REACTIVE SCHEDULING FOR PARALLEL I/O SYSTEMS

By

Robert B. Ross

A DISSERTATION

Submitted to
Clemson University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Computer Engineering

December 2000

Walter B. Ligon III

December 15, 2000

To the Graduate School:

This dissertation entitled "Reactive Scheduling for Parallel I/O Systems" and written by Robert B. Ross is presented to the Graduate School of Clemson University. I recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy with a major in Computer Engineering.

Walter B. Ligon III, Advisor

We have reviewed this dissertation
and recommend its acceptance:

Robert Geist

Adam Hoover

Ron Sass

Accepted for the Graduate School:

ABSTRACT

Parallel computing is integral to high performance computing, but it is not uniquely sufficient. With the adoption of parallel computing, some additional supporting technologies are required. Parallel I/O is one such supporting technology, providing high speed data storage in parallel computing environments. Parallel I/O systems have emerged and are beginning to see use in the main stream; however, research into optimizing these systems is still an open area. In particular, techniques for optimizing parallel I/O have focused on disk performance optimization when other resources might have equal or greater impact on overall performance. Other work has looked at adaptive techniques for optimizing in these systems, but has focused on caching and prefetching only.

In this work we present *reactive scheduling* (RS), which combines adaptive methods with scheduling algorithms to optimize service order to the workload and system state. First the Parallel Virtual File System (PVFS), a parallel file system developed for researching issues in parallel I/O, is described. Second a selection of additional scheduling algorithms are added to PVFS and a study of four workloads is performed on the system. Third a model is developed which matches PVFS performance under the workloads studied. Fourth the model and scheduling techniques are combined to create an implementation of RS for PVFS. This implementation is tested and compared against the best case performance seen

in our workload study, showing as much as 60% improvement in task service time over the original PVFS system. Finally conclusions are drawn on the viability of the RS technique and directions of future study are described.

TABLE OF CONTENTS

List of Tables

LIST OF FIGURES

Chapter 1

Introduction

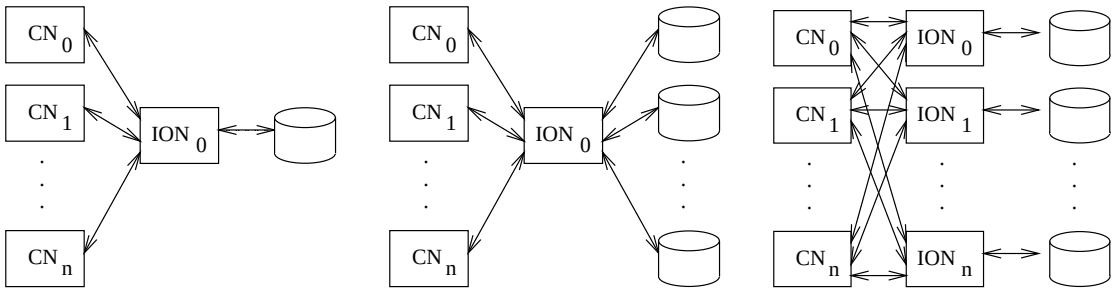
Performance improvements in computing technology have vastly out-paced improvements in storage technology. However, computing power as a whole depends not only on raw processing power but input and output (I/O) capability as well [18]. Many of the applications utilizing high performance computing have significant input and output (I/O) requirements, and the disparity in advances between computation and I/O hardware has forced new approaches to providing I/O for these applications. Examples of such applications include satellite image processing, which often involves very large, very detailed, multi-band images, and genome research, which often centers around matching sequences to others in very large databases.

1.1 Disk Arrays and RAID

One important advancement in high performance I/O was the use of disk arrays [24] and redundant arrays of inexpensive disks (RAIDs) [40]. Disk arrays are simply collections of

smaller, less expensive disks used to replace larger ones. Connected to a single host, these disks are used in concert to provide a higher bandwidth and potentially higher capacity storage element. RAID extends this to account for the decrease in mean time between failure (MTBF) of the array as a whole by using a parity or mirroring scheme to provide data redundancy.

While disk arrays and RAID do provide higher performance storage for a single machine, parallel computing adds another dimension to the I/O problem. Parallel computing involves the utilization of *multiple* computers or processors to simultaneously solve a given problem. Typically these processors communicate through some network to share data during the solution process. Figure 1.1 illustrates the bottlenecks of using single disks and disk array configurations. Compute nodes (CNs) are the processors used to perform computation in a parallel system, while I/O nodes (IONs) are used for data storage. In the single disk case (Figure 1.1a) we see that all CNs must access data on a single disk, which is typically the biggest performance bottleneck. In the disk array case (Figure 1.1b) there are multiple disk resources, but all CNs must access these resources through a single host. This leads to the connection between the single host and the CNs becoming a bottleneck. To avoid these problem, some method of spreading file data throughout the collection of processors or providing multiple points of access must be devised. Parallel I/O solutions solve this problem by utilizing multiple I/O nodes, as seen in Figure 1.1c.



a) Single disk b) Disk array c) Parallel I/O

Figure 1.1: I/O Configurations

1.2 Parallel I/O

The parallel I/O concept is to direct I/O operations in a parallel machine so that multiple resources connected via multiple links are used to service them. This is achieved in a number of ways, including the use of separate I/O networks, additional I/O nodes attached to the existing network, or attachment of storage systems directly to the compute processors. One particular area of study in parallel I/O is parallel file systems, in which multiple resources are presented in a manner similar to traditional file systems. By presenting the resources this way, the semantics used to access traditional, sequential files may be maintained.

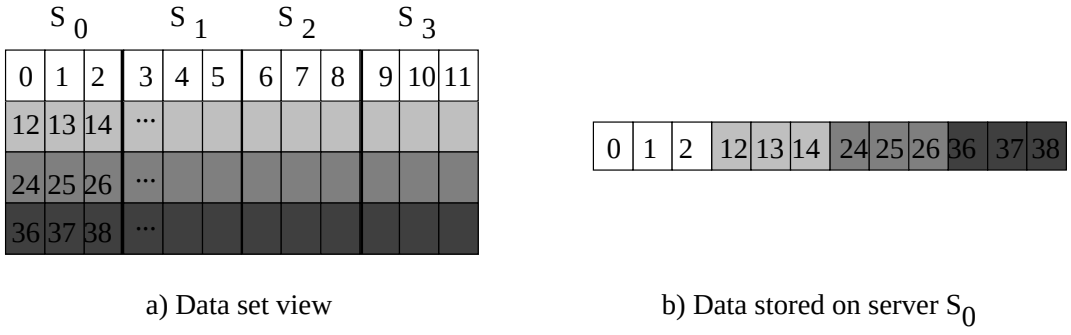
The first generation of parallel file systems were constructed for high performance commercial parallel machines such as the IBM SP-1 and the Intel Paragon. These machines were primarily designed for parallel computation and had high performance communication systems which severely outperformed the disk systems attached to the machine. As many research groups were using these machines for parallel I/O studies, the techniques for improving I/O performance were tailored to this environment.

1.3 Optimization Techniques

At least in part because of the environments in which research was taking place, early techniques tended to focus on optimizing disk performance. There are three techniques that have become commonly used and discussed. The data sieving technique [9] reduces the number of disk requests at the cost of increased network traffic, as does the two-phase technique [3]. The disk-directed I/O technique [21] reorders data transfer to force blocks to be accessed in disk-optimal order, which may have negative effects on the utilization of the network. These techniques all perform well in the tested systems in part because of the gap in performance between disk and network throughput in these systems.

More recently new parallel computing platforms have emerged, including clusters of workstations [6], Piles-of-PCs [42], and Beowulf computers [42]. Beowulfs are constructed from commodity components, which commonly includes fast ethernet and IDE disk interfaces. These components are of roughly the same order of magnitude of performance, where in commercial parallel machines the network typically has much lower latency and much higher throughput than the storage system. This, coupled with commodity software, makes it less obvious what resources, if any, will consistently outperform the other components in the system.

The emergence of new and different platforms has created a situation where the traditional parallel I/O optimizations, while useful in some contexts, are not flexible enough to be applied in all cases. What is needed is a more general approach to optimization in parallel I/O. Such an approach should recognize the components of the system and the parameters of workloads that determine performance: disks, network, and cache. It should



a) Data set view

b) Data stored on server S_0

Figure 1.2: Example Workload

provide a mechanism for determining what resources are currently limiting the performance of the system as a whole based on both system parameters and workload characteristics. Finally, using this information it should schedule I/O operations and apply an I/O technique to best utilize the available resources. This new approach could be the cornerstone of the next generation parallel I/O systems, and it could be added to existing systems in order to provide better performance over a wider range of situations.

1.4 The Optimization Problem

Now we will discuss an example workload and how one might attempt to optimize on the server side for improved performance. In our workload a two-dimensional data set is stored on a set of I/O servers in row-major order. The entire data set is written by the application at once by all tasks that make up the application simultaneously, and no other traffic is seen by the file system at this time. Figure 1.2 shows the two-dimensional data set and the distribution of blocks on a given server (stored as a one-dimensional sequence).

a) Disk-oriented optimization
Service order: 0,1,2,12,13,14,...,38

b) Network-oriented optimization
Service order: 0,12,24,36,1,13,...,38

Figure 1.3: Optimization Alternatives

One approach to optimization would be to attempt to access disk blocks sequentially. This should lead to the highest possible disk performance. Figure 1.3a shows the order in which blocks would be serviced for this optimization.

An alternative approach would be to attempt to optimize more for network traffic. One common approach here would be to service all requests simultaneously on the assumption that this will provide the network layer with the most opportunities to transfer data. With protocols such as TCP, for example, it is difficult to attain peak network utilization with only a few request streams. This approach is shown in Figure 1.3b.

Both of these approaches have advantages; if the disk is our bottleneck, then optimally accessing blocks will be a win. If, on the other hand, our network is our bottleneck, then it is more likely that the network optimization is the right choice.

While it is useful to theorize on the effects of these optimizations, in order to really know how these approaches will affect performance we must observe the actual behavior of the system. In Figures 1.4a-d we see performance measurements from a similar workload run on a real system. Two I/O servers were used to serve fourteen client tasks. The x-axis of each graph tracks the amount of data accessed on a single server. We compare performance using two of our algorithms, the default network-oriented one (Net Opt) which services

all ready requests, and the most disk-oriented algorithm (Disk Opt) which services in disk location order.

In Figure 1.4a the application service time is shown. By application service time we mean the time it takes the tasks, as a collection, to complete their operation. This means that the time shown here is the maximum time required by any one task to complete. The vertical bar indicates the amount of physical memory available for user processes on the system and serves as an upper bound for the amount of cache available at run time. We see that for large accesses, the disk optimization provides lower application service times.

Figure 1.4b focuses on small accesses. Here we see that there is a small but consistent advantage to using the network optimization.

Results are as expected for mean task service time and variance (Figures 1.4c and 1.4d respectively); for the entire range of access sizes, the disk optimization provides lower mean service times but substantially higher variance between tasks. This is because for this workload the disk optimization services one request at a time, which means that the first tasks to be serviced are finished quickly, while others wait a long time before service even starts.

We can see from our performance data that for “small” accesses the network optimization is best given these constraints, while for larger ones the disk optimization is preferred. Also as expected, the availability of cache seems to have an effect on where this transition takes place. Noting the effects of these optimizations leads us to create scheduling algorithms which try to apply these optimizations. In the following chapters we will attempt to more closely examine the correlation between resources, workload, and performance and apply reactive scheduling to automatically choose appropriate scheduling algorithms.

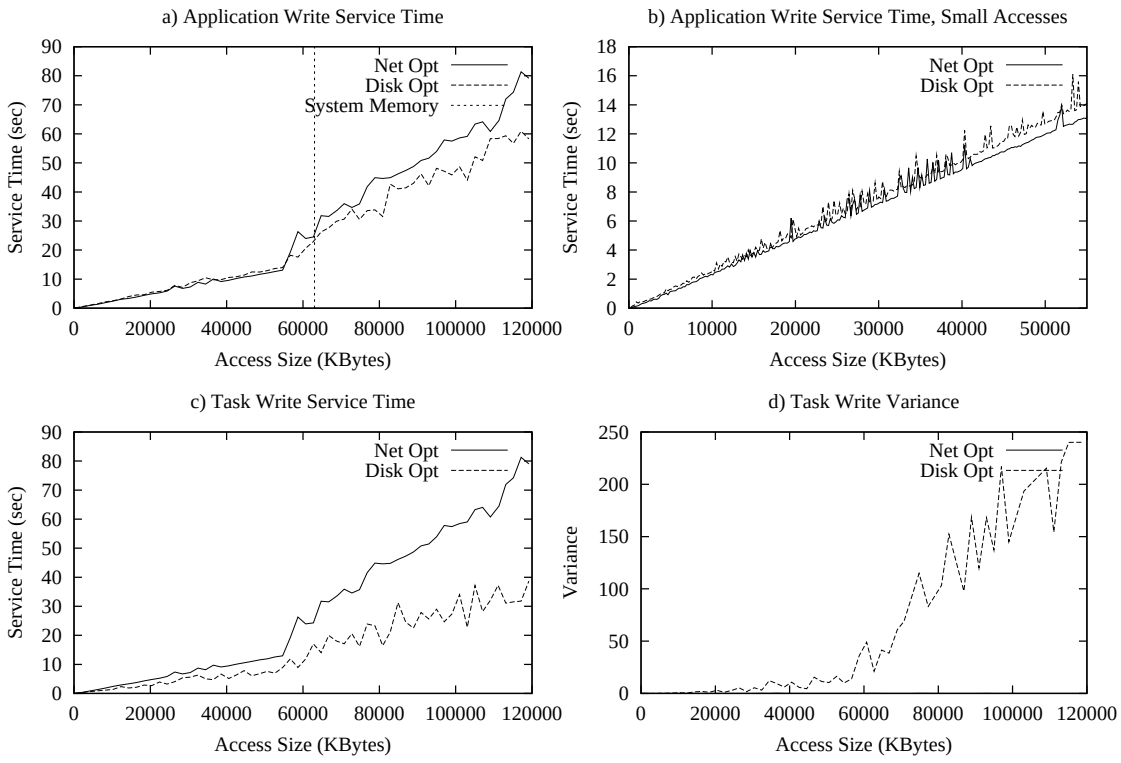


Figure 1.4: Performance Measurements

1.5 Reactive Scheduling

The *reactive scheduling* (RS) approach is one method of selecting and applying the correct optimization for a given workload. Reactive scheduling is a decentralized, server-side approach to optimization in parallel I/O. The focus is on utilizing system state and workload characteristic data available locally to servers, potentially along with application hints passed with application requests, to make decisions on how to schedule the service of application requests. This approach is resource-oriented in that it focuses on the relationship between workload and the state of the system resources as a whole in order to affect improvements to system performance. This is opposed to the more common file-oriented approach seen in most adaptive cache policy systems which apply rules by looking at behavior at the file level. This allows us to more effectively focus on how multiple accesses affect the system as a whole, and by doing so should allow RS to provide benefits in more complicated workloads.

