the correct state machine to construct based on the type of message arriving over the network.

3.1.3.4 I/O Package

The I/O package is used by both the Client and Server packages to transmit file data between the client and server. As in PVFS, the basic mechanism used is a data flow that processes the I/O request to determine the amount of data to transmit and whether the transmission involves reading or writing to disk or memory based on whether the flow endpoint is a client or server and whether the request is a read or write.

3.1.3.5 Layout Package

The Layout package acts as a central registry for looking up file names, file handles, and file attributes. The use of the central repository of data makes it easy for the client or server to quickly determine whether a file exists, or where it resides without storing the response from the simulated server.

3.1.3.6 OS Package

The OS package provides the operating system components useful to our simulation. Specifically, the OS layer includes a POSIX file system interface for use by the Server and I/O packages and internally simulates a Linux file system's disk scheduler, block cache, and file system inode and data block assignment schemes.

3.1.3.7 Physical Package

The Physical package includes the model of a simulated hard disk, an integration layer for accessing the INET networking components, and the BMI protocol client and server models used to access the network.

3.1.3.8 Messages Package

The Messages package defines all of the messages sent between simulator models. We use the high-level OMNeT++ message definition language to describe all of the messages for each component of the system architecture, and then automatically generate C++ code using the provide code generation tool.

3.1.3.9 Common Package

The Common package contains all the widely useful type definitions (e.g. FileHandle, FileSize, FileOffset) and widely reusable classes. Some of the most useful classes in the Common package include the Filename class which allows path components to be easily extracted from file names and the FSOperation and FSState classes used to build nested state machines in the Client and Server packages.

3.1.4 User Interfaces

HECIOS provides two interfaces for executing simulation scenarios: a command line interface (CLI) and a graphical user interface (GUI). The command line interface provides updates on the simulation state at tunable intervals (every 100,000 events by default). Output includes the total number of events processed, current simulation time, events per second, events per simulation second, and the total user time spent executing the simulation. The command line environment is provided so that simulations can be executed on

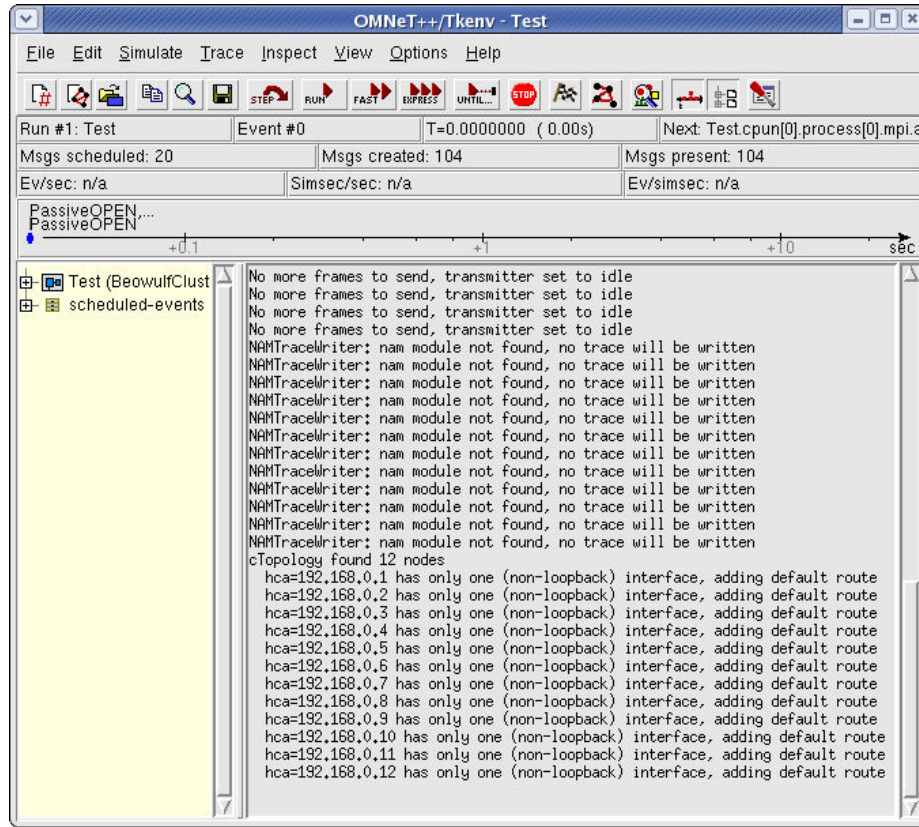


Figure 3.3: HECIOS Main Window

the distributed processing resources available on campus.

HECIOS also provides a graphical user interface based on the TCL/TK environment supported by the OMNeT++ simulation package. Figure 3.3 shows the main GUI window which presents the same progress statistics reported in the command line version of HECIOS. The main window includes a window for viewing all of the simulation defined output messages (warnings, diagnostics, etc.), and basic controls for starting the simulation, stopping the simulation, and controlling the level of detail provided in the interaction animations. Figure 3.4 shows the main visualization window for HECIOS. The main visualization window displays the top level component models and highlights the models and connections as the messages flow between the simulation models. The main visualization

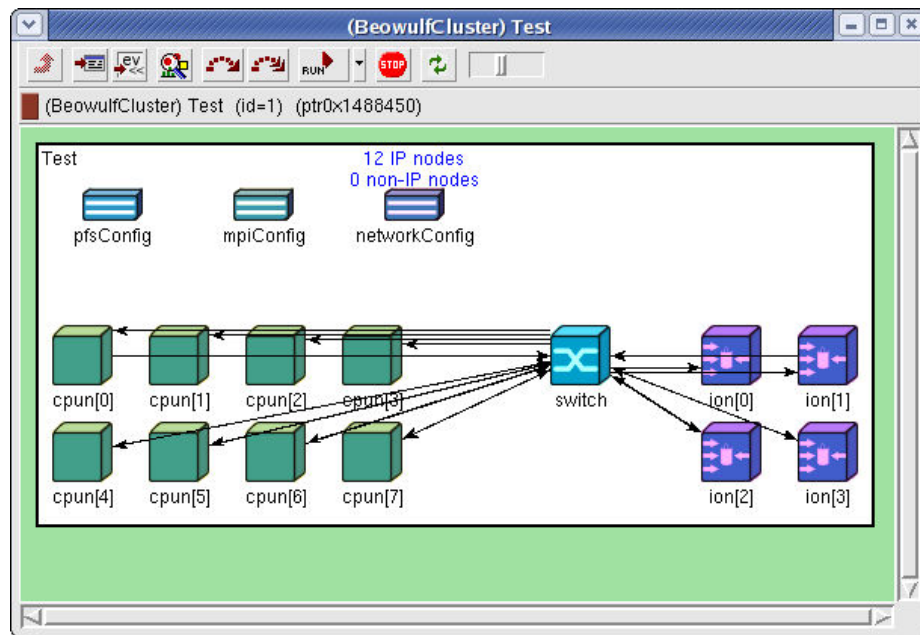


Figure 3.4: HECIOS

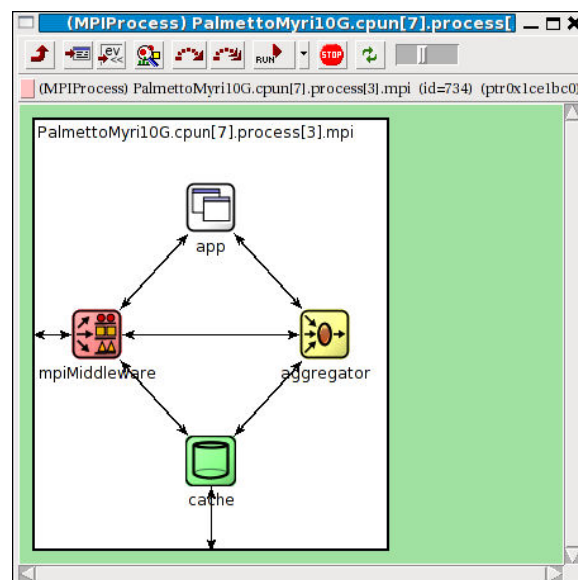


Figure 3.5: HECIOS

window also supports the ability to drill down on the compound models and examine the simulation models contained within the compound model. Figure 3.5 shows the drilled down view of the process running on the compute node. Icons represent the application process, middleware cache, parallel file system client, and a management interface for accessing the BMI abstraction layer of the network. The icons are highlighted while handling a simulation event, and the links are animated as a message is sent between the models.

3.2 Simulator Validation

One of the primary difficulties in executing a simulation-based performance study is ensuring that the experimental results are a valid approximation of the real world system to be simulated [33]. One aspect of simulator validation is code verification: the process of ensuring the model is correctly coded and performs all of the steps the software developer intended the simulator to perform. Essentially, verifying the simulation model is the same as ensuring that the developed software is *correct*. To that end, we have used recognized best practices and methods for developing correct software to ensure the verification of our simulator. In particular, we focused on using a spiral model for software development that continuously refines and improves the software by implementing, testing, and integrating each new simulator feature as completely as possible before beginning a new feature. Secondly, as part of our testing approach we performed both high-level black box testing that ensured the simulator performed the same steps as the real world system (PVFS) and low-level white-box testing aided by the use of a unit testing framework (CppUnit) and the development of significant test scaffolding to allow software modules to be tested in isolation from the remainder of the software system. Finally, we have performed a series of experiments that attempt to measure how well our simulator models the real world systems, and how well our model predicts the expected response when the real world system

is modified.

In performing our validation experiments we have tried to perform a series of experiments that measure the simulator's performance over a variety of inputs. For example, we have performed sets of experiments that are latency constrained and sets of experiments that are bandwidth constrained. Although we have attempted to build an accurate software model, we are aware that our model does make several simplifying assumptions. We will note those assumptions when relevant, particularly when the assumption leads to performance predictions that differ from the real world system. The remainder of Section 3.2 focuses on describing the experimental details of the individual validation experiment configurations and the observed results of the simulated and real world system.

3.2.1 Clemson University's Palmetto Cluster Computer

In order to compare the results of HECIOS and a real system running PVFS it was first necessary to choose an actual existing computer system to target for simulation. At Clemson University the natural choice is the Palmetto Cluster, the fastest computer at the university and the 60th fastest computer in the November 2008 Top 500 list, a listing of the 500 fastest publicly known supercomputers.

The Palmetto Cluster is a collection of 6,168 dedicated computing cores capable of achieving a peak performance of 56.55 Teraflops and sustaining 45.61 Teraflops of computational performance. As shown in Table 3.1, Palmetto is composed of 3 different types of high-end computational nodes, each with dual 64-bit quad-core processors. Every processor is rated at 2.3GHz or higher and has at least 4MB of L2 cache. The Dell nodes contain 12GB of memory while the Sun Microsystems nodes provide 16GB of total RAM. Local storage on the compute nodes is provided by Western Digital WD800JD-75MS 80GB drives with 8MB of disk cache. In general, Palmetto has 36 nodes per rack and a total of

Num Nodes	Vendor	Processor Type	Num CPUs	Num Cores	Memory (GB)
257	Dell	Intel Xeon E5345 @ 2.33 GHz	2	8	12
258	Dell	Intel Xeon E5345 @ 2.33 GHz	2	8	12
256	Sun	AMD Opteron 2356 @ 2.3 GHz	2	8	16

Table 3.1: Palmetto Compute Node Architecture

24 racks in the entire system (some of the racks host switching gear, and do not contain a full 36 nodes). All of the compute nodes run a 64-bit CentOS-5 Linux distribution with the 2.6.18-92.1.10.el5 version of the distribution kernel.

In order to run a computational job on Palmetto, the user must submit a job execution script to the Maui scheduler, which then interacts with the cluster’s Torque resource scheduler to determine when the available computing resources are available to satisfy the job’s computing requirements. The computational jobs may be queued for several hours, or even days, while waiting for the requested computational resources. Due to the heterogeneity of the compute nodes, in particular the difference in the amount of main memory available on the Dell nodes versus the Sun nodes, we submitted all of our validation timings to nodes that satisfied the “Intel” resource request, ensuring our jobs executed on the Dell nodes that provide only 12GB of main memory.

Palmetto also provides two independent interconnection networks for each compute node: a Gigabit Ethernet network interface and a Myrinet Myri-10G network interface. The Gigabit Ethernet connection is provided by an integrated Broadcom BCM5708 network interface chip on each compute node. The GigE network is available for use by applications; however, it is also used as a cluster management network for spawning jobs, monitoring compute node resources, and accessing network shares. Each compute node

also has a Myrinet Myri-10G network connection that is dedicated for application use only. The Myrinet 10G network is a high-performance interconnect capable of low-latency message passing and 1.2 GB/s of sustained network bandwidth. The Myri-10G network can be accessed as a low-latency, 9.8 Gb/s IP network or with the Myrinet Express (MX) RDMA interface capable of achieving full line-rate bandwidth and a message passing latency of $2.3\ \mu\text{s}$.

3.2.1.1 Measuring PVFS Performance

By default, the Palmetto cluster does not run the PVFS parallel file system. Instead, Palmetto uses the Lustre file system to provide a high-performance parallel storage system. Although the Lustre file system provides a high-performance storage interface, it is not possible for a non-privileged user to change the number of I/O nodes or to access the storage system over the Myrinet 10G network. Further, since the file system is in production use it is not possible to guarantee that other user's jobs are not performing significant amounts of "interference" I/O operations. Even if PVFS was available on Palmetto, we would find it necessary to run our instance of the file system to avoid interference operations and to have the option to configure the file system with varying number of I/O nodes.

In order to construct a PVFS file system as part of our computational job, we are using the PVFS Automatic Volume Service. The PVFS Auto Volume service, or PAV, is a set of scripts included with PVFS that start remote I/O server processes on a subset of the nodes allocated to the computational job. We have further modified PAV to enable the PVFS I/O servers to utilize the Myrinet network interface in addition to the default TCP/IP-based network connection.

3.2.2 Validating Simulated Gigabit Ethernet Performance

Our HECIOS simulator uses the INET network simulation package to construct a realistic model of a TCP/IP network transport over a switched Ethernet physical local area network. In particular, we use the Ethernet2 model-set that allows full duplex network connections and a TCP Reno congestion avoidance algorithm. One shortcoming of the HECIOS network simulation model is that all of the compute nodes are connected with a single network switch. In reality, each Palmetto rack uses a single 48-port Dell PowerConnect 2848 Gigabit Ethernet switch with a single 10 Gigabit port trunked to a single vlan on a Cisco Nexus 7000 backbone switch. Effectively, the inter-rack bisection bandwidth is only one-fifth of the intra-rack bisection bandwidth, meaning that benchmark performance may differ greatly depending upon the number of racks any computational job is scheduled across.

Although the INET package has the capability to construct complex networks, the default networking components do not allow the interconnection of switches or port trunking to provide the bandwidth achieved by link aggregation. Instead, it would be necessary to construct multiple network routers in order to connect the switches, thus resulting in a larger number of network hops than is required in the physical system. The other alternative is to develop improved Ethernet switch models that support direct switch connections and port trunking. We did not feel that the added accuracy in network topology modeling would balance out the development time required to implement an Ethernet switch with link aggregation support and direct switch connection capabilities. The following sections attempt to measure and define the validity of the Gigabit Ethernet networking model used by HECIOS.

3.2.2.1 Network Latency Benchmark

The ping utility is a useful tool for measuring the *round-trip* latency between two nodes connected over an IP-based network. Ping measures the round-trip latency by sending a series of “echo request” ICMP packets from a network node to a destination IP. The recipient node then responds to each echo request by sending an “echo response” ICMP packet back to the originating node. The originator then prints out the time between sending the packet and receiving the response which is the round-trip latency. In our simulator, the ping time is primarily affected by the parameters setting each node’s IP processing delay and the switch processing time. While it would be possible to set the parameter values such that the simulator exactly matches the observed ping times, the parameter values also needed to provide accurate real-world performance over our entire array of validation benchmarks, which sometimes led to difficulties in balancing the overall system validity as opposed to the validity of a single complex benchmark.

Table 3.2 shows 30 collected ping times for intra-rack ping, inter-rack ping, and simulated ping times. Our simulator does not implement multiple hierarchical switches interconnected to form a large network, instead we implement a single unified switch that connects all of the nodes in a flat topology. In this case we have decided that a simpler implementation is worth the loss of accuracy because it improves the simulator performance (i.e. benchmarking sessions perform much faster) and it reduces the overall simulator complexity. Due to the simplified switch topology, our simulated ping time lies between the two extremes, with a bias toward the intra-rack ping time average. Ideally, we could perform a T-Test on the simulated value to insure that the simulated ping time does not differ significantly from the mean; however, since the data is bi-modal we are not aware of a suitable hypothesis test using parametric statistics. In order to generate a data sample that we can perform statistical analysis upon we then tried to re-sample the data using greater

Trial	Intra-rack Ping Time (ms)	Inter-rack Ping Time (ms)	Simulated Ping Time (ms)
1	0.116	1.600	0.147968
2	0.092	0.158	0.097864
3	0.087	0.153	0.097864
4	0.087	0.161	0.097864
5	0.087	0.265	0.097864
6	0.086	0.158	0.097864
7	0.089	0.183	0.097864
8	0.088	0.224	0.097864
9	0.092	0.159	0.097864
10	0.089	0.197	0.097864
11	0.086	0.161	0.097864
12	0.091	0.183	0.097864
13	0.088	0.133	0.097864
14	0.091	0.132	0.097864
15	0.088	0.136	0.097864
16	0.087	0.132	0.097864
17	0.089	0.136	0.097864
18	0.086	0.143	0.097864
19	0.088	0.181	0.097864
20	0.088	0.135	0.097864
21	0.086	0.168	0.097864
22	0.100	0.152	0.097864
23	0.088	0.163	0.097864
24	0.088	0.184	0.097864
25	0.087	0.233	0.097864
26	0.087	0.155	0.097864
27	0.089	0.195	0.097864
28	0.088	0.162	0.097864
29	0.087	0.183	0.097864
30	0.086	0.253	0.097864
Mean	0.089	0.219	0.0995341

Table 3.2: Ping times recorded for Palmetto over the Gigabit Ethernet Interface

randomization. We are using Palmetto's node named node0254 as the source node for all ping times because we can use that node freely to ping other nodes. We then randomly select another node to ping by choosing random numbers in the set $[0..770]$ (excluding 254 as a possibility). We then contact the selected node using ping. We further randomize ping time selection by choosing a random number from the set $[2..100]$, and then collecting the i -th ping where i is the selected random number. We discard all of the initial pings due to the possibility of additional work involving NIC and switch-based address resolution time being included in the ping time.

Table 3.3 shows the randomly selected ping destinations, random sample numbers, and the resultant ping times. The mean ping time is 0.112ms and the median ping time is 0.095ms. As the large time difference between the mean and median ping times indicates, the randomly sampled data still does not correspond to a normal distribution; however, this is to be expected. Ping time values are bound on one side only. That is, the minimum possible time for a ping request to be acknowledged is limited by the wire propagation delay of the electronic network signal, whereas there does not exist any theoretical upper bound on the ping time given enough other network traffic. The sampled data did not conform to any probability distribution used by the Minitab simulation package, therefore we used a bootstrapping procedure to determine robust confidence intervals for the mean and median ping times.

Table 3.4 shows the computed mean and median ping times using a bootstrapping procedure with 1000 repetitions. Our simulated ping time of 0.097864ms does not fall within the bootstrapped mean ping time range. However, because the sample size includes a significant sample size (30 trials), in our opinion the median value is a better characterizing measure of the Ethernet interface's ping performance. Our simulated ping time of 0.097864ms lies within the 95% confidence interval for the bootstrapped median and is statistically indistinguishable from the bootstrapped median of 0.098ms at two significant

Trial	Node Name	i-th Ping	Ping Time (ms)
1	node0687	8	0.095
2	node0491	38	0.094
3	node0521	91	0.094
4	node0123	6	0.165
5	node0172	78	0.118
6	node0272	2	0.104
7	node0349	86	0.095
8	node0678	30	0.094
9	node0237	25	0.146
10	node0714	97	0.094
11	node0139	63	0.131
12	node0752	7	0.095
13	node0583	82	0.094
14	node0425	33	0.104
15	node0464	43	0.129
16	node0292	29	0.119
17	node0207	31	0.123
18	node0581	78	0.095
19	node0390	37	0.094
20	node0717	96	0.094
21	node0308	23	0.185
22	node0440	6	0.093
23	node0310	57	0.094
24	node0310	23	0.111
25	node0684	54	0.095
26	node0636	94	0.095
27	node0080	43	0.205
28	node0117	62	0.094
29	node0611	11	0.124
30	node0107	54	0.094

Table 3.3: Ping times recorded for Palmetto over the Gigabit Ethernet Interface

Bootstrap Mean Ping Time (ms)		Bootstrap Median Ping Time (ms)	
Average	0.112	Average	0.098
Std Deviation	0.0051	Std Deviation	0.0057
Min	0.100	Min	0.094
Max	0.128	Max	0.124
95% C.I.	[0.104,0.121]	95% C.I.	[0.0945,0.111]

Table 3.4: Mean and Median Ping Times for Palmetto using bootstrapping with 1000 repetitions

digits.

3.2.2.2 MPI I/O Test Benchmark

While network latency is an important metric for evaluating the validity of our parallel file system simulator, it is perhaps more important to have an accurate model of network bandwidth and network utilization characteristics. The MPI I/O Test benchmark is a useful tool for evaluating the aggregate file system bandwidth available to clients of the parallel file system. The benchmark is an MPI code that has each of the participating processes open a shared file and write 16 Mebibytes(16×10^2) of data to the file before closing it. The processes then collectively re-open the file (requiring synchronization) and read the 16 Mebibytes of data from the file before closing the file and calculating the aggregate bandwidth in Megabytes per second.

Figure 3.6 shows the aggregate bandwidth curve for the MPI I/O Test benchmark running on 8 nodes with 8 processes per node. The x -axis measures the number of I/O nodes participating in the file and the y -axis measures the aggregate bandwidth in Megabytes per second. For each file system configuration, 10 trials were measured using the Gigabit Ethernet network on the Palmetto cluster. Each individual trial is graphed as a single scatter-point. The measured bandwidth using an identical file system configuration in simulation is show using the connected line-points.

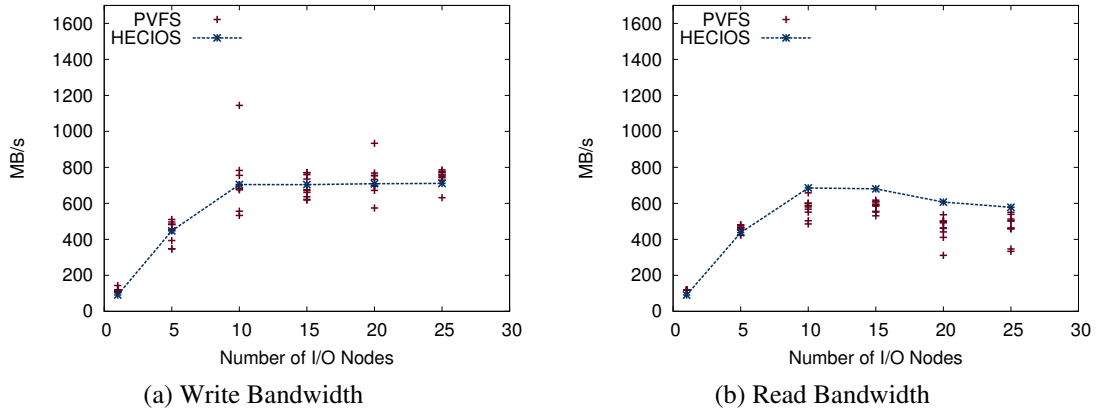


Figure 3.6: Comparison of empirical and simulated parallel file system bandwidth using 64 file system clients (8 processes per node) connected with Gigabit Ethernet

The write bandwidths measured empirically and in simulation track very closely in Figure 3.6(a). Figure 3.6(b) shows the simulated and real world performance for benchmark's aggregate read bandwidth. The read performance curves indicate the simulated system provides slightly higher read bandwidth; however, the overall read performance and the performance trends are very similar to the real world system. The other major trend we note in Figure 3.6(b) is the increasing variance in the real world system's performance. As the aggregate read bandwidth graph shows, the performance variance of the MPI Test I/O code running on Palmetto is relatively large, and grows as the number of I/O nodes increases. One likely cause of the diminished performance that leads to the increased variance is switch in-cast. In-cast occurs when many network clients try to send data simultaneously to a single host. The queuing buffer for the switch port is overwhelmed with data and the switch is forced to drop some number of the frames sent to the port. Although both Ethernet and TCP have mechanisms for avoiding congestion, both are more effective for preventing frame drops at network nodes rather than at the network switch level.

Figure 3.7 shows the aggregate read and write bandwidth data for both PVFS and HECIOS using 128 total processes (16 client nodes with 8 processes per node). Again,

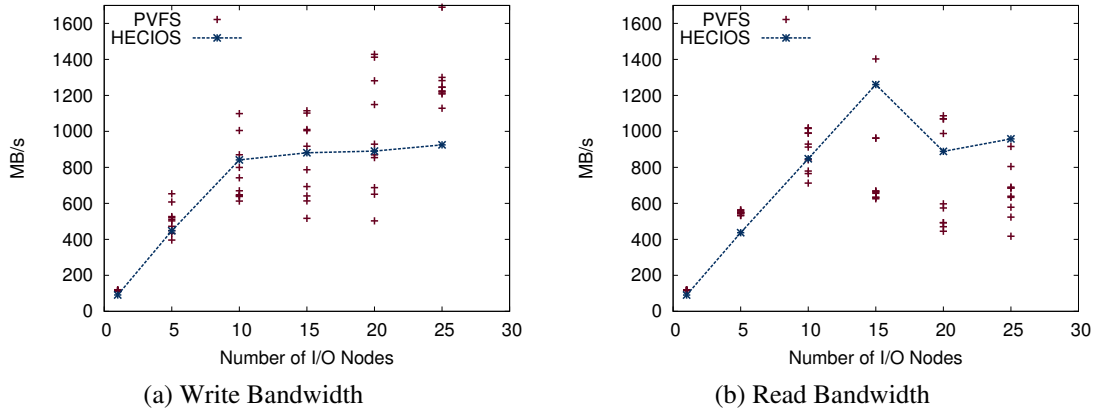


Figure 3.7: Comparison of empirical and simulated parallel file system bandwidth using 128 file system clients (8 processes per node) connected with Gigabit Ethernet

we see that the simulated write performance closely mimics the real world measurements, although the simulated bandwidth at 25 I/O nodes is much slower than real world system's write performance. The simulated aggregate read performance shown in Figure 3.7(b) also tracks closely to the results observed for Palmetto. However, with a 25 I/O node configuration the aggregate read performance is higher on the simulator versus the real system. Although, in this case, the simulated and observed values are much closer.

Finally, Figure 3.8 shows the aggregate bandwidth performance for our last Ethernet network configuration, 32 nodes with 8 processors per node for a total of 256 benchmark processes. In general, we are pleased with the accuracy of the simulated file bandwidth performance, however, two configurations, one write and one read, exhibit much higher performance on the simulator than on the Palmetto cluster. The simulated write performance shown in Figure 3.8(a) tracks very closely to the real world system with one outlier when configured with 10 I/O nodes. For the read performance in Figure 3.8(b) we observe that the simulator provides higher read performance for 20 and 25 I/O nodes; however, the high simulated performance with 20 I/O nodes is a much larger outlier than the 25 I/O node configuration. Again, the difficulty in matching the simulated performance with

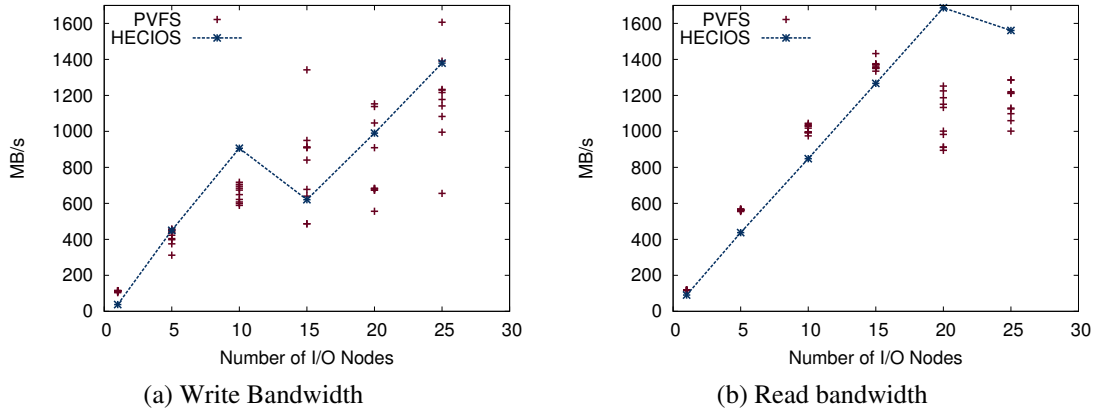


Figure 3.8: Comparison of empirical and simulated parallel file system bandwidth using 256 file system clients (8 processes per node) connected with Gigabit Ethernet

the observed performance is attempting to provide adequate matching performance over a wide range of benchmarks while also simplifying the system so that a simulator can return results faster than a real world system. The major cause of the performance difference again appears to be the lack of a hierarchical switch architecture as opposed to a flat network topology. Even with the various inaccuracies we have described in our system, we feel that the simulated values could easily approximate an execution path on the Palmetto cluster, even if it does not match one of the 10 trials we observed at each configuration. For that reason, we are satisfied with the performance of our simulator on the MPI-I/O Test benchmark on our simulated Palmetto system.

3.2.2.3 The FLASH I/O Benchmark Routine

The final benchmark we use to validate our simulator is FLASH I/O, a parallel, multi-dimensional, adaptive-mesh-refinement, hydrodynamics code useful for simulating thermonuclear flashes for astrophysics applications [21]. The adaptive mesh refinement algorithm subdivides the computational domain into Cartesian grids and sub-grids capable of arbitrary refinement levels. The grids and sub-grids are composed of three-dimensional

arrays called blocks that are distributed across the participating MPI processes. A single block represents a structured mesh and is implemented as an $8 \times 8 \times 8$ array with an additional 4 elements in each dimension called guard cells that hold ghost data to avoid nearest neighbor communication during refinement calculations. The FLASH I/O kernel uses the HDF5 library to produce three files: a plotfile with centered data, a plotfile with corner data, and a checkpoint file. The FLASH I/O benchmark routine duplicates the FLASH code's I/O kernel to allow experimentation with improving I/O performance without spending time performing domain calculations [46].