The RS approach consists of three components: a set of scheduling algorithms, a system model, and a selection mechanism. The scheduling algorithms give us options in terms of how we service the collection of requests that the system is presented at any given time, allowing us to tailor service to optimize more for disk access, network access, or cache utilization. The system model uses system-specific known values such as network and disk bandwidth, system state such as cache availability, and information describing the workload in progress to predict how the available optimizations would perform if utilized. The selection mechanism maps available system and workload information into the parameters used by the model. It then uses the model to predict performance using each of the avail-

able optimizations and chooses the one that it predicts will perform best. Together these three components provide a means for increasing performance in parallel I/O systems by adapting the scheduling of service to best fit the workload presented to the system.

This work describes this new approach for optimization in parallel I/O. We present a model for I/O performance which accounts for disk, network, and memory resources. We validate this model on a Beowulf parallel computer by adding reactive scheduling to the Parallel Virtual File System (PVFS), a parallel file system we have designed and implemented for testing parallel I/O techniques in cluster of workstation environments. Using our model, we develop a technique for predicting scheduling algorithm performance that uses both the system parameters described above and measured workload characteristics. This technique is then used to select a scheduling algorithm tailored to the workload and system in question. The reactive scheduling approach is validated for a number of workloads, and areas of future development are discussed.

1.6 Contributions

In this work we describe the reactive scheduling technique, which allows a parallel I/O server to react to changes in workloads and system state by changing the method in which it schedules service. A number of contributions have been made as a result of this work.

The first of these is the parallel file system, PVFS, which was designed and implemented to facilitate parallel I/O research. PVFS combines conventional wisdom on application I/O patterns with a new scheduling technique (stream-based I/O, which is described later in this work) to create a working parallel file system and test bed for parallel I/O re-

search. Without this open-source, freely available parallel file system, this work would not have been possible.

The second contribution of this work is the workload study itself, which presents evidence of the correlation between workload characteristics and the performance of various scheduling algorithms. Previous work indicated that disk-directed approaches (described later in this work) were appropriate for all situations.

Finally we contribute the reactive scheduling technique, which combines system modeling and I/O scheduling into a system which increases system performance in real systems. The technique can be extended to include new scheduling approaches or more thorough models in order to further increase effectiveness.

1.7 Outline

In Chapter 2 we will discuss related work, including previous workload studies, strategies for caching and prefetching, transfer mechanisms, and other adaptive I/O systems.

Chapter 3 will describe the design and performance of PVFS. This will include a discussion of the issues involved in designing a parallel I/O system, a description of a related parallel file system, an overview of the components of the system, an outline of the interfaces available for application designers, and a study of the system performance on a 64-node PC cluster.

Chapter 4 describes the scheduling algorithms added to PVFS in order to facilitate implementing RS. We first cover how the algorithms operate, then cover performance of

the system under four workloads with each of the scheduling algorithms. This provides the data necessary to validate our system model developed in Chapter 5.

Chapter 5 presents the design of a reactive scheduling system and its validation. We first build the model component of the RS system by developing a simple algorithmic model which predicts performance using a round-robin scheduling algorithm. This model is then supplemented with an additional component to account for scheduling changes. The model is tuned to data collected in our workload study and shown to be an accurate indicator of performance. This completed, we then cover the implementation of the selection mechanism and integration of the model within the PVFS file system and the constraints placed on the system by the input data available at run time. Next we present performance data and compare our RS implementation with best-case data obtained in Chapter 4. Finally we point out system features which would facilitate a more effective implementation of RS.

Finally, Chapter 6 draws conclusions and point out future areas of study. We cover the effectiveness of our new scheduling algorithms, what we learned from our workload study, the accuracy of our system model, and the practical implications of integrating RS into a real parallel file system.

Chapter 2

Related Work

A number of research projects influenced the development of this work. In particular workload characterization studies, previous work in optimization techniques, and other attempts at adaptive I/O systems have directly impacted the work described here. These previous efforts will be covered in this chapter.

2.1 Workload Studies

Workload studies have helped researchers in parallel I/O better understand the behavior of both applications and some parallel I/O systems. This knowledge has influenced the direction of work in the field. In this section we cover three studies which are of particular importance to this work because they point out commonalities between access patterns that have lead to a greater understanding of how applications use parallel I/O systems.

2.1.1 Concurrent File System Analysis

Intel's Concurrent File System (CFS) was the subject of a number of early performance studies in parallel I/O. One study of CFS was performed on the Touchstone Delta machine [2]. Simple I/O tests were used to characterize the performance of the system. They noted that throughput to a single node was limited by how fast a node could generate requests, and small requests (4K) were adequate to obtain good performance. They found that read performance to a single node dropped off when a large number of disks (> 32) was used, indicating that limiting the number of disks accessed simultaneously by a single client is beneficial.

In [23, 39] Krystynak and Nitzberg examine the performance characteristics of CFS on the iPSC/860 with 128 compute nodes and 10 I/O nodes. A set of simple read and write tests were used to characterize the performance. They found that file pre-allocation, double buffering of I/O, and large block sizes were necessary to obtain maximum performance. In addition they found it was necessary to split compute nodes into groups of approximately 16 nodes and restrict file system access to one group at a time when reading, in order to place a limit on the number of concurrent accesses. This allowed them to maintain near-peak performance by allowing the system to better prefetch and cache data. Allowing more nodes to simultaneously access the system reduced performance.

These tests in particular are interesting because they point to the necessity of prefetching and caching for maintaining high performance in some parallel I/O systems but at the same time show that prefetching and caching, by themselves, are not adequate for maintaining performance with workloads with many clients in this environment.

2.1.2 CHARISMA Project

In [38] the results of the CHARISMA project are described. This was a project designed to characterize the behavior of production parallel workloads at the level of individual reads and writes. The authors traced workloads on two machines, a CM-5 and an iPSC/860, both with numerous scientists running applications on them. They were hoping to discover some commonalities between parallel applications on the two machines in terms of the number of files read and written, the size of the files, the typical read and write request sizes, and how these requests were spaced and ordered.

They differentiate between *sequential* and *consecutive* requests, where sequential accesses begin at a higher file offset than the point at which the previous request from the same process ended, while consecutive ones begin at exactly the point where the last request ended. Almost all write-only files were accessed sequentially, and many of the read-only files were as well. Most of the write-only files were written consecutively. This was likely because in many applications each process would write out its data to a separate file. Read-only files were accessed consecutively much less often, indicating that they were read by multiple applications. About a third of the files were accessed with a single request.

By examining the sizes of the intervals between requests, they found that most files were accessed with only one or two interval sizes. The request sizes were also very regular, with most applications using no more than three distinct request sizes. The traces showed that a simple-strided access pattern was most common. A simple-strided pattern is one where requests are separated by a regular spacing (or stride). In files accessed with this pattern there were often only a very few segments (groups of requests that made up a simple-strided

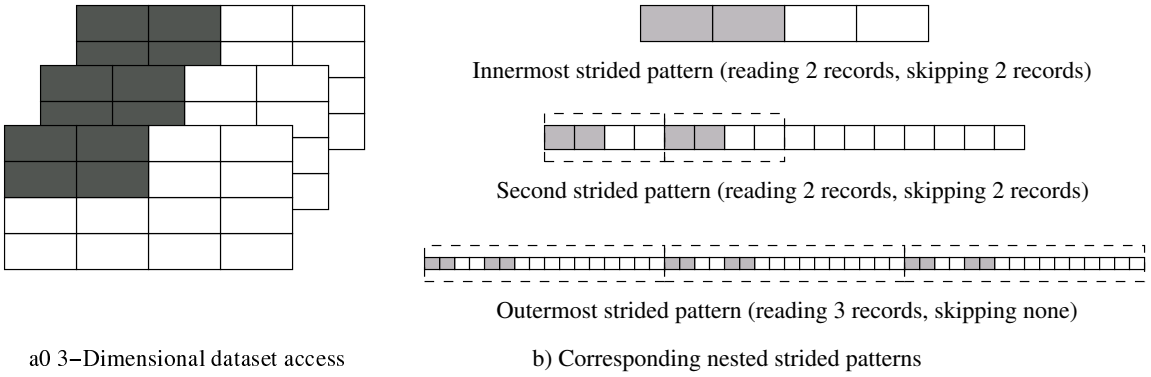


Figure 2.1: Nested-Strided Example

pattern), although in some cases there were as many as 60 to 80. These segments usually consisted of between 20 and 60 separate requests.

In addition to simple-strided patterns, nested-strided patterns were also common. A nested-strided pattern is simply the application of multiple simple-strided patterns, allowing the user to build more complex descriptions of stored data. Figure 2.1 shows an example of a nested-strided access, in this case utilizing three strided patterns in order to access a block of a 3D data set. First a simple-strided pattern is used to describe the data to retrieve from a row, in this case two of the four records. Next this pattern is used inside a second strided pattern to describe the data to retrieve from a plane, again two of the four records, where "records" in this case are rows. The regions described by the inner-most strided pattern are denoted by the dashed boxes. Finally a third strided pattern utilizes the previous two to describe the data volume to access as one unit, in this case three "records" with no skipping.

Nieuwejaar et. al. conclude that parallel I/O consists of a wide variety of request sizes, that these requests often occur in sequential but not consecutive patterns, and that there

is a great deal of interprocess spatial locality on I/O nodes. They believe that strided I/O request support from the programmer's interface down to the I/O node is important for parallel I/O systems because it can effectively increase request sizes, which lowers overhead and provides opportunities for low-level optimization. These results directly impacted the design of a number of parallel file systems and highlight the importance of non-contiguous request capability.

2.1.3 Scalable I/O Initiative Characterization Work

Another parallel I/O study was performed by Crandall *et al* on three scientific applications designed for the Paragon [14]. Selected by the Scalable I/O Initiative Applications Group, these applications were all I/O intensive and provided a variety of access patterns.

The premise of this work was to examine the patterns of access of these applications, determine what generalizations could be made about the patterns of access in high performance applications from these access patterns, and finally to discuss with the authors the reasons for their decisions regarding I/O patterns in order to determine why they chose certain approaches.

They instrumented the applications using the PABLO performance environment [41] in order to obtain information on the parameters and duration of I/O calls. They found that the three applications exhibited a variety of access patterns and requests sizes, so that there was no simple characterization that could be made. From conversations with the application programmers, they found that the I/O capabilities of the system did not match the desired ones, which resulted in complications in application code and also reduced the

scope of feasible problems. For example, in a couple of the applications the programmers found it easier to read data into a single node and then distribute to the remaining nodes instead of using the parallel access modes, because the available modes did not allow for the particular pattern of access they needed.

In addition they found that small requests were common. While application programmers can aggregate requests in the application, providing capabilities in the file system for transforming requests via prefetching and caching is preferred. This leads to the idea that file system policies should not be fixed but should adapt to the patterns of access of currently executing applications. They name two approaches for providing this adaptation: hints from the applications on future access patterns, and pattern identification in the file system via matching access characteristics to known patterns. Furthermore they point out that the small use of temporary files in high performance I/O systems leads to a change in file system priorities in general; the system should be attempting to maximize utilization of bandwidth to the I/O system instead of trying to reduce the volume of I/O to disk (which is common in interactive systems).

This work emphasizes the use of pattern matching for altering policies such as prefetching and caching. This could be used in conjunction with reactive scheduling to aid in maintaining peak performance by tuning policies in addition to scheduling to match the current workload.

2.2 Caching, Prefetching, and Writeback Strategies

While some early studies seem to indicate that double buffering is adequate for high performance parallel I/O [21], more recent studies indicate that the availability and proper use of memory for prefetching and caching is paramount for I/O performance [39, 14].

In [22] Kotz and Ellis discuss and test various caching and prefetching strategies for parallel file systems. They associate caches with files, caching logical blocks rather than physical disk blocks. They notice that so called "delayed writeback" policies, typical in multiuser systems, are tailored to a type of write traffic not usually seen in parallel I/O. They promote instead a scheme they call *WriteFull*, which writes a buffer back to disk when the number of bytes written to the buffer is equal to the size of the buffer in bytes. An aging system is used to push blocks out to disk eventually if all bytes are not written, effectively falling back to delayed writeback. This scheme is tailored to sequential access patterns, which are common in parallel workloads as seen in [38].

In their tests they use 20 processes, 20 disks, 1K blocks and buffers, and a 4 Mbyte write size. In each test a single file is accessed and a single application run. For many tests they found that caches of 40–80 blocks performed best, with 40 blocks being equivalent to double-buffering. However, they note that with one particular application, in which the disks were written to in order by all processes simultaneously, that a large cache (100+ buffers) allowed the application to achieve higher bandwidth.

They test WriteFull against three other policies:

- WriteThru – forces a disk write on every write request
- WriteBack – forces a disk write when a buffer is needed for another block

- WriteFree – forces a disk write when a buffer becomes replaceable (not in use by a processor)

They note that the choice of writeback algorithms, in addition to this choice of number of buffers per process, had a large impact on performance. WriteThru wrote back too often. WriteBack often delayed writes too long, resulting in poor performance as the disk tried to catch up later in the write stream. WriteFree is a compromise between these, making some mistakes but not delaying writes as long as WriteBack. The WriteFull policy consistently performed at or near the best performance.

This work, while somewhat limited in scope, clearly demonstrates the effects caching and writeback policies can have on parallel I/O. While they conclude that large caches are only useful for cases with "high disk contention", the case where this occurs is simply one where all the processes begin writing to the same disks at the same time. This, unfortunately, can be a common occurrence when parallel applications are sequentially accessing data sets in strided patterns or when disk performance simply does not match the rest of the system.

More importantly, policies show the advantages of appropriately scheduling I/O on disks. This led directly to the disk-directed I/O work, discussed later in Section 2.3.3. Further work in adaptive caching and writeback is covered in Section 2.4.

2.3 Transfer Mechanisms

Investigation of performance problems in I/O systems led researchers to reconsider how data should be moved between applications and I/O servers. New approaches for request-

ing data, for ordering accesses more optimally, and for better utilizing the network and I/O bandwidth were devised. Many techniques build on two basic concepts: non-contiguous requests and collective I/O. By non-contiguous requests we mean that the request mechanism is extended to allow some number of disjoint regions to be requested as a single unit; for example, a column of records in a matrix stored in row-major order might be requested via a single non-contiguous request. Collective I/O is an approach to I/O that focuses on the relationships between task I/O requests in a parallel application. Specifically collective I/O approaches combine or organize the access of groups of tasks in an application in order to attain higher performance.

In this section we describe four techniques for more optimally performing data transfer for some system architectures.

2.3.1 Data Sieving

Data sieving is a technique for efficiently accessing noncontiguous regions of data in files [9]. In this work they first define the *direct read method*, the naive approach, in which one would perform a single I/O call for each contiguous region. They note that this can result in a large number of calls and low granularity of data transfer, both of which typically result in poor performance. In the data sieving technique, a number of disjoint regions are accessed by reading a block of data containing all of the regions, then the regions of interest are extracted. They note that this has the advantage of a single I/O call, but that additional data is read from the disk and must be stored in memory during the sieving process. They

find for the system in question, an Intel Touchstone Delta, that the advantages outweigh the disadvantages for a large percentage of accesses.