Because the FLASH I/O benchmark uses the popular HDF5 I/O library to access the underlying file system [25], we consider it representative of many I/O bound application workloads that perform small, unaligned, and independent file writes to construct hierarchical data formats. Each benchmark process produces 7.5MB of data total, and the total file sizes scale linearly with the number of participating benchmark processes. For each output file, all of the participating processes open the file (truncating the size to zero if necessary), perform roughly 250 small, unaligned file writes, and close the file. The small size of each individual file write and the small size of total I/O performed by each process has traditionally made it challenging to achieve a high-level of parallel file system I/O throughput with this benchmark. In particular, achieving scalable parallel file system performance is traditionally difficult with workloads dominated by small, unaligned file reads and writes.

Figure 3.9 shows the observed total runtime for the FLASH I/O benchmark using 128 client processes and 256 client processes respectively. In general, the total runtimes on Palmetto and on HECIOS' simulated Palmetto environment are very similar.

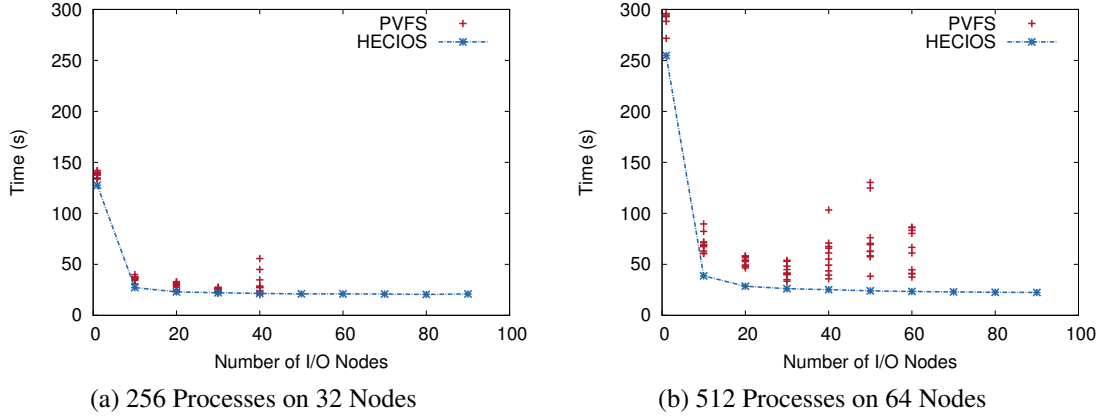


Figure 3.9: Comparison of empirical and simulated parallel file system execution time running the FLASH I/O benchmark on Palmetto with the Gigabit Ethernet interface (lower is better).

3.2.3 Validating Simulated Myri-10G Performance

While HECIOS’ Gigabit Ethernet network-based configuration provides a reasonably accurate model of the physical system, we are more interested in measuring the effects of our proposed file system modifications using the most recent high-performance hardware capabilities. Gigabit Ethernet provides a reasonable amount of network bandwidth, however, the high network latency caused by copying data from user space into kernel memory in order to send TCP packets performs much worse than several popular alternative networking technologies. Palmetto’s high-performance, alternative interconnection network is provided in the form of Myrinet’s Myri-10G high-end networking fabric. In addition to providing 10 times the bandwidth of Gigabit Ethernet, Myri-10G also exhibits substantially lower network messaging latency. Many of the performance improvements made available by Myrinet are due to the use of an independent processor on the network interface card that calculates message check-sums, sends route discovery messages, and performs source routing in conjunction with switch support for wormhole routing. Palmetto’s Myri-10G network uses both line cards and edge switches to construct an interconnection network

with full crossbar bandwidth (each node has a full 10 Gigabit connection to every other node). A Myrinet 10G-SW32LC-16M line card is used to connect 16 nodes with a full crossbar connection. The line card has an additional 16 ports that connect into a Myrinet Edge enclosure that connects the line cards using a 16 port CLOS topology network as the back plane architecture. However, even with the high degree of network bandwidth available, switch in-cast can still be a problem when many nodes send data simultaneously to a single node, although wormhole routing does avoid the extreme in-cast penalties imposed by Ethernet because packets are queued at the sender rather than simply dropped by the switch.

For our simulator we did not have access to a network package that correctly simulates Myrinet, so instead we have modified the Ethernet network settings to approximate the performance of a high-performance networking fabric such as Myrinet. The chief difference between our simulated configuration and a real Myrinet network is again the flat network topology that comes from using a single network switch, and also the fact that our switch drops in-cast packets, rather than queuing the packets at the sender. We have been able to modify the media access times of the network card and switch so that the network provides low latency message passing, and tuned the switch and network card buffer sizes so that the network performance under load approximates the high-performance network on Palmetto.

3.2.3.1 Network Latency Benchmark

The Myrinet Express (MX) software interface is able to avoid the system call overhead associated with copying message buffers between user space memory and kernel page memory by providing an interface that sends network messages directly from user space without the requirement to copy. Even when Myri-10G is used as a medium for an IP network, Myrinet is able to provide lower latency due to the dedicated checksum processor,

tighter physical medium qualities and more aggressive signal timings. In order to compare the network latencies used by our simulator we are again using the Ping utility, this time running on Palmetto's IP over Myrinet interface.

Table 3.5 shows the Myrinet network ping time in milliseconds between two nodes on the same line card (column 2) and two nodes on different line cards connected over the edge switch (column 3). Column 4 shows the simulated ping time using HECIOS configured with a Myrinet Myri-10G network. The first trial for each test again appears to be an outlier, probably related to network address resolution overhead. Similar to our earlier Ethernet ping time experiments, the observed ping times again appear to be bimodal with our simulator value adhering much more closely to the ping times within a single line card. We performed the same randomization as described in Section 3.2.2.1 (randomly choosing both the destination node and sample number to observe) and generated a sample of ping time values using Palmetto's node0254 as the ping source node.

The results of our randomized process are shown in Table 3.6. The mean ping time is 0.048 milliseconds and the median ping time is 0.036 milliseconds. The difference between the observed data mean and median values indicates that we should again be concerned that the data is not normally distributed. An Anderson-Darling test for normality returned a value of 2.387 with an accompanying P-Value of less than 0.005 indicating that the data does vary significantly from the normal distribution [36]. However, for this set of observations, the Minitab statistical package was able to find a matching probability distribution for the data by applying the following Johnson transformation:

$$J = 1.37919 + 0.575550 * \ln((R - 0.0226194)/(0.192187 - R)) \quad (3.1)$$

where R is the observed ping time and J is the transformed value [37]. The transformed data was verified to correspond to a normal distribution using the Anderson-Darling normality

Trial	Intra-line Ping Time (ms)	Inter-edge Ping Time (ms)	Simulated Ping Time (ms)
1	0.076	0.096	0.0601568
2	0.036	0.091	0.0367264
3	0.026	0.056	0.0367264
4	0.024	0.112	0.0367264
5	0.025	0.123	0.0367264
6	0.024	0.065	0.0367264
7	0.024	0.078	0.0367264
8	0.024	0.125	0.0367264
9	0.024	0.125	0.0367264
10	0.026	0.117	0.0367264
11	0.025	0.226	0.0367264
12	0.025	0.084	0.0367264
13	0.025	0.130	0.0367264
14	0.023	0.183	0.0367264
15	0.025	0.096	0.0367264
16	0.024	0.127	0.0367264
17	0.026	0.095	0.0367264
18	0.025	0.109	0.0367264
19	0.036	0.065	0.0367264
20	0.026	0.076	0.0367264
21	0.025	0.125	0.0367264
22	0.026	0.070	0.0367264
23	0.024	0.063	0.0367264
24	0.024	0.062	0.0367264
25	0.024	0.096	0.0367264
26	0.033	0.121	0.0367264
27	0.054	0.065	0.0367264
28	0.025	0.085	0.0367264
29	0.025	0.128	0.0367264
30	0.026	0.134	0.0367264
Mean	0.029	0.156	0.0375074

Table 3.5: Ping times recorded for Palmetto using the Myrinet Interface

Trial	Node Name	i-th Ping	Ping Time (ms)
1	node0208	73	0.045
2	node0546	29	0.080
3	node0592	41	0.061
4	node0032	24	0.042
5	node0417	21	0.030
6	node0711	34	0.032
7	node0730	65	0.038
8	node0192	38	0.027
9	node0586	62	0.026
10	node0177	62	0.098
11	node0699	32	0.031
12	node0721	70	0.036
13	node0741	10	0.031
14	node0528	49	0.068
15	node0498	14	0.024
16	node0153	69	0.026
17	node0380	36	0.026
18	node0390	77	0.036
19	node0745	62	0.031
20	node0451	91	0.052
21	node0363	48	0.156
22	node0329	13	0.112
23	node0160	46	0.025
24	node0265	92	0.058
25	node0623	50	0.036
26	node0637	92	0.067
27	node0388	39	0.029
28	node0122	90	0.075
29	node0117	55	0.027
30	node0280	25	0.023

Table 3.6: Randomized ping times recorded for Palmetto over the Myrinet 10G Interface

test and resulted in an A-D value of 0.180 with a P-value of 0.907 allowing us to continue our statistical analysis with an alpha-level of 10% (i.e. the probability the data is not normal is less than 10%).

The simulated ping time, 0.0367264ms, becomes the transformed value -0.00197. A one sample T-Test indicates that the observed and simulated means are identical with a probability of 0.981 (or $\alpha < 0.05$). We conclude that our simulated ping time is not significantly different than the average of the randomized pings we observed on Palmetto.

3.2.3.2 MPI I/O Test Benchmark

In order to validate the bandwidth accuracy of our simulator we are again using the MPI I/O Test benchmark developed at Argonne National Laboratory. For all of our observed trials we are using PVFS2 configured to use the Myrinet MX software package, a network communication layer that is based on dedicating, or *pinning*, pre-allocated user-space memory buffers for sending and receiving network messages. This approach avoids the expensive calls to convert user-space memory into kernel space packets and is more generally known as a remote direct memory access method, or RDMA for short.

In Figure 3.10 we have again overlaid the observed read and write aggregate bandwidth curves using the MPI I/O Test benchmark. In this configuration, the benchmark program is running on 8 nodes with 8 processes per node and all parallel file system access uses only the Myri-10G network using the RDMA-based MX network driver. The x -axis measures the number of I/O nodes participating in the file and the y -axis measures the aggregate bandwidth in Megabytes per second. For each file system configuration, 10 trials were measured using the Gigabit Ethernet network on the Palmetto cluster. Each individual trial is again graphed as a single scatter-point. The measured bandwidth using an identical file system configuration in simulation is shown using the connected line-points. The write bandwidths shown in Figure 3.10(a) are approximately the same with the simulated

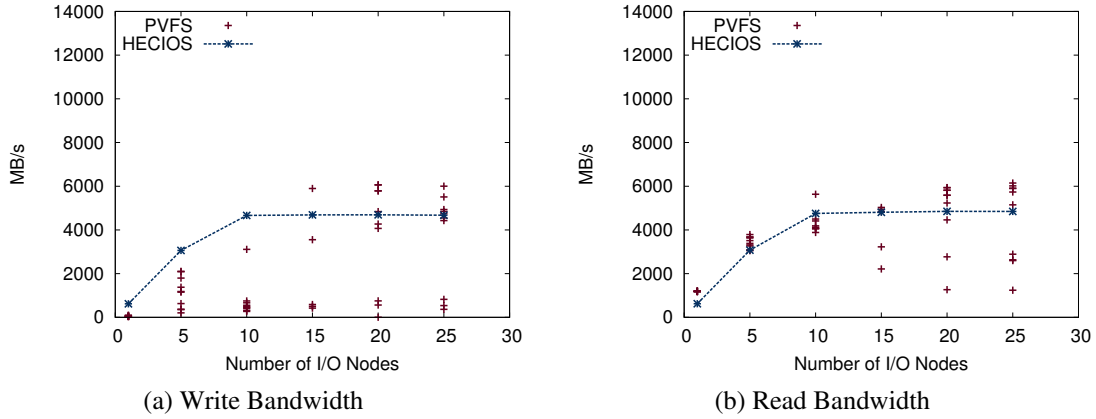


Figure 3.10: Comparison of empirical and simulated parallel file system bandwidth using 64 file system clients (8 processes per node) connected with Myrinet Express (MX) using Myri-10G fabric.

network providing higher aggregate bandwidth for several configurations, but generally tracking the observed results. Figure 3.10(b) shows the aggregate read bandwidth curves, and generally provides a good model of the real world system.

Figure 3.11 shows the same aggregate read and write bandwidth curves we have observed in the previous graphs this time with the MPI I/O Test benchmark running on 32 nodes with 8 processes per node. In Figure 3.11(a) we note an exaggerated inflection point in the simulated write performance at 5 I/O nodes and again at 20 I/O nodes. The inflection points are performance outliers, however only the configuration with 20 I/O nodes appears to be substantially higher than the real system. The simulated read performance shown in Figure 3.11(b) closely tracks the real system performance, although the simulated system provides slightly higher read bandwidths for all configurations with more than 10 I/O nodes.

Finally, Figure 3.12 shows aggregate read and write bandwidth for the MPI I/O Test benchmark running on 64 nodes with 8 processes per node. Figure 3.12(a) shows large differences between the simulated write bandwidth and Palmetto's write bandwidth

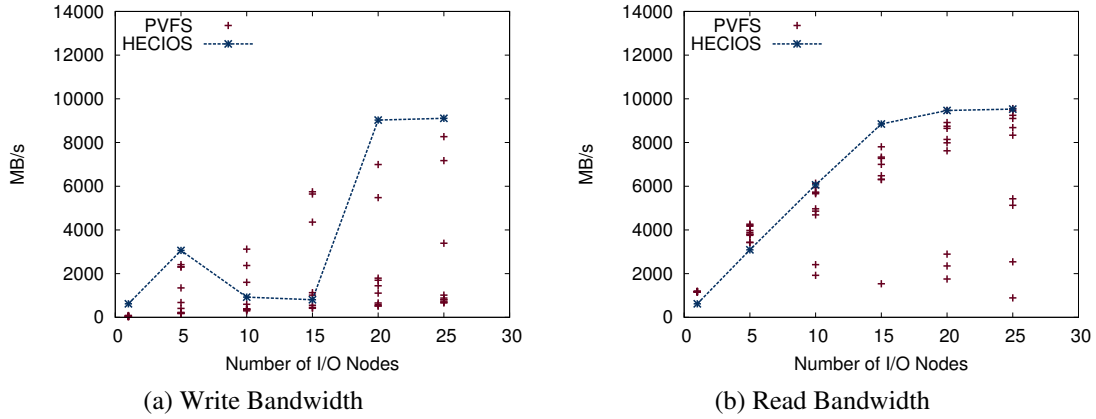


Figure 3.11: Comparison of empirical and simulated parallel file system bandwidth using 128 file system clients (8 processes per node) connected with Myrinet Express (MX) using Myri-10G fabric.

for configurations with less than 20 I/O nodes. In this case we wonder if PVFS or Palmetto is not simply misconfigured when performing I/O bandwidth tests using large numbers of nodes. We have modeled the extreme in-cast exhibited for configurations larger than 20 I/O nodes; however, we were unable to configure the simulator to match the depressed write performance with small numbers of I/O nodes. The simulated read performance shown in Figure 3.12(b) much more closely approximates the real system, although by limiting the buffer sizes to induce switch in-cast for file writes we have also induced some degree of switch in-cast in the final read bandwidth data point. We include further discussions of switch in-cast as we encounter them in our later experiments.

3.2.3.3 FLASH I/O Benchmark

For the Myrinet network we tried to run the FLASH I/O benchmark at larger scales, however we were only able to successfully run the benchmark using 8 and 16 client nodes, as the Myrinet hardware was unreliable when running the FLASH I/O benchmark on both a large number of client nodes and I/O nodes simultaneously. In general, the limiting fac-

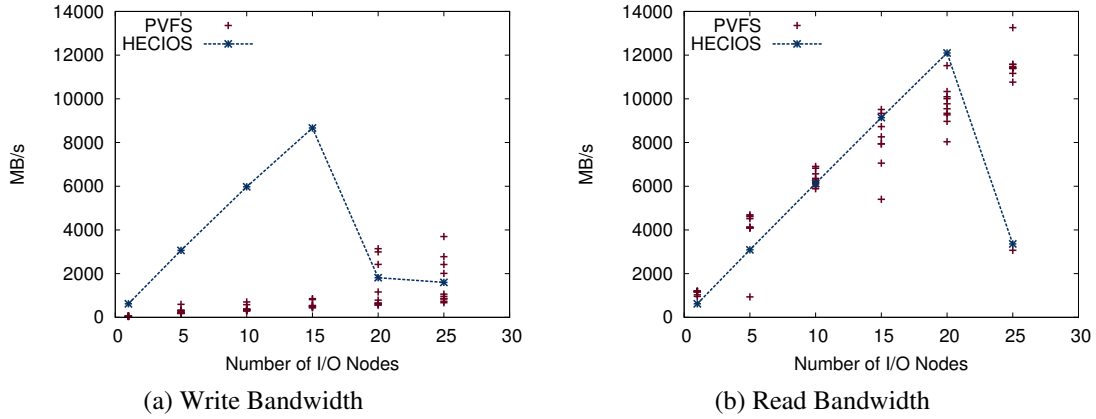


Figure 3.12: Comparison of empirical and simulated parallel file system bandwidth using 256 file system clients (8 processes per node) connected with Myrinet Express (MX) using Myri-10G fabric.

tor for the performance of the FLASH benchmark is network and disk latency rather than the available I/O bandwidth. Although the network provides low latency message passing, the intensive nature of the FLASH I/O benchmark overwhelms the network queues at the clients, limiting the scalability of the benchmark. Figure 3.9 shows the simulated and observed total runtimes for the FLASH I/O benchmark using 64 client processes and 128 client processes respectively. Our simulated results are generally very accurate, although HECIOS provides better performance in the case of a single I/O. Because the single I/O node case is mainly useful for computing theoretical speedups rather than a configuration used for a parallel file system (obviously 1 I/O node exhibits no parallelism), we are satisfied with the simulations FLASH I/O results.

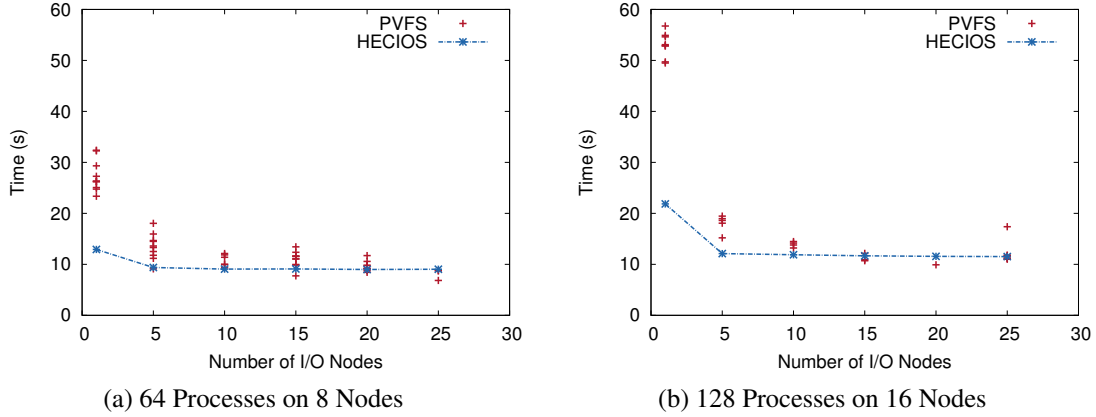


Figure 3.13: Comparison of empirical and simulated parallel file system execution time running the FLASH I/O benchmark on Palmetto with the Myrinet 10G interface (lower is better).

3.3 A Case Study: Using Server-to-Server Communication in Parallel File Systems

HECIOS is designed to closely mimic the existing parallel file system, PVFS. As such, we expect it to provide an accurate model of the parallel file system performance for an application's trace. However, in order for HECIOS to be a viable tool for our study of caching behavior in parallel file systems, our simulator must also easily provide an accurate model of novel parallel file system enhancements. In order to verify the effectiveness of HECIOS as a tool for modeling parallel file system improvements, we have chosen to implement a cutting edge performance improvement and verified the simulated performance against a prototype implementation in PVFS.

3.3.1 Prototype Implementation

In addition to a simulation study, we have also implemented the proposed optimizations in PVFS, a widely available parallel file system. Our experiments were performed

using the number of file system servers as the independent variable and the operation execution time as the dependent variable. We gathered 35 samples for each file system configuration. The first two samples were discarded to avoid experimental noise related to startup costs. The remaining 33 samples achieved an approximate normal distribution about the sample mean and standard deviation. The presented data omits extreme outliers. An extreme outlier is defined as being less than $(Q_1 - 3(IQR))$ or greater than $(Q_3 + 3(IQR))$, where Q_1 and Q_3 represent the first and third quartile of the sample set, and IQR represents the maximum of the interquartile range or $5\mu s$.

3.3.1.1 Jazz System Configuration

In order to time the individually optimized file system operations we used Argonne National Laboratory's Jazz cluster. Jazz is made up of 350 compute nodes, allowing us to scale each operation over a large number of I/O nodes to determine the operation performance at scales rarely seen for production parallel file systems. Each node contains a 2.4GHz Pentium Xeon processor, at least 1GB of RAM, and an 80GB IBM hard disk. The compute nodes run a Linux 2.4.26 kernel with ext3 local file systems – it is on these nodes that we configured and ran PVFS servers, transforming the compute nodes into *de facto* I/O nodes for our purposes. The nodes are interconnected with both a 100Mb/s Fast Ethernet network and a high-performance Myrinet-2000 network based on the PCI64C network interface card with a message latency of $6.7\mu s$ [48]. The transmission time of small messages sent during metadata operations are primarily limited by message latency, rather than the available bandwidth.

3.3.2 Server-to-Server Communication

In his dissertation, Carns proposed collective communication between the I/O servers as a mechanism to simplify consistency control and improve the performance of three important file metadata operations: create, stat, and delete [9]. Using his techniques and prototype as a reference, we are able to evaluate the effectiveness of HECIOS in measuring the performance of an innovative parallel file system modification.

Carns' proposed technique is based on the observation that as parallel file systems increase in scale, efficient metadata access becomes more difficult. Although client driven serial metadata techniques may perform adequately for a few hundred clients accessing tens of metadata servers; when thousands of application processes attempt to simultaneously create files, remove files, and list the contents of a directory, the performance of client driven metadata operations directly impacts the number of storage nodes that can be deployed in a parallel file system. Additionally, the difficulty in maintaining a consistent view of the file system during independent and simultaneous multi-step metadata operations encourages file system developers to deploy complicated distributed locking approaches that increase fragility and further impact scalability. The use of server-to-server communication neatly addresses both of these problems. Server-to-server communication in a parallel file system improves the scalability of the file system by simplifying consistency control for metadata operations and improving performance by leveraging collective communication techniques to perform metadata operations more efficiently.

3.3.3 File Creation

The process of creating a file in a parallel file system requires a considerable amount of work. A metadata object must be initialized and populated with values, data objects must be initialized on each storage server, and a directory entry must be added in the parent di-

rectory. Although it is possible to perform some of the work in an “as needed” fashion; lazy creation techniques are unlikely to improve the performance of typical application work loads. Typical file creation use cases (e.g. copying files or data collection) immediately follow file creation with writing a substantial amount of file data to the file system. In such cases the notion that the file system can populate only the metadata on an eager basis, and then initialize the data storage resources on demand is unlikely to result in better performance. Object pre-creation strategies have been shown to improve performance [20], but our optimizations are independent of such techniques, and so we present metadata algorithms that do not rely on pre-created data objects.

The file creation algorithm listing (and all following listings) use the following conventions for brevity:

- C: represents a client process
- D: represents a data server
- M: represents a metadata server
- P: represents a parent directory server
- \rightarrow : represents a request sent from a client to a server (response is implied)

For example, the statement “C \rightarrow M create metadata object” indicates that a request was sent from the client to the metadata server in order to create a metadata object. Each algorithm step is implemented with atomic semantics. The client-driven file creation process is shown in Figure 3.14.

First, the client retrieves the parent directory’s attributes to verify that creation permissions exists. The client then creates a metadata object on the metadata server, and creates the data objects on the data servers simultaneously. Once all the data objects are

```
1 C → P get parent directory attrs
2 C → M create metadata object
3 for each D:
4     C → D create data object
5 C → M set file attributes
6 C → P create directory entry
```

Figure 3.14: Client-initiated file create algorithm

initialized, the file metadata is populated and the parent's directory entry is created. By ordering the operations carefully, the need for a distributed locking system that acquires all the resources preemptively is avoided. In the scenario where two clients attempt to simultaneously create the same file, only one client will be able to successfully write the directory entry; however, the client that fails to create the file will need to perform further work to clean up the orphaned metadata and data objects.

In addition to the possible leaked resources during creation failure, data object creation is slower than it needs to be. By making the client responsible for performing all the communication necessary to create the data objects, the server disk activity can be overlapped but the client's network link becomes a serialization point and a bottleneck for interacting with the data servers. The collective algorithm shown in Figure 3.16 resolves both of these issues.

In this algorithm, the server responsible for the parent directory entry is contacted by the client to perform the file creation. The parent directory server serializes local access to the parent directory and then creates the metadata object. The parent server and data servers collectively implement the binary tree communication algorithm shown in Figure 3.15 to create all of the data objects for the file. The parent server populates the metadata and creates the directory entry before unlocking the parent directory and signaling success to the client.

This algorithm simplifies consistency management because the parent directory can

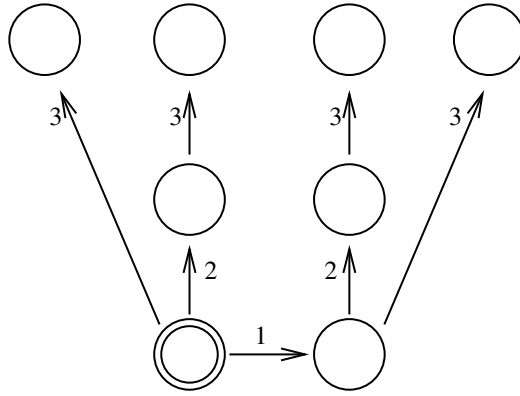


Figure 3.15: Binary tree collective communication

-
- 1 C → P file create request
 - 2 P locks parent directory
 - 3 P → M create metadata object
 - 4 for each D: (collective)
 - 5 P → D create data object
 - 6 P → M set file attributes
 - 7 P creates directory entry
 - 8 P unlocks parent directory
 - 9 C ← P aggregate response
-

Figure 3.16: Collective file create algorithm

perform local serialization on the directory entry. There is no possibility of two clients making partial progress toward creating the same file. Additionally, if the collective communication fails, the parent can simply unlock the parent directory locally rather than depend on a remote lock timeout mechanism. The algorithm also improves performance because the binary tree collective communication creates all of the data objects in time proportional to $O(\log_2(n))$ rather than $O(n)$ where n is the number of data objects for the file.

Figure 3.17 shows the time to create a single file for each file system configuration (i.e. the number of file system I/O nodes). Figure 3.17(a) shows the predicted time to create a file using both the client-based create and the collective create algorithm using the HECIOS simulator tuned to match the performance of the Jazz cluster. Figure 3.17(b)

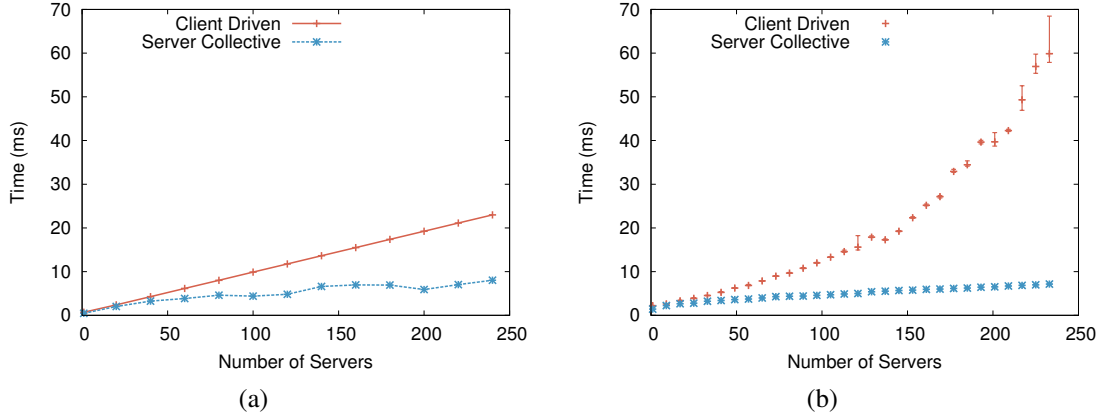


Figure 3.17: File Create Performance on Jazz (Simulation)

shows the creation times for both algorithms as measured on Jazz. The increasing slope demonstrated on the real system is largely due to the increasing cost of the *poll* system call as the number of active sockets grows [9]. The Linux 2.4.26 kernel in use on Jazz does not support the more modern *epoll* system call that provides roughly $O(c)$ socket polling performance. Our simulator does not impose an increasing performance cost as the number of sockets increases and thus the file creation performance using the client-driven algorithm is roughly linear in the number of I/O servers.

3.3.4 File Removal

File removal can be a very resource intensive operation for parallel file systems. File data may be distributed over tens or hundreds of data servers and orphaned data objects may result in a significant loss in storage capacity until the file system can be repaired (usually via an offline file system check). Figure 3.18 lists the basic client-driven file remove algorithm.

The client-initiated file removal algorithm demonstrates the difficulties in developing a file system server without the use of distributed locking. Consider the scenario where

```

1 C → M get file attributes
2 C → P remove directory entry
3 C → M remove metadata object
4 for each D:
5     C → D remove datafile object

```

Figure 3.18: Client-initiated file remove algorithm

one client attempts to delete a file while another client is simultaneously modifying the file system permissions. Client 1 initiates the remove, and succeeds in removing the parent directory's entry for the file. Client 1 then attempts to continue the removal process by deleting the file metadata, but a network timeout occurs causing the metadata removal to fail. At the same time, client 2 modifies the permissions of the parent directory so that further modifications by client 1 are not allowed. Client 1 can then attempt to recreate the directory entry in the parent directory; however, since client 2 has modified the parent permissions, the file has been deleted. This outcome results in a large number of orphaned data files that waste significant storage space. Even in the case where the application code is simply interrupted immediately after the parent directory entry has been removed will result in orphaned data objects and wasted space until a file system check can be performed to recover the storage space.

```

1 C → P aggregate remove request
2 P locks parent directory
3 P → M get metadata attributes
4 for each D and M: (collective)
5     P → D remove data object
6     P → M remove meta object
7 P remove directory entry
8 P unlock parent directory
9 C ← P aggregate response

```

Figure 3.19: Collective file remove algorithm

The server-driven file removal listing in Figure 3.19 does not exhibit this behavior.

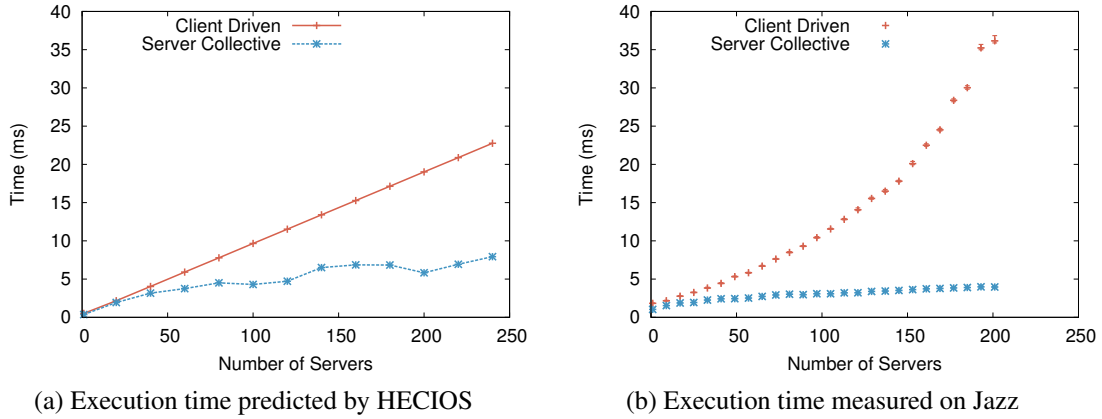


Figure 3.20: File Remove Performance

By having the parent directory's server control the remove, access to the parent directory can be trivially serialized with a local lock, and a change in the permission's of the file targeted for removal is detected by the server and no repairs are necessary. Even if the client is interrupted, the parent directory server will fully complete the remove operation.

Figure 3.20 shows the time to perform a single file remove, i.e. remove the file data, metadata, and parent directory entry for each file system configuration (i.e. the number of file system I/O nodes). Figure 3.20(a) shows the predicted time to remove a file using both the client-based file delete and the collective file deletion algorithm using the HECIOS simulator tuned to match the performance of the Jazz cluster. Figure 3.20(b) shows the deletion times for both algorithms as measured on Jazz. We again see the increasing slope due to the increased cost of the *poll* as the number of I/O nodes participating in the file delete increases.

3.3.5 File Stat

File stat, while rare in parallel applications, is common during system administration and interactive data set management activities. One of the most common ways a

user invokes the file stat command is using the UNIX utility *ls*. The *ls* command lists the contents of a directory and, optionally, each entries attributes (e.g. permissions, last modification time, file size). Efficient performance in the file stat command is critical to easing data management activities and improving the file system's interactivity. PVFS, like most file systems, employs a client-side attribute cache to avoid retrieving file metadata before every file interaction (e.g. to check permissions); however, metadata fields such as the file size and access time (atime) are not kept up to date for the same reason – to avoid writing metadata attributes after every successful file I/O. The file stat operation allows the user to query all of the file's metadata. The client-initiated file stat algorithm is shown in Figure 3.21.

```
1 C → M get metadata attributes
2 if file size is requested:
3     for each D:
4         C → D get data attributes
5     C compute logical file size
```

Figure 3.21: Client-initiated file stat algorithm

One important feature to note is that the client is only required to contact the data servers if the user has requested the file size. The more common operation of requesting the file's permissions only requires checking the attribute cache and contacting the file's metadata server if the cache entry does not exist.

The collective file stat operation in Figure 3.22 differs from create and remove in that the file's metadata server initiates the collective communication rather than the parent directory server. Also, the collective data attribute request (Steps 5 and 6) acts like a gather operation instead of a reduction. The metadata server receives all of the data object attributes rather than just the computed size or latest access time. For simplicity, we prefer to perform the metadata calculations at the meta server, but a distributed reduction algorithm

```

1 C → M aggregate stat request
2 M locks metadata object
3 M get metadata attributes
4 if file size is requested:
5     for each D: (collective)
6         M → D get data attributes
7     M computes logical file size
8 M unlocks metadata object
9 C ← M aggregate response

```

Figure 3.22: Collective file stat algorithm

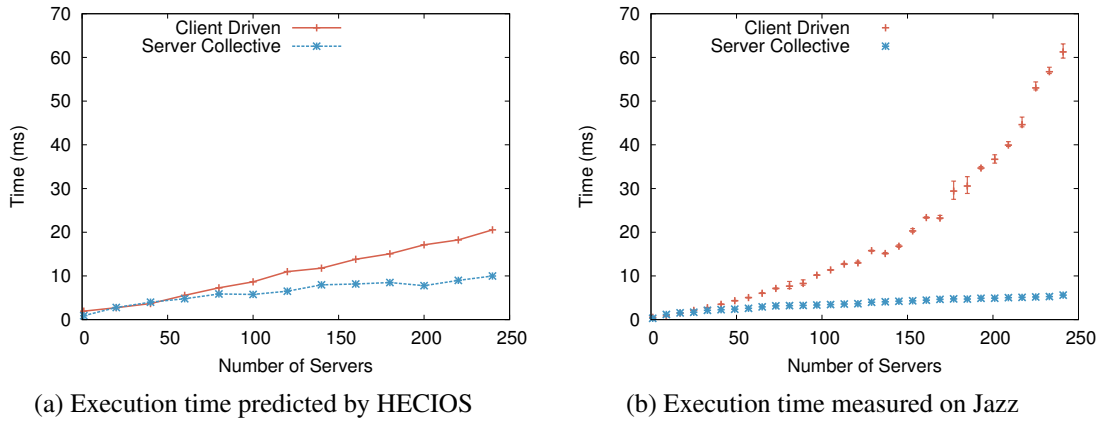


Figure 3.23: File stat Performance

would be a reasonable improvement upon this algorithm.

Figure 3.23 shows the time to perform a single file full stat command, i.e. an operation that contacts each of the I/O nodes to determine an accurate accounting of file metadata such as *file size* or *last access time*. Figure 3.23(a) shows the predicted time to stat a file using both the client-based file get-attributes and the collective file get-attributes algorithms using the HECIOS simulator tuned to match the performance of the Jazz cluster. Figure 3.23(b) shows the stat times for both algorithms as measured on Jazz. We again see the increasing slope in the observed data from Jazz due to the increased cost of the *poll* as the number of I/O nodes participating in the file delete increases [9].

3.3.6 Interactive Workload Evaluation

We also measured the performance of the improved collectives in real world scenarios that require the manipulation of a large number of files. We chose three tasks commonly performed by both software developers and system administrators. In the first representative test, we extracted all of the files from a Linux-2.6.9 source file archive in tar format. In the second test, we performed a full listing of all of the files in the resulting source tree, which relied heavily on the implementation of the file stat operation. Finally, we performed a recursive remove of all of the source files. The 2.6.9 Linux kernel source is composed of over 1,000 directories and 16,000 files, with most of the files having a small or moderate size. Figure 3.24 shows the runtime in seconds for the three file manipulation tasks on a parallel file system configured with 74 dedicated storage nodes using both the simulation software and the prototype implementation running on the Adenine cluster (100Mbit/s network).

Adenine System Configuration Clemson University's Adenine cluster is composed of 75 compute nodes. Each compute node contains dual Pentium III 1GHz processors with 1GB of RAM and 30GB Maxtor hard drives. All nodes are connected by a 100Mb/s (Fast Ethernet) network, while 48 nodes also have a 1Gb/s (GigE) network connection. Each network uses a single, independent dedicated switch. The compute nodes run a Linux 2.6 kernel with an ext2 file system.

The measurements in Figure 3.24 exemplify the idea that optimizing the performance of the most critical operations can provide substantial execution time improvement. Each of the three tasks is composed of file system operations other than file creation, removal, and stat; for example, archived file extraction requires file I/O, directory creation, file creation, and metadata updates for atime. Still, the performance improvement of nearly 49% can be attributed entirely to the improved efficiency of the collective file creation

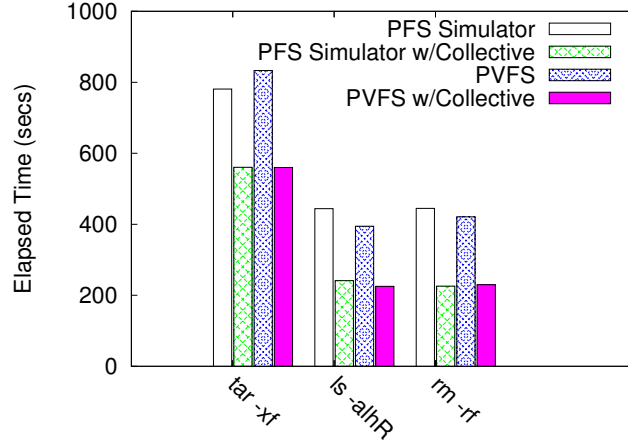


Figure 3.24: Kernel Manipulation Performance on Adenine

technique. Similarly, recursive file listing and recursive file removal require additional metadata operations such as *readdir*; however, the performance speedups in using collective file stat and collective file remove improve the benchmark execution times by 75% and 83%, respectively. In summary, with only the collective metadata operations presented in this paper, we were able to significantly improve the performance and interactivity of these three common developer and administrative tasks.

3.4 Summary

Although our simulator was not able to provide a perfect simulation of the Jazz cluster due at least in part to the behavior of the outdated *poll* system call, HECIOS was able to demonstrate the basic performance characteristics of the algorithmic improvement from linear time to logarithmic time. Our simulator was able to provide an adequate simulation of how the proposed modifications would effect a diverse metadata intensive workload. HECIOS has been designed to provide an accurate model of actual MPI I/O codes and benchmarks rather than single operation micro-benchmarks, so it is not surprising that

HECIOS provides more accurate results with trace-based workloads. In summary, we believe that this case study demonstrates that our simulator is able to provide a realistic platform for performing experiments in parallel I/O. The simulation runs and prototype implementation yielded similar results for a complex parallel file system optimization.

Chapter 4

Fixed-Size Page Caching

Traditionally, data caching is used to improve application performance by improving the locality of reference for frequently used data locations. For example, the Linux kernel's page cache uses any available main memory to cache frequently accessed disk blocks to avoid spending time repeating disk accesses to the same file locations. Similarly, the disk cache on modern hard drives attempts to store the most recently accessed disk sectors on a small RAM buffer on the drive, and, more speculatively, the disk controller will attempt to pre-fetch sectors into cache that are likely to be requested in the future. Network file systems, including parallel file systems, typically employ a client-side name cache and a client-side metadata cache in order to avoid repeatedly checking the permissions of parent directories on a remote server and to avoid beginning all path resolutions with the root directory server.

One of the fundamental decisions in the architecture of any cache is: How to map the address space of possible values into cache locations? Most approaches begin by evenly subdividing the address space into fixed-size blocks (usually called lines in processor caches or pages for disk-backed caches). The advantage of using a fixed-size cache page is the ease of implementing a mapping algorithm from the real address space to the corre-

sponding cache location, the ease of evicting data from the cache, and the small amount of cache metadata required to describe the cache address space and determine the next eviction.

One of the simplest address mapping transformations, called direct mapping, can be calculated by taking the address modulo the number of blocks the cache can store. More concisely, a direct mapped cache uses the following mapping function to determine the cache location for an address:

$$Address_{cache} = Address_{global} \pmod{Size_{cache}}. \quad (4.1)$$

A direct cache mapping is simple and fast to calculate, although it may lead to continual block evictions if two frequently used addresses translate to the same cache address. Fully associative mapping schemes enable any cache block to become a duplicate of any aligned block of the address space, and while expensive to implement in hardware, software implementations are straightforward and reasonably efficient.

The second major advantage of fixed-size blocks for caching is the ease in determining which blocks to replace once the cache is full and more data is requested. The most efficient choice for the next fixed-size block to evict from the cache is the block that will be used the furthest into the future. Generally, the information needed to implement this scheme is not available and an approximation called the “Least Recently Used” cache algorithm is often preferred. In software, a linked list can be updated each time a cache block is accessed to ensure that list is sorted from most recently used to least recently used. Evictions blocks are calculated by accessing the tail of the linked list. Hardware cache implementations that have limited resources for maintaining an LRU list may simply resort to randomized eviction.

Finally, caches with a fixed block size typically require smaller amounts of storage

overhead than alternative designs. The implementation details for fully associative caching have been described in detail in literature (e.g. Hennessy and Patterson [26]), essentially the overhead boils down to nk tag bits per cache block, where n is the cache capacity in blocks and k is at most the number of bits in the full address store, and $n\log_2(n) + 2$ bits for storing the LRU list pointers. Because $\log_2(n)$ must be less than k , and k must be less than n , the overhead of a direct cache is $O(n)$, or simply, linear in the number of cache entries. As the small cache overhead implies, fixed-size block caches are relatively easy to implement and the data blocks within the cache are easy to locate and evict.

4.1 File Data Cache Architecture

A contiguous page of file data is the basic block of storage within the file data caches we examine in this chapter. The size of the contiguous pages is a cache parameter that is supplied at run time, allowing us to sweep the entire cache page size parameter space to determine the effects of page size on application performance. Although we have performed our parameter sweeps using powers of two, it is possible that some application workloads would achieve better page alignment with a non-power of two page size. Additionally, because our underlying file system does not provide its own caching or a mechanism for performing advisory locking on some fundamental block size we have been able to examine the performance of page sizes smaller than 4KB, the traditional locking granularity on many Posix file systems.

In order to transparently store and retrieve file data at the client, our middleware cache intercepts and interprets all MPI-I/O calls to file open, close, read and write from the application to the file system. Essentially, our cache is implemented as a file system driver within MPI-IO that wraps the actual target file system. For each file open and close the cache respectively increments and decrements the reference count for the file name within

the cache. When the reference count reaches 0, all dirty pages associated with the closed file are flushed from the cache. As we will discuss in section 4.1.1, this behavior performs a critical role in our selected data consistency scheme.

Cache page lookups are performed using a compound key based on the file name, and the file page id, which is simply the file page number determined by calculating the desired file offset divided by the page size.

$$Key_{filename} = Request_{filename}$$

$$Key_{pageid} = Request_{offset} / Cache_{pagesize}$$

The cache is fully associative so that any page of the file can be stored in any block in the local cache. As described earlier, the storage overhead associated with the cache $O(n)$.

The cache uses a write-back model for updates, meaning file writes that can be successfully copied into the cache are allowed to complete immediately. As part of any cache update (due to a read or write), evicted dirty pages are written back to the file system synchronously before completing the operation. The cache page eviction model is based on a simple sorted LRU list where, such that each cache page update or access results in the key being moved to the tail of list.

Finally, it is important to note that file reads and writes that are larger than the entire cache capacity bypass the cache rather than causing a series of updates and evictions. In the case of a large file write that triggers a cache bypass and provides new data for pages currently in cache, the cache data will be updated and the bypass write will occur as describe before. Similarly, a large bypass read will interact with the file system first, and then any newer data residing in the cache will be copied on top of the read. Additionally, while a bypass read or write is occurring, the cache must block until the bypass operation

completes its interaction with the underlying file system. This is necessary to insure the consistency of the file data returned by the bypass operation.

4.1.1 Data Consistency

The data consistency model describes the ordering model for how updates to shared file regions are applied to actual underlying files. In general, programmers typically expect the ordering model to provide *sequential consistency*, that is, file writes should be performed atomically in the order they are completed and all file reads should return the value of any file write that has previously completed. We can paraphrase the formal definition provided by Lamport as follows [39]:

A parallel I/O program is sequentially consistent if the contents of each file is the same as if operations of all the processes were executed in some sequential order, and the operations of each individual process in this sequence in the order specified by its program.

Unfortunately, sequential consistency is difficult to provide in general, and, since it requires a centralized clock or synchronization point, sequential consistency is unlikely to provide high levels of performance.

Fortunately, more relaxed consistency models are both popular and ubiquitous in parallel computing. Our caches typically use the popular close-to-open file consistency model employed by NFSv4 [24]. Close-to-open consistency is an eager release consistency model that ensures all persistent file data is updated when the file is closed and that any subsequent file opens are guaranteed to see the file data committed during all previous file close operations. In order for a client process to see the most up to date version of the file data, it only needs to close all references to the file and perform a new file open.

One important element of close-to-open consistency that may not be immediately obvious, is the removal of any guarantees of atomicity in write operations. That is, if two processes, A and B, begin simultaneously writing bytes 10 to 20 of file `/foo`, there is no requirement that a later reader, called process C, sees only exclusively the contents of the write by A or exclusively the contents of the write by B. It *is* perfectly valid for some of the first ten bytes to come from process A's write, and some of the ten bytes to come from process B. Note that all of the bytes must come from either process A's file write or process B's file write; it is not valid for any of the original file data to remain. Also note that it is possible to use synchronization via `MPI_Barrier` calls to ensure that the contents of the file are updated atomically; however, writes to the file must be serialized in order to achieve an atomic update semantic.

In some of our experiments we will further relax the consistency model beyond the allowances of close-to-open consistency. We provide a further explanation of the relaxed consistency model as we describe the results of those experiments.

4.1.2 Multiple Writers

In our previous discussion of data consistency we described the consistency requirements at the I/O operation level (e.g. a file read or write), however, one may naturally assume that the concern about generating a sequential program ordering is only critical for the file regions that are read and written by multiple processes. Unfortunately, when using a fixed-size block cache this assumption is false.

The problem is that when a fixed-size block cache is employed, a file write does not just affect the file region described by the write, but instead all of the cache pages the region lies upon. Again consider two processes, A and B, writing to a file, `/foo`. Process A is writing the file byte range 0 - 9, and process B is writing the file byte range 10 - 19. It

would appear that the file must contain the data from both writes once both processes have written and closed the file. However, if the cache page size is 100 bytes, the individual caches cannot just update the ten bytes in question. Instead, the entire page must be evicted from the process caches and either the first 100 bytes from process A will be in the file, or the first 100 bytes from process B will be in the file. Either way, the result is not what a programmer would expect, and close-to-open consistency is violated (the updated data for one of the file regions will not appear to future readers after a successful file close).

This problem, called *false sharing*, results when two unrelated data regions cause incorrect result to be committed to the file system wholly due to the page granularity used in the cache architecture. Fortunately, false sharing is a well researched problem, with many different solutions [17].

4.1.2.1 Cache Page Directory

The false sharing problem described in the previous section is a problem of cache coherence: when the second process attempts to write the updated cache page to file, it no longer has a coherent view of the file. The obvious way to resolve false sharing is to insure that each page of file data is only cached at one location at a time. While this idea returns the correct result, it is overly pessimistic in requiring a cache page to reside at only one location. Instead, we must ensure that a cache page is only *updated* at one location at a time, and any cache page that a file write needs to update must include other pending updates for that page (note that this condition provides a stronger guarantee than close-to-open consistency, but is still a sufficient implementation of close-to-open consistency).

In order to ensure that pages are updated at only one cache at a time we can construct a state directory that maintains the state of every cache page [18]. In order to update a cache page, a client process must hold the cache page in an *exclusive* state (i.e. only the one client cache has the page stored locally). On the other hand, if a process wishes to read from a

cache page, many client processes can simultaneously hold the page in a *shared* state. The cache page directory then must be able to store the current state of the cache page and locators for finding the client process or processes that locally store the page.

Because of the frequent messaging required to update the state of the cache pages, a distributed directory approach is usually preferred over a single centralized cache page directory [11, 41]. The most straightforward scheme is to simply store the directory entry for cache pages on the *home node*, or the same node that stores the backing store's version of the data. For a parallel file system, the home node is simply the I/O server that persists the corresponding portion of the file, and the file system server process must be modified to maintain a cache page state directory alongside the file data.

There are several problems with the I/O server as home node approach. First, current cluster messaging schemes typically only support message transmissions between processes within a single MPI execution context (e.g. a parallel job). However, I/O servers are UNIX daemons that are not part of any MPI context, and further, multiple MPI jobs may attempt to simultaneously access the same file requiring the I/O servers to concurrently participate in multiple MPI execution contexts. One possibility for implementing such a scheme is the use of multipurpose daemons (MPDs) that manage all cluster processes and may be implemented with a mechanism to send directory cache protocol messages to any process running on a cluster computing system [7].

Another problem with directory-based approaches is integrating directory messages into our proposed shared caching enhancements. In a shared cache, multiple processes share the cache pages, and, concomitantly, must also share the cache page state. With multiple processes sharing the data, the question becomes which process should the home node contact to notify the processes of a cache state change (e.g. from shared state to invalid state). If the process contacted is performing intensive processing, the cache interruption could trigger significant performance loss due to context switching costs. On the

other hand, if the message is queued and handled during the next I/O phase, the processes requesting the state change are forced to idle while waiting for cache directory service.

One solution is to simplify the cache directory protocol to exist only in the client processes as part of the middleware cache. In this approach, we can segment the cache directory entries evenly amongst all of the client processes (using round-robin page assignment or some other scheme) and manage the directories within the middleware cache. This approach is feasible; however, it is unlikely to be an effective scheme for experimenting with cache sharing due to the previously stated problems of determining which process to contact for directory state changes. Additionally, our simulator is not well instrumented to capture the performance costs of the cache protocol overhead. For these reasons, we have decided not perform a performance analysis of a directory-based middleware caching protocol.

4.1.2.2 Cache Page Differences

The complexity of directory-based caching protocols has led us to evaluate a different false sharing avoidance schemes that leverages a simpler protocol at the expense of additional cache storage overhead. The TreadMarks system [1], originally used to implement a shared processor virtual memory system, provides a simple, but effective scheme to avoid writing inconsistent file data that has resulted from false sharing. Before performing any write that does not encompass a full cache page, the cache must first read in the data page from the file system. The cache then makes a duplicate of the read page in memory, and updates the duplicate data. When the page is evicted from the cache (due to replacement or a file close), the updated duplicate page and the original page are compared, and only the differences between the two pages, or “diffs”, are written back to the file system.

The major drawbacks of this scheme are the additional cache storage required and the possibility that small, unaligned writes will accumulate sparsely over each page re-

sulting in the same poor performance we sought to improve in the first place. Below, we describe how our implementation of a page differencing scheme attempts to overcome several of these limitations.

4.2 Page Delta Cache

In Section 4.1.2.2 we described the basic scheme used by TreadMarks to avoid false sharing in cache pages. Our system modifies the basic TreadMarks approach so that when partial page data is written to the duplicate page, the original copy of the page is also updated in order to act as dirty mask for the updated data. The simplest way to accomplish the dirty mask operation is to perform a modified *memcpy* operation that writes data to the client's requested region on both the original copy and the duplicate copy. However, the data written to the original copy of the data should be the complement of the data written to the duplicate cache page.

The reason for modifying the original TreadMarks scheme is that we desire to increase the granularity of file system interactions, and rather than writing only the differences, which may be a disjoint subset of the updates, we prefer to update the maximum contiguous region possible. Additionally, our simulator cannot calculate the actual deltas on the file data because our file traces only include metadata describing the file I/O performed rather than the actual updated file data. One possible optimization is to use an actual dirty mask to track the updated file regions. The implementation details and an analysis of the storage overhead of such an approach is presented in Chapter 5 as part of our progressive paging description.

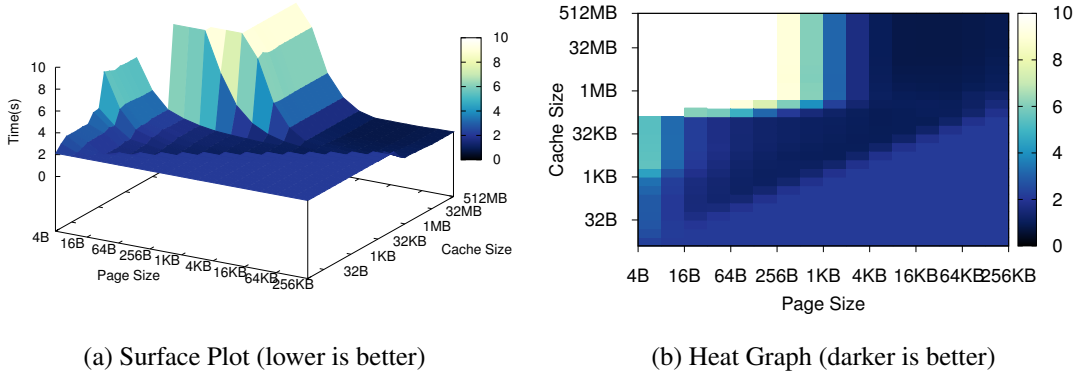


Figure 4.1: FLASH I/O benchmark execution time on 8 CPUs, 1 process per node, and 4 I/O Nodes over Gigabit Ethernet.

4.2.1 Measuring the Effects of Cache Parameters

In the following experiments, we attempt to gain an understanding of how the cache block size and cache capacity affect performance across several cluster platform configurations. We are using the popular FLASH I/O benchmark as a tool for examining how the different cache and platform architectures interact to affect application run times. Additionally, we use this analysis to guide cache reorganizations in order to improve performance.

Figure 4.1 shows a surface plot and heat graph expressing the total execution time of the FLASH I/O benchmark configured to run on a cluster with 8 single-core compute nodes connected via Gigabit Ethernet to a parallel file system consisting of 4 I/O nodes. The x -axis shows the cache page size in bytes and the y -axis shows the total cache data capacity in bytes. In the case of impossible page and capacity combinations (e.g. a 256KB cache page size with a 1KB cache capacity) the baseline performance of the benchmark execution time without caching has been provided instead. The non-caching times appear as an flat triangular plane on the surface plots and as a lower right triangle on the heat graphs. Finally, we have clamped execution times in excess of 10 seconds in order to improve the appearance of the graphs. Benchmark run times in excess of 10 seconds are

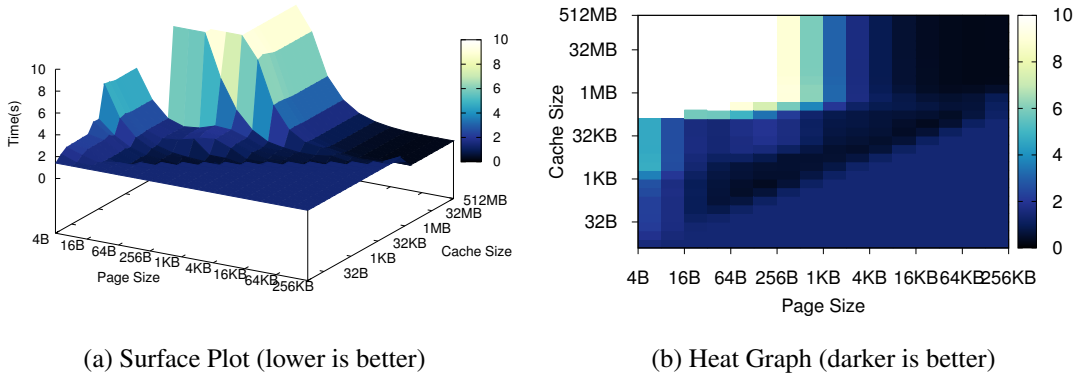


Figure 4.2: FLASH I/O benchmark execution time on 8 CPUs, 1 process per node, and 4 I/O Nodes over Myrinet Myri10G.

exhibiting an execution time slowdown of at least 100% versus the time required to run without caching.

Figure 4.2 shows the same 8 compute nodes, 1 process per node, and 4 I/O node configuration as in figure 4.1; however, in these results, the simulated interconnection network is using our Myrinet Myri10G network settings. The obvious performance feature for both configurations is the large run time spike for cache configurations with smaller page sizes (1KB and less) and large cache capacities. Secondly, we can see that the cache is effective at improving the benchmark performance when the cache capacity is sufficiently large, but not too large.