This technique can in fact also benefit systems with slow networks as well in that it reduces the number of requests, for which there is often significant startup time. However, the percentage of data transferred that is desired must be high for this to pay off. A more appropriate technique for reducing the overhead of multiple requests in network bound systems would be the use of more descriptive requests, which allow the I/O server to either perform non-contiguous reads, if the capability is available, or to perform this sieving on the server side, reducing network traffic.

2.3.2 Two-Phase I/O

The *two-phase access* strategy is described in [3]. This is an attempt to avoid the performance penalties often incurred when directly mapping from the distribution of data on disks to the distribution in processor memories. In this technique, data is first read from the disks by a number of processors who then redistribute the data to the processors in the final, processor-memory distribution. The strategy was tested on the Intel Touchstone Delta using the Concurrent File System (CFS). The 512 processor Delta has a limited number of I/O nodes (32) and substantial network bandwidth between processors, which are arranged in a mesh topology.

First the *direct access* method is defined and discussed, which is essentially the use of the direct read method covered above by a number of processors at once. In this approach each processor reads data from I/O nodes directly using single requests for each contiguous

region. They test performance using row block, column block, and cyclic distributions and note that in the cases where transposition is necessary, performance of the direct access method is orders of magnitude worse than the cases where data can be accessed in large chunks.

The two-phase technique obviously breaks I/O into two stages. In the first phase, the number of processors involved is chosen to match the I/O nodes. A single processor is typically used to request all the needed data from a given disk. This reduces the number of requests hitting each disk, avoiding the poor performance seen in many direct access tests when a large number of processors hit the smaller number of I/O systems. In the second phase the processors who previously read data from the disks calculate the final destination for each block of data and perform the necessary transfers. This takes advantage of the additional bandwidth between compute processors to more quickly complete the I/O process.

In order for this technique to be of use, compute processes must communicate and organize the transfer, which means that collective I/O must be available. The authors promote an interface which allows the user to declare the data decomposition into processor memories. This decomposition can be chosen from a number of popular distributions and can be changed dynamically. A run-time system is provided to perform the necessary I/O while utilizing the underlying parallel file system (in this case CFS) to handle actual physical storage.

This technique is an excellent example of this assumption that disks are the bottleneck. Here data transfer is organized in such a way that single request streams hit each disk, so fewer, larger requests may be made. This does turn out to help with disk accesses; however,

the cost is a second transfer over the network for potentially large amounts of data. Clearly the network must significantly outperform the disks, which it did in the tested systems.

However, there are a number of ideas here that might be useful in systems where other resources might be a bottleneck. For example, by reducing the number of requests to the file system to this stream of large requests, they reduce the total number of requests and tend to create larger packets. In some systems it may be paramount that the network traffic consist of large packets for optimal performance. Additionally for some systems the number of simultaneous, independent accesses hitting an I/O server can impact performance as discussed in Section 2.1.1, so the reduction of clients actually accessing I/O servers will benefit some systems as well.

2.3.3 Disk-Directed I/O

Disk-directed I/O (DDIO) is a combination of a number of other techniques for data transfer in parallel I/O systems [21]. DDIO was developed after both the data sieving and two-phase techniques and the CHARISMA study, and it is designed assuming non-contiguous request capabilities and collective I/O are available. In addition to these constraints, a number of other requirements must also be met:

- the interface between the application interface and the I/O system must pass this information to the I/O system
- the I/O system must have direct access to the disk(s) used to store data
- the I/O system must have an accurate method of predicting optimal disk access orders

Given that all the requirements are met, the disk-directed technique uses the information passed to it about the data requested in the collective request in order to determine a list of physical blocks to retrieve from the disk that contain data. It then sorts these blocks into some optimal access ordering and uses double-buffering to transfer data to/from disk with one buffer while transferring data to/from compute nodes with the other.

DDIO has some potential advantages over two-phase I/O. First, the disk I/O and permutation steps are overlapped in time in the DDIO technique. In the two-phase technique, permutation occurs on application nodes, after all disk I/O has occurred. Second, data items move through the network only once in DDIO, but some will pass through the network twice in two-phase I/O. Finally, communication to all nodes is spread more thoroughly through the disk access process in DDIO, whereas in two-phase the whole of the network isn't used until the second, permutation phase. In some situations, mainly when network performance is on the order of disk performance, DDIO will perform better because of reduced total communication and single-phase operation. In systems where network performance severely out-paces disk performance, the two-phase technique is likely to be the better performer.

As far as contributions to a well-rounded I/O transfer method, DDIO has a number of things to offer. First, they do make use of non-contiguous requests, which can result in fewer, larger packets. Second, they promote the use of an ordering scheme for optimization on the server side. While it might not always make sense to optimize for disk access, and a static scheme such as the one used in their examples might not help in a complex system, it does make sense to have a system capable of determining the cost of transferring particular packets and ordering transfers accordingly.

A derivative of disk-directed I/O, called server-directed I/O, was proposed and implemented [48]. Part of the Panda system, this technique utilizes a high-level multidimensional data set interface, performs array chunking, and uses disk-directed techniques at the logical level. Instead of determining physical block locations, they use logical file locations to determine their optimal ordering, as file data is stored on underlying local file systems. These characteristics combined proved capable of utilizing almost full capacity of the disk subsystems in their test system for a range of array sizes and numbers of nodes.

2.3.4 Stream-based I/O

Stream-based I/O is an approach that attempts to address network bottlenecks in parallel I/O systems [25]. The stream-based I/O technique was developed as part of the Parallel Virtual File System (PVFS) project [25], which is described in detail in Chapter 3. With stream-based I/O, this concept of combining small accesses into more efficient, large ones is applied to data transfer over the network. Here data to be transferred is considered as a stream and packetized in a manner which best fits the protocol in use. This is similar to a technique known as message coalescing in interprocessor communication; however, an integral part of stream-based I/O is the removal of control messages from the stream as well. This further reduces the total amount of data to be transferred and is accomplished by calculating the data ordering on each side of the stream beforehand.

This is strictly a technique for optimizing network traffic. When coupled with a server that focuses on the network (almost "network directed I/O"), peak performance can be

maintained for a variety of workloads, particularly when network performance lags behind disk performance or when most data on I/O servers is cached.

2.4 Adaptive I/O Systems

Perhaps the previous work that most closely matches the work presented here is the work performed by the Pablo Scalable Performance Tools group at University of Illinois at Urbana-Champaign. This group has developed a number of techniques for analyzing workloads and choosing policies based on this analysis. This technique has been integrated into their parallel file system as well. Both their selection system and the file system will be covered here.

2.4.1 Adaptive Policy Selection

In [30] they focus on an artificial neural network (ANN) approach to classification of I/O access patterns in parallel systems. In [31] they improve on this approach by using hidden Markov models (HMMs) to perform classification, which provides more thorough control over policies than the neural network. This system is added to the Portable Parallel File System (PPFS) [19] in order to test the performance benefits in a real system.

In all of these works they concentrate on the patterns of file access and how the caching and prefetching policies might be tuned to optimize. This is used throughout their I/O system, at the server and client levels. In [28] they continue to discuss this approach of tuning prefetching and caching. They add a component for *performance-based steering*,

which uses a number of "performance sensors" monitoring such things as mean disk service time and cache hits to help further optimize prefetching and caching.

This system could be used in addition to reactive scheduling to help maximize the benefits of caching and prefetching. Additionally, their methods of classification might also eventually be applied to the problem of reactive scheduling to improve our model. The promotion of performance-based steering in [28] further supports the reactive scheduling concept by showing another example of dynamic tuning in action.

In [29] the authors discuss the use of adaptive prefetching as an alternative to collective I/O. They compare their performance to that of disk directed I/O for a number of workloads and make the point that prefetching can be performed at the user level, as opposed to disk directed I/O, which must be part of the parallel I/O system proper. This makes implementation of prefetching easier, especially on commercial machines where I/O system modifications are not feasible.

2.4.2 Portable Parallel File System II

In [50] the implementation of the Portable Parallel File System II (PPFS II) is described. This implementation includes many of the advances the Pablo group has made in adaptive cache policy selection and also in adaptive disk striping.

The PPFS II system includes a set of new managers which handle reception of input data on the state of the distributed machine, make policy decisions based on this data, and direct the remaining components of the system to implement the desired policies. In addition to an implementation of adaptive caching, they also include a system for adaptive

disk striping. This scheme alters striping based on previous work indicating that optimal striping units are dependent on resource contention [7, 46].

While the work performed by this group shares many characteristics with the work presented here, there are many significant differences. First, the approach used in PPFS II is a centralized one, where sensor data is gathered at one point and used to make global decisions. Our system, on the other hand, will rely only on local data for decision making and will thus be a decentralized system.

Second, the PPFS II system uses input strictly for caching and striping decisions, while our system will modify ordering of network and disk transfer. This provides potentially more direct control over performance, but will possibly require more effort to implement.

Third, the PPFS II algorithms concentrate on data on a by-file basis and attempts to identify specific patterns of access, while the reactive scheduling approach is resource oriented and focuses on general characteristics of the workload. Thus we are in some sense more coarse-grain, but we do not rely on a fixed set of patterns.

In fact *both* of these techniques could be used cooperatively in a single I/O system. The Pablo group's policy selection system could be used to more effectively cache while the reactive scheduling system works to optimize data transfer. This potential will not be investigated here but will instead be left as future work.

Chapter 3

Parallel Virtual File System

Cluster computing has emerged as a mainstream method for parallel computing in many application domains, with Linux leading the pack as the most popular operating system for clusters. As researchers continue to push the limits of the capabilities of clusters, new hardware and software have been developed to meet cluster computing's needs. In particular, hardware and software for message passing have matured a great deal since the early days of Linux cluster computing; in many cases, cluster networks rival the networks of commercial parallel machines. These advances have broadened the range of problems that can be effectively solved on clusters.

One area in which commercial machines still maintain great advantage, however, is that of parallel file systems. A production-quality high-performance parallel file system has not been available for Linux clusters, and without such a file system, Linux clusters cannot be used for large I/O-intensive parallel applications. To fill this need, we have developed a parallel file system for Linux clusters called the Parallel Virtual File System (PVFS). PVFS is being used by a number of groups, including ones at Argonne National Laboratory and

the NASA Goddard Space Flight Center. Other researchers are using the PVFS system as a research tool [52].

We designed PVFS with the following goals in mind:

- It must provide high bandwidth for concurrent read/write operations from multiple processes or threads to a common file.
- It must support multiple APIs: a native PVFS API, the UNIX/POSIX I/O API [20], as well as other APIs such as MPI-IO [17, 32].
- Common UNIX shell commands, such as `ls`, `cp`, and `rm`, must work with PVFS files.
- Applications developed with the UNIX I/O API must be able to access PVFS files without recompiling.
- It must be robust and scalable.
- It must be easy for *someone else* to install and use.

In addition, we were and are firmly committed to distributing the software as open source.

In this chapter we describe how we designed and implemented PVFS to meet the above goals. We also present some performance results on a Linux cluster at NASA Goddard Space Flight Center; a detailed performance study is in progress and will be reported in a future paper. We outline both the strengths and the weaknesses of the system and discuss areas for future improvement.

The rest of this chapter is organized as follows. In the next section we discuss related work in the area of parallel file systems, focusing first on basic approaches to implementing

software support for parallel I/O and then concentrating on another parallel file system, Galley. In Section 3.2 we describe the design of PVFS. Performance results are presented in Section 3.3. In Section 3.4 we discuss the lessons we learned from our experiences with this design and point out areas for improvement in future releases.

3.1 Related Work

There are three basic approaches to providing the software support for parallel I/O:

- user libraries
- user level file systems
- kernel-supported file systems

The architecture chosen for this underlying software constrains the capabilities of the software to handle complex I/O requests and take best advantage of system resources. At the same time, software complexity increases as one attempts to more directly control disk, network, and cache. In the following subsections we will cover these three general approaches to parallel I/O software and discuss the implications of each.

3.1.1 User Libraries

User libraries are a fairly straightforward and simple method for providing parallel I/O for applications. A library of calls is created that a parallel application can use to perform I/O, usually to disks local to each of the application tasks. These libraries can be made portable and are easy to implement and modify, making them excellent tools for research.

However, they require that any application wishing to access the data stored by them link to the library, which makes common file system tools useless for accessing these parallel files. They typically only support one interface, which is often tailored to a single class of application, and can suffer from poor performance due to the use of generic message passing interfaces and the storage of data on local node file systems.

In general, libraries for parallel I/O are best used in conjunction with a set of applications which all access data sets in a similar manner and are very latency tolerant. This type of support is often used by groups that are developing compilers that support parallel I/O and need library functions to perform the I/O for the application. The user-level paging mechanism used in [44] is a good example of a library implementation. Here a paging mechanism is designed that allows data pages from remote processes to be accessed. A simple polling mechanism is built into the library and handles requests for pages from other application processes without the need for additional I/O processes or kernel support. Disks local to each application process are used for data storage.

The Jovian library [1] is something of a hybrid implementation, with a library of routines used by application processes to perform I/O through a separate set of processes, known as coalescing processes, which exist for the duration of application execution in order to handle I/O. These coalescing processes (CPs) receive requests from application processes, determine which blocks need to be read from data files, and sort and combine these blocks to more optimally access the disk local to each CP. The use of coalescing processes has the additional advantage of allowing disks located on nodes not involved in computation to be accessed. Furthermore by executing coalescing processes only during the lifetime of the application, the design of the coalescing process is greatly simplified.

3.1.2 User Level File Systems

User level file systems are a more complicated method for implementing parallel I/O. These involve creating a set of processes which will handle the tasks of accessing file data and performing permission checks. These processes, or a subset of them, are constantly available to handle requests from any application in the system. These requests are often made by applications using a generic interface, as opposed to the domain-specific ones we discussed in the previous section.

There are a number of advantages to this method. First, the use of a low level interface for communicating with the file system processes allows for a number of application level interfaces to be developed which can all interact with the file system while reusing the underlying file system code. Second, the flexible interface can also be used to allow common file utilities to interact with the file system without rewriting the utilities. Third, the use of processes to store file data allows for additional optimization of file storage and data transfer methods. These processes also allow nodes to be used for I/O when they are not being used as compute nodes, which is not the case with user libraries.

At the same time, this type of system is less portable than a user library as often times the processes or data transfer mechanisms are optimized for the target platform. Development and implementation of such a system takes more time as the I/O process code must be developed in addition to the user interfaces.

The Portable Parallel File System (PPFS) is a user-level file system designed for rapid experimentation on a variety of platforms[19, 50]. It implements various caching, prefetching, data placement/striping, and data sharing policies. So-called caching agents are used

to provide shared caches between application tasks and data servers which handle I/O to underlying UNIX file systems. MPI or NXLIB is used to pass messages between processes.