The relatively narrow band of high-performance cache configurations in figures 4.1 and 4.2 is particularly troubling. When the cache has too few pages, the performance is barely improved over the base case, and when the cache has too many pages, the performance may be much worse than if no cache had been employed at all. To improve our insight into this problem, we have re-indexed the data from figure 4.2 into figure 4.3 such that the x -axis shows the cache capacity in number of pages rather than cache size in bytes. The execution time for each page size is then plotted using scatter points. We have limited

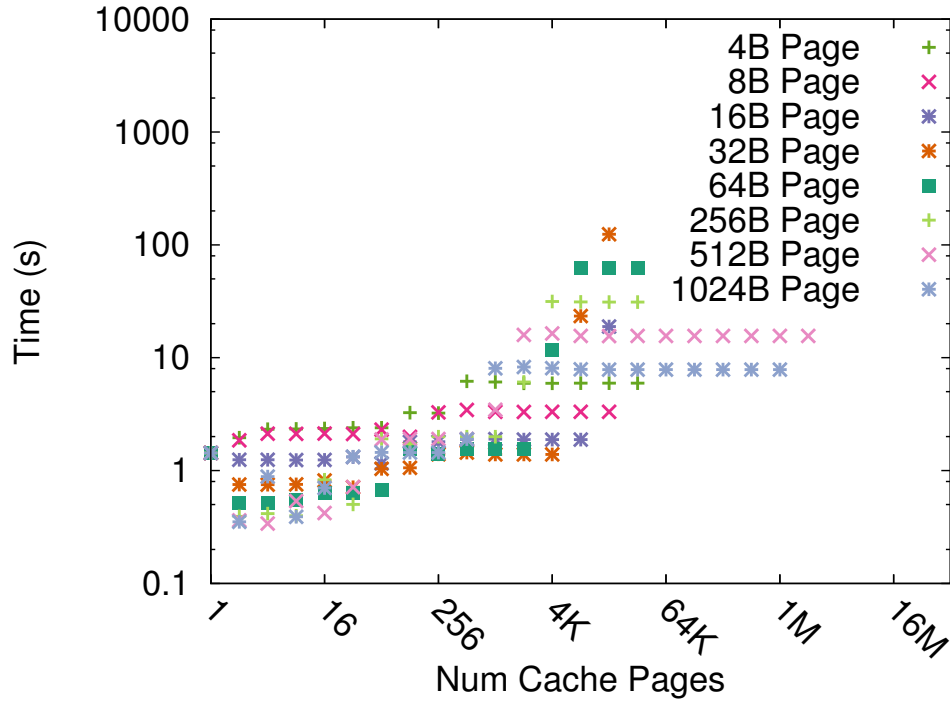


Figure 4.3: FLASH I/O benchmark execution time sorted by cache capacity in pages (log-log scale).

figure 4.3 to only the six smallest configurations to examine how the number of pages in the cache effects the overall execution time for small cache page sizes. As evidenced by the graphic, cache configurations with less than 64 pages of capacity tend to provide stable performance, but as the page capacity grows beyond 256 pages the run time for some configurations begins to increase rapidly (note: the y-axis has a logarithmic scale). Obviously, this leads to tuning difficulty for system administrators and end users because specifying a “too small” cache is unlikely to maximize performance, while selecting a “too large” cache may lead to even worse performance than not using a cache at all.

Tables 4.1 and 4.2 show the cache configurations that result in the best performance for each of the interconnection network types, respectively. We note that the timing results are for the entire FLASH I/O benchmark routine execution rather than just the time spent performing file I/O. Our cache cannot improve the performance of the benchmark’s non-

Page Size	Cache Size	Total Exec. Time	Speedup
64KB	16MB	0.8174	2.62
32KB	16MB	0.8186	2.62
16KB	16MB	0.8216	2.61
8KB	16MB	0.8296	2.59

Table 4.1: Best performing cache configurations on Gigabit Ethernet (8 CPU, 4ION).

Page Size	Cache Size	Total Exec. Time	Speedup
32KB	64KB	0.2546	5.62
32KB	128MB	0.2551	5.61
64KB	8MB	0.2718	5.27
128KB	16MB	0.2733	5.24

Table 4.2: Best performing cache configurations on Myrinet Myri10G (8 CPU, 4ION).

I/O related operations (e.g. file opens, barriers, and broadcasts). The highest speedup in table 4.1 is 2.62, which is far short of the maximum possible theoretical speedup of 7.21 available if each I/O node is accessed at the maximum possible I/O bandwidth of 117MB/s per I/O node – which is, admittedly, very unlikely. Similarly, the best performing cache configurations for the Myrinet network results in a remarkable speedup of 5.62; however, this is still well short of the maximum possible theoretical throughput.

4.2.1.1 Addressing Performance Volatility

The performance volatility demonstrated for small page sizes in figures 4.1 and 4.2 is largely an artifact of the network and file system becoming overwhelmed by the bursty stream of relatively small page updates. When the cache page size is small and the number of cache pages is small, the cache acts as a governor, ensuring that file system updates are fed to the file system at a steady, sustainable pace. As the number of cache pages increases the networking queues are overwhelmed and the file system performance becomes unstable; larger writes will force many page evictions while small file writes may

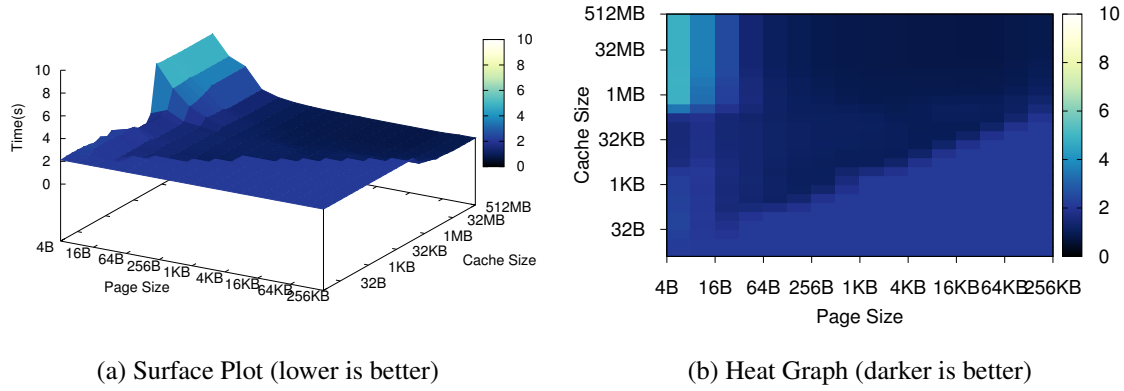


Figure 4.4: FLASH I/O benchmark execution time with the Block-Indexed data type optimization on 8 CPUs, 1 process per node, and 4 I/O Nodes over Gigabit Ethernet.

not force any evictions at all, reducing the average throughput. The unstable performance is not apparent with larger page sizes because the average file system interaction size is larger, reducing the total network traffic.

In order to improve the update granularity we have chosen to exploit the high level data type interfaces that parallel file systems such as PVFS support. Equivalent to the MPI File View interface available in MPI-IO, our simulator uses the file system support for non-contiguous data types (based on the interfaces of PVFS). We maintain the same fixed-size paging architecture of our first cache design. However, instead of interacting with the file system by reading and writing individual cache pages, we construct a Block-Indexed data type describing each of the pages to be read or written from the file system and perform all of the partial page evictions resulting from a write in a single large operation.

Figures 4.4 and 4.5 show the results of using the Block-Indexed data type to access the file system with both Ethernet and Myrinet, respectively. The data clearly indicates that employing the block-indexed data type to map the cache data into the file system improves the access granularity, reduces the performance volatility, and formulates a cache design that does not exhibit significant performance volatility or penalize the performance

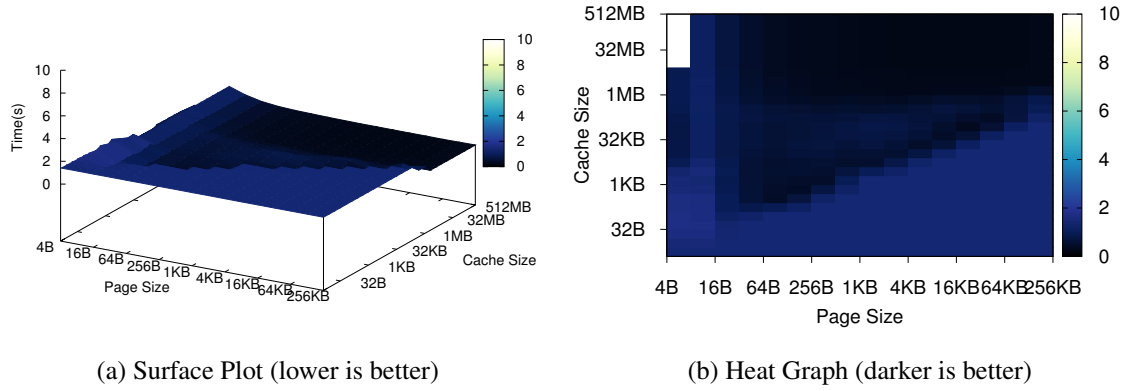


Figure 4.5: FLASH I/O benchmark execution time with the Block-Indexed data type optimization on 8 CPUs, 1 process per node, and 4 I/O Nodes over Myrinet Myri10G.

Page Size	Cache Size	Total Exec. Time	Speedup
32KB	16MB	0.8081	2.66
64KB	16MB	0.8086	2.65
16KB	16MB	0.8108	2.65
8KB	16MB	0.8170	2.63

Table 4.3: Best performing cache configurations using block-indexed access on Gigabit Ethernet (8 CPU, 4ION).

of cache configurations with small page sizes.

Tables 4.3 and 4.4 show the four best performing cache configurations with the block-indexed data type cache optimization for both Gigabit Ethernet networks and Myrinet Myri10G networks. The block-indexed data type appears to remove the penalty for large cache capacities, resulting in lower performance volatility, while providing roughly identical performance for the fastest cache configurations (we do not consider the performance improvements to be statistically significant).

Page Size	Cache Size	Total Exec. Time	Speedup
64KB	16MB	0.2367	6.04
16KB	16MB	0.2411	5.94
32KB	16MB	0.2452	5.84
8KB	16MB	0.2462	5.81

Table 4.4: Best performing cache configurations using block-indexed access on Myrinet Myri10G (8 CPU, 4ION).

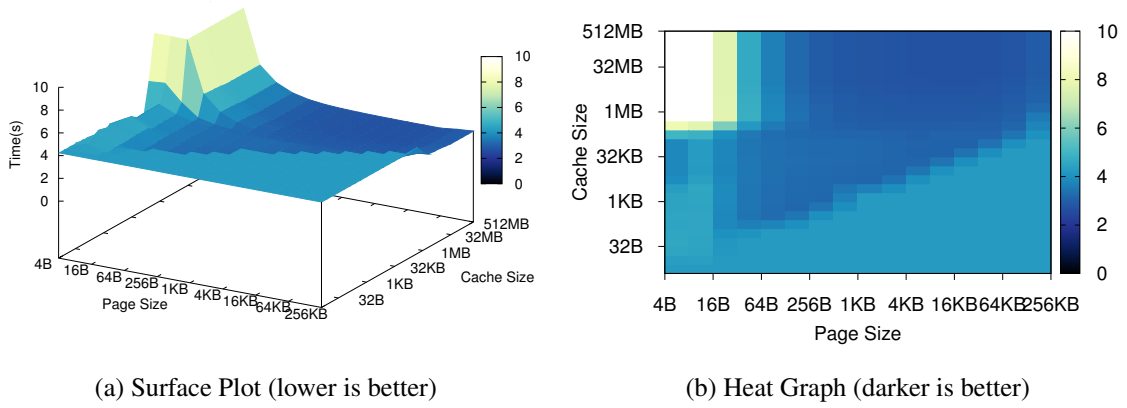


Figure 4.6: FLASH I/O benchmark execution time with the Block-Indexed data type optimization on 1 CPU, 8 processes per node, and 4 I/O Nodes over Gigabit Ethernet.

4.2.1.2 Leveraging Modern Cluster Hardware

Until this point, we have assumed that each compute node is running a single process with a single dedicated network link. Modern clusters, however, leverage the capabilities of multi-core processors to improve the system processing density which leads to several processes running on a single compute node, all sharing the same network interface. The increased number of clients per network connection typically leads to greater network utilization at the cost of decreased performance for each individual process due to network queue overruns and collisions. In this section we look at how this more popular cluster configuration affects caching performance.

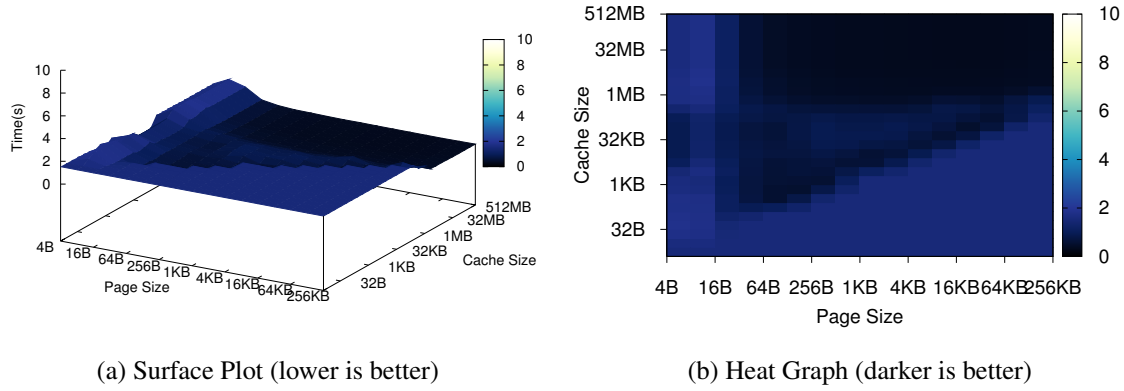


Figure 4.7: FLASH I/O benchmark execution time with the Block-Indexed data type optimization on 1 CPU, 8 processes per node, and 4 I/O Nodes over Myrinet Myri10G.

Page Size	Cache Size	Total Exec. Time	Speedup
32KB	16MB	2.7081	1.57
64KB	16MB	2.7089	1.57
16KB	16MB	2.7107	1.57
8KB	16MB	2.7107	1.57

Table 4.5: Best performing cache configurations using block-indexed access on Gigabit Ethernet (1 CPU, 8PPN, 4ION).

Figures 4.6 and 4.7 show the Flash I/O benchmark performance running with the block-indexed data type on a single node with 8 cores and a single network link. Although the file system performance is generally stable, we can again see that when the page size is small (64 byte pages or smaller), the performance is again unstable. Because a 64 byte page is unlikely to appear in a realistic system, we do not consider that a significant performance problem. On the other hand, in these configurations it is important to note that eight individual caches exist on each node, meaning that the largest cache configuration, with 512MB of cache, actually uses 4GB of memory per node in total, an unrealistic amount for a compute node that is likely configured with less than 16GB of total memory (the maximum available on Palmetto).

Page Size	Cache Size	Total Exec. Time	Speedup
64KB	16MB	0.3170	4.84
32KB	16MB	0.3200	4.80
16KB	16MB	0.3209	4.78
4KB	8MB	0.3219	4.78

Table 4.6: Best performing cache configurations using block-indexed access on Myrinet Myri10G (1 CPU, 8PPN, 4ION).

Fortunately, the small size of the Flash I/O benchmark working set means that small caches are capable of providing a substantial performance improvement. Tables 4.5 and 4.6 show the fastest configurations of the per process caches for Gigabit Ethernet and Myrinet. On parallel codes that require a larger working set, the size of the caches may need to grow as well. One of the original ideas discussed in this dissertation was the application of a single cache to multiple processes running on a multi-core node.

4.2.1.3 Shared Caching

This time in order to improve the caching systems performance we look at the effects of sharing the cache data for all the MPI processes participating on the node. The shared memory primitives in Linux do not allow us to simply pin and unpin buffers into shared memory, instead data will need to be copied into and out of the shared cache which can require mutually exclusive access to cached file pages; however, with memory bandwidths currently 10 times faster than network bandwidth, the costs should be easily amortized over the gains made in improving the page coverage of partial page writes. Additionally, our shared cache also includes the block-indexed data type optimization we discussed earlier (it still provides a benefit in terms of performance volatility).

One caveat with the shared file cache is that the close-to-open consistency model described in section 4.1.1 is further relaxed. Rather than flushing all the dirty cache pages

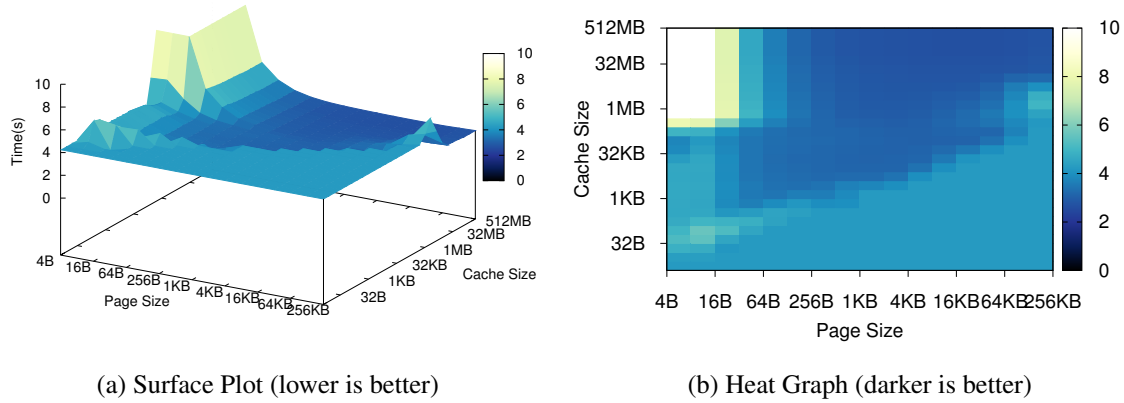


Figure 4.8: FLASH I/O benchmark execution time with a shared cache on 1 CPU, 8 processes per node, and 4 I/O Nodes over Gigabit Ethernet.

Page Size	Cache Size	Total Exec. Time	Speedup
16KB	64MB	2.6649	1.60
32KB	128MB	2.6673	1.60
64KB	128MB	2.6688	1.60
8KB	64MB	2.6765	1.59

Table 4.7: Best performing shared cache configurations on Gigabit Ethernet (1 CPU, 8PPN, 4ION).

on each file close, we now wait for all the local processes accessing the file to close it before flushing the dirty pages into the file system. Essentially, the consistency granularity is altered from process-centric to node-centric, as a client opening a file on one node can only see the file updates from nodes where every process that is updating the file has performed a file close.

Figures 4.8 and 4.9 show the benchmark performance data for our final cache configuration. Here we see that the cache performance is very similar to the performance of the non-shared cache. However, in these configurations the shared cache is using one-eighth the memory resources per node to achieve the same benchmark performance.

Tables 4.7 and 4.8 again show the four fastest configurations for each of the inter-

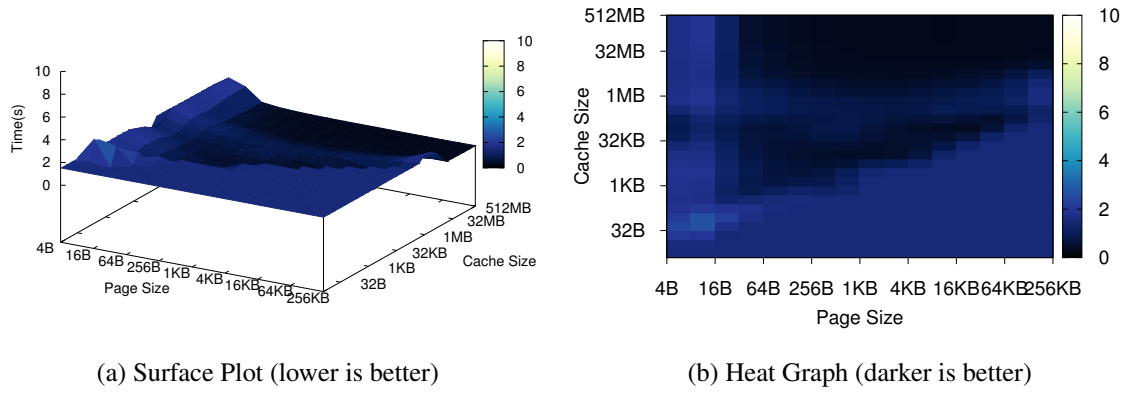


Figure 4.9: FLASH I/O benchmark execution time with a shared cache 1 CPU, 8 processes per node, and 4 I/O Nodes over Myrinet Myri10G.

Page Size	Cache Size	Total Exec. Time	Speedup
8KB	64MB	0.2928	5.25
16KB	64MB	0.2931	5.24
64KB	128MB	0.2933	5.24
32KB	128MB	0.2940	5.23

Table 4.8: Best performing shared cache configurations on Myrinet Myri10G (1 CPU, 8PPN, 4ION).

connection network types. In the case of the Gigabit network the additional speedup from cache sharing is negligible; however, two of the configurations halve the memory requirement per node to achieve nearly identical performance. On the Myrinet interconnect the shared cache exhibits better performance than the non-shared caches in table 4.6, although the memory consumption per node is basically unchanged.

4.2.2 Cache Performance At Scale

Based on our experimental analysis, we have been able to construct a middleware cache that consistently achieves high performance for our benchmark program at small scales. We now wish to explore the effects of middleware caching on our benchmark application at larger scales.

4.2.3 Gigabit Ethernet Network Experiments

Because of the extended runtimes required to simulate large-scale benchmarks and cache configurations we are leveraging our earlier small-scale results to determine a subset of representative configurations that should illustrate the effects of cache page size and capacity on benchmark execution time. We have chosen page sizes of 512B, 4KB, 64KB, and 256KB in order to measure page size effects at scale. For each page size we perform simulation executions using a cache size of 2MB and 16MB, respectively. We avoided larger cache sizes simply because we do not believe application developers will be willing to dedicate much more system memory to middleware caching (e.g. 1GB to 8GB of the available system memory at 512 processes, or 2GB to 16GB at 1024 processes). Note that the sizes shown are on a cache basis rather than a node basis and include only the data capacity, not the overhead associated with maintaining cache keys and each cache page's LRU status. The shared cache configurations are using one-eighth as much memory as

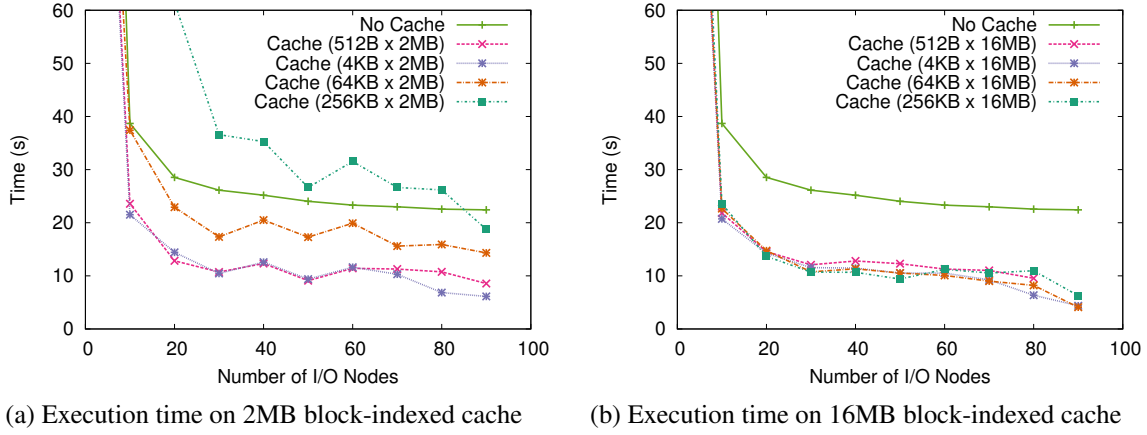
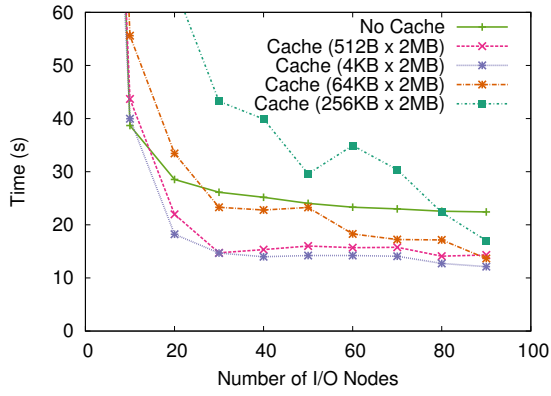


Figure 4.10: FLASH I/O benchmark execution time with a block-index cache on 512 processes (64 Nodes, 8PPN) over Gigabit Ethernet (lower is better).

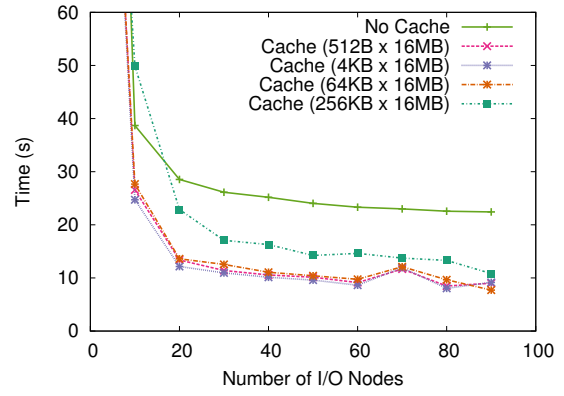
the non-shared cache configurations due to the dual quad-core processors in our Palmetto simulation profile.

Figures 4.10 and 4.11 show the execution times for the FLASH I/O benchmark running as 512 processes on 64 Gigabit Ethernet nodes on file systems ranging from 1 I/O node to 90 I/O nodes. The left hand figures demonstrate the performance of each cache page size on a cache configured with 2MB of capacity and the right hand figures show the benchmark execution time on caches configured with 16MB of capacity. As you would expect, the larger caches provide better performance, however the benefits of even a small cache can be substantial if the cache page size is well chosen. Only one of the selected cache configurations (256KB pages in a 2MB cache) fails to improve the performance of the benchmark over most of the selected file system sizes.

Similarly, figures 4.12 and 4.13 show the execution times for the FLASH I/O benchmark running as 1024 processes on 128 Gigabit Ethernet nodes. Again the parallel file system configurations range from 1 I/O node to 90 I/O nodes. At the thousand process scale, most of the smaller 2MB cache configurations only slightly improve performance. The larger 16MB caches (shown in the right side column) are still able to provide substantial

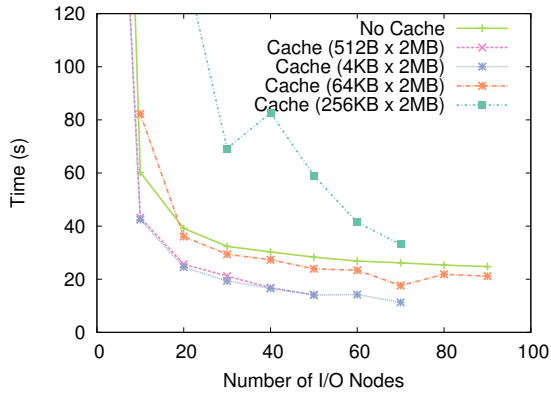


(a) Execution time on 2MB shared cache

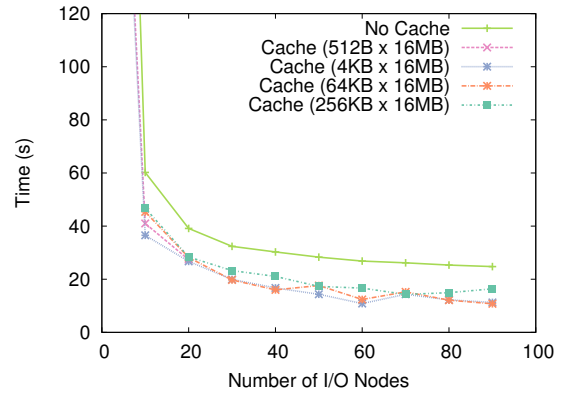


(b) Execution time on 16MB shared cache

Figure 4.11: FLASH I/O benchmark execution time with a shared cache on 512 processes (64 Nodes, 8PPN) over Gigabit Ethernet (lower is better).



(a) Execution time on 2MB block-indexed cache



(b) Execution time on 16MB block-indexed cache

Figure 4.12: FLASH I/O benchmark execution time with a block-index cache on 1024 processes (128 Nodes, 8PPN) over Gigabit Ethernet (lower is better).

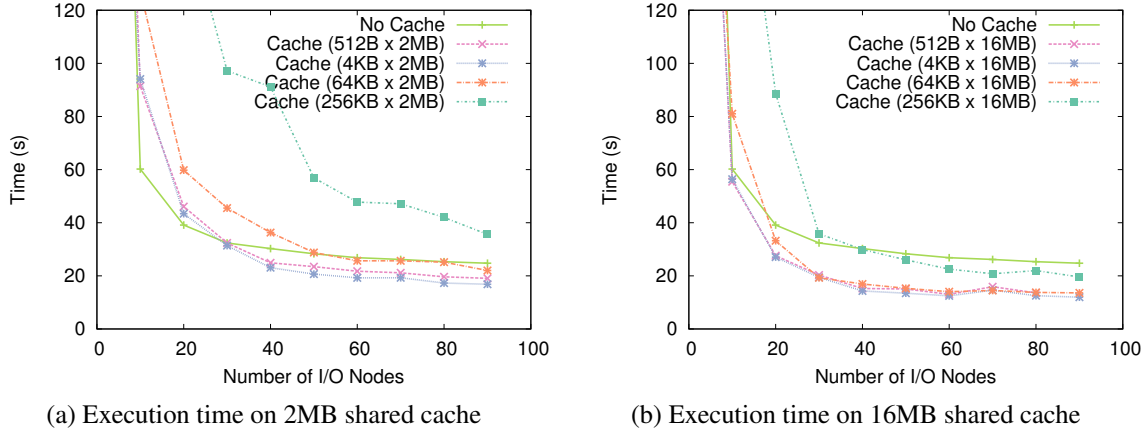


Figure 4.13: FLASH I/O benchmark execution time with a shared cache on 1024 processes (128 Nodes, 8PPN) over Gigabit Ethernet (lower is better).

performance improvements with the smaller page sizes.

4.2.4 Myrinet Myri10G Network Experiments

As we described in Chapter 3, our simulation of the Myrinet networking technology is not as accurate as our simulation of Ethernet networks. The extremely variable cache performance shown in figure 4.14 bears this out as inadequate switch buffering completely clouds any insight we have into cache performance. When the Flash I/O benchmark is run without caching, the poor network utilization of the application results in stable performance because the amount of switch buffering provided by the simulator is adequate for the inefficient network access. However, upon adding the client-side cache, the network bandwidth utilization increases due to the larger average message size and the upper limit on the switch buffer capacity is quickly reached causing a large number of dropped frames and the accompanying packet retransmissions and lowered network throughput.

On a real Myrinet network, frames cannot be dropped; rather, the messages will enqueue into network buffers all the way back to the originating node eventually forcing

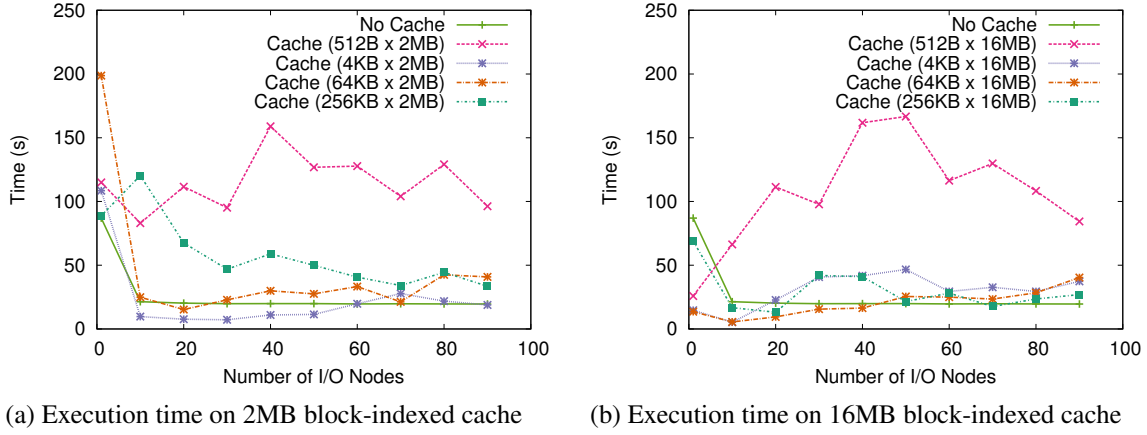
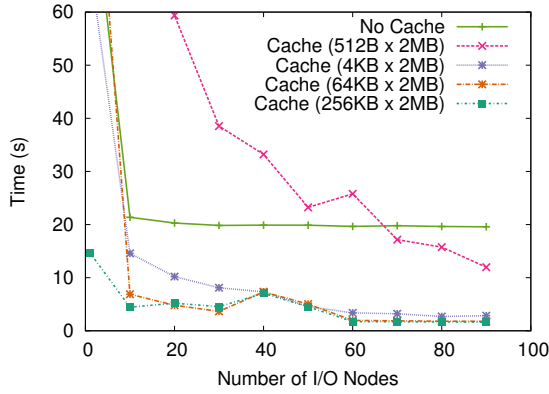


Figure 4.14: FLASH I/O benchmark execution time with a block-index cache on 512 processes (64 Nodes, 8PPN) over Myrinet Myri10G.

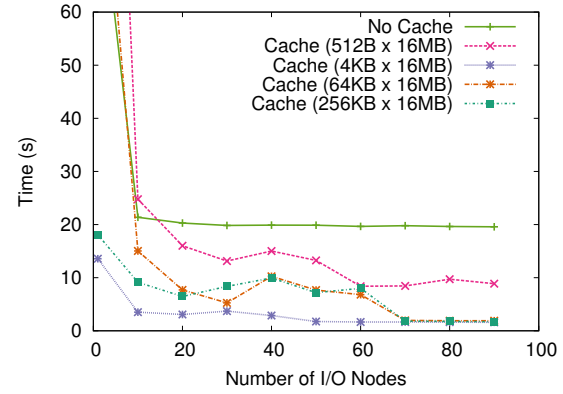
the sender to block until the desired network route is clear. Because our simulation does not provide this key functionality, in Chapter 3 we configured the switch buffer size to grow in proportion to the number of nodes simulated; however, we also instituted a maximum allowable switch buffer size for Myrinet simulations. The buffer size limit improves the quality of our simulator validation results; however, it now appears to limit the simulated performance gains we can achieve by improving the efficiency of the benchmark’s network access. In order to provide results that may better reflect how large scale Myrinet installations perform, our remaining results use a reconfigured Myrinet simulation that allows the switch size to grow beyond the buffer size limit we enforced during our earlier performance validation tests.

Figures 4.15 and 4.16 show the performance of the FLASH I/O benchmark simulated on the enhanced Myrinet network infrastructure. Again, the use of a client-side cache provides excellent performance improvements with both the cache per process and cache per node configurations.

Figures 4.17 and 4.18 shows the execution time for the benchmark running on a non-shared and shared cache. For the 2MB cache per node configuration we can see that

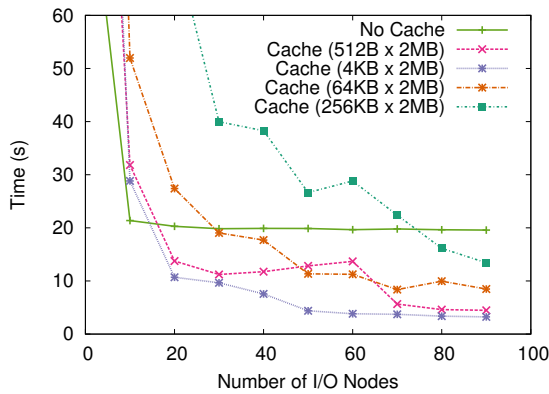


(a) Execution time on 2MB block-indexed cache

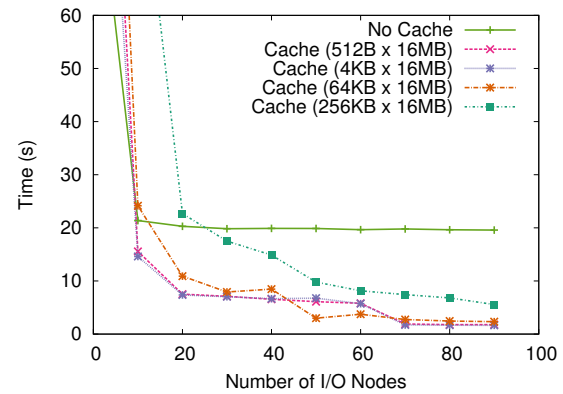


(b) Execution time on 16MB block-indexed cache

Figure 4.15: FLASH I/O benchmark execution time with a block-index cache on 512 processes (64 Nodes, 8PPN) over Myrinet Myri10G with fully dynamic switch buffer size (lower is better).

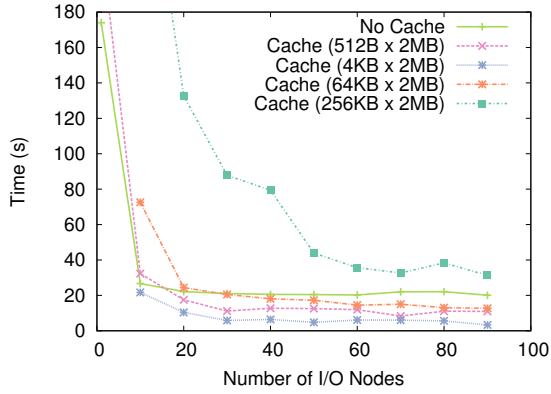


(a) Execution time on 2MB shared cache

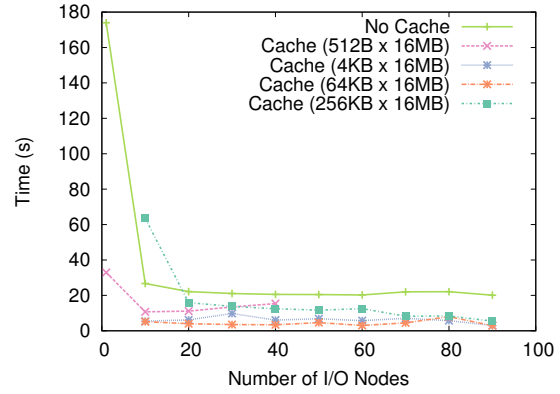


(b) Execution time on 16MB shared cache

Figure 4.16: FLASH I/O benchmark execution time with a shared cache on 512 processes (64 Nodes, 8PPN) over Myrinet Myri10G with fully dynamic switch buffer size (lower is better).

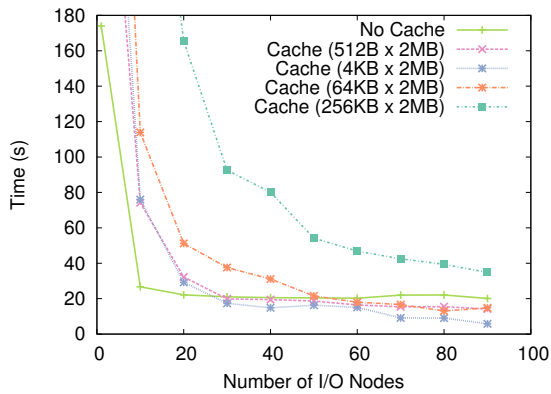


(a) Execution time on 2MB block-indexed cache

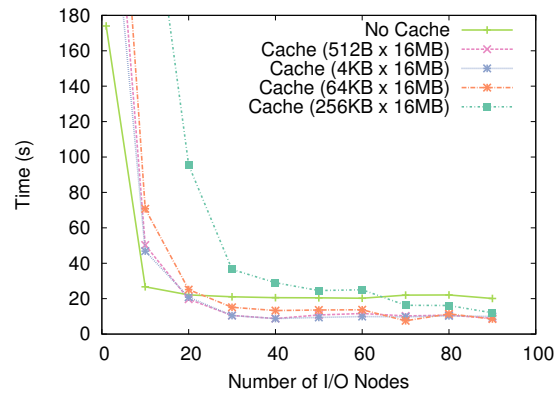


(b) Execution time on 16MB block-indexed cache

Figure 4.17: FLASH I/O benchmark execution time with a block-index cache on 1024 processes (128 Nodes, 8PPN) over Myrinet Myri10G with fully dynamic switch buffer size.



(a) Execution time on 2MB shared cache



(b) Execution time on 16MB shared cache

Figure 4.18: FLASH I/O benchmark execution time with a shared cache on 1024 processes (128 Nodes, 8PPN) over Myrinet Myri10G with fully dynamic switch buffer size (lower is better).

the performance is not generally improved, although the benchmark exhibits better scaling than in the non-cached case. When the cache size is increased to 16MB per node, both caches are able to again improve performance for file systems composed of more than 20 I/O nodes for all page sizes except 256KB, which only improves performance for file systems with more than 60 I/O nodes.

4.3 Summary

In this chapter we have described a file data caching middleware that transparently provides coherent file access and improves application performance. Our file caching scheme improves application I/O granularity by using fixed-size pages for parallel file system access. False sharing between caches is prevented by using a diff-based scheme similar to the TreadMarks Shared Virtual Memory system. We were able to use small-scale experiments with the FLASH I/O benchmark to empirically evaluate our cache design parameters over a variety of hardware configurations. We determined that the block-indexed data type can be leveraged within our fixed-size page caches to reduce performance volatility and lessen the impacts of poorly chosen cache parameters. We also examined the effectiveness of cache sharing at improving performance and reducing the memory requirements necessary for caching at the cost of loosening our selected data consistency scheme.

Finally, we were able to leverage the results of our small-scale analysis to find a representative set of cache configurations to examine the effects on benchmark performance at scale. We found that at 512 applications processes the standard fixed-size page cache and shared cache were able of significantly improving performance for both Gigabit Ethernet clusters and Myrinet clusters. We also found that our benchmark running as 1024 processes experienced significant performance improvements with our fixed-size page caches, however, the increased network utilization led to increased volatility. This result is not sur-

prising as even our largest file system configuration, 90 I/O nodes, is still far less than the 128 compute nodes running the application benchmark.

Chapter 5

Progressive Page Caching

The primary problem associated with fixed-size page caching, in all of its incarnations, is false sharing. The only mechanism for reducing the degree of false sharing is to reduce the page size, and as we saw in our analysis of fixed-size page caching in Chapter 4, caches configured to use small page sizes are the most likely to generate unstable performance (though we were able to largely mitigate the performance volatility). More problematically, some parallel file systems such as Lustre and GPFS may be unable to benefit from cache page sizes smaller than a system memory page (typically 4KB) or a disk block (typically 512 bytes) respectively, because the mechanisms used to provide advisory locking within the file system lead to contention and poor performance with smaller granularity accesses [43]. In this chapter we look at an alternative caching scheme that seeks to avoid false sharing at the cost of additional overhead in time and space.

5.1 Architecture Overview

Our progressive page cache appears to the end user as identical to our earlier fixed-size page cache designs. A middleware caching infrastructure intercepts and interprets all

calls to file open, close, read and write from the application to the file system in a manner mostly transparent to the user. Just as in fixed-size page caching, the only differences the client sees between using a caching middleware and accessing the file system directly is the change to close-to-open file consistency and, hopefully, improved performance. While an application scientist's view of the I/O system is mostly unchanged, the internal architecture of a progressive page cache is significantly different from a fixed-size page cache.

In progressive page caching, the file is still segmented into fixed-size pages as described in the chapter on basic page caches. We are still able to use our same page identification formula to construct cache keys:

$$Key_{filename} = Request_{filename}$$

$$Key_{pageid} = Request_{offset} / Cache_{pagesize}$$

However, if the initial lookup succeeds, we will need to examine the actual contents of the cache page in order to ensure the correct page contents are present. In the case of a file read, it still may be necessary to request several file pages from the backing store/file system based on the actual cache contents. But in the case of a file write, the data can be immediately copied into the cache without a requirement to pre-acquire the contents of any additional file pages at all!

5.1.1 Progressive Page Cache I/O

Fundamentally, file I/O is described by a file offset, a file extent, and a data buffer. The offset describes where in the file to begin reading or writing data, the extent describes how much data to read or write, and the data buffer is used to locally store the file contents.

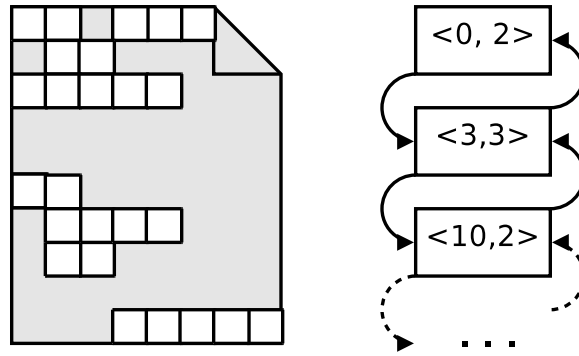


Figure 5.1: A sample progressive cache page with an accompanying list of the valid file regions. The valid file regions are shown in black and described by the file region list.

In both our fixed-size page cache and our progressive page cache the data buffer is stored locally on the cache page. The major difference is that our progressive page cache does not attempt to update the rest of the page with valid data on a file write. Instead, the cache maintains a set of valid file offsets and extents for each cache page.

Recall that false sharing may occur when a cache page being committed back to the file system is only partially updated, and thus stale data on the cache page may be written back to the file system if the file system has been updated since the page was cached. Progressive paging explicitly avoids false sharing by maintaining a set of valid *file regions*, or offset and extent pairs, within the page. Figure 5.1 shows a representation of a progressive cache page that is partially filled with file regions and the page's accompanying set of file region descriptions. During cache page evictions and replacements, the library ensures that only the updated file regions are written into the backing store/file system.

5.1.1.1 Client Reads

A progressive page middleware cache satisfies file reads in a manner similar to our previous fixed-size page cache. First, the middleware attempts to satisfy the read with data from the cache. For each portion of the read that cannot be satisfied from the cache, the

satisfying file page is added into a single block-indexed data type. The block-indexed data type is then used to read all of the non-resident pages from the file system in a single request. We use the block-indexed data type based on the results of our prior study; however, it is possible that for some workloads single page requests could work just as well.

The major difference in read processing for the progressive page cache versus the fixed-size page cache discussed in Chapter 4 is how the read data is merged into the progressive cache. With a fixed-size page cache design, it is impossible to have a partially empty cache; instead, the cache page must be filled before any write to that page can be performed. A progressive page cache is not so simple, and portions of the page may be dirty, requiring the read data to be merged into the progressive page selectively. We discuss these issues more fully in Section 5.2.

5.1.1.2 Client Writes

In a progressive page cache, file writes are even simpler to process than file reads, the data is simply merged into the cache. Any existing pages are updated with the new data; for pages that do not exist a blank page is constructed, and the file regions are written into the blank page and the file region set for the page is updated.

5.1.1.3 Progressive Page Cache Evictions

The progressive page cache uses the same close-to-open consistency model we described in Chapter 4. Cache page evictions occur under three scenarios: the cache is full and a client read operation requires a new cache page, the cache is full and a client write operation requires a new page, or a file is closed. Because our cache uses a write back policy for committing data to the file system, only during page evictions is data actually stored to the underlying parallel file system. If the evicted page contains file regions that have been updated in the cache, but not sent to the backing store, then the regions are writ-

ten to the file system. Based on the results of our earlier work we construct all of the fully updated pages into a single request using a block-indexed data type, and all of the partial pages are written with a request using the more general indexed data type. Requests using a file view based on the indexed data type are able to describe the irregular file regions typical of partial cache pages [55].

5.2 Implementation Details

The chief problem with implementing a progressive page cache organization is determining how to organize the actual progressive pages. In a fixed-size cache the page organization is straightforward: the keys are held in a fast lookup data structure such as a hash table, a list stores the LRU status, and the pages can be stored consecutively in an array or managed dynamically with little penalty in either case. The only additional state required for the data pages is a single bit to indicate if a page contains data not yet committed to the file system (i.e. a dirty bit).

The basic organization of the progressive state cache is similar: a hash table for keys and a list for LRU sorting. However, for a progressive page the amount of state to be stored with the page is considerable, and thus how we store the data pages will depend significantly upon how we choose to store the state information. Obviously, our page implementation will attempt to balance ease of implementation, runtime performance, and storage overhead.

In selecting implementation appropriate data structures, we need to first examine exactly what type of operations a progressive page cache will need to perform in order to read, write, and evict cache pages. Upon receiving a file read or file write, the cache first needs to determine if the file regions described by the requested I/O exist in the cache. Because the page may not be fully populated, the cache lookup may require traversing the

file regions to find the updated regions. In the case of a file write, it will also be necessary to update each requested page with new file regions, and/or merge with existing file regions. Finally, during page eviction, we must extract all of the updated file regions from a page in order to perform file data write backs. More generally, we can categorize the three fundamental operations as cache lookup, cache insertion, and cache eviction.

Based on the description of these operations, we see that we will need state information so that the cache can determine which information on a page is valid. In the case where the cache page is the result of a file read, this determination can be done simply by adding a single bit identifying if the page has been read from the backing store or is the result of client file updates. Analogous to the fixed-size page caches dirty-bit, we call this the valid-bit, and it describes whether the non-updated data in the cache page is valid, or not.

Although figure 5.1 shows the set of updated file regions maintained as a doubly linked list, there are several possible data structures we could choose to maintain the set of active file regions within the cache page. In the next sections we look at two possibilities, a dirty-mask and a file region tree, and examine the costs and benefits of each in terms of runtime and cache memory requirements.

5.2.1 Page Dirty-Mask Overview

The first mechanism we describe for maintaining the set of updated file regions is a dirty-mask assigned to each page in the progressive page cache. In this approach, we choose not to describe the updated regions explicitly, but rather for every updated byte in the page, we set a corresponding dirty bit in the page's dirty-mask bit vector. For example, in a progressive page cache with 512 byte pages, each cache page will have 512 state bits to describe the dirty status of each cache page. This approach has the disadvantage of

describing the state of pages that may not be valid at all; however, for some workloads it will provide the smallest possible runtime and storage overhead. We begin our analysis of the dirty-mask approach by examining the runtime of the three critical cache operations we described earlier: insertion, lookup, and eviction.

5.2.1.1 Insertion Runtime

Given a request to insert into the progressive page cache of size R , the time spent updating the dirty mask on a file region insertion is the time it takes to update R bits in the dirty-mask. Figure 5.2 shows one possible implementation for the dirty mask insertion algorithm in C. As the code listing shows, the total execution total cost of updating the page dirty mask during a file region insertion is $17 + \lceil R/8 \rceil$ integer operations and $1 + \lceil R/8 \rceil$ comparison operations. The runtime asymptote is $O(R)$.

5.2.1.2 Lookup Runtime

File region lookups into the progressive page cache occur when trying to satisfy a file read in the local cache. Given a lookup request for a file region of size, R , the total time spent inspecting the cache pages spanned by R is the time it takes to read the value from R bits of dirty mask. The code listing supplied in figure 5.2 can be easily modified to test that each bit for the desired file region is set using $15 + \lceil R/8 \rceil$ integer operations and $3 + \lceil R/8 \rceil$ comparison operations. The runtime asymptote is again $O(R)$.

5.2.1.3 Eviction Runtime

Finally, we look at the time required to perform a cache page eviction. On a cache page eviction, it is necessary to update the status for the entire bit vector so that the page may be safely used with its newly mapped data. Although it may be possible to combine the dirty-mask eviction update with the update required for the new data insertion, the runtime

```

1 void insert(char* page_mask, int offset, int extent)
2 {
3     /* Update the first byte */
4     int first_byte = offset/8;
5     int first_byte_off = offset \% 8;
6     int first_byte_lo = 8 - first_byte_off;
7     if (extent <= first_byte_lo)
8     {
9         /* Update the only byte for this page */
10        char mask = ((1 << extent) - 1) << first_byte_off;
11        page_mask[first_byte] |= mask;
12    }
13    else
14    {
15        /* Update the first byte */
16        char mask = ((1 << first_byte_lo) - 1) << first_byte_off;
17        page_mask[first_byte] |= mask;
18
19        /* Update the last byte */
20        int last_byte = (offset + extent)/8;
21        int last_byte_lo = (offset + extent) \% 8;
22        mask = ~(((1 << (8 - last_byte_lo)) - 1) << last_byte_lo);
23        page_mask[last_byte] |= mask;
24
25        /* Update the remaining bytes */
26        int second_byte = first_byte + 1;
27        int second_last_byte = last_byte - 1;
28        for (int i = second_byte; i <= second_last_byte; i++)
29        {
30            page_mask[i] |= -1;
31        }
32    }
33 }

```

Figure 5.2: Example of dirty mask region insertion code in C

is still proportional to $O(nP)$, where n is the number of pages to evict and P is the size of a page in the progressive page cache; or, more simply, $O(P)$. The total number of integer operations required to perform the update is $\lceil P/8 \rceil$ operations.

5.2.2 File Region Set Overview

The obvious alternative to the mask-based approach described in the previous section is to simply store the offset and extent pairs for just the updated regions in a cache page. Then as lookups, inserts, and evictions occur the set of file regions can be searched, appended, and erased as necessary. Not surprisingly, the data structure we use to store the data set has a significant impact on storage overhead, cache performance, and even implementation difficulty.

In figure 5.1 we saw the file regions stored in a doubly linked list, but a quick analysis demonstrates that this organization is unlikely to yield optimal results. For example, if we wish to insert adjoining file regions, from an overhead standpoint, we would ideally merge the two regions together. Given a linked list of updated file regions, it is necessary to check every region in the list for an overlapping sub-region. Sorting the list by the offsets seems like it may help, but without constant time access to random list elements, insert time is still proportional to the number of file regions in the list. Fortunately, we can more simply select a binary search tree as our data structure, and leverage the resulting \log_2 performance for search and insertion.

Figure 5.3 shows a visual representation of a file region set implemented as a binary search tree [12]. The tree sort key is the file region offset and the tree nodes have been modified to store additional data in the form of the file region extent. In addition to the left and right child node pointers typical of a tree structure, nodes also maintain a parent node pointer. The parent node pointer is needed to allow easy iteration over the tree nodes, an

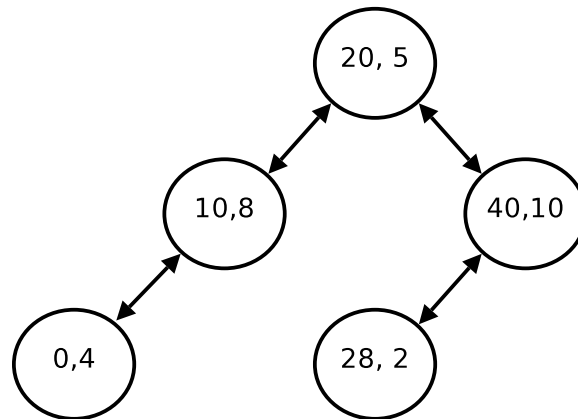


Figure 5.3: A binary search tree for file regions. The offset is used as the key field, with the extent tacked on as additional node data.

operation required to maintain the constraint that no file regions in the tree overlap. The non-overlapping constraint is important in achieving adequate performance for lookups and reducing the tree's memory footprint.

5.2.2.1 Insertion Runtime

Inserting into the file region tree can be decomposed into three basic steps. First, find the tightest lower bounding file region and resolve any overlapping regions with the file region to insert. Second, check the upper bounding regions to determine if the new region overlaps any following regions, and resolve those overlapping areas. Finally, insert the new region if no overlapping regions were located.

Listing 5.1 shows the algorithm for finding the next lowest file region before the region to insert. The algorithm is basically the same as a binary search tree insert, although we return the parent node, rather than add a child to the node.

Using the lower bound algorithm, we can then formulate an algorithm for inserting into our non-overlapping file region tree. Listing 5.2 shows the algorithm for inserting into the file region tree. In part I, we find the lower bounding region and adjust it if it overlaps.

Algorithm 5.1: LowerBound(Tree tree, FileOffset offset) : Node

```
1 begin
2    $node \leftarrow \text{Root}(tree)$ 
3    $lb \leftarrow \text{NIL}$ 
4   while  $node \neq \text{NIL}$  do
5     if  $offset < node.offset$  then
6        $lb \leftarrow node$ 
7        $node \leftarrow node.left$ 
8     else
9        $node \leftarrow node.right$ 
10    end
11  end
12  return  $lb$ 
13 end
```

In part II, we subsume the regions that follow the new region, and add those as overlapping if necessary. Finally, we insert the new region just above the lower bound if no overlaps occur. Following the insertion, the tree may require re-balancing. In our implementation we used a standard red-black tree approach to maintain a balanced search tree [13].

In order to analyze the runtime for Listing 5.2 let us define the following cache parameters:

- Let P be the size of a cache page, and
- Let R be the size of the new file region, and
- Let Q be the number of regions already stored on the page, and
- Let V be the number of regions the new file region overlaps on the page.

In Part I of the insertion algorithm, the call to LowerBound has runtime in the same order as a standard tree insertion, $O(\log_2(Q))$ with at most 10 integer operations and 2 comparison operations of overhead. The time to iterate over the succeeding regions to find subsumed regions is proportional to $O(V)$ with a total overhead of $6V + 2$ integer operations and $2V$

Algorithm 5.2: Insert(SortedFileRegion regions, FileRegion newRegion)

```
1 begin
  // Part I.
2  hasPriorOverlap  $\leftarrow$  false
3  lb  $\leftarrow$  LowerBound (regions, newRegion.offset)
4  lbEnd  $\leftarrow$  lb.offset + lb.extent
5  if newRegion.offset  $\leq$  lbEnd then
6    | prior.extent  $\leftarrow$  Max (priorEnd, newEnd) - prior.offset
7    | newRegion  $\leftarrow$  prior
8    | hasPriorOverlap  $\leftarrow$  true
9  end

  // Part II.
10 newEnd  $\leftarrow$  newRegion.offset + newRegion.extent
11 iterator  $\leftarrow$  getNext(regions, iterator)
12 while iterator.offset  $\leq$  newEnd do
13   | nextEnd  $\leftarrow$  iterator.offset + iterator.extent
14   | newRegion.extent  $\leftarrow$  Max (newEnd, nextEnd) - newRegion.offset
15   | Erase (regions, iterator)
16   | iterator  $\leftarrow$  getNext(regions, iterator)
17 end

  // Part III.
18 if hasPriorOverlap = false then
19   | AddChild(lb, newRegion)
20 end
21 end
```

comparison operations. Finally, the time to perform insertion if no overlapping regions is performed requires only 1 comparison operation and 6 integer operations, because the new region can simply be inserted as a child of the lower bound node, so $O(c)$. In the worst case, the value for V is $P/2$, resulting in an algorithm runtime of $O(P)$; however, as that requires updating every other byte in the page, we classify the runtime of tree insertion as $O(\log_2(Q))$.

5.2.2.2 Lookup Runtime

In order to perform a Lookup operation on a file region, we need to search for the element in the tree with the largest file region offset that is smaller or equal to the lookup region's offset. Just as in the earlier insert algorithm, the *LowerBound* algorithm described in Listing 5.1 provides exactly the behavior we desire. If the located lower bound region does not fully overlap the lookup file region, then the entire page will need to be retrieved to satisfy the read. For the Lookup operation to provide correct behavior, we must ensure that no file regions in the tree overlap. Of course, the insertion algorithm provided in Listing 5.2 maintains the non-overlapping region constraint, thus a single call to *LowerBound* can be used to perform file region lookups. As discussed earlier, the runtime of *LowerBound* is asymptotically the same as a binary tree search, or simply $O(\log_2(Q))$, where Q is the number of non-overlapping file regions stored for the cache page.

5.2.2.3 Eviction Runtime

During a page eviction, we need only to erase the nodes in the file region tree. So the runtime for page eviction is simply proportional to the number of updated regions in the page, or $O(Q)$. As we stated earlier, in the worst case, the value for Q is $P/2$, resulting in an algorithm runtime of $O(P)$.

Procedure	Dirty-Mask		File Region Tree	
	Runtime Order	Add. Overhead	Runtime Order	Add. Overhead
<i>Insert</i>	$O(R)$	$17\text{ops}_{\text{int}} + 1\text{ops}_{\text{cmp}}$	$O(\log_2(Q) + V)$	$(6V + 8)\text{ops}_{\text{int}} + (2V + 1)\text{ops}_{\text{cmp}}$
<i>Lookup</i>	$O(R)$	$15\text{ops}_{\text{int}} + 1\text{ops}_{\text{cmp}}$	$O(\log_2(Q))$	1ops_{int}
<i>Evict</i>	$O(P)$	0	$O(Q)$	0

Table 5.1: Expected case runtimes for performing the three critical progressive page caching operations. Worst case performance is triggered by inserting a region that fully writes a mostly updated page ($R = P$).

5.2.3 Data Structure Comparison

Table 5.1 shows the run time asymptotes for the dirty-mask and file region tree progressive page implementations. The expected runtime performance and the worst case performance are shown for both implementations. For the purpose of comparing the runtimes of both approaches easily, we define the worst-case for the dirty mask to be when the page has been fully updated minus one byte. The file region tree implementation presents worst-case performance when the page has been updated with an alternating byte pattern (every other byte), resulting in a fully populated file region tree.