PIOUS is a parallel file system developed at Emory University for use on clusters of workstations [34, 33]. PIOUS consists of PIOUS data servers (PDSs), a PIOUS service coordinator (PSC), and a library of functions to access the file system. The PSC is involved in major system events, such as opening and closing files, but not in general file accesses. PIOUS supports what they call parafiles, which are files with multiple subfiles (of zero or more bytes). There are three modes for accessing these files:

- global — the file is seen as a linear sequence of bytes with a shared file pointer
- independent — the file is seen as a linear sequence of bytes with private file pointers
- segmented — each process sees a separate segment.

In the global and independent modes the subfiles are mapped into a single linear address space using a fixed stride and a round-robin ordering.

PIOUS uses a transaction-based approach to I/O, so that writes can be guaranteed to either completely succeed or completely fail. Logging is used to guarantee this in the case of system failure, but this logging can be turned off via the "volatile" option. This is mainly for fault tolerance.

The Vesta File System has also been implemented as a user-level file system. In their implementation for the IBM SP-1 dedicated I/O nodes run Vesta server code and store data on a local disk partition using the AIX Journaled File System as the underlying file system. Preliminary measurements indicate that they are suffering only a small ($<$ 20 percent)

penalty over implementing their own device drivers, but obviously this number will vary depending on access patterns [11, 12].

3.1.3 Kernel-Supported File Systems

The last and most complex implementation is one which is built into the kernel of the machine or is given direct access to devices in some similar manner such as the *sfs* file system for the CM-5 [27]. There are a number of advantages to this implementation choice. Usually implementation in the kernel allows existing programs to interact with the new file system without any modification or recompilation. Access to kernel data structures and devices avoids extra data copies that can affect performance, and file system characteristics such as caching and prefetching can be directly controlled as needed. This said, little work has focused on the performance benefits of in-kernel support.

However, modifying a kernel to add a new file system, especially one which must communicate with other kernels to provide parallel I/O, is not a trivial task. Often one must have the source to the kernel in order to modify it, and debugging can be more troublesome than with user level code. The resulting code is very kernel-dependent as well, making porting to other operating systems difficult.

3.1.4 Galley Parallel File System

The Galley parallel file system was developed at Dartmouth University and designed to operate on supercomputers and clusters of workstations [36]. The authors assume that some nodes will be used for I/O and others for computation, but not both; however, nothing in the

design precludes overlap of I/O and compute nodes. The focus of Galley's implementation is on low-level support for disk-directed I/O [21]. That is, it allows the servers to specify the order in which data will be transferred in order to optimize the traffic to the disk.

In this section we will discuss the file format of Galley, the I/O system design, and performance of the system.

Instead of the declustering and striping common with parallel file systems, Galley allows for files to be created with what they call *subfiles*, with a maximum of one subfile per I/O node. The number and location of these subfiles is fixed at file creation. Each of these subfiles consists of one or more independent, named forks, which are linear sequences of bytes. Forks can be added or removed at any time. There is no explicit connection between subfiles other than the name, and there is no requirement that subfiles have the same forks.

Once all subfiles are created, a higher level library can be used to handle declustering into them in any manner desired. For example, a single subfile might be used on each I/O node to hold stripes of file data. This system of file storage is extremely flexible, but it places a much larger burden on application interfaces and the underlying client-side implementations to perform mapping into the subfiles.

Galley is multi-threaded. The threads perform a number of distinct tasks:

- cache manager thread
- disk manager thread
- name space manager thread
- incoming connection thread

- service threads

The cache manager handles cache allocation; a least recently used (LRU) policy is used for replacement. The disk manager thread handles local disk I/O. No prefetching is done by the system, and support is included for storing data on existing file systems and raw devices. The name space manager handles metadata operations. Metadata for files is stored on the I/O processors and a hashing function is used to locate data for a specific file. A service thread is started for each compute process, who use TCP/IP for moving data between themselves and clients.

Their interface supports strided and batch accesses. There is no collective I/O support; presumably this would be implemented at a higher level. All application tasks separately open and close files.

In [37] the performance of the system is tested using a simulated disk, giving them the level of control over the disk and file system policies that would only be available inside the kernel. In two separate situations they run into problems with regards to network utilization and eventually note that they need a better algorithm to handle network flow control to account for these situations.

The Galley system approaches the problem of providing parallel I/O support in clusters from a very different angle than PVFS. The two systems differ in the way they view data local to I/O processors, the manner in which metadata is stored, the degree to which they rely on client-side libraries, and the resource for which they optimize. In the next section we will detail the PVFS design, including how it approaches these issues.

3.2 PVFS Design

As a parallel file system, the primary goal of PVFS is to provide high-speed access to file data for parallel applications. In addition, PVFS provides a cluster-wide consistent name space, enables user-controlled striping of data across disks on different I/O nodes, and allows existing binaries to operate on PVFS files without the need for recompiling.

PVFS is a client-server file system with multiple servers, called *I/O daemons*. I/O daemons typically run on separate nodes in the cluster, called *I/O nodes*, which have disks attached to them. Each PVFS file is striped across the disks on the I/O nodes. Application processes interact with PVFS via a *client library*. PVFS also has a *manager daemon* that handles only metadata operations such as permission checking for file creation, open, close, and remove operations. The manager does not participate in read/write operations; the client library together with the I/O daemons handle all file I/O without the intervention of the manager.

It is not necessary that the clients, I/O daemons, and the manager be run on different machines. Running them on different machines may result in higher performance, however, because it alleviates competition between compute and I/O processes for network, CPU, and memory resources.

PVFS is implemented as a user-level implementation; no kernel modifications are necessary to install or operate the file system.[1] PVFS uses TCP/IP (sockets) for all internal communication. As a result it is not dependent on any particular message-passing library.

[1] A PVFS kernel module is under development.

3.2.1 PVFS Manager and Metadata

A single manager daemon is responsible for the storage of and access to all the metadata in the PVFS file system. Metadata, in the context of a file system, refers to information describing the characteristics of a file, such as permissions, the owner and group, and, most importantly, a description of the physical distribution of the file data. In the case of a parallel file system, the distribution information must include both file locations on disk and disk locations in the cluster. Unlike a traditional file system, where metadata and file data are all stored on the raw blocks of a single device, parallel file systems must distribute this data among many physical devices. In PVFS, for simplicity, we chose to store both file data and metadata in files on existing local file systems rather than directly on raw devices.

PVFS files are striped across a set of I/O nodes in order to facilitate parallel access. The specifics of a given file distribution are described with three metadata parameters: base I/O node number, number of I/O nodes, and stripe size. These parameters, together with an ordering of the I/O nodes for the file system, allow the file distribution to be completely specified.

An example of some of the metadata fields for a file `/pvfs/foo` is given in Table 3.1. The *pcount* field specifies that the data is spread across three I/O nodes, *base* specifies that the first (or base) I/O node is node 2, and *ssize* specifies that the stripe size—the unit by which the file is divided among the I/O nodes—is 64 Kbytes. The user can set these parameters when the file is created, or, if not specified, PVFS will use a default set of values.

Table 3.1: Metadata Example

File `/pvfs/foo`

inode	1092157504
⋮	⋮
base	2
pcount	3
ssize	65536

Application processes communicate directly with the PVFS manager (via TCP/IP) when performing operations such as opening, creating, closing, and removing files. When an application opens a file, the manager returns to the application the locations of the I/O nodes on which the file data is stored. This information allows applications to communicate directly with I/O nodes when file data is accessed. In other words, the manager is not contacted during read/write operations.

One issue that we have wrestled with throughout the development of PVFS is how to present a directory hierarchy of PVFS files to application processes. At first we did not implement directory-access functions and instead simply used NFS [51] to export the metadata directory to nodes on which applications would run.[2] This provided a global name space across all nodes, and applications could change directories and access files within this name space. This method had some drawbacks, however. First, it forced system administrators to mount the NFS file system across all nodes in the cluster. As PVFS emerged as a candidate file system for large clusters (on the order of 1,000 nodes), this

[2]Note that NFS was used to store only metadata, not the actual file data. The actual data was striped across the local disks on the I/O nodes.

restriction was problematic because of limitations with NFS scaling. Second, the default caching of NFS caused problems with certain metadata operations.

These drawbacks forced us to reexamine our implementation strategy and eliminate the dependence on NFS for metadata storage. We have done so in the latest version of PVFS, and, as a result, NFS is no longer a requirement. Specifically, this dependence was removed by additionally trapping system calls related to directory access. A mapping routine determines if a PVFS directory is being accessed, and if so, the operations are redirected to the PVFS manager. This trapping mechanism, which is used extensively in the PVFS client library, is described in Section 3.2.4.

3.2.2 I/O Daemons and Data Storage

An ordered set of I/O daemons run on the I/O nodes in the cluster. The I/O nodes are specified by the user when the file system is installed. These daemons are responsible for using the local disks on each I/O node for storing data for PVFS files. These I/O daemons utilize a local file system to store data. For each PVFS file handled by the daemon, a local file is created on an existing local file system. These files are accessed using standard unix `read()`, `write()`, and `mmap()` operations. This means that all data transfer occurs through the kernel block and page caches and is scheduled by the kernel I/O subsystem.

Figure 3.1 shows how the example file `/pvfs/foo` is distributed in PVFS based on the metadata in Table 3.1. Note that although there are six I/O nodes in this example, the file is striped across only three I/O nodes, starting from node 2, because the metadata file specifies such a distribution. Each I/O daemon stores its portion of the PVFS file in a file

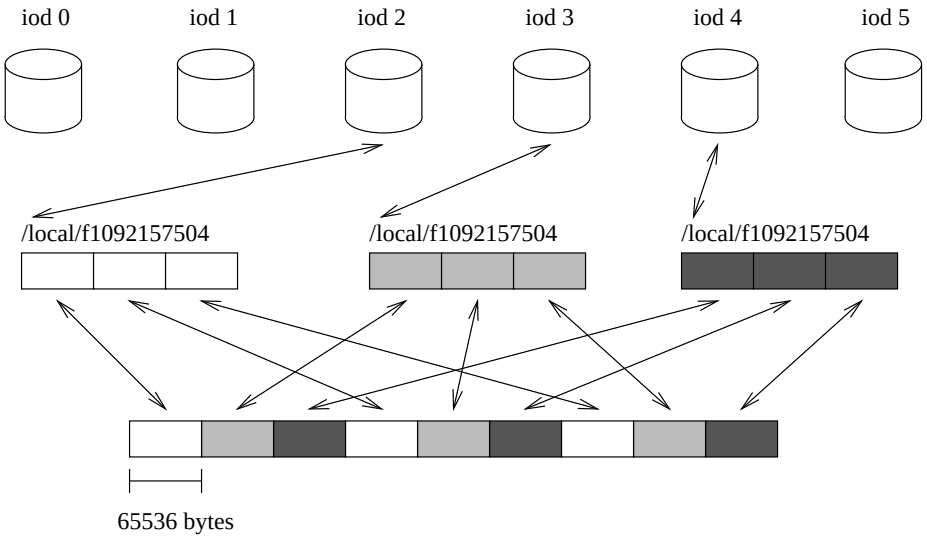


Figure 3.1: File Striping Example

on the local file system on the I/O node. The name of this file is based on the inode number that the manager assigned to the PVFS file (in our example, 1092157504).

As mentioned above, when application tasks (clients) open a PVFS file, the PVFS manager informs them of the locations of the I/O daemons. The clients then establish connections with the I/O daemons directly. These connections are used strictly for data requests. When a client wishes to access file data, the client library sends a descriptor of the file region being accessed to the I/O daemons holding data in the region. The daemons determine what portions of the requested region they have locally and perform the necessary data transfers using TCP/IP.

Figure 3.2 shows an example of how one of these regions, in this case a regularly strided logical partition, might be mapped to the data available on a single I/O node. (Logical partitions are discussed further in Section 3.2.3.) The intersection of the two regions defines what we call an *I/O stream*. This stream of data is then transferred in logical file order

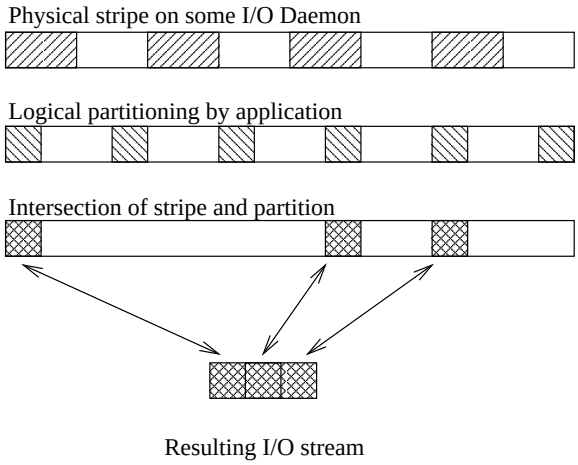


Figure 3.2: I/O Stream Example

across the network connection. By retaining the ordering implicit in the request and allowing the underlying stream protocol to handle packetization, no additional overhead is incurred with control messages at the application layer.

Figure 3.3 shows in greater detail what happens when a client accesses data from PVFS I/O daemons. In black we see that data that was requested, which corresponds to a portion of a logical partition as described in Figure 3.2 (partitioning is covered in more detail in Section 3.2.3). In the two shades of gray we see the portions of this data which are stored on the two I/O nodes across which this file is striped.

When this region is accessed, the I/O daemons each send back an I/O stream containing the requested data that they possess. These two streams are then merged into a final, contiguous data buffer on the client.

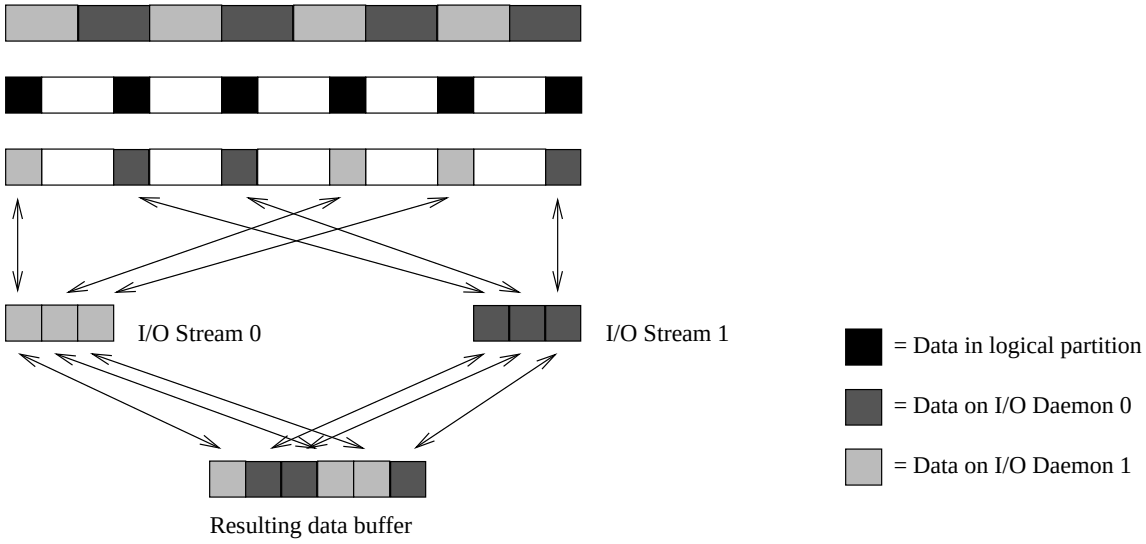


Figure 3.3: File Access Example

3.2.3 Application Programming Interfaces (APIs)

PVFS can be used with multiple APIs: a native API, the UNIX/POSIX API [20], MPI-IO [17, 32], and an array I/O interface called the Multi-Dimensional Block Interface (MDBI).