Table 5.1 shows that asymptotically, insert performance will on average be no worse for the region tree than it is for the dirty mask; however, the large additional overhead (which does not include any costs associated with maintaining the balanced tree constraint or dynamic memory management) dominates the logarithmic performance of the file region tree for all but the largest page sizes. The simplicity, and easy optimization of the dirty-mask approach means that it will be faster for typical page sizes, and even more so when the newly inserted region overlaps many existing file regions.

Page Size	Dirty-Mask Overhead	File Region Tree	
		Maximum	Break-even
512 bytes	64 bytes	1344 bytes	2 regions
4KB	512 bytes	14592 bytes	18 regions
64KB	8KB	315392 bytes	292 regions
256KB	32KB	1425408 bytes	1149 regions

Table 5.2: The memory overhead required to store file region metadata for 4 page sizes. The break-even column describes the number of file regions that can be stored in the same amount of storage as the dirty-mask.

For the lookup operation, the relationship between the sizes of R and Q likely determines which data structure produces better search performance. Because the file region tree is implemented as a balanced binary search tree, lookups can easily be performed in time proportional to the logarithm of the number of existing file regions with very low overhead. The dirty-mask approach can locate the file region to examine in constant time, but must process the region a bit or byte at a time, resulting in time proportional to the requested region size. In terms of the problem domain of our cache middleware, small, unaligned file regions, we expect that R will generally be larger than Q , and thus the performance of the file region tree is likely to be better than the dirty mask performance.

Finally, we look at the eviction operation. Although the file region tree approach has the opportunity to do less work than the dirty-mask when $Q \ll P$, the existence of hardware support for the *memset* function probably means that the dirty mask provides similar, if not better performance for most realistic cache page sizes.

Table 5.2 shows the progressive page file region storage overhead for both the dirty-mask and the file region tree. As you can see the overhead for the dirty-mask is always the same, and provides fairly good compression with the bit-per-byte mapping scheme, meaning that the file region set can be represented with one-eighth of the cache page size. The memory overhead asymptote is $O(P)$.

The file region tree, not surprisingly, results in much worse compression when the maximum number of file regions are populated. The file region offset and extent can be stored in $2\log_2(P)$ bits. Assuming we are pre-allocating the storage for the file region tree, we can store the left, right, and parent node pointers in $3\log_2(P/2)$ bits. The total storage required per node is $5\log_2(P) - 3$. The maximum number of file regions per page is $P/2$. Multiplying the two terms together we see the asymptotic bound on the memory overhead is $O(P\log_2(P))$, a log factor worse than the dirty-mask approach.

Fortunately, this memory layout is not the only possibility. If we instead use 64-bit pointers to represent the left, right, and parent pointers for each node in the file region tree, we can dynamically allocate the nodes (in fact, we could store the tree data in an array that grows dynamically and use the smaller pointers described above, but the resulting copy operations and memory management is beyond the scope of this dissertation). The final column of table 5.2, titled **Break-even**, shows the number of such dynamically allocated file region nodes the tree can store *per page* before consuming more storage than the per page storage used by the dirty-mask scheme. For context, recall that the FLASH I/O benchmark performs less than 250 writes per process before closing the file, thus forcing cache page evictions. Further, those writes do not all occur on the same page and are not all on disjoint file regions. For many workloads, the dynamically allocated file region tree will consume less memory and provide better runtime performance than the dirty-mask approach.

5.3 Performance Analysis

Given our earlier analysis of the overheads associated with the two competing approaches to storing progressive page cache metadata, it would be useful to examine how differing cache page sizes and capacities perform. Our simulator is intended to provide ac-

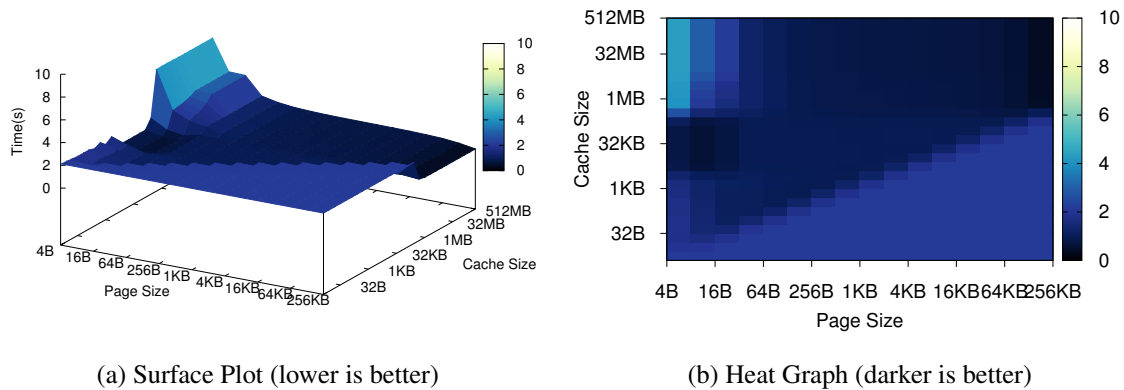


Figure 5.4: FLASH I/O benchmark execution time with a progressive paged cache on 8 CPUs, 1 process per node, and 4 I/O Nodes over Gigabit Ethernet.

curacy for high latency network and disk operations, rather than the much smaller latency associated with simulating a microprocessor. Fortunately, our earlier analysis that demonstrated that at the request and page sizes we are interested in benchmarking, the runtimes of our progressive page processing are likely to be suitably fast (logarithmic or linear time), and sufficiently similar. In particular, when compared to the order of magnitude slower network and disk latencies associated with parallel file system access, we feel confident that discarding algorithm runtime is unlikely to contaminate our results.

5.3.1 Baseline Performance

As in the last chapter, we begin our examination of progressive page cache performance by examining how our cache affects performance at small scales. Figures 5.4 and 5.5 shows a surface plot and heat graph expressing the total execution time of the FLASH I/O benchmark configured to run on a cluster with 8 single-core compute nodes accessing a parallel file system running of 4 I/O nodes. In figure 5.4 we see that using the Gigabit Ethernet network, we are able to generate a relatively stable performance improvement at most

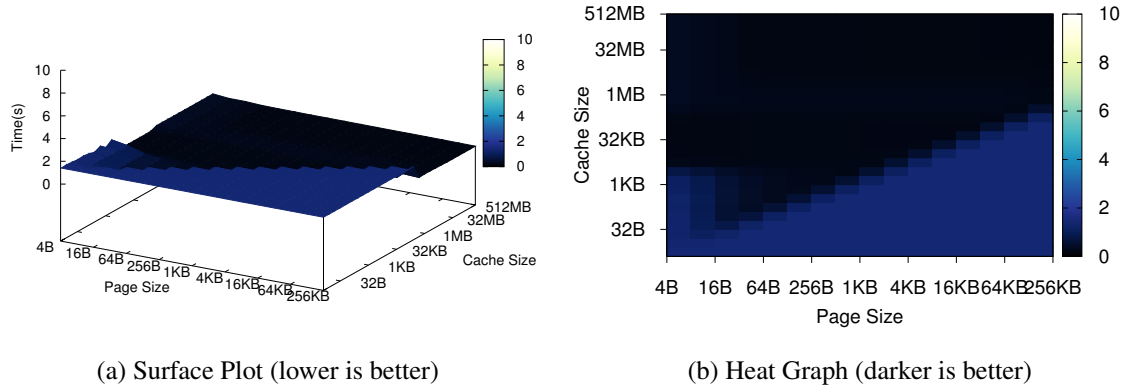


Figure 5.5: FLASH I/O benchmark execution time with a progressive page cache on 8 CPUs, 1 process per node, and 4 I/O Nodes over Myrinet Myri10G.

Page Size	Cache Size	Total Exec. Time	Speedup
256KB	8MB	0.2770	7.75
256KB	16MB	0.2772	7.74
256KB	2MB	0.2808	7.65
256KB	4MB	0.2831	7.58

Table 5.3: Best performing cache configurations with progressive page caching on Gigabit Ethernet (8 CPU, 4ION).

cache page sizes. The smallest page sizes exhibit a small degree of performance degradation, but the unstable performance of our previous cache designs appears to be largely gone. This is in part due to the progressive nature of the cache and the intelligent application of high-level data types to reduce the number of file system transactions.

Figure 5.5 shows execution time using the same simulation configuration with the exception of the network, which is configured as a Myrinet Myri10G interconnect. We can see that the progressive page cache pairs with the low latency, high bandwidth Myri10G network to produce excellent performance over the entire range of cache configurations.

Tables 5.3 and 5.4 show the four best performing cache configurations with progressive page caching for Gigabit Ethernet networks and Myrinet Myri10G networks, re-

Page Size	Cache Size	Total Exec. Time	Speedup
256KB	16MB	0.1667	8.58
128KB	16MB	0.1801	7.94
64KB	16MB	0.1872	7.64
32KB	16MB	0.1892	7.56

Table 5.4: Best performing cache configurations with progressive page caching Myrinet Myri10G (8 CPU, 4ION).

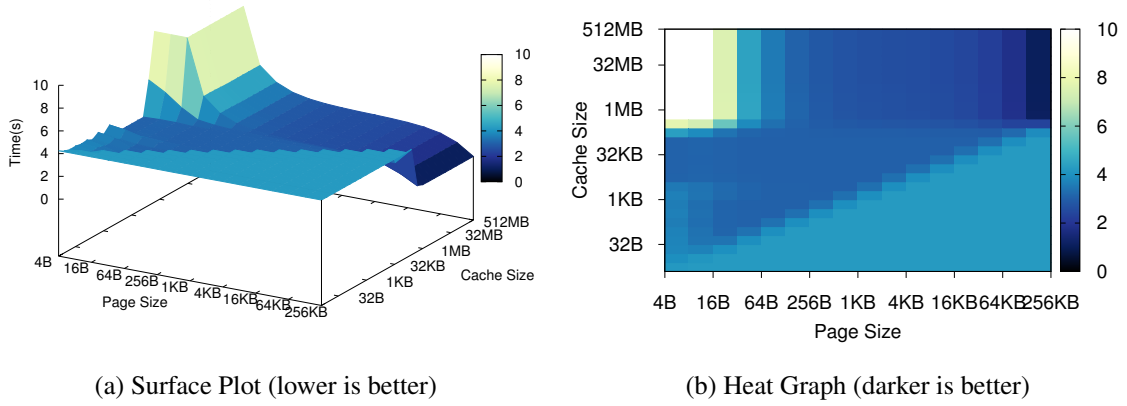


Figure 5.6: FLASH I/O benchmark execution time with the Block-Indexed data type optimization on 1 CPU, 8 processes per node, and 4 I/O Nodes over Gigabit Ethernet.

spectively. The speedups are impressive and improve upon the results of our fixed-size page caches from the last chapter. Even more interesting is the clear correlation between the size of the cache page and performance. Clearly cache capacity is a relevant factor, but the emergence of page size as a strong factor in cache performance has several important ramifications in how we select the data structure to store the cache metadata, and thus provides important insight into progressive caching.

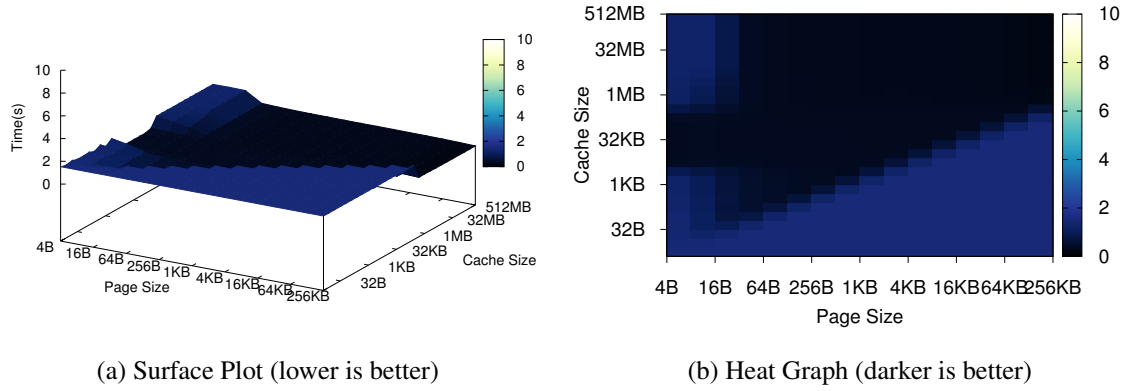


Figure 5.7: FLASH I/O benchmark execution time with the Block-Indexed data type optimization on 1 CPU, 8 processes per node, and 4 I/O Nodes over Myrinet Myri10G.

Page Size	Cache Size	Total Exec. Time	Speedup
256KB	16MB	0.5903	7.21
256KB	8MB	0.5998	7.10
256KB	4MB	0.6008	7.08
256KB	1MB	0.6020	7.07

Table 5.5: Best performing cache configurations with progressive page caching on Gigabit Ethernet (1 CPU, 8PPN, 4ION).

5.3.2 Performance on Multi-core Nodes

Figures 5.6 and 5.7 show the Flash I/O benchmark performance running with a progressive page cache on a single node with 8 cores and a single network link. Here, we again see the return of performance instability for small page sizes when using the Ethernet interconnect. The simulated cluster configuration with the Myrinet interconnect shown in figure 5.7 exhibits much less performance degradation, again likely due to the superior performance characteristics of Myri10G compared to Gigabit Ethernet.

Tables 5.5 and 5.6 show the fastest configurations of the progressive page caches for multiple processes running on Gigabit Ethernet and Myrinet nodes. Again we note the

Page Size	Cache Size	Total Exec. Time	Speedup
256KB	16MB	0.1851	8.30
256KB	8MB	0.1947	7.89
256KB	4MB	0.2044	7.51
256KB	2MB	0.2096	7.33

Table 5.6: Best performing cache configurations with progressive page caching on Myrinet Myri10G (1 CPU, 8PPN, 4ION).

appearance of cache page size as an apparent governing factor on performance, as all of the fastest configurations use the largest tested page size, 256KB. And cache capacity appears again as a predictor of performance, with more cache capacity resulting in more impressive speedups.

5.3.2.1 Cache Sharing

Our final small scale configuration again looks at using cache sharing to improve cache performance volatility and overall cache performance. The difficulties of using shared memory management as an implementation medium for the file region tree may make a shared implementation unlikely; however, the simulated performance results provide an interesting data point. Additionally, because multiple processes are sharing the same cache pages, the number of updated cache regions may grow rapidly leading to large metadata storage requirements for a region tree based progressive page cache. For a shared progressive page cache, a dirty-mask based implementation may be more feasible given limited budgets and schedule time. Also, we again note that shared caching violates close-to-open-consistency due to close-based evictions only occurring for the last file close on the node. A possible solution to this problem is to attach the process rank to each updated region, and writing the page contents back to disk, but not evicting file pages that are closed by one process on the node but remain open for other job processes.

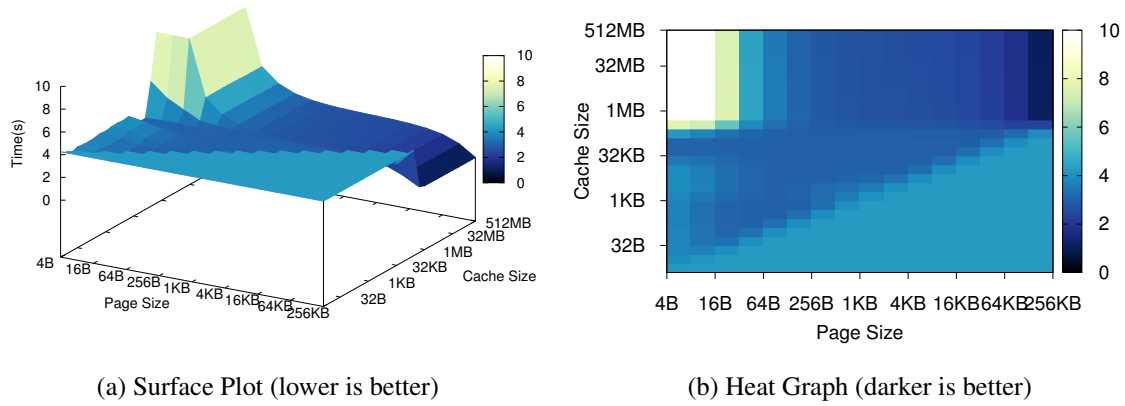


Figure 5.8: FLASH I/O benchmark execution time with a shared progressive page cache on 1 CPU, 8 processes per node, and 4 I/O Nodes over Gigabit Ethernet.

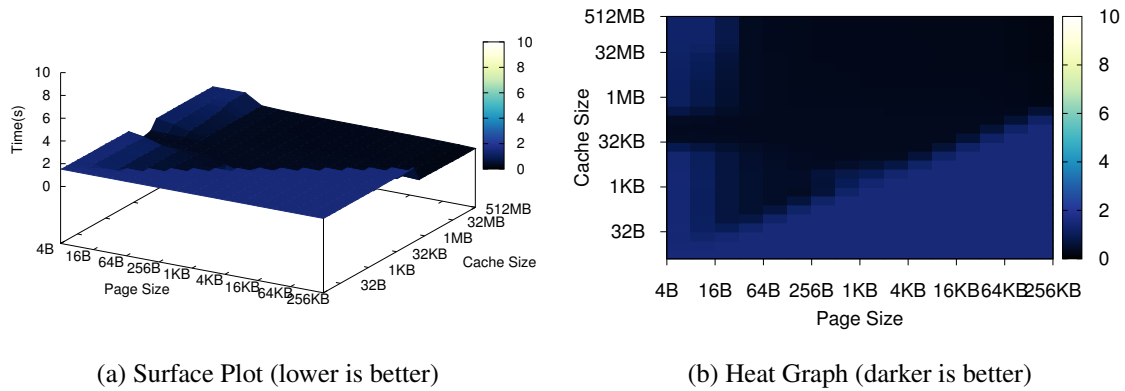


Figure 5.9: FLASH I/O benchmark execution time with a shared progressive page cache 1 CPU, 8 processes per node, and 4 I/O Nodes over Myrinet Myri10G.

Page Size	Cache Size	Total Exec. Time	Speedup
256KB	64MB	0.5799	7.34
256KB	32MB	0.5919	7.20
256KB	16MB	0.5960	7.15
256KB	8MB	0.5973	7.13

Table 5.7: Best performing shared progressive page cache configurations on Gigabit Ethernet (1 CPU, 8PPN, 4ION).

Page Size	Cache Size	Total Exec. Time	Speedup
256KB	64MB	0.1786	8.61
256KB	32MB	0.1865	8.24
256KB	16MB	0.2011	7.64
256KB	4MB	0.2012	7.64

Table 5.8: Best performing shared progressive page cache configurations on Myrinet Myri10G (1 CPU, 8PPN, 4ION).

Figures 5.8 and 5.9 show the benchmark performance data for our final cache configuration. Here we see that the cache performance is very similar to the performance of the non-shared cache. However, in these configurations the shared cache is using one-eighth the memory resources per node to achieve the same benchmark performance.

Tables 5.7 and 5.8 again show the four fastest configurations for each of the inter-connection network types. Although the performance is slightly better than in the non-shared case, the appeal of the shared cache is again the more efficient utilization of cache space and the opportunity for increased request bundling. By dedicating only 8MB per node to cache data (not including the file region metadata overhead), a shared cache can provide significantly reduced benchmark execution times.

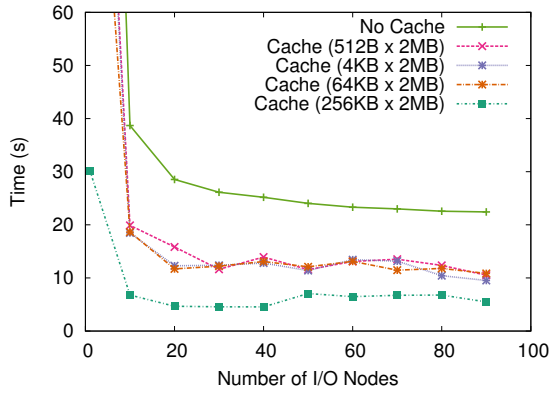
5.3.3 Cache Performance At Scale

Based on our experimental analysis, we have been able to construct a middleware cache that consistently achieves high performance for our benchmark program at small scales. We now wish to explore the effects of middleware caching on our benchmark application at larger scales.

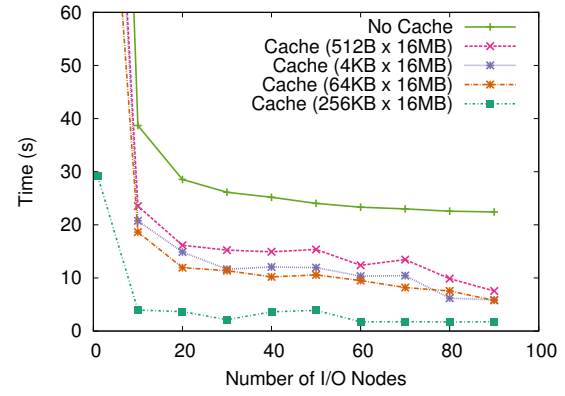
5.3.3.1 Gigabit Ethernet Network Experiments

Due to the extensive simulation time required to produce results for large-scale benchmarks, we are again limiting the number of measured cache configurations to a representative subset of the entire configuration space. We have selected page sizes of 512B, 4KB, 64KB, and 256KB in order to measure page size effects at scale. For each page size we perform our simulations using caches configured with data capacities of 2MB and 16MB. We do not include the space for progressive page overhead in our cache size configurations; however, considering the progressive page overhead can be considerable, we do not think extremely large client-side cache sizes are likely to be interesting to research scientists. Again, given 512 processes with 16MB progressive page caches, the cached data alone may consume as much as 8GB of main memory. With progressive page overhead we can imagine another GB of RAM dedicated to caching rather than application domain code. Dedicating significantly more RAM to a file data cache is unlikely in our opinion.

Figures 5.10 and 5.11 show the execution times for the FLASH I/O benchmark running as 512 processes on 64 Gigabit Ethernet nodes on file systems ranging from 1 I/O node to 90 I/O nodes. The left hand figures demonstrate the performance of each progressive page size on a cache configured with 2MB of capacity and the right hand figures show the benchmark execution time on caches configured with 16MB of capacity. As you would expect, the larger caches provide better performance; however, the performance im-

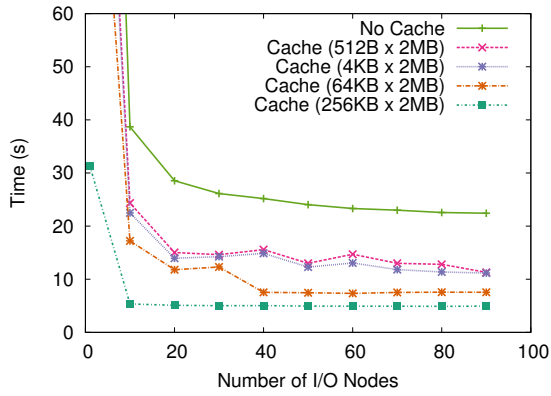


(a) Execution time on 2MB block-indexed cache

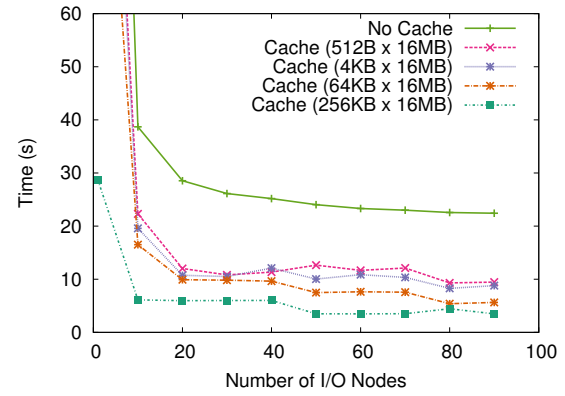


(b) Execution time on 16MB block-indexed cache

Figure 5.10: FLASH I/O benchmark execution time with a progressive page cache on 512 processes (64 Nodes, 8PPN) over Gigabit Ethernet (lower is better).



(a) Execution time on 2MB shared cache



(b) Execution time on 16MB shared cache

Figure 5.11: FLASH I/O benchmark execution time with a shared progressive page cache on 512 processes (64 Nodes, 8PPN) over Gigabit Ethernet (lower is better).

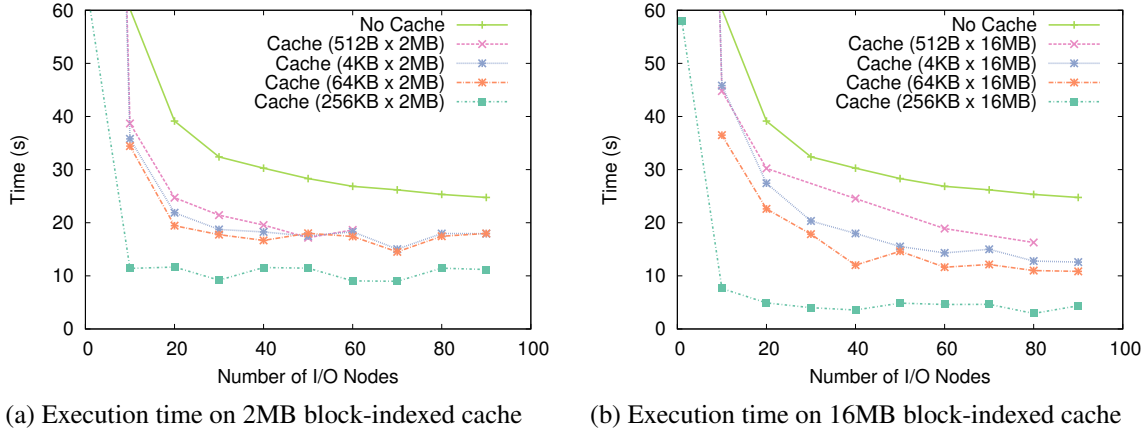


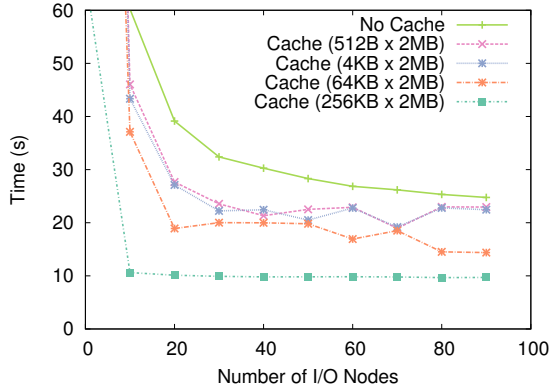
Figure 5.12: FLASH I/O benchmark execution time with a progressive page cache on 1024 processes (128 Nodes, 8PPN) over Gigabit Ethernet (lower is better).

provement appears to be significantly impacted by cache page size, a much different result than for our earlier fixed-size page designs. All of the configurations provide significant improvement over the non-cached execution.

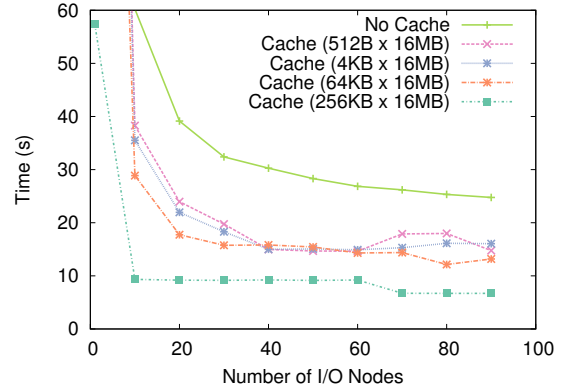
Similarly, figures 5.12 and 5.13 show the execution times for the FLASH I/O benchmark running as 1024 processes on 128 Gigabit Ethernet nodes. Again the parallel file system configurations range from 1 I/O node to 90 I/O nodes. At the thousand process scale, the performance improvements are greater for the larger cache sizes; however, increasing the progressive page granularity seems to have the strongest effect on benchmark execution time. All of the configured caches are able to improve the performance of the FLASH I/O benchmark.

5.3.3.2 Myrinet Myri10G Network Experiments

In Chapter 4 we saw that our flawed simulation of the Myrinet interconnect likely invalidated our large-scale cache configuration testing and results. We modified our simulator to uncap the buffer size in hopes of providing a simulation of how a high-performance network might perform, even though our simulation of such a network departs from reality.



(a) Execution time on 2MB shared cache



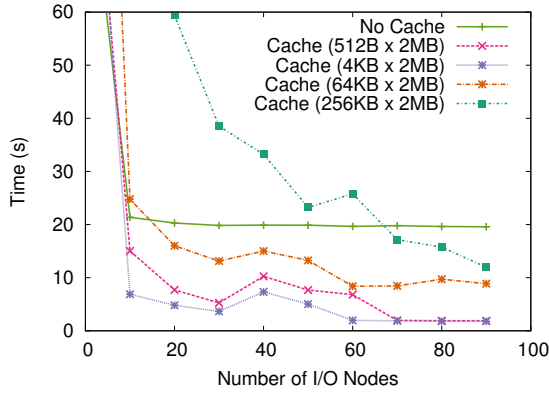
(b) Execution time on 16MB shared cache

Figure 5.13: FLASH I/O benchmark execution time with a shared progressive page cache on 1024 processes (128 Nodes, 8PPN) over Gigabit Ethernet (lower is better).

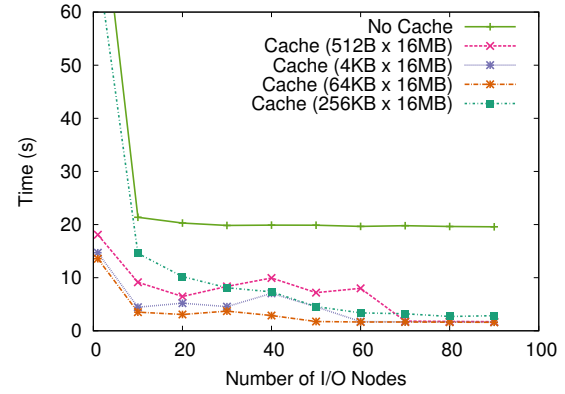
For these results, we perform the same modifications to our network model. As before, we believe that this set of results is useful for describing how a low-latency, high-performance interconnection network may perform with a progressive cache.