The native API for PVFS was heavily influenced by two research projects in progress at the time, Charisma [38] and Vesta [10]. The Charisma project focused on characterizing the I/O workloads on parallel machines. One of their observations was that a large fraction of I/O accesses occurred in what they called "simple strided" patterns. These patterns are characterized by fixed-size accesses that are spread apart by a fixed distance in the file. The Vesta parallel file system had an interesting API that allowed processes to partition a file logically such that a process could only "see" a subset of the file data.

We combined these two concepts into a "Partitioned File Interface," which forms the basic API for PVFS. This API provides the same functionality as the regular UNIX file I/O functions, but also provides a partitioning mechanism. With this API, applications can

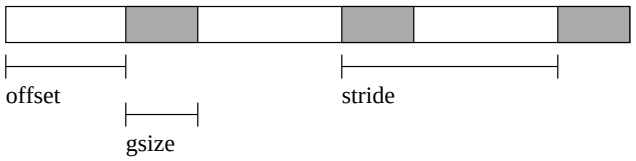


Figure 3.4: Partitioning Parameters

specify partitions on files, which allow them to access simple strided regions of the file with single read/write calls, thereby reducing the number of I/O calls for many common applications. This idea is similar in some ways to the "file view" concept in MPI-IO [17, 32], although the MPI-IO technique is considerably more flexible.

Partitioning allows for noncontiguous file regions to be accessed with a single function call. A special `ioctl` call is used to set a partition. Three important parameters are involved in partitioning a file: *offset*, *gsize*, and *stride*, as shown in Figure 3.4. The *offset* parameter defines how far into the file the partition begins relative to the first byte of the file. The *gsize* parameter defines the size of the simple strided regions of data to be accessed, and the *stride* parameter defines the distance between the start of two consecutive regions.

It should be noted that there is a significant difference between the semantics of strided access in PVFS as compared to Galley. In Galley the I/O servers do not have information on the overall ordering of file data, so strided requests received by them are serviced relative to the data available on that node only. In contrast to this, PVFS strided accesses are relative to the file as a whole. This seems to better fit the access patterns seen in previous studies, which were looking at files in their entirety [38] and is also easier to apply in practice.

As parallel programming and parallel I/O has matured, it has become obvious that certain applications could benefit from interfaces tailored to their native views of file data.

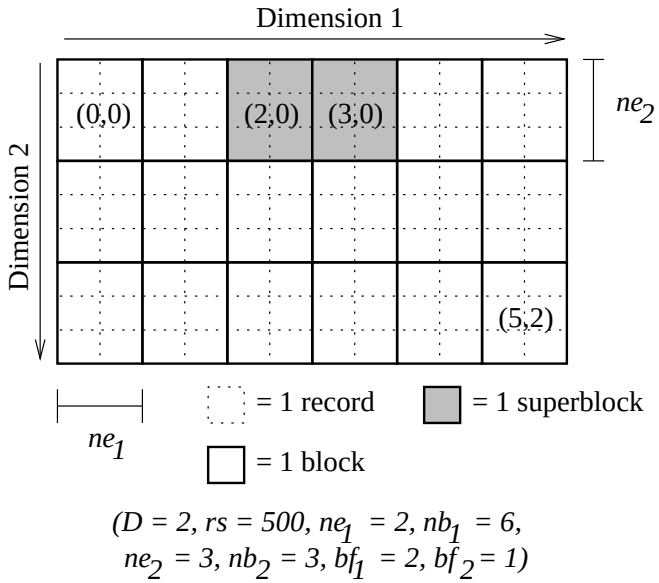


Figure 3.5: Specifying the Blocking of a 2-D Matrix

One popular application-specific interface is the multidimensional array or block interface [5, 11, 49]. These interfaces are tailored for operation on data sets that consist of records stored in some array format.

The MDBI interface is a library of calls designed to help in the development of out of core (OOC) algorithms operating on multi-dimensional datasets by making it easier to manage the movement of data in and out of core. It allows the programmer to describe an open file as an N-dimensional matrix of elements of a specified size, partition this matrix into a set of blocks, then read or write blocks by specifying their indices in the correct number of dimensions. In addition, it supports buffering and read-ahead of blocks via the definition of *superblocks* (a poor choice of terms in retrospect). The programmer specifies superblocks by giving their dimension in terms of the previously defined blocks of the file. Any time a block is accessed all other blocks in the superblock are read and held in a transparent user-space buffer on the compute node.

A set of parameters is first passed to the library to describe the logical layout and superblocks of the file:

- the number of dimensions, D,
- the size of a record, rs,
- D (block size, block count) pairs, giving the number of elements ne_i in a block and the number of blocks nb_i in the file $\forall i : 1 \leq i \leq D$, and
- D blocking factors $bf_{1..D}$, defining the size of superblocks in each dimension.

For example, Figure 3.5 shows a file described to the library as a two dimensional matrix containing a 12×9 array of 500 byte records stored in row major order. This matrix is grouped into a 6×3 array of blocks, each of which is a 2×3 array of records.

Once the file has been described to the library, accesses to blocks can be specified simply by giving the coordinates. Blocks that are read are placed into a multidimensional array of records on the compute node of size $ne_1 \times ne_2 \times \cdots \times ne_D$. In addition blocks residing in the same superblock are also placed into a user-space buffer. For example, if block (2,0) were accessed in the matrix in Figure 3.5, block (3,0) would be read into the user-space buffer for the process as well, as it resides in the same superblock as defined by the blocking parameters.

The application library uses the coordinates of the request, the blocking values, the buffer factors, and the partitioning mechanism supported by PVFS to minimize the number of requests to the file system. In the two dimension case, accesses are converted into a strided access. This allows the entire block or superblock to be read with a single request.

When the data is defined to be of more dimensions, multiple requests, a batch request, or a nested-strided request mechanism must be used.

The MPI-IO interface has also been implemented on top of PVFS. For this purpose, the ROMIO implementation of MPI-IO developed at Argonne National Laboratory was used [43]. ROMIO is designed to be ported easily to new file systems by implementing only a small set of basic I/O functions called ADIO, an abstract-device interface for I/O [54, 55], on the new file system. This feature was key to porting ROMIO to PVFS.

PVFS also supports the regular UNIX I/O functions, such as `read()` and `write()`, and common UNIX shell commands, such as `ls`, `cp`, and `rm`. Furthermore, existing binaries that use the UNIX API can operate on PVFS files without recompiling. The following section describes how this feature is implemented.

3.2.4 Trapping UNIX I/O Calls

System calls are low-level methods that applications can use for interacting with the kernel (for example, for disk and network I/O). These calls are typically made by calling wrapper functions implemented in the standard C library, which handle the details of passing parameters to the kernel. Trapping calls to UNIX system calls has been used in previous works to allow users to use new capabilities in their applications without modifying their code. A straightforward way to trap system calls is to provide a separate library to which users relink their code. This approach is used in the Condor system to help provide checkpointing in applications [26]. This method, however, requires relinking of each application that will use the new facilities.

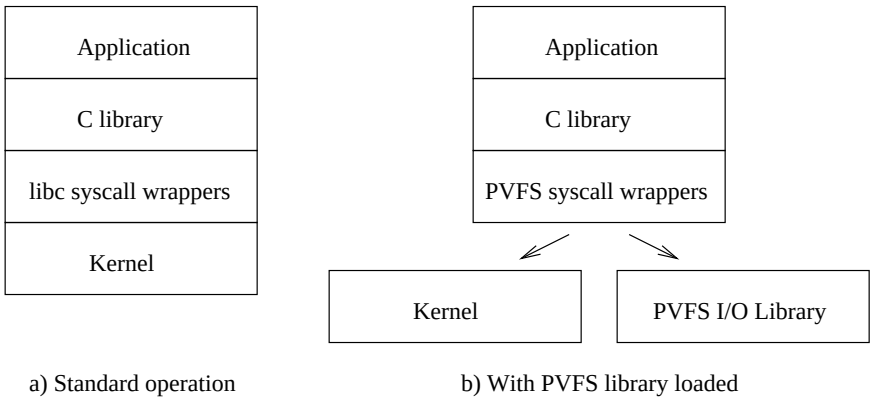


Figure 3.6: Trapping System Calls

When compiling applications, a common practice is to use dynamic linking in order to reduce the size of the executable and to use shared libraries of common functions. A side effect of this type of linking is that the executables can then take advantage of new libraries supporting the same functions without recompilation or relinking. We use this method of linking in the PVFS client library to trap I/O system calls before they are passed to the kernel.

We take advantage of the Linux environment variable `LD_PRELOAD` to allow existing binaries to operate on PVFS files. We provide a library of system-call wrappers that are loaded *before* the standard C library by using this environment variable. This method allows us to catch I/O calls before they pass on to the operating system (without relinking or recompiling the application).

Figure 3.6a shows the organization of the system-call mechanism before our library is loaded. Applications call functions in the C library (`libc`), which in turn call the system calls through wrapper functions implemented in `libc`. These calls pass the appropriate values through to the kernel which then performs the desired operations.

Figure 3.6b shows the organization of the system-call mechanism again, this time with the PVFS client library in place. In this case the `libc` system-call wrappers are replaced by PVFS wrappers that determine the type of file on which the operation is to be performed. If the file is a PVFS file, the PVFS I/O library is used to handle the function. Otherwise the parameters are passed on to the actual kernel call.

This method of trapping UNIX I/O calls has limitations, however. First, a call to `exec()` will destroy the state that we save in user space, and the new process will therefore not be able to use file descriptors that referred to open PVFS files before the `exec()` was called. Second, porting this feature to new architectures and operating systems is nontrivial. The appropriate system library calls must be identified and included in our library. This process must also be repeated when system libraries change. For example, the GNU C library for Linux (`glibc`) is constantly changing, and, as a result, we have had to constantly change our code!

3.2.5 PVFS Request Servicing

The PVFS request processing and service mechanisms are the building blocks on which we will build our RS implementation. In this section we provide a short overview of the PVFS I/O implementation. First we cover how PVFS receives requests, how these are processed, and the structure in which they are stored for service. Next we will discuss how the system handles multiple simultaneous requests. We will concentrate on read operations in this discussion, and the actual system calls used by the I/O servers (iods) to perform local file system and socket accesses are described specifically.

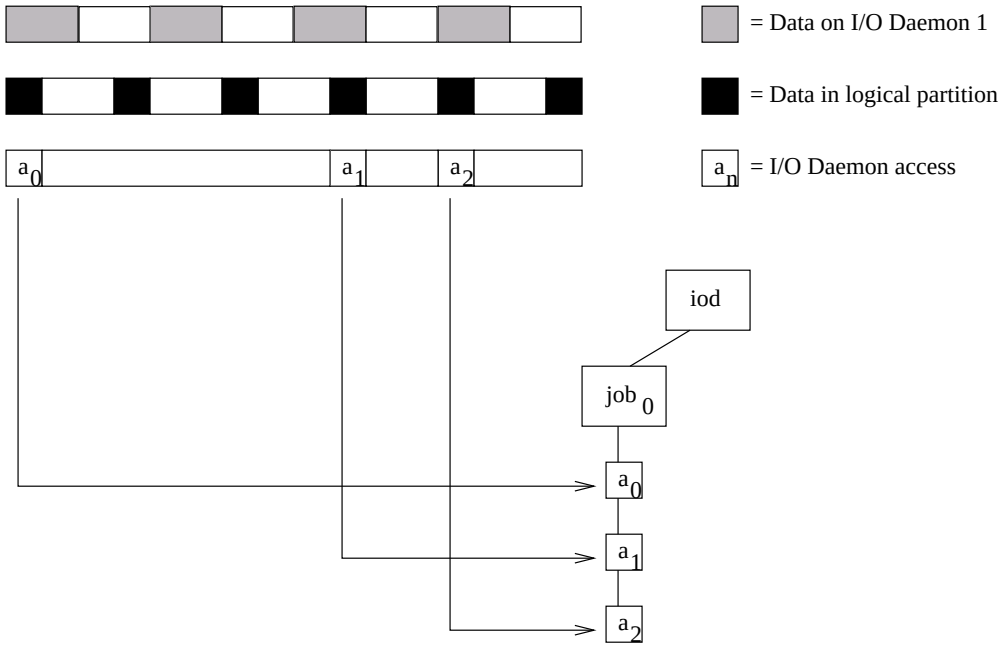


Figure 3.7: Creating Accesses from a Request

Receiving and Queuing Requests

Since all PVFS I/O is currently performed over TCP, all PVFS communication is through the UNIX sockets interface. PVFS I/O servers maintain a set of open sockets that are checked for activity in a loop. One of these sockets is an “accept” socket which is used by clients to establish connections for service. The other two possible states for an open socket are that it is connected but has no outstanding request, or that it is in active use for servicing a request.

PVFS I/O servers are single-threaded entities which rely on the `select()` call to identify connections which are ready for service. One of the sockets on which the server selects is the accept socket. When this socket (or any other open socket not involved in a request) is ready for reading, the I/O server attempts to receive an I/O request. Once the

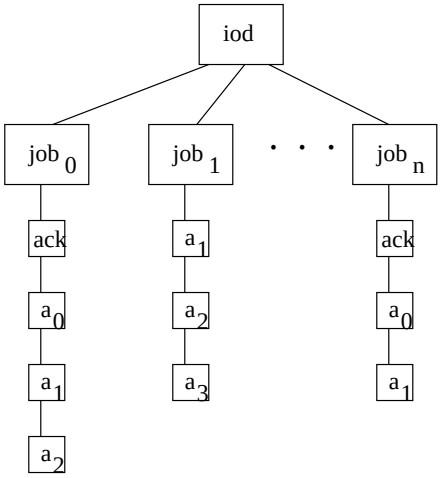


Figure 3.8: Jobs in Service

request is received it is parsed, and if the request requires data transfer a *job* is created to perform the necessary I/O. Figure 3.7 shows the job data structure as it is being created to service a request. The job is associated with a socket and file, and attached to the job is a list of *accesses*, which are data transfers which must be performed to service the request. A typical job may have 10's or 100's of accesses.

First the I/O server allocates the job structure and breaks the request into contiguous accesses based on the intersection of the requested data and the data available locally on the server. This is the process diagrammed in Figure 3.7. Following this an acknowledgment is prepended to the access list for passing status information back to the client. This job is then added to the collection of jobs which the I/O server is processing.

Figure 3.8 shows multiple jobs in service simultaneously. As the I/O server processes these jobs, accesses associated with the job are updated and removed when completed. In this example, job 1 has already had its acknowledgment sent, so it is no longer present on the access list.

Servicing Requests

Typically each task in a parallel application will create a job on each I/O server when performing an I/O operation. It is easy to imagine then that for a large parallel application a large number of jobs might be in service on an I/O server at one time. This large number of jobs, for which there are by definition no inter-job dependencies, provide us with an opportunity to optimize by selecting what jobs will be serviced and in what order. The server can perform all or part of any number of accesses for the selected job before selecting another job to service. Thus a number of jobs can be processed concurrently. Usually we do not want the server to block waiting for any single job to complete when other jobs could be serviced, thus, the server normally performs only the operations for a selected job that will not cause the server to block. This selection of jobs to service is the level at which we will apply all of our scheduling optimization in this work.

I/O servers, when not idle, sit in a loop servicing requests:

```
while (job list not empty) {
   select next ready job to service
   complete accesses from job until we would block
}
```

Selection of a job can be performed by any means we wish, including examining the characteristics of the job such as size or next access type and position. This gives us the flexibility to implement our scheduling algorithms.

Once a connection is selected for service, the I/O server refers to the access list in order to determine what operation should be performed next. In the case of a read operation

the I/O server first uses `mmap()` to map the data region into its address space. This is performed on a region of 128 Kbytes in the implementation tested, which previously was found to be a reasonable trade-off between mapping too large a region and performing too many mapping operations.

Once the region is mapped into memory, `send()` is used to send the data from the desired region to the remote host. The `O_NONBLOCK` flag is set on the socket using `fcntl()` prior to sending data so that the server will not block on the socket. When a new region of the file is needed by this connection for I/O, the old region is unmapped with `munmap()` before the new region is mapped.

3.3 Performance Results

We present some performance results using PVFS in a parallel test application on the Beowulf cluster at NASA Goddard Space Flight Center. At the time we performed our experiments, the cluster was configured as follows. There were 64 nodes; each node had two 200-MHz Pentium-Pro processors, 128 Mbytes of RAM, a 6-Gbyte Seagate IDE drive, and a 100 Mbit/sec Intel EtherExpress Pro network card that could operate in full-duplex mode. A Foundry Network FastIron II switch connected the nodes. A separate front-end node was connected to the switch via a 1-Gbit/sec full-duplex connection. The nodes were running Linux 2.2.5 and MPICH 1.1.2.

Seagate Medalist Pro 6540 disks were used in the system, with 512 Kbyte caches, 5400 RPM spindle speeds, and advertised sustained transfer rates of 7.9 Mbytes/sec. The performance of these disks measured using the `Bonnie` file-system benchmark [4] showed

a write bandwidth of 8.8 Mbytes/sec with 27.1% CPU utilization and a read bandwidth of 7.5 Mbytes/sec with 17.3% CPU utilization when operating on a 256-Mbyte file in a sequential manner. The write performance measured by `Bonnie` is slightly higher than the advertised sustained rates, perhaps because the test accessed the file sequentially, thereby allowing file system caching, read ahead, and write behind to better organize disk accesses. Write performance is likely higher than read performance due to write behind effects as well.

Since PVFS currently uses TCP for all communication, we measured the performance of TCP as well. Using `ttcp` version 1.1 [53], we found a TCP throughput of 11 Mbytes/sec.

To measure PVFS performance we performed experiments using concurrent writes and reads with the native PVFS calls. We varied the number of I/O and compute nodes used, and, in all cases shown here, the I/O and compute nodes were distinct. Test data was removed and the disks synchronized between each test iteration.

3.3.1 Scaling I/O Nodes

The first set of test runs was designed to test the performance of PVFS as the number of I/O nodes (IONs) is scaled. The amount of data accessed on each I/O node was held constant for each number of application processes (compute nodes) at 2 Mbytes per process. For example, for the case where 26 application processes accessed 8 I/O nodes, each application task wrote out 16 Mbytes of data, for a total of 416 Mbytes stored in the parallel file.

Figure 3.9 shows that the maximum read bandwidth obtained was around 30 Mbytes/sec for 4 I/O nodes, 60 Mbytes/sec for 8 I/O nodes, 120 Mbytes/sec for 16 I/O nodes, and

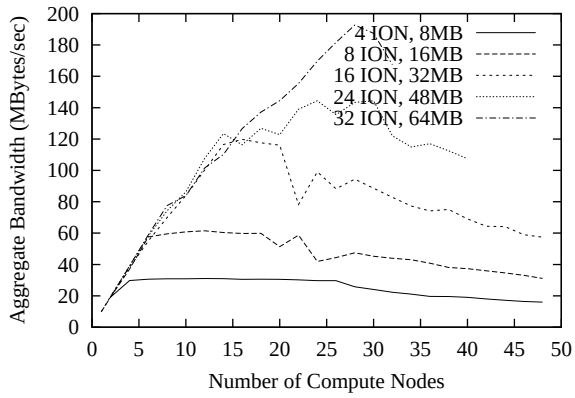


Figure 3.9: PVFS Read Performance

190 Mbytes/sec for 32 I/O nodes. These values closely match the maximum performance we expect to get out of the disks on each node, although it is likely that at the points where these peaks occur, we are seeing the benefits of caching done by the local file system on the I/O nodes. In all cases, we find that network performance is a bottleneck for small numbers of application tasks, but it appears that I/O performance is the bottleneck for larger numbers of application tasks (and thus larger amounts of data). For writing, we observed maximum bandwidths of 30 Mbytes/sec, 70 Mbytes/sec, 120 Mbytes/sec, and 150 Mbytes/sec for the different numbers of I/O nodes, as shown in Figure 3.10.

3.3.2 Read Performance Limitations

Figure 3.11 shows the read performance of PVFS using four I/O daemons and by varying the number of compute nodes. We see that as the size of the aggregate requested region increases, we reach a point where performance drops off significantly, irrespective of the number of clients (compute nodes). This point occurs after approximately 6 clients reading 32 Mbytes each, 14 clients reading 16 Mbytes each, and 26 clients reading 8 Mbytes each.

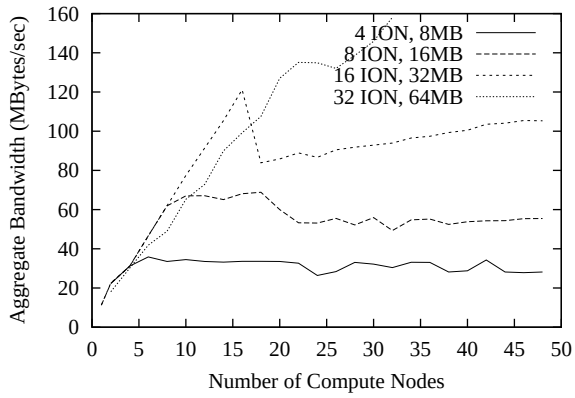


Figure 3.10: PVFS Write Performance

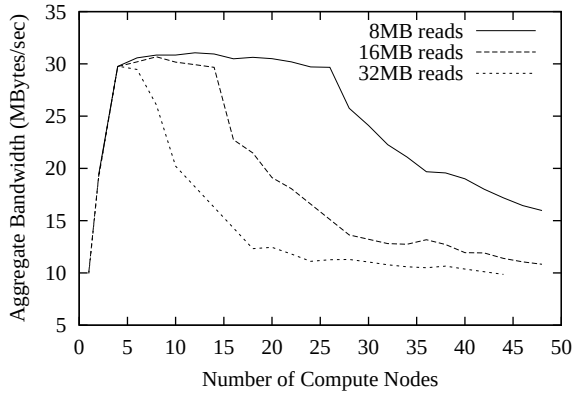


Figure 3.11: Read Performance Dropoff

This point corresponds to approximately 200 Mbytes of total file data or 50 Mbytes of file data per server. We believe that this drop-off is related to the size of the available cache on the I/O node and is due to the network-centric design of our I/O-server routines. This issue is discussed further in Section 3.4.

We are working on a detailed performance study of PVFS on a 256-node cluster at Argonne, called *Chiba City* [8], which will be reported in a future paper.

3.4 Lessons Learned

In the present design of PVFS, we made a number of decisions to ease implementation, such as using a single metadata manager, using local files for data and metadata storage, not implementing directory calls initially, and matching PVFS request types to system calls. In some cases we have already identified problems with these decisions and implemented changes, such as the inclusion of directory calls in the PVFS system to eliminate the NFS dependency. Other decisions have turned out to be acceptable for now; for example, using a single metadata manager and using local files (as opposed to raw devices) for storing metadata and file data has not, so far, caused problems with performance or configurability.

Some of the decisions that seemed appropriate at the time, however, have turned out to have nontrivial and detrimental implications. We have realized that improvements are needed in the following areas, and we plan to incorporate them into future releases of PVFS.

- A more flexible I/O-description format between clients and I/O daemons is needed.
- The data-transfer system should be implemented as a replaceable module.
- A better low-level client-side interface is needed.
- Low-level PVFS operations should be implemented in a way that can be easily integrated with VFS [56].
- More disk optimizations are needed in the I/O daemons.

While PVFS support for simple strided accesses (through partitioning) is a step in the right direction in terms of supporting common I/O patterns, more powerful request mech-

anisms are needed to take advantage of the descriptive capabilities of new interfaces such as the Scalable I/O (SIO) low-level API [13] and MPI-IO [17, 32]. Work in parallel I/O libraries indicates that providing library implementers with native scatter-gather support is highly desirable for obtaining the highest-possible performance [55].

The existing implementation is tightly coupled with TCP data transfer. While at the time of the original design this choice seemed benign, with the advent of mainstream high-performance data-transfer mechanisms such as GM [35], VIA [16], and ST [45], the need for an internal communication abstraction that can effectively use a variety of networking technologies has become apparent.

Although PVFS was originally intended mainly for Linux clusters, there is a demand for porting it to clusters that run other operating systems as well. The implementation of the client-side interfaces in PVFS needs to be better organized for ease of porting to other operating systems. The integration of the system-call-wrapper technique into the library is currently made in the calls themselves, which makes the client library difficult to port. The inclusion of the partitioning system directly in the low-level library has also resulted in a few complications in implementation. In retrospect, we should have created a different lower-level interface, built the partitioned file interface on top of it, and included the system-call wrappers as a separate library. This would separate the important functionality of the base client library, namely, performing basic I/O operations, from the administrative tasks of keeping up with partitioning information and determining file types.

The requests used by the client library to communicate with the PVFS daemons are modeled after the system calls that they help perform. When implementing a system such as PVFS at the user level, this is the most obvious method of choosing request types. These

calls, however, do not map well into the types of operations seen in the VFS system [56]. This has created problems for us with respect to implementing a client-side kernel interface. Mapping VFS-like operations into system-call operations would be a simpler task.

The design of the I/O daemon focuses on extracting maximum performance from the network. To this end, when reading a PVFS file, the I/O daemon code attempts to fill all available network buffers for all outgoing connections. As discussed in Section 3.3.2, this approach may lead to performance drop-offs. A better approach that we plan to implement is to also reorder read operations on the I/O daemons for better disk-read performance.

PVFS brings high-performance parallel file systems to Linux clusters and, although more testing and tuning is needed for production use, it is ready and available for use *now*. The software can be downloaded freely from `http://www.parl.clemson.edu/pvfs`. The inclusion of PVFS support in the ROMIO MPI-IO implementation makes it even easier for applications (written portably with the MPI-IO standard [17, 32]) to take advantage of the available disk subsystems lying dormant in most Linux clusters.

We are also working actively towards developing the next-generation PVFS system, taking into account the lessons we have learned from our experiences with the existing system and feedback from PVFS users. The new version will have improvements in the client-server interface, the I/O-description format, data-transfer mechanism, and I/O-server-scheduling algorithms. A kernel VFS implementation is also in the works, which will allow applications to access PVFS files without the need for the system-call wrappers.

Chapter 4

Workload Study

In Chapter 3 we covered PVFS, the parallel file system used in our experiments. In this chapter we will discuss the workloads we examined in order to ascertain the effectiveness of our scheduling algorithms at servicing three different test patterns:

- single block accesses
- strided accesses
- random block accesses

First we will describe the system used for these tests and the scheduling algorithms added to PVFS for the purpose of this work. Next we will discuss the test applications used to gather data. Following that we will discuss the results for each of the test patterns. Finally we will make observations on the various algorithms and when each is appropriate.

4.1 Test System

The Beowulf machine on which this work was performed is a 17-node cluster connected by an Intel fast ethernet switch. Each node has the following specifications:

- Pentium 150MHz CPU
- 64 Mbytes EDO DRAM
- 64 Mbytes local swap space
- 2.1 GByte IDE disk
- Tulip-based 100Mbit fast ethernet card

One node runs the PVFS manager daemon and handles interactive connections while the other nodes are used as compute nodes, I/O nodes, or both. Each node runs Linux v2.2.13 with tulip driver v0.89H. The IDE disks provide approximately 4.5 Mbytes/sec with sustained writes and 4.2 Mbytes/sec with sustained reads, as reported by Bonnie, a popular UNIX file system performance benchmark [4]. When idle, approximately 8 Mbytes of memory are used on each node by the kernel and various system processes, including PVFS, leaving approximately 56 Mbytes of space which could be used by the system for I/O buffers.

For these tests two nodes were used as I/O nodes, and 14 nodes were used for computation. This combination allowed us to separate the I/O nodes from the compute ones, to provide a number of simultaneous requests which the I/O nodes can schedule, and to ensure that no single-disk optimizations are used on the I/O nodes (mapping PVFS file locations

to local file ones is simpler in the single-disk case). It is important to understand that the "scheduling" we refer to here is at the request level, not the disk level or network level. All scheduling of requests to the actual devices (network and disk) is performed by the kernel. We are simply choosing what to hand to the kernel from the user level by choosing the order of socket operations and local file operations we must perform.

4.2 Algorithms

In its base configuration the PVFS system optimizes for the network by using the stream-based I/O technique and by servicing all requests that are ready at any given time, regardless of data location. This algorithm will be maintained for use when network-limited behavior is detected and will be called *Opt 1* for the purposes of discussion. This is similar to a first-come first-served (FCFS) algorithm based on readiness of receivers to accept data (in the read case), or incoming data (in the write case).

A selection of three, increasingly disk-oriented, approaches will be implemented to better address disk-limited workloads. In all cases data ordering within a given request will remain the same; we are only modifying the order in which we service requests.

It is important to note that in all cases we will utilize *file* position as our indication of spatial locality as opposed to *disk block* position. This has been shown effective in previous works [48] and will avoid additional modification to the parallel file system. We will perform all testing within a single file to maximize the reliability of this measurement.