Figures 5.14 and 5.15 show the performance of the FLASH I/O benchmark simulated on the enhanced Myrinet network infrastructure. Again, the use of a client-side cache provides excellent performance improvements with both the cache per process and cache per node configurations.

Finally, figures 5.16 and 5.17 show the execution time for the benchmark running on a 1024 Myrinet interconnected processes. For the 2MB cache per node configuration we can see that the performance is generally improved; however, users should clearly favor a large page size. Similarly, when the cache size is increased to 16MB per node, the cache is able to again improve performance for most file system configurations, but the best performance is achieved with the largest page size, 256KB.

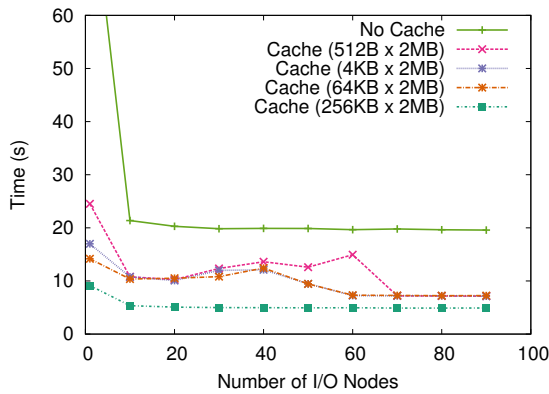


(a) Execution time on 2MB block-indexed cache

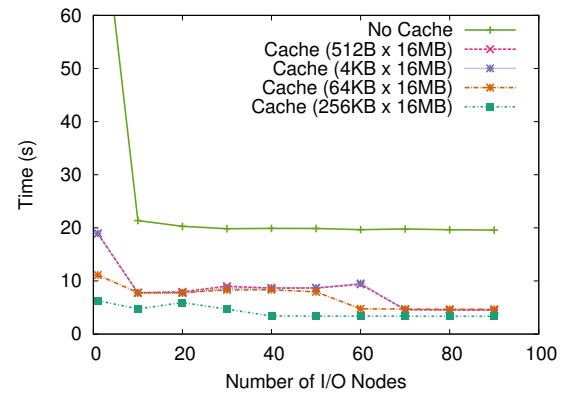


(b) Execution time on 16MB block-indexed cache

Figure 5.14: FLASH I/O benchmark execution time with a progressive page cache on 512 processes (64 Nodes, 8PPN) over Myrinet Myri10G with fully dynamic switch buffer size (lower is better).

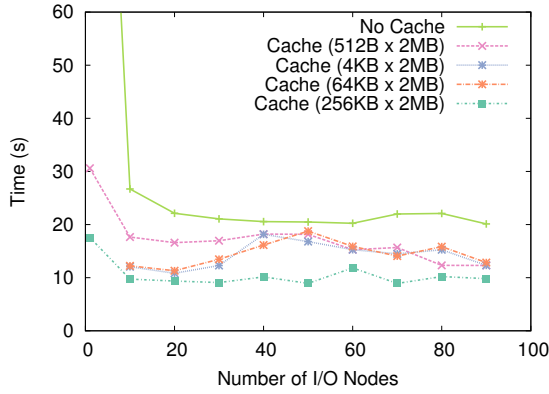


(a) Execution time on 2MB shared cache

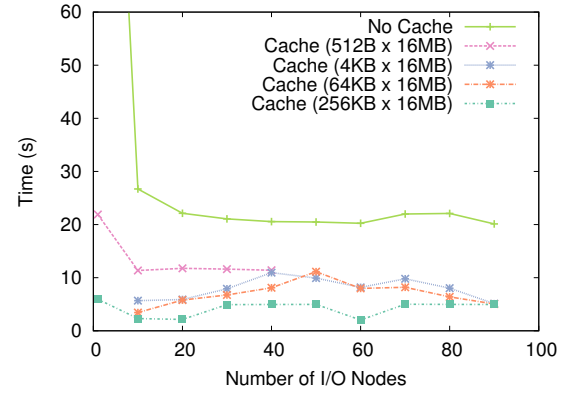


(b) Execution time on 16MB shared cache

Figure 5.15: FLASH I/O benchmark execution time with a shared progressive page cache on 512 processes (64 Nodes, 8PPN) over Myrinet Myri10G with fully dynamic switch buffer size (lower is better).

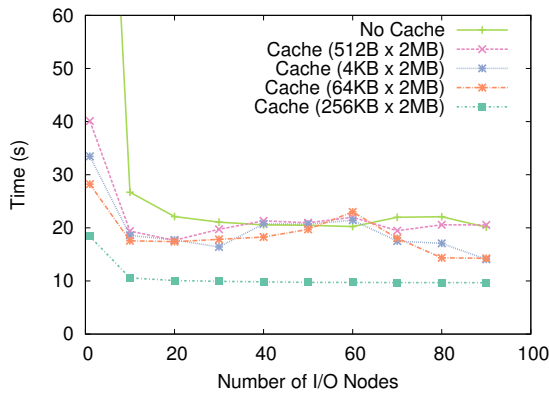


(a) Execution time on 2MB block-indexed cache

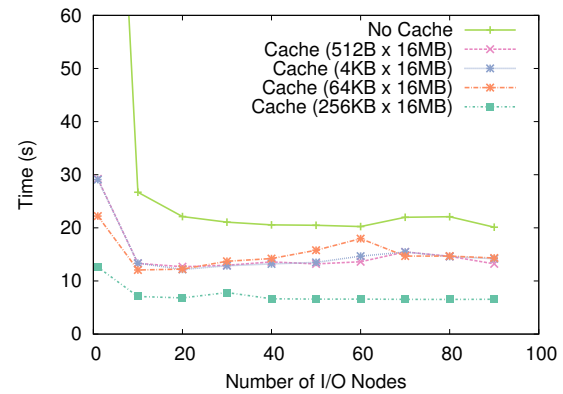


(b) Execution time on 16MB block-indexed cache

Figure 5.16: FLASH I/O benchmark execution time with a progressive page cache on 1024 processes (128 Nodes, 8PPN) over Myrinet Myri10G with fully dynamic switch buffer size.



(a) Execution time on 2MB shared cache



(b) Execution time on 16MB shared cache

Figure 5.17: FLASH I/O benchmark execution time with a shared progressive page cache on 1024 processes (128 Nodes, 8PPN) over Myrinet Myri10G with fully dynamic switch buffer size (lower is better).

5.4 Summary

In this chapter we introduced a new middleware caching paradigm for parallel file systems, progressive page caching. We looked at two possible implementations for maintaining the updated regions in a progressive page: a dirty-mask and a file region tree. We did an in-depth analysis of the runtime and storage requirements of each approach, and learned that if the number of updated regions in a page is relatively small compared to the size of the page, the file region tree offers better asymptotic performance and smaller storage overhead. These results were further strengthened by our small scale performance results, which indicated that large page sizes were a strong component in configuring a high-performance cache.

At large scale, the performance impacts of progressive page caching and the file region tree scheme became even clearer. A correctly configured progressive cache has the opportunity to strongly improve performance, and there is little risk that performance will be made worse by selecting unfortunate cache parameters. On the other hand, it is important to be selective in choosing the file I/O workloads with which to apply progressive page caching. It is not hard to imagine workloads that result in large amounts of file region meta-data storage, and perhaps even poor performance due to the growth in runtime proportional to large page sizes.

Chapter 6

MPI View Aware Aggregation

In our study of caching up to this point, we have frequently noted that the performance benefits of file data caching did not rely so much upon reading data already resident in the cache, but rather in improving the utilization patterns of the application to more closely match the file system’s capabilities. In this chapter we examine a technique that is independent of caching, but seeks to improve application performance by improving the application access granularity directly.

6.1 Collective I/O Aggregation

One of the benefits of the collective I/O interface available in MPI-IO is that it allows the application developer to provide more information to the middleware layers allowing for further optimization. In a traditional POSIX-style read or write, the file system is provided only a file handle, an offset, and an extent. In order to access sparse fields within a file, the application developer must issue an independent series of accesses, which makes it difficult for the underlying libraries to optimize access for the entire application rather than simply for each request. The anticipatory disk scheduler provided by the Linux

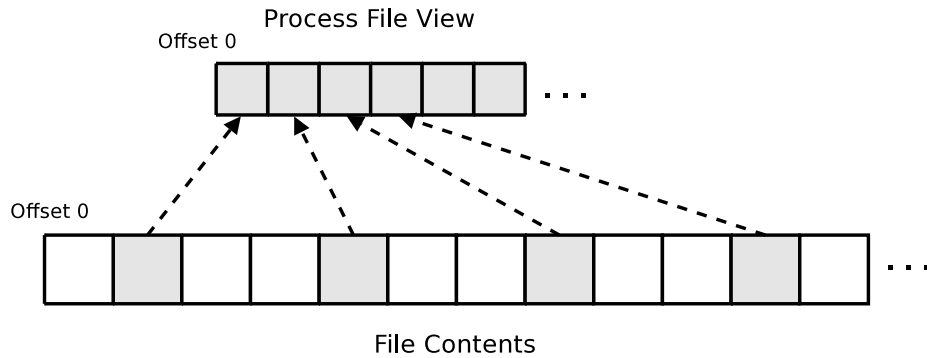


Figure 6.1: An example of a vectorized MPI file view. The file regions are assigned to the processes in a round robin fashion, assuring that file reads and writes access only the file data assigned to the individual process.

operating system was designed to address this issue by waiting for a small amount of time after receiving an I/O request to see if the next arriving I/O request should be scheduled in conjunction with the first request [30]. Although clever, this approach is not optimal and has been replaced in the Linux kernel [53].

To address this problem, POSIX file I/O has been extended to allow a list of offset and extent pairs to be included in a single I/O request. While the list I/O interface may be adequate for a single executing process, parallel jobs are composed of multiple processes, and the POSIX list I/O interface cannot incorporate the data from multiple processes into a single request. The standardization committee for MPI-IO recognized this shortcoming in the POSIX I/O specification and created *collective I/O* operations that allow the application writer to specify a single I/O request that spans multiple processes. In addition to the collective I/O interface, MPI-IO provides an alternative scheme for defining regular and repeating file regions called file views.

Similar to database views, MPI file views allow each process participating in a collective I/O to access only the disjoint set of file regions interesting to that process. For example, if four processes are participating in a collective I/O operation and accessing the file in a round-robin fashion, the application writer can specify a vector-based file view that

describes how the entire file is partitioned among the cooperating processes. Figure 6.1 shows an example of how an MPI process views the file after applying a file view. The utilization of MPI file views allows I/O library implementers to have greater knowledge of how the application writer intends to access the file. Any MPI data type can be used to build a file view, thus the application writer can specify vectorized access patterns, tiled access patterns, or combinations of any valid MPI data type.

6.1.1 Two-Phase I/O with Data Sieving

Two-phase I/O is one of the earliest aggregation approaches applied to MPI collective I/O operations [54]. In two-phase, the requests are forwarded to a subset of the client processes called aggregators. After all of the assigned collective I/O requests arrive at the aggregator, the library extracts the minimum file offset and maximum file offset from the assembled requests and requests all of the file data between the minimum and maximum offset. Once the contiguous data region has been retrieved from the file system, the process of extracting only the portions requested by the processes is called data sieving. In the case where the requests as a group are significantly less partitioned than each request independently the disk and network access performance increases dramatically.

Unfortunately, it is not difficult to imagine collective requests that will cause two-phase to request far more data than is truly necessary. For example, if a collective I/O intentionally accesses widely spaced out data regions, data sieving will retrieve the spaces in between the data as well as the desired data, resulting in pathologically poor performance. The implementers of data sieving have conceived of three solutions to this problem.

First, the file is partitioned round-robin among all of the aggregators, so that each aggregator only receives collective requests for a portion of the file. The reorganization of the requested file domains between the processes is called the shuffle step and may require

additional remote communication between the job processes. Secondly, the data sieving distance is limited to a configurable buffer size (by default 4MB) that prevents requesting massive amounts of data between the file regions accessed by the application. Finally, aggregation implementation determines if the data sieving step will request primarily non-requested data, and if that is the case, simply forwards the individual collective requests to the file system in that case.

6.1.2 View-based Aggregation

View-based aggregation modifies the basic data sieving aggregation implementation by sending the file views to the aggregators when the file view is set [5]. Then, rather than sending the individual requests to the aggregator nodes, it is only necessary to send the data buffer. At the aggregator, the buffer is mapped into the file views and the starting and ending offsets are extracted from the file view information. A single request spanning the entire file data range from the starting offset to the ending offset is then sent to the file system. In the case where remote communication is required between the job processes and aggregators, view-based aggregation offers opportunities for better performance by reducing the costs associated with the data shuffling step and does not result in worse performance than data sieving with two-phase I/O.

6.1.3 View-aware Aggregation

In this dissertation we present a new approach to aggregation we call view-aware aggregation. Similarly to view-based aggregation, the file views are sent to the aggregator and used to perform the request aggregation. However, rather than assume that the aggregator is a remote node, we operate under the assumption that aggregation is performed for all of the processes running on a single compute node. Thus the file views are accumulated

into shared memory at the aggregator without the need for a data shuffle step. As the file views are accumulated into shared memory, a new file view is constructed that combines all of the existing file view data types into a single data type if possible. When a collective I/O operation is performed on a compute node, all local processes participating in the collective operation will issue their requests into the shared buffer. The collective participant that arrives last in the buffer will then perform the file system request using the shared buffer of client data and the aggregated file view constructed earlier.

6.2 Implementation Details

At present, we are unaware of a valid algebra that can be used to combine arbitrary data types during request aggregation. MPI data types are very similar to a restricted set of regular expressions, but the Kleene algebras that exist for regular expression languages are descriptive and do not include operators for unification or equivalence among expressions [38]. Instead, we are relying on the assumption that most collective I/O operations will attempt to access the file with similar data types. Thus we can apply a heuristic approach to combining MPI file views.

In general, we believe that most scientific data solvers tend to operate on similar MPI data types. For example, if a scientist requires a matrix-matrix multiplication, most implementations will attempt to read the matrix data from the file using roughly equally sized chunks of the matrix at each process. In this case, the MPI file views are likely to be very regular, and a heuristic approach to combining the types can be effective. Similarly, if a vector of data is to be distributed among all of the processes, a heuristic approach can construct a single high-level data type of the complementary strided access patterns. On the other hand, if the file views in use are related in a subtle, or hard to detect fashion, our heuristic approach will fail, and our implementation will have to resort to simple data

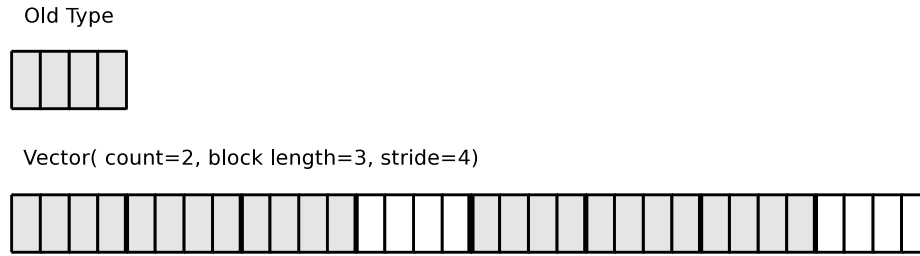


Figure 6.2: An example of a MPI vector data type.

sieving, or simply issue each request individually. Even when no prepackaged heuristic is capable of performing the view union, it may be possible for the application developer to provide a custom procedure for performing an application specific view aware data type re-combination such that the aggregators can provide improved performance.

6.2.1 File View Unions

At present, we have only developed a small number of union algorithms for use with our view-aware aggregator. In general, our approach to view joining has been to detect a fundamental period for the data types in use, and then perform the combination based on the size of the contiguous regions within the fundamental period. For example, consider a set of file views based on the vector data type. The vector data type is parameterized with a count, a block length, a stride, and the type of data stored in the vector (called the old type). The count is the number of blocks in the vector. The block length is the number of elements per block, and the stride is the number of elements between the start of each block.

Figure 6.2 shows an example of the valid data regions specified by a vector data type. In general, we expect that the vector described by the vector data type will be partitioned relatively evenly amongst all of the processes participating in a collective file I/O. In the case where all of the file views use the same vector block length and stride, with only

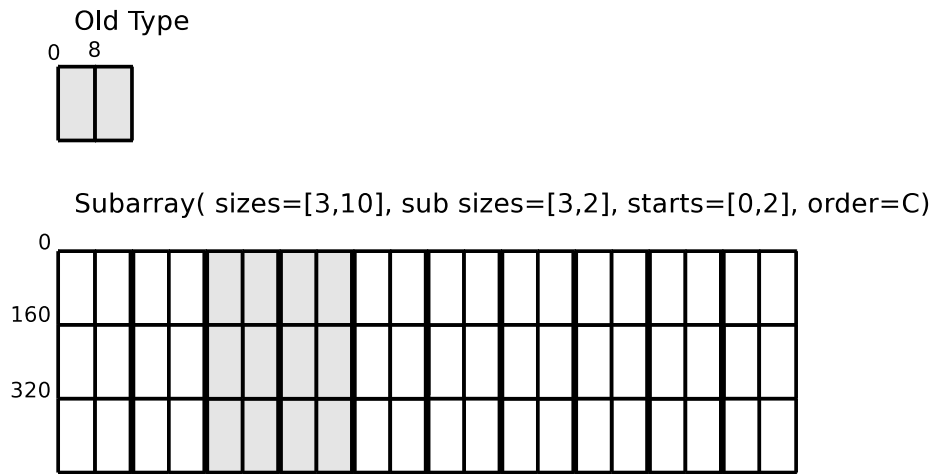


Figure 6.3: An example of a MPI subarray data type.

differing initial type displacements or counts, it is easy to assemble the beginning and ending offsets for a block into a set of offset and extent pairs, and then extract the contiguous regions for each block. Our code to perform this combination uses the code similar to the file region tree described in Chapter 5.

We perform a similar composition with file views using the Subarray data type. The Subarray data type is specified by an array of sizes, an array of sub-sizes, an array of starts, and a data ordering. The sizes array describes the size of the full array dimensions. The sub-sizes array describes the size of the sub-array dimensions. The starts array describes the offset into each full array dimension, and the data ordering specifies whether the data is stored in row-major order as in C, or in column-major order as in FORTRAN. Figure 6.3 shows an example of the data described by a subarray data type in MPI. In order to combine subarray data types to construct a unified data view, we essentially require that the sizes array must be identical for all composition data types, the sub-sizes array must be identical in all dimensions not storing data contiguously, and the starts array must generate a valid data period. While this set of constraints is stringent, if all of the array data is to be distributed relatively evenly among the processes, these constraints are likely to be true for

most of the collective participants. The unified Subarray data type can be constructed by collecting the length of elements in the contiguous dimension. If all of the contiguous data forms a single contiguous region, we construct a new Subarray data type. If the regions are not contiguous, we instead construct an Indexed data type to describe the disjoint data regions.

These same techniques can be used to combine Darray, Indexed, and Block-Indexed data types – provided that the individual data types all have the same fundamental period size. We have not explored any heuristics for combining irregular and non-periodic data types such as the MPI struct data type. If there does exist a fundamental period allowing a single data type to represent the data for multiple processes, users should be able to extend our approach to provide further data type combining heuristic approaches.

6.2.2 MPI Tile I/O Benchmark

In order to evaluate our view-aware aggregation scheme we are using a popular non-contiguous I/O test called the MPI Tile I/O benchmark [47]. The MPI Tile I/O benchmark accesses the file as a two dimensional array, with each process in the MPI job assigned a portion of the array called a tile. Each tile is 80 bytes by 1000 bytes, and before performing any file I/O the benchmark sets a Sub-array data type as the MPI file view to ensure that each process retrieves only the tile assigned to that process. All processes then participate in a collective file write operation or a collective file read operation depending on whether the benchmark is set to generate the tile data or access existing tile data.

Figure 6.4 shows how the tile I/O data is partitioned among the MPI processes. During a file read, all of the tile data will be retrieved from disk due to the simulated operating system buffer cache beginning in an empty state. During a file write of the tiled data the data will primarily be written into the operating system buffer cache without a

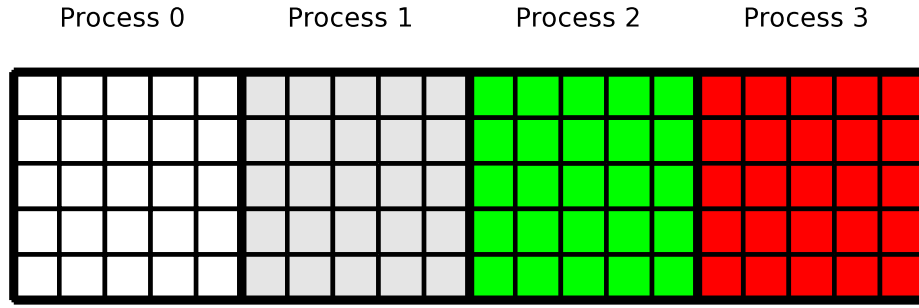


Figure 6.4: The MPI Tile I/O benchmark distributes two-dimensional tiles of data from within a single file to all of the job processes. The contiguous row data is limited to 80 bytes per process with 1000 columns per tile for a total I/O size of 80 kilobytes of data per process.

synchronous flush to disk. In theory the tile I/O read performance is heavily governed by the disk performance while the tile I/O write performance is primarily limited by the network bandwidth. In fact, due to the challenging nature of the tile I/O benchmark and the large number of nodes configured for our simulations we will see that switching hardware will often become a critical bottleneck for many of our optimization attempts.

It is important to note that we do not use the current default behavior of the PVFS driver in MPICH2 to access the tiled file data. The current driver accesses each contiguous tile row with an individual request; thus, for an 80x1000 tile, 1000 requests are generated to access the file data for each MPI process. Instead we use the high-level data type interfaces available in PVFS to access the tiles using a sub-array data type directly from the file system. By utilizing the file views within the MPI driver for PVFS we are able to avoid the overhead of deconstructing each non-contiguous file region into a separate I/O request, greatly improving file system performance in the base case. Additionally, we have modified the benchmark to access the tile I/O pattern collectively multiple times accessing data further along in the file each time without any overlap in file regions. We perform 100 access iterations in succession for the small scale benchmarking results and 50 access iterations for the large scale results. A single access iteration did not result in long enough

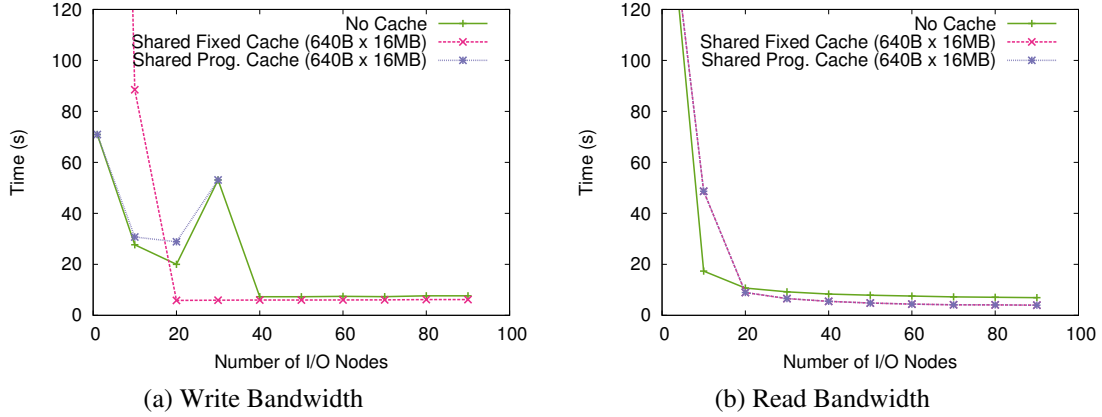


Figure 6.5: Simulated execution time for MPI Tile I/O benchmark using 64 file system clients (8 processes per node) connected with Gigabit Ethernet

runtimes to perform a straightforward data analysis.

Figure 6.5 shows the simulated baseline performance for the tile I/O benchmark executing on 64 compute nodes with 8 processes per node using the Gigabit Ethernet management network. The x -axis shows the number of I/O nodes employed by the parallel file system and the y -axis shows the benchmark execution time. We present the file write and file read times separately, and note that the file write is primarily network bound while the file read is primarily disk bound. We have also included timings for a shared fixed-size page cache and a shared progressive page cache, both configured with a cache page size matching the longest aligned contiguous region accessed in the file. Although our earlier caching results were able to significantly improve the performance of the flash I/O benchmark, the high-level Subarray data type already exploited by the tile I/O benchmark is able to provide better performance than the more general page-based data types used by the cache.

Figure 6.6 shows the same benchmarking result, this time using a Myrinet network without a global limit to the total switch buffer size (though the amount of buffer available to each network port is still limited). We have again included the benchmarking times

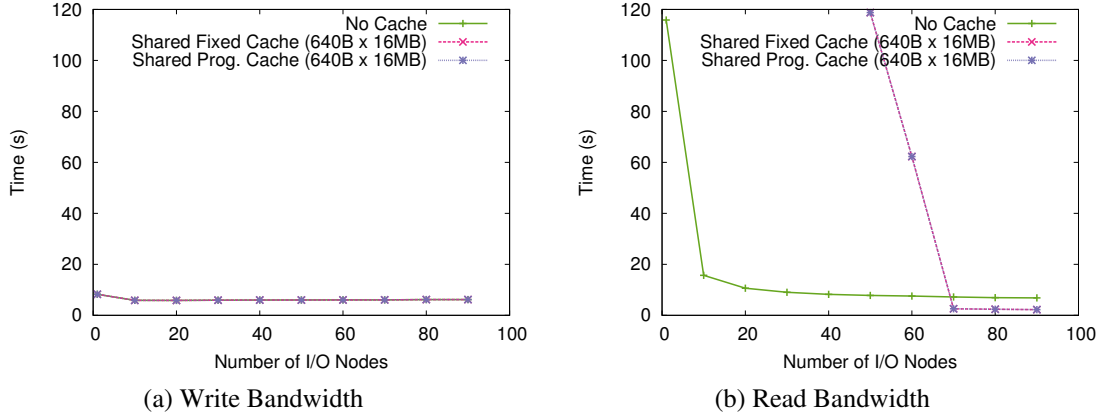


Figure 6.6: Simulated execution time for MPI Tile I/O benchmark using 64 file system clients (8 processes per node) connected with Myrinet

using our earlier caching designs; however both fixed-size page caching and progressive page caching were unable to significantly improve the benchmark performance.

6.3 Performance Analysis

Like our earlier shared caching approaches, view-aware aggregation seeks to improve performance by increasing the granularity of file system access. Multiple collective I/O requests are assembled into a single I/O request at each computation node. By reducing the costs associated with sending multiple requests over the network and updating the data file once rather than multiple times, we anticipate that view-aware aggregation can increase performance for applications that leverage collective I/O.

6.3.1 Small Scale Evaluation

As has been our practice in previous chapters, we begin our performance evaluation with a single multi-core computation node and a small file system configuration. Table 6.1 shows the execution time of the MPI Tile I/O benchmark running as 8 processes running

Aggregation Scheme	Total Time (GigE)		Total Time (Myrinet)	
	Read secs.	Write secs.	Read secs.	Write secs.
None	2.89164	2.66067	1.23024	0.436352
Data Sieving	2.92259	N/A	1.13668	N/A
View-Aware	2.92322	2.32597	1.13785	0.231396

Table 6.1: Tile I/O execution time on 1 CPU, 8PPN, 4ION.

Aggregation Scheme	Disk Time (GigE)		Disk Time (Myrinet)	
	Read secs.	Write secs.	Read secs.	Write secs.
None	0.96861	0.17585	0.97289	0.17720
Data Sieving	0.82454	N/A	0.82563	N/A
View-Aware	0.82552	0.03222	0.82738	0.03220

Table 6.2: Tile I/O disk access time on 1 CPU, 8PPN, 4ION.

on a single computation node. The underlying parallel file system is configured with 4 I/O servers. The difference in performance between the direct file system access, data sieving aggregation, and view-aware aggregation is modest at small scales; however there is clearly a small degree of performance improvement when writing the tiled data over the Myrinet network.

Table 6.2 shows the maximum time spent accessing the disk on the I/O nodes for each aggregation scheme. As the timing data shows, one of the advantages of aggregation schemes is the improved efficiency achieved in accessing the underlying file system storage components. Data sieving and view-aware aggregation effectively result in the same disk access time in this benchmark execution because both schemes are able to aggregate all of the collective I/O requests together into a single contiguous request. Of course, for this configuration, data sieving could be performed without a shuffle-step because the entire array of data was accessed by a single compute node. In the more general case, data sieving will require an extra communication phase because each compute node accesses eight possibly contiguous sub-array regions, but the data in between the sub-array rows is

larger than the sub-array, and thus the aggregator will request far more data than is actually required by local processes. The only solution provided by the full two-phase solution is the shuffle step that requires an additional one-to-many communication for each process to every aggregator allowing the requests to be reorganized into contiguous file domains assigned to each aggregator. Otherwise, the aggregator requests will include large regions of data not explicitly requested by the constituent processes.

6.3.2 Large Scale Evaluation

For our evaluation of view-aware aggregation at large scales we have traced the MPI Tile I/O benchmark on 512 processes. Because of the extended simulation time required to measure execution time for 512 processes, we only perform the tile I/O access pattern 50 consecutive times per process rather than the 100 iterations we performed for our small scale testing.

6.3.2.1 Gigabit Ethernet Network Experiments

Our simulated Gigabit Ethernet based on the hardware in Palmetto struggled to take advantage of the improved performance associated with view-aware aggregation. Although the disk access times for the view-aware approach were significantly improved, the network switch was unable to cope with the large amount of data arriving to and from each network port. As the switch buffers overflowed, Ethernet frames required multiple back-off and re-transmission cycles eventually leading to collapse in TCP throughput. This problem, called switch in-cast, has been frequently observed for large scale storage system access [59].

Figure 6.7 shows the execution time for both writing and reading the two-dimensional tile data from a file. In the case of the file read, the disk access appears to act as a governor, limiting the network traffic and achieving a reasonable performance improvement

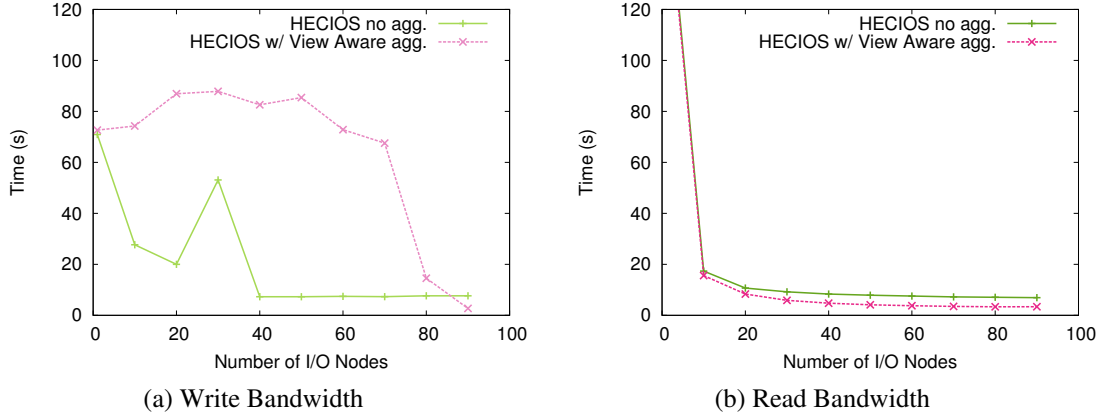


Figure 6.7: MPI Tile I/O benchmark execution time with view-aware aggregation on 512 processes (64 Nodes, 8PPN) over Gigabit Ethernet (lower is better).

over the base case execution time. The file write, however, triggers TCP collapse, and the primarily network bound tile data write with view-aware aggregation achieves much worse performance than if the optimization is not applied at all.

Due to the large scale of modern cluster computers several solutions have been proposed to lessen switch in-cast and TCP collapse in storage systems [59]. In this dissertation, we apply the simplest solution, we double the size of the switch buffers. Figure 6.8 shows the performance of our view-aware aggregation optimization on the exact same configuration as figure 6.7 with the exception that the amount of memory available within the network switch has been doubled. With this cluster configuration it is easy to examine the improved benchmark performance achieved via view-aware aggregation.

6.3.2.2 Myrinet Myri10G Network Experiments

Because our Myrinet simulation suffered from a similar in-cast induced performance problem earlier, we have again uncapped the limit on the available amount of memory in the entire switch; however, the amount of memory per switch port is still limited to 4000 bytes. Figure 6.9 shows the performance of the Tile I/O benchmark with view-aware

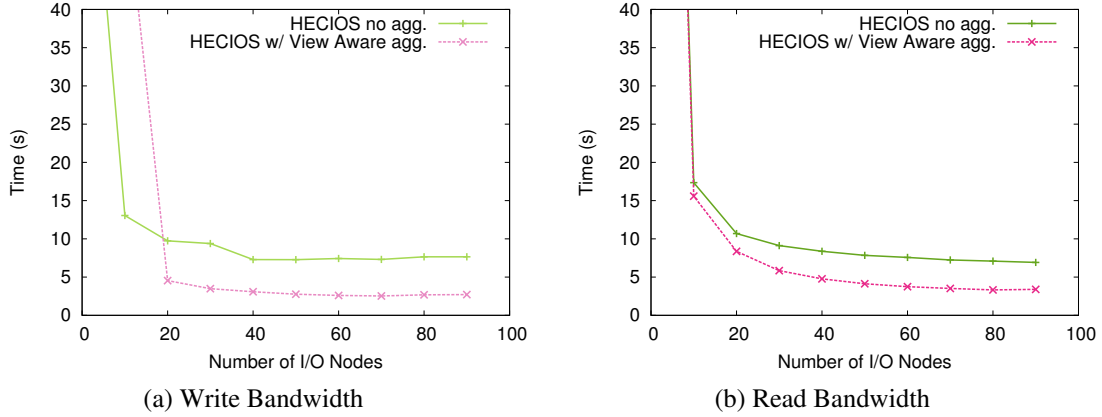


Figure 6.8: MPI Tile I/O benchmark execution time with view-aware aggregation on 512 processes (64 Nodes, 8PPN) over Gigabit Ethernet with increased switch buffering (lower is better).

aggregation at each compute node over the simulated Myrinet Myri10G interconnect. The initial file write performance instability is triggered due to insufficient switch buffering; but as the number of switch ports in use is increased due to a larger number of I/O nodes, the view-aware aggregation optimization demonstrated considerable performance improvements. In the case of the tile data read, the view-aware aggregation again significantly improves upon the performance of the default file system interface.

6.4 Summary

View-aware aggregation is a promising approach to improving the performance of applications already leveraging high-level data types and collective I/O operations. Unfortunately, view-aware aggregation does not pair well with our existing cache designs due to loss of typing data that occurs as data is copied into the cache. A useful modification to our earlier caching schemes could allow the cache to somehow use the data type supplied by the aggregator for later file system access. Unfortunately, we never found an approach for such a mechanism that could meaningfully improve performance beyond that provided by

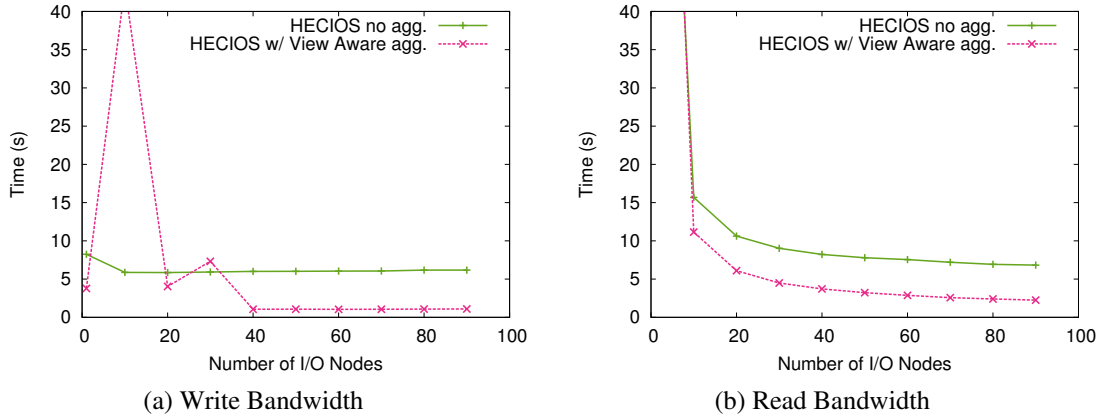


Figure 6.9: Comparison of empirical and simulated parallel file system bandwidth using 64 file system clients (8 processes per node) connected with Myrinet

the plain view-aware aggregator.

One drawback to the increased granularity of access provided by view-aware aggregation is the requirement to use extremely high-end switching hardware to achieve actual performance improvements. Modest commodity network switches simply will not provide the degree of network buffering needed to support applications running on hundreds or thousands of processes and accessing large-scale file systems running on tens or hundreds of I/O nodes. As the core density of modern clusters increases, the advantages of high-level request aggregation will become more and more relevant, but the network buffering capacity of storage networks will need to grow proportionately to support the increasing number of execution threads per network interface.

Chapter 7

Conclusions

In this dissertation we have focused on performing a rigorous study of the performance impacts of several different client-based caching architectures. In Chapter 1 we describe several workloads that exhibit performance problems for parallel file systems and described 3 file data cache designs to improve performance: shared caching, progressive page caching, and view-aware data caching. Chapter 2 describes the background literature for data caching in parallel applications and parallel file systems. Chapter 3 provides a detailed description of our simulator model and presents all of the results of our validation experiments. Chapters 4, 5, and 6 presents the architectural overviews for our cache designs, important cache implementation details, and our experimental results with each of the cache designs. In this chapter we summarize our experimental results, describe relevant future work that follows from this dissertation, and describe our contributions to the high performance computing and parallel file system communities.

7.1 Fixed-size Page Caching

We found that a fixed-size page caching approach was effective for improving the performance applications performing small, unaligned file access – provided the cache was able to sufficiently improve the granularity of file system access. By utilizing file system interfaces supporting high-level data type access we were able to significantly improve the performance of the FLASH I/O benchmark, and demonstrated a shared caching approach that provided high performance with only moderate memory requirements. We also demonstrated that it is possible to efficiently achieve close-to-open cache coherence for our file system by using a false-sharing avoidance scheme based on the Treadmarks system. One disappointing note is that although the page caching approach improved application performance, we still encountered little scalability as the number of I/O nodes used in the parallel file system were increased.

7.2 Progressive Page Caching

Progressive page caching provides a further improvement upon the performance of the fixed-size page caching design by eliminating the need to maintain cache coherence at the page granularity. Progressive page caching maintains the cache coherence only for data actually updated by the application. We examined two alternative data structures for storing the additional metadata required to implement progressive page caching, a dirty-mask approach and a file region tree, with an assessment that the density of the page data is likely to determine which data structure is more appropriate. Finally, we performed a series of benchmarks with our progressive page caching scheme and learned that large progressive cache page sizes generally provide better performance than smaller cache pages, and are as an important factor in application I/O performance as the amount of memory devoted to

the cache.

7.3 File View Aware Aggregation

While our caching designs attempted to find a regular page-based data structure in a series of small, unaligned file accesses, our MPI file view-aware aggregation approach attempts to leverage the data type information provided by the collective I/O interfaces provided by MPI. View-aware aggregation improves file system access granularity by combining the data types of a collective I/O operation participants into a single data type driven request. By combining the data types supplied by each processes file view, the aggregator is able to bundle file access operations together and improve performance. Earlier aggregation approaches relied on the application data arriving at the aggregator as mostly contiguous requests, but view-aware aggregation attempts to create a new sparse data type from the already sparse data types, improving performance even in the case where the data is highly structured, but also spaced out within the file.

7.4 Future Work

Our study of parallel file system caching has resulted in many useful discernment into how caching mechanisms can actually improve file system performance. Equally important, our study has also provided an insight into what caching modifications and refinements merit further study. The benefit of performing simulation-based evaluations is the low costs associated with looking at widely varied and speculative approaches to parallel file system performance. In this section we look at several caching modifications that may provide improved file system performance or improved application I/O semantics.

7.4.1 Cache Page Replacement

During our study of page-based caching we briefly discussed the *working set*, or the amount of cache space required to service an application's computational phase between each file access. Unfortunately, given the finite quantities of physical memory available to software developers, the amount of cache space available may not always exceed the size of the application's working set. In such cases, the replacement policy used by our file system cache may be a critical factor in cache performance. For example, if an application is writing data to the file system in some non-sequential order, the cache may need to re-read the data for a cache page many times in order to write small regions to the page. Obviously, re-reading a cache page multiple times is bad for performance (progressive paging is designed to avoid this overhead entirely), but an intelligent cache replacement algorithm may be able to avoid this overhead by only evicting cache pages that will not be accessed later.

Our caching studies used a least recently used (LRU) cache page replacement policy entirely. LRU page replacement can be the worst replacement policy for applications performing file accesses in a cyclical manner, and a configurable approach that allows the application developer to select the best replacement policy may be of some use. The developer's of Sun Microsystem's ZFS file system contend that a system that combines LRU data with frequency of page access called adaptive replacement caching is an effective scheme for balancing the most popular types of file system workloads. A workload characterization study coupled with an examination of page replacement schemes could provide useful insight into the importance of cache page eviction policies.

More speculatively, we could attempt to construct a cache page replacement algorithm that attempts to evict pages based on how quickly the evicted data can commit to the underlying file system. For example, multiple full cache pages can be efficiently de-

scribed with a single block-indexed data type, and written to the file system in a single request. Because we do not generally anticipate file re-read and re-write access patterns, a fastest-commit first eviction policy could result in significantly improved performance. Additionally, the policy may choose to write data pages that do not require eviction, thus further increasing the granularity of file system access without requiring any additional cache tuning by the application developer. One difficulty in developing a fastest-commit-first eviction policy is the need for a page scoring system that can accurately rank how quickly partially updated pages can be committed into the file system.

7.4.2 Consistency

One of the limitations of the caches described in this dissertation is the reliance upon close-to-open consistency. Although close-to-open consistency is apt to become the de facto standard for parallel file systems due to its adoption by NFSv4, some application developers are likely to need stronger consistency semantics. An exploration of directory-based caching for parallel file systems is one possibility for providing a stronger semantic at the cost of a much more complicated caching protocol. Because directory-based caches require that a page exists in a modified state on only one file system client, an eager-release consistency semantic can be implemented for each file I/O (requiring a synchronization at each I/O as well) rather than just a synchronization triggered by file open and close calls. One of the difficulties of a directory-based scheme is communicating from the file system servers to the various client processes. Advanced messaging daemons such as the MPDs used by MPICH2 may be a critical factor for implementing directory-based caching at the file system servers.

The second major consistency issue we discussed in this caching study is the violation of close-to-open consistency triggered by shared caching. Both our shared fixed-size

page caching and shared progressive page caching designs violated close-to-open consistency by requiring every process on the node to close the file in order to trigger the cache flush to disk. In the case of progressive page caching, it is possible to provide close-to-open consistency even in the shared cache case, although likely at the cost of some performance. For example, by modifying the file region tree to include the originating process rank along with each dirty region in the tree, it would be possible to flush only the correct dirty regions to the file system. In the case that multiple processes update the same file regions, then the affected file regions would need to maintain a list of modifying processes rather than simply a single field. Of course, the region tree will become more segmented and the benefits of sharing the cached file data may disappear as the amount of cache metadata increases, but in the case of few overlapping file regions, the storage and performance overhead may be acceptable for applications that require strict close-to-open consistency.

7.4.3 Writeback Buffers

One of the cache modifications we attempted to study, but were unable to achieve positive performance improvements was a cache writeback buffer. A writeback buffer allows file reads and writes to complete immediately without waiting for any resulting evictions to commit to the underlying file system. Unfortunately, in our studies of writeback buffering we found that the increased rate of file system I/O resulted in performance degradation. A more thorough examination of fixed-size writeback buffering may be able to determine how much writeback buffering is required to improve cache performance and at what point the amount of available writeback buffer leads to network congestion and unstable performance in file system access.

7.4.4 File System Tuning

Although the large-scale performance improvements using our middleware cache were noteworthy, we were disappointed that varying the number of file system I/O nodes did not alter the overall benchmark performance significantly. In our simulation trials, we kept the parallel file system striping factor – the amount of contiguous file data mapped to each server in turn – at the default size of 64KB. Although our caching middleware clearly impacts the granularity of file system interactions, it is possible that the default stripe size is simply too large and a smaller striping factor, possibly less than the cache page size, may result in greater file system scalability. Another possibility is two dimensional file data distributions that stripe data ranges into subsets of the available data servers, meaning that a larger contiguous range may map to a smaller subset of I/O nodes, but the file data in whole is still mapped evenly over all of the file system’s available I/O nodes.

One element of file system tuning that we know improves performance is the use of the high-level data type interfaces available in PVFS. We feel that the exploration of other non-Posix file I/O APIs is a useful study, and likely to result in further performance improvements. During our study of view-aware caching we discussed the need to copy data into the cache to construct a contiguous buffer for the PFS client interface. A more general API that can work with multiple client memory buffers can prevent those additional copies and further improve performance with no additional costs to applications performing Posix-based I/O. Another possibility is the ability to pin cache pages into the file system directly, allowing the file system and user applications to share memory directly.

Finally, extremely general scatter-gather I/O primitives have proved extremely useful for many types of I/O domains and may conceivably provide better performance for many typical cache access patterns. One problem is a lack of understanding in how request data fragmentation affects parallel file systems. The traditional focus on file system

interfaces for contiguous data has left a glaring hole on how high-level interfaces for file systems impact performance. A detailed study of how request fragmentation affects parallel file system performance is warranted, particularly given that the costs associated with fragmentation may disappear with emerging disk technologies (e.g. solid-state and phase-change storage devices).

7.5 Contributions

This work provides four significant contributions to the field of parallel file systems and parallel storage:

1. The availability of HECIOS, the High-End Computing I/O Simulator. Upon completion of this work, HECIOS will be released under the GNU Public License, an open source software license that allows anyone to modify and use the code for research purposes. One goal of HECIOS is to provide a suitable environment for easily measuring prospective parallel file system extensions. Although the HECIOS simulation model closely follows the PVFS software model, HECIOS is designed to be extended. Adding in constructs unique to other parallel file systems, such as a distributed locking service, allows HECIOS to become a general parallel file system research tool. In this way, HECIOS will advance the research and development of PVFS, and other production file systems deployed in multiple high-end computing systems.

In our studies we were able to configure HECIOS to accurately model Clemson University's Palmetto cluster, including its Gigabit Ethernet network and Myrinet Myri10G network. We have performed extensive parameter sweeps over cache page sizes and cache capacities at small scale and large scale for both interconnection networks, and

we have been able to use HECIOS to design middleware components that provide stable performance over a wide range of configurations and system scales.

2. In order to perform interesting studies with the HECIOS simulator, it was important to use representative application execution traces. As part of this work we have performed traces of several popular parallel I/O benchmarks. The availability of Clemson University's Palmetto cluster made it possible to generate traces for applications running simultaneously on thousands of processes which is a significant improvement over the existing 128 node traces currently available from Los Alamos National Laboratory.
3. The most important component of our contributions is a study of the performance improvements available by leveraging the high-level data type parallel file system interfaces with shared caching and request aggregation. Our initial fixed-size page cache and progressive page cache were effective at improving the granularity of access of the parallel file system, but the performance stabilized and improved more generally when we employed shared cache architectures that increased the density of data on each page allowing the caches to access the file system with highly efficient block-indexed data types. Effectively, our caching infrastructure allowed the caches to find some amount of regular structure in the data even though the file data accesses dispatched by the application were presented in an irregular ordering. Similarly, our view-aware aggregation scheme improved the granularity of applications already leveraging the high-level data type accesses by aggregating the data types at each multi-core compute node and constructing a new data type encompassing all of the constituent data types. The performance advantages of high-level file system interfaces has long been important, but our techniques have been shown to access this performance without further modifying and complicating end-user code.

4. Finally, our contributions include a detailed study of the performance of our middle-ware component designs over a wide array of parameter settings and at large scales. Past studies have been limited to small scales and limited parameter sweeps due to the inability to get significant time on a high performance cluster. Additionally, file system enhancement trials are often contaminated because the parallel file systems under study are often servicing other applications during the benchmarking process. This interference leads to contaminated results and many repeated benchmarks to try and average out the noise in the experimental results. Because we invested the time in building and validating our simulator, we were able to more efficiently collect experimental results and leverage the cluster hardware available at Clemson, but we instead leveraged it as a reference system for validation and computing resource for simulation experiments rather than as an actual benchmarking system.

7.6 Closing

In this dissertation we have endeavored to understand how middleware-based data caching and request aggregation affect the performance of parallel file systems. We have shown that for some workloads, and for some cluster configurations, that client-side data caching can provide substantial performance improvements. On other workloads, it is reasonable to assume client-side caching will detract from overall application performance rather than improve it – particularly those applications already relying upon high-level data type access. This mixed benefits scenario is unfortunately the nature of most sufficiently complex systems. Given the constraints of the physical devices employed by parallel file systems, there is no single best approach to improving the performance of a parallel file system.

In the coming months and years it is likely that CPU performance will continue

to improve, network performance will improve, and even disk performance will increase – perhaps dramatically. It is also undoubtedly true that with increasing performance will come increasing application demands in the forms of higher fidelity data sets, increased computational scale, and emerging scientific fields with completely novel computational requirements. While the improvements in the performance of the underlying technologies will no doubt increase parallel file system performance, access to high-level file system interfaces will still exist as an important optimization for accelerating the file-based I/O of demanding parallel applications with storage workloads dominated by small, unaligned file accesses. The benefits of latency reduction and request bundling are fundamental in improving the throughput of parallel file systems and will be important in building file systems capable of meeting the demands of future scientific efforts.

Bibliography

- [1] Cristiana Amza, Alan L. Cox, Hya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29:18–28, 1996.
- [2] Siddhartha Annapureddy, Michael J. Freedman, and David Mazières. Shark: scaling file servers via cooperative caching. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 129–142, Berkeley, CA, USA, 2005. USENIX Association.
- [3] Rajive Bagrodia, Stephen Docy, and Andy Kahn. Parallel simulation of parallel file systems and I/O programs. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–17, New York, NY, USA, 1997. ACM.
- [4] A. Batsakis and R. Burns. NFS-CD: Write-enabled cooperative caching in NFS. *IEEE Transactions on Parallel and Distributed Systems*, 19(3):323–333, March 2008.
- [5] Javier Garcia Blas, Florin Isaila, David E. Singh, and J. Carretero. View-based collective I/O for MPI-IO. pages 409–416, May 2008.
- [6] Daniel P. Bovet and Marco Cesati. *Understanding the LINUX Kernel, 3rd Edition*, pages 611–622. O'Reilly, 2006.
- [7] Ralph Butler, William Gropp, and Ewing Lusk. A scalable process-management environment for parallel programs. In *In Euro PVM/MPI*, pages 168–175. Springer-Verlag, 2000.
- [8] Surendra Byna, Yong Chen, Xian-He Sun, Rajeev Thakur, and William Gropp. Parallel I/O prefetching using mpi file caching and I/O signatures. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '08)*, 2008.
- [9] Philip H. Carns. *Achieving Scalability in Parallel File Systems*. PhD thesis, Clemson University, Clemson, SC, May 2005.

- [10] Philip H. Carns, Walter B. Ligon, III, Robert Ross, and Pete Wyckoff. BMI: A network abstraction layer for parallel I/O. *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, April 2005.
- [11] David Chaiken, John Kubiawicz, and Anant Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV*, pages 224–234. ACM, 1991.
- [12] Thomas H. Cormen, Charles E. Leiserson, and Ronald D. Rivest. *Introduction to Algorithms*, pages 244–262. The MIT Press, 1990.
- [13] Thomas H. Cormen, Charles E. Leiserson, and Ronald D. Rivest. *Introduction to Algorithms*, pages 263–280. The MIT Press, 1990.
- [14] CppUnit – C++ unit testing framework. <http://cppunit.sourceforge.net/>.
- [15] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach, 3rd Edition*, pages 273–283. Morgan Kaufman, 1999.
- [16] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach, 3rd Edition*, pages 286–291. Morgan Kaufman, 1999.
- [17] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach, 3rd Edition*, pages 328–329. Morgan Kaufman, 1999.
- [18] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach, 3rd Edition*, pages 553–571. Morgan Kaufman, 1999.
- [19] Michael Dahlin, Randolph Wang, Thomas E. Anderson, and David A. Patterson. Co-operative caching: Using remote client memory to improve file system performance. In *Operating Systems Design and Implementation*, pages 267–280, 1994.
- [20] A. Devulapalli and P. Wyckoff. File creation strategies in a distributed metadata file system. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10, 26-30 March 2007.
- [21] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo. FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes. *The Astrophysical Journal Supplement Series*, 131:273–334, November 2000.

- [22] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *SIGARCH Comput. Archit. News*, 18(3a):15–26, 1990.
- [23] Network Working Group. NFS version 3 protocol specification. <http://www.ietf.org/rfc/rfc1813.txt>, 1995.
- [24] Network Working Group. Network file system (NFS) version 4 protocol. <http://www.ietf.org/rfc/rfc3530.txt>, 2003.
- [25] HDF5 scientific file format. <http://hdf.ncsa.uiuc.edu/HDF5/>.
- [26] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 3rd Edition*, pages 678–679. Morgan Kaufman, 2003.
- [27] James V. Huber, Jr., Andrew A. Chien, Christopher L. Elford, David S. Blumenthal, and Daniel A. Reed. PPFS: a high performance portable parallel file system. In *ICS '95: Proceedings of the 9th international conference on Supercomputing*, pages 385–394, New York, NY, USA, 1995. ACM.
- [28] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope consistency: a bridge between release consistency and entry consistency. In *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 277–287, New York, NY, USA, 1996. ACM.
- [29] Florin Isaila, Guido Malpohl, Vlad Olaru, Gabor Szeder, and Walter Tichy. Integrating collective I/O and cooperative caching into the "clusterfile" parallel file system. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 58–67, New York, NY, USA, 2004. ACM.
- [30] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous I/O. *SIGOPS Oper. Syst. Rev.*, 35(5):117–130, 2001.
- [31] R. Colin Johnson. Intel's teraflops chip uses mesh architecture to emulate mainframe. *Electronic Engineering Times*, page 12, February 12 2007.
- [32] Wei keng Liao, Kenin Coloma, Alok Choudhary, Lee Ward, E. Russell, and S. Tide-man. Collective caching: application-aware client-side file caching. In *HPDC '05: Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing, 2005 (HPDC-14)*., pages 81–90, Washington, DC, USA, 2005. IEEE Computer Society.
- [33] Jack Kleijinen and Willem Van Groenendaal. *Simulation: A Statistical Perspective*, pages 205–219. John Wiley and Sons, 1992.

- [34] Igor Kotenko and Alexander Ulanov. The software environment for multi-agent simulation of defense mechanisms against ddos attacks. In *CIMCA '05: Proceedings of the International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce Vol-1 (CIMCA-IAWTIC'06)*, pages 283–289, Washington, DC, USA, 2005. IEEE Computer Society.
- [35] David Kotz and Carla Schlatter Ellis. Practical prefetching techniques for multiprocessor file systems. *Distributed Parallel Databases*, 1(1):33–51, 1993.
- [36] Samuel Kotz, Campbell B. Read, N. Balakrishnan, and Brani Vidakovic. Anderson darling test of goodness of fit. In *Encyclopedia of Statistical Sciences, 2nd Edition*, volume 4, pages 2881–2884. John Wiley and Sons, 2005.
- [37] Samuel Kotz, Campbell B. Read, N. Balakrishnan, and Brani Vidakovic. Johnson’s system of distributions. In *Encyclopedia of Statistical Sciences, 2nd Edition*, volume 6, pages 3751–3763. John Wiley and Sons, 2005.
- [38] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Infor. and Comput.*, 110(2):366–390, May 1994.
- [39] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.
- [40] ScaLAPACK – scalable linear algebra PACKage. <http://www.netlib.org/scalapack/>.
- [41] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 148–159, New York, NY, USA, 1990. ACM.
- [42] Jianwei Li, Wei keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing (CDROM)*. ACM Press, 2003.
- [43] Wei-Keng Liao, Avery Ching, Kenin Coloma, Alok Choudhary, and Lee Ward. An implementation and evaluation of client-side file caching for mpi-io. pages 1–10, March 2007.
- [44] Wei-Keng Liao, Avery Ching, Kenin Coloma, Arifa Nisar, Alok Choudhary, Jacqueline Chen, Ramanan Sankaran, and Scott Klasky. Using MPI file caching to improve parallel write performance for large-scale scientific applications. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–11, New York, NY, USA, 2007. ACM.

- [45] Xiaosong Ma, M. Winslett, Jonghyun Lee, and Shengke Yu. Faster collective output through active buffering. *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 34–41, 2002.
- [46] Michael Zingale. FLASH I/O Benchmark Routine – Parallel HDF 5. http://flash.uchicago.edu/zingale/flash_benchmark_io/, May 2002.
- [47] Parallel I/O benchmarking consortium. <http://www.mcs.anl.gov/research/projects/pio-benchmark/>.
- [48] Myri-10G and Myrinet-2000 performance measurements. <http://www.myri.com/scs/performance/>.
- [49] PETSc portable, extensible toolkit for scientific computation. <http://www-unix.mcs.anl.gov/petsc/petsc-as/index.html>.
- [50] F. Praus, W. Granzer, G. Gaderer, and T. Sauter. A simulation framework for fault-tolerant clock synchronization in industrial automation networks. *ETFA: IEEE Conference on Emerging Technologies and Factory Automation, 2007*, pages 1465–1472, Sept. 2007.
- [51] ROMIO: A high-performance, portable MPI-IO implementation. <http://www.mcs.anl.gov/romio>.
- [52] Bradley W. Settlemyer. A mechanism for scalable redundancy in parallel file systems. Master’s thesis, Clemson University, Clemson, SC, May 2006.
- [53] D. John Shakshober. Choosing an I/O Scheduler for Red Hat Enterprise Linux 4 and the 2.6 kernel. *Red Hat Magazine*, (8), June 2005.
- [54] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in romio. In *In Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, 1998.
- [55] Rajeev Thakur, William Gropp, and Ewing Lusk. Optimizing noncontiguous accesses in MPI-IO. *Parallel Computing*, 28(1):83–105, January 2002.
- [56] Nancy Tran and Daniel A. Reed. Arima time series modeling and forecasting for adaptive I/O prefetching. In *ICS ’01: Proceedings of the 15th international conference on Supercomputing*, pages 473–485, New York, NY, USA, 2001. ACM.
- [57] U.S Department of Energy. U.S. Department of Energy’s new supercomputer is fastest in the world. <http://www.energy.gov/news/6321.htm>, June 9 2008.
- [58] U.S. Department of Energy. The office of science data-management challenge, March-May 2004.

- [59] Vijay Vasudevan, Hiral Shah, Amar Phanishayee, Elie Krevat, David Andersen, Greg Ganger, and Garth Gibson. Solving TCP incast in cluster storage systems. 2009.
- [60] Murali Vilayannur, Anand Sivasubramaniam, Mahmut Kandemir, Rajeev Thakur, and Robert Ross. Discretionary caching for I/O on clusters. In *CCGRID '03: Proceedings of the 3rd International Symposium on Cluster Computing and the Grid*, page 96, Washington, DC, USA, 2003. IEEE Computer Society.
- [61] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. Dynamic metadata management for petabyte-scale file systems. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 4, Washington, DC, USA, 2004. IEEE Computer Society.
- [62] Ying Xu and Brett D. Fleisch. NFS-cc tuning NFS for concurrent read sharing. *International Journal of High Performance Computing and Networking*, 1(4):203–213, 2004.