In the first new approach the algorithm used to select buffers to service will be modified to utilize data location information while retaining data transfer ordering. This data will be

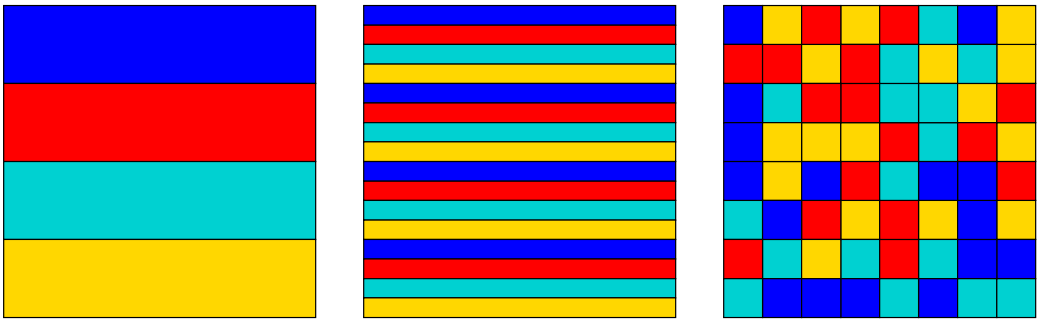
used to order the servicing of regions in a similar manner to circular scan (CSCAN) [47]. Once the regions are ordered, they are serviced in order to attempt to better match disk and file system read-ahead. This algorithm will be denoted *Opt 2* in further discussion.

The second new approach will extend the last one by selecting a subset of the disk from which to service requests at a given time, limiting the number of requests which we will service simultaneously depending on the size and distribution of requests. Specifically a logical "window" is defined to be of a size near the size of physical memory on the machine. Based on the last file position serviced, ready requests fitting inside this "window" (ie. having a starting file position no more than one half the window size bytes from the last position serviced) are serviced. If no requests fit this requirement, the nearest ready request is serviced instead. This should allow the system to better exploit caching and spatial locality at the expense of network buffer utilization. This will be known as *Opt 3*. This is similar to a window scan (WSCAN) algorithm.

The final new approach will order service strictly on location, servicing the request closest to the last access next and waiting on this request if it is not yet ready for service. This will mimic disk-directed I/O for many workloads and will be referenced as *Opt 4*. In many ways this is analogous to the greedy, or shortest seek time first (SSTF) [15], algorithm.

4.3 Test Applications

Three test applications in a total of four configurations were used in this workload study. These workloads represent a number of access patterns seen in some traditional applications, particularly ones operating on dense multidimensional matrices. We include both



a) Contiguous b) Strided c) Random

Figure 4.1: Test Workloads

single contiguous block access, strided access, and multiple random block access workloads in order to cover a wide range of possible workloads. Figure 4.1 shows the general pattern of access for these workloads.

In all cases a MPI application was used to create a set of application tasks which independently accessed a PVFS file system using the native PVFS libraries. MPICH version 1.2.0 was used. As mentioned before, in all tests a single PVFS file was used to store data.

In the single block access tests, contiguous regions of the data file were simultaneously accessed by each task. The tasks synchronize before the access, and the times to complete access were recorded. We consider the longest service time of any one task to be the application service time. We also calculate the mean service time for all tasks and the variance of task service time. Patterns such as these are seen in applications accessing dense matrices in a block manner, in checkpoint applications, and in some out of core applications.

In the strided access tests, multiple discontiguous regions of the data file were simultaneously accessed by each task using a single operation. Again the tasks synchronize before

the access, and the times to complete the operation were recorded. As before we consider the longest task service time to be the application service time, and we also calculate mean service time and the variance. Strided accesses are often seen as a result of row cyclic distribution of data sets and access to portions of records of fixed size.

The purpose of the random block access tests is to observe the system serving an application with an irregular access pattern. The file is logically divided into a number of blocks, and these blocks are randomly assigned to the tasks in the application such that each task will access an equal number of blocks. Tasks are synchronized before any accesses begins, and tasks access all blocks in random order using a native PVFS operation to access each block, one block at a time. The time to access all blocks is recorded for each task, the largest of these total times is considered the application service time, and mean service time and variance are also calculated. This pattern might be created by an application accessing pieces of a multidimensional data set or reading arbitrary records from a large database.

In all cases, our goal is to analyze the effects of the four algorithms on the performance of the system, using three metrics:

- application service time
- mean task service time
- task service time variance

Which of these metrics is most important depends on the nature of the applications run on the system and the policies in place for the system. For a system serving single parallel applications, application service time might be the most appropriate metric. For a system running multiple parallel applications or many serial ones, mean task service time might be

a more appropriate metric. Groups interested in fairness of access might instead desire to minimize variance.

For our write tests, the data file was removed before each test run and the disks synchronized. A thirty second delay between test runs was also imposed to allow the `update` process to write any metadata to disk on I/O nodes.

For read tests, before each run a local data file larger than the size of a node's memory was read in its entirety on each I/O node to remove all PVFS file data from cache. A separate run of read tests was additionally performed in which we allowed file data to remain in cache (ie. did not read a local data file between runs). These results are compared to small accesses without cache in the following sections as well.

Graphs of each of the three metrics are provided for all tests. In all cases we present output for all four of the algorithms described earlier in this work. The total data accessed on a single I/O node is shown on the X axis, and either service time or variance is provided on the Y axis. The data presented is the average of three test runs.

4.4 Single Block Accesses

The results for uncached single block accesses are shown in Figure 4.2. Opt 4 provides the lowest average task service time due to servicing single tasks one at a time, while Opt 1 and 2 provide the lowest variance by cycling through all requests in a more "fair" manner. Opt 3 provides a compromise between the two, allowing for a smaller service time but retaining a lower variance than Opt 4 due to added flexibility in servicing tasks.

For single block writes, shown in Figure 4.3, we observe less benefit to picking the best algorithm. Opt 4 appears to provide some benefit in terms of task write service time, most likely due to servicing one task at a time.

An interesting effect is seen when read access with and without caching is compared. When all file data is flushed (Figure 4.4) , Opt 4 provides by far the best task read service time (beating the worst performers, Opt 2 and 3, by as much as 28%) and only slightly higher variance with a similar application read service time. Recall that Opt 4 is our algorithm most similar to disk-directed I/O; only the request closest to the last accessed file position will be serviced. It is apparent that we are more effectively utilizing disk resources with Opt 4 in this case.

In the case where cached data is available (Figure 4.5), we see that Opt 4 results in the slowest application read service time, and highest variance, but again with good task read service time, again beating the worst performer by 30%. However, Opt 3 seems to be a more appropriate overall choice in this case. Recall that Opt 3 relaxes the strict ordering of requests, allowing for requests within a position window to be serviced and always allowing the nearest ready request to be serviced. When cached data is available, Opt 3 provides a better application service time while also resulting in a competitive mean task service time and lower variance than Opt 4.

4.5 Strided Accesses

For our strided access pattern we arbitrarily chose to access 16 disjoint regions with each task access. The size of the disjoint regions was varied throughout the tests.

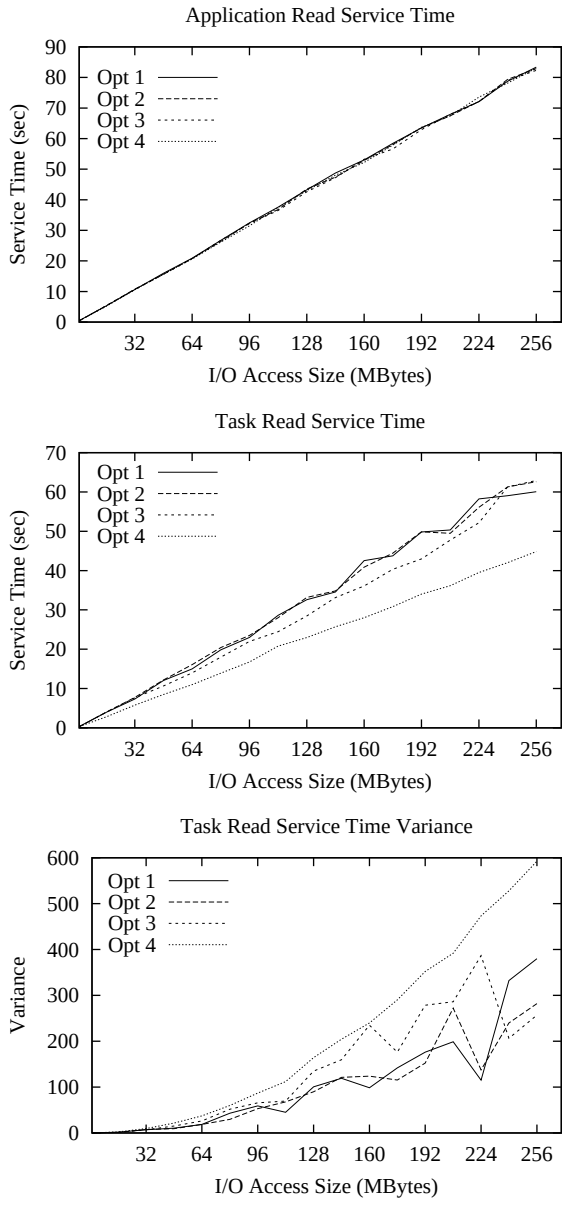


Figure 4.2: Single Block Read Performance

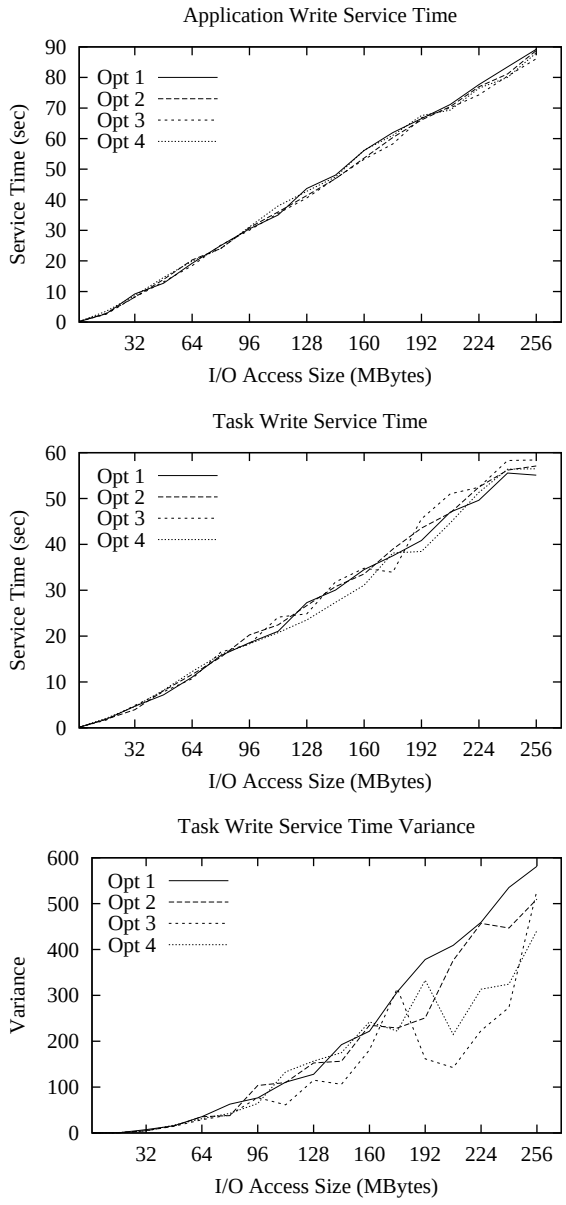


Figure 4.3: Single Block Write Performance

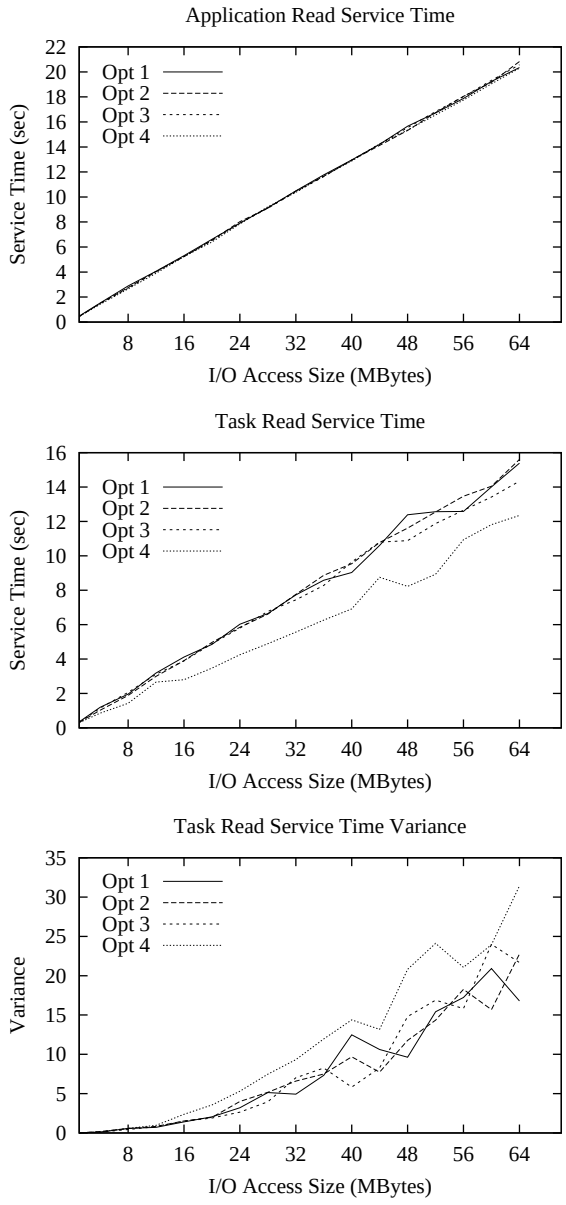


Figure 4.4: Single Block Read Performance, Small Uncached Accesses

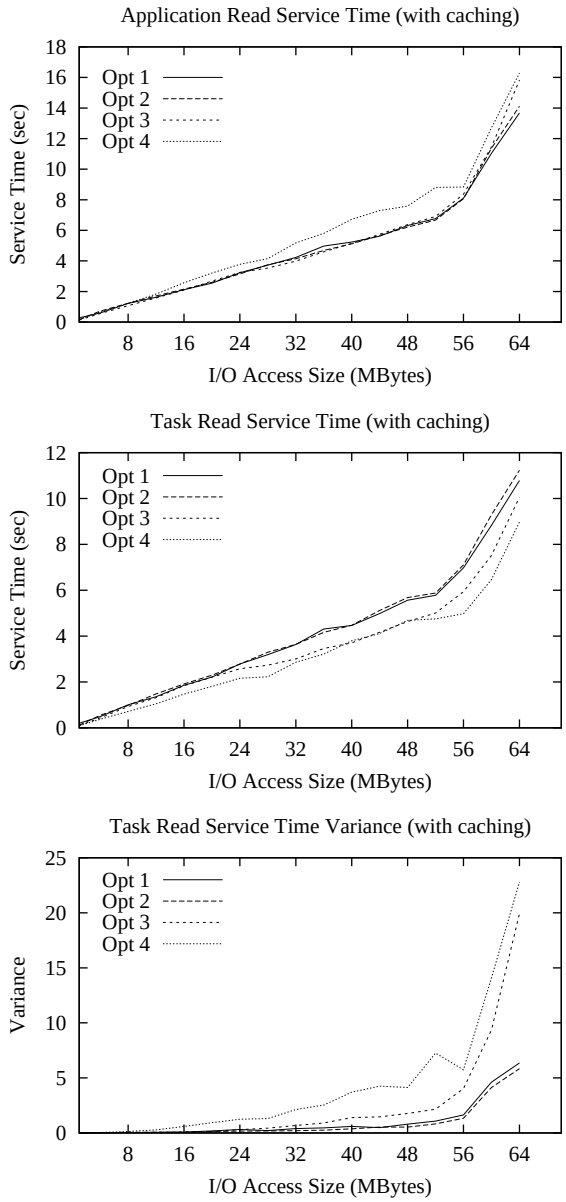


Figure 4.5: Single Block Read Performance, Small Cached Accesses

When servicing strided read requests, we see in Figure 4.6 that Opt 1 provides the lowest mean task service time, most likely due to the fact that file data is interleaved between the application tasks, resulting in Opt 1 performing similarly to Opt 4 but without its ordering restrictions. All algorithms result in approximately the same application read service time over the wide range of access sizes.

For writes (Figure 4.7) we see that Opt 4 provides the best application service time and variance, while no single algorithm stands out in terms of task service time performance.

For small reads we again see in Figures 4.8 and 4.9 that cached data availability has an impact on algorithm selection. For the case where cached data is not available, we see that Opt 1 provides gains in task service time at the cost of higher variance, reducing service time by 14% over the worst performing algorithm. When data does reside in cache, however, we see that Opt 1 provides a very competitive application and task service times while resulting in low variance. With data in cache, service times become more consistent even when we service in an order that is not disk optimal.

4.6 Random Block Accesses

In addition to tests focusing on known patterns we also chose to study random block access. An interesting characteristic of these tests is that only a fraction of the total data to be accessed is being requested at any one time since multiple operations are required to access the randomly distributed blocks (using native PVFS calls). This is in contrast to the previous tests, where all data to be accessed is requested in single calls. The result of this is that the total size of requests at any point in time is no more than S_{tot}/N_{blks}, where S_{tot} is

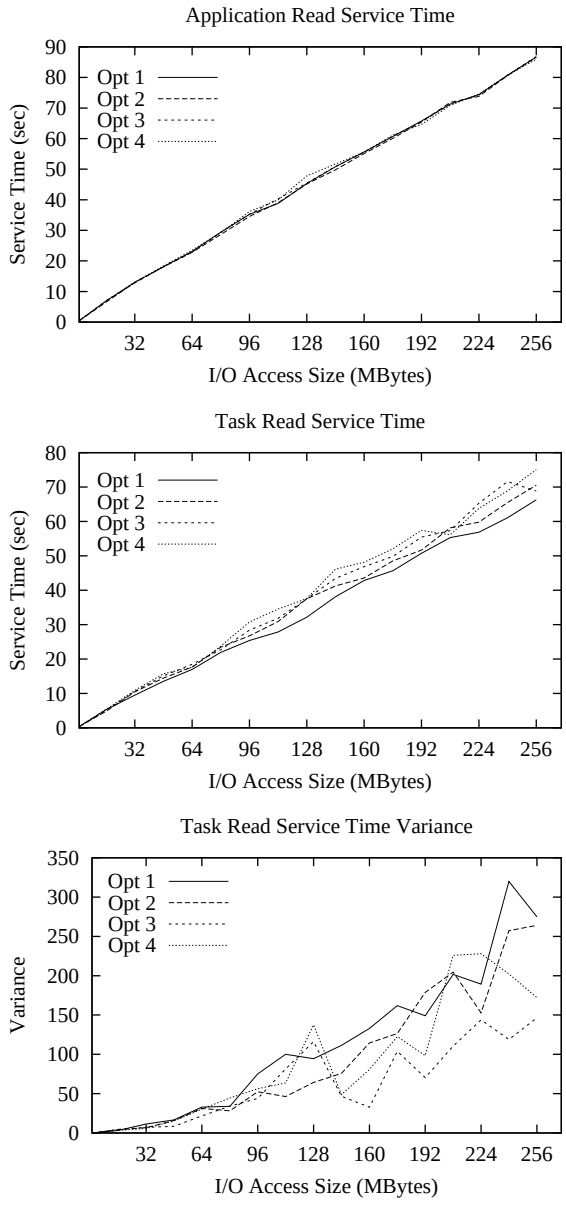


Figure 4.6: Strided Read Performance

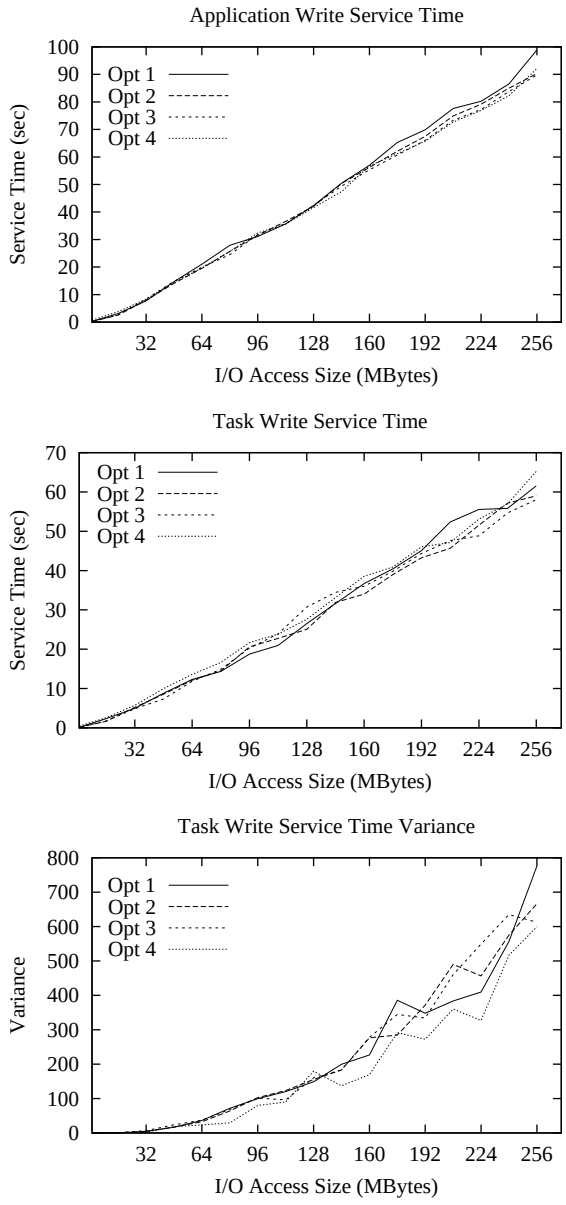


Figure 4.7: Strided Write Performance

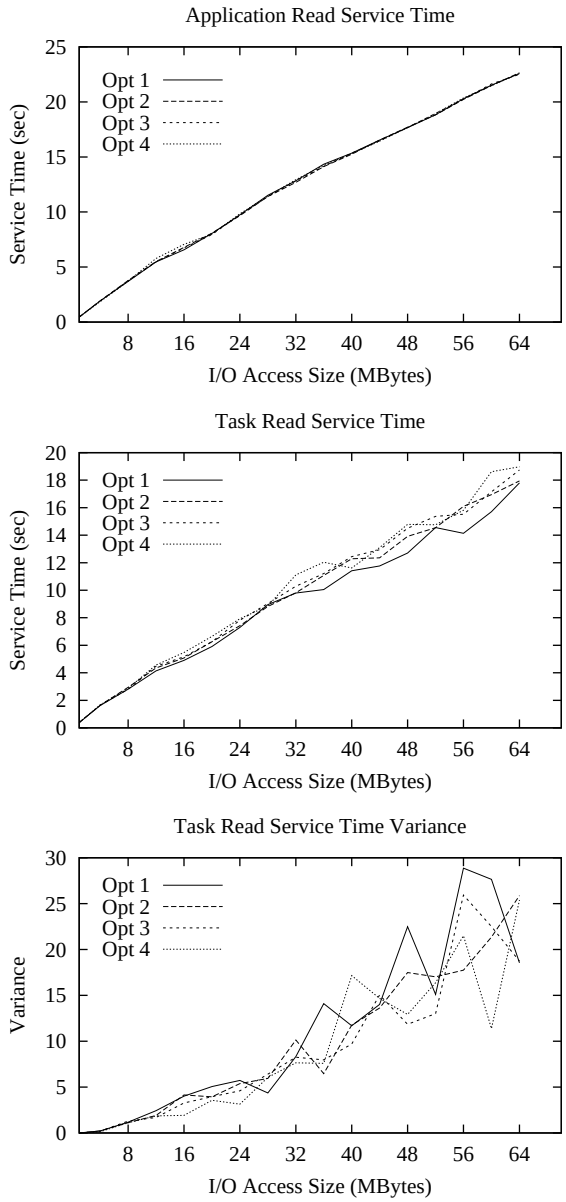


Figure 4.8: Strided Read Performance, Small Uncached Accesses

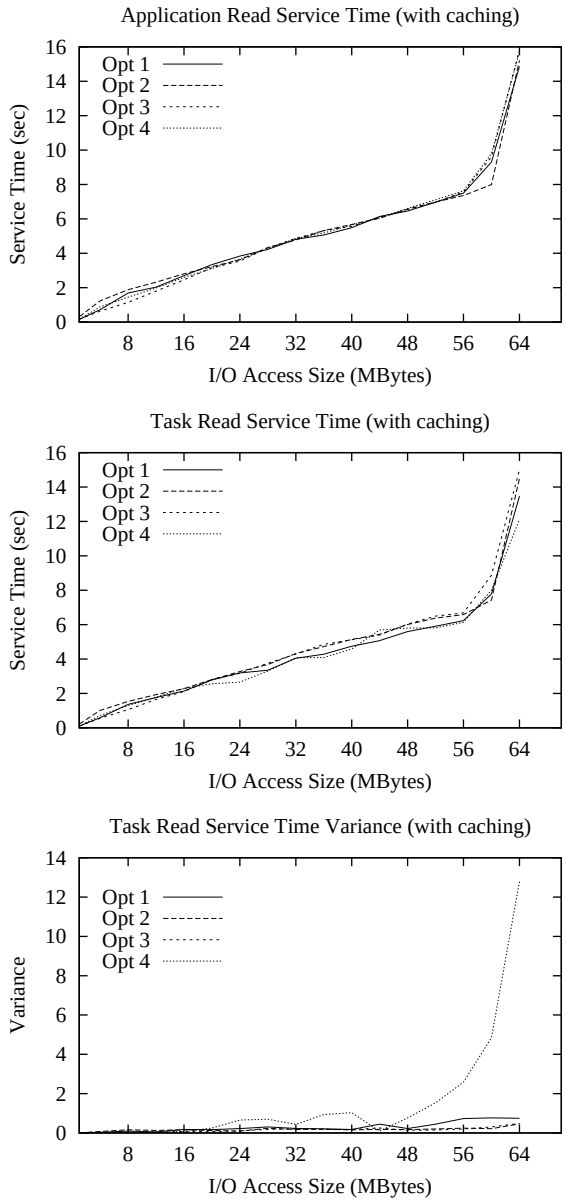


Figure 4.9: Strided Read Performance, Small Cached Accesses

the total amount of data that will be accessed and N_{blks} is the number of blocks into which the data is split (per task).

Figures 4.10 and 4.11 show performance when the data is split into 16 blocks per task. In the random 16 block write tests we see that Opt 2 provides a slight application performance benefit when more than 100 Mbytes of data are accessed per I/O daemon. In addition Opt 2 gives us by far the lowest variance of the bunch, especially as data sizes increase. The consistent ordering of request service by Opt 2 allows for this.

We note that Opt 3 and Opt 4 have extremely high variance on the read test, while Opt 2 variance is quite low. The difference in task service times is fairly small here (Opt 4 consistently outperforms Opt 2 by around 6%).

When we focus on small read accesses, we see that Opt 4 provides slightly higher performance than the rest when cache is not available, at the expense of a particularly high variance (Figure 4.12). Opt 4 is helpful in situations such as the one created in the random block tests, where blocks of contiguous data, potentially separated by unwanted data, are accessed simultaneously because it avoids traversing these unwanted data regions more often than necessary. As expected Opt 1 provides the best performance when data is cached (Figure 4.13), with both low service times (providing a modest 5% performance gain over Opt 2) and variance.

In Figures 4.14 and 4.15 we see the random 32 block test results. Opt 2 again provides a consistently low variance in conjunction with competitive service times for writes. Performance is slightly lower than in the 16 block case, consistent with the fact that twice as many requests are used to perform the same transfer of data. For reads we again see Opt 3 and Opt 4 providing slightly better performance at the expense of high variance.

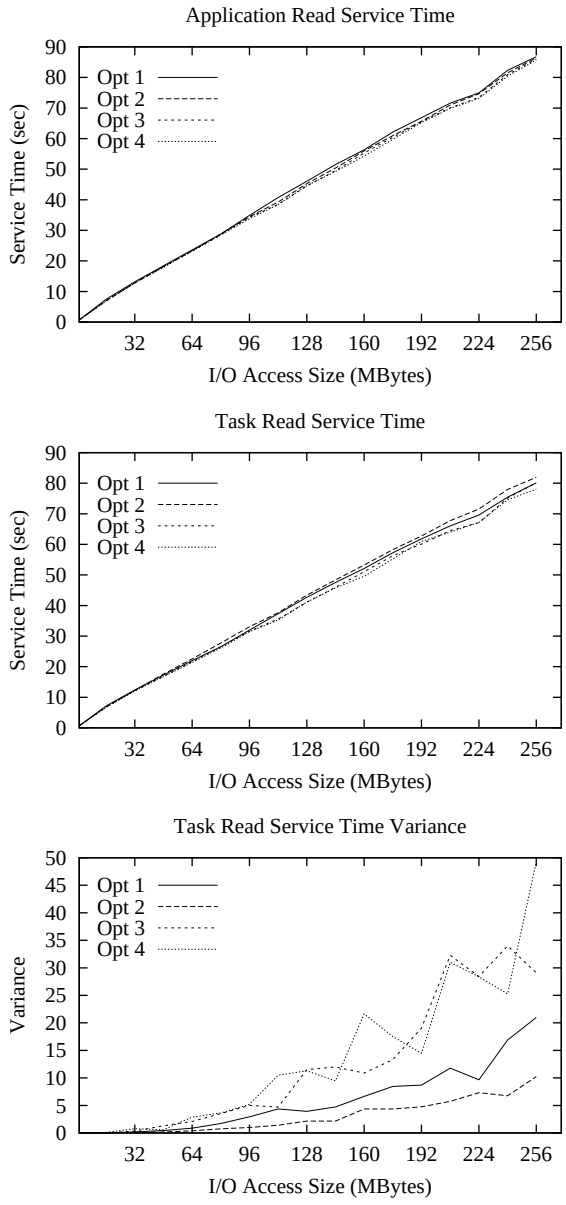


Figure 4.10: Random (16 Block) Read Performance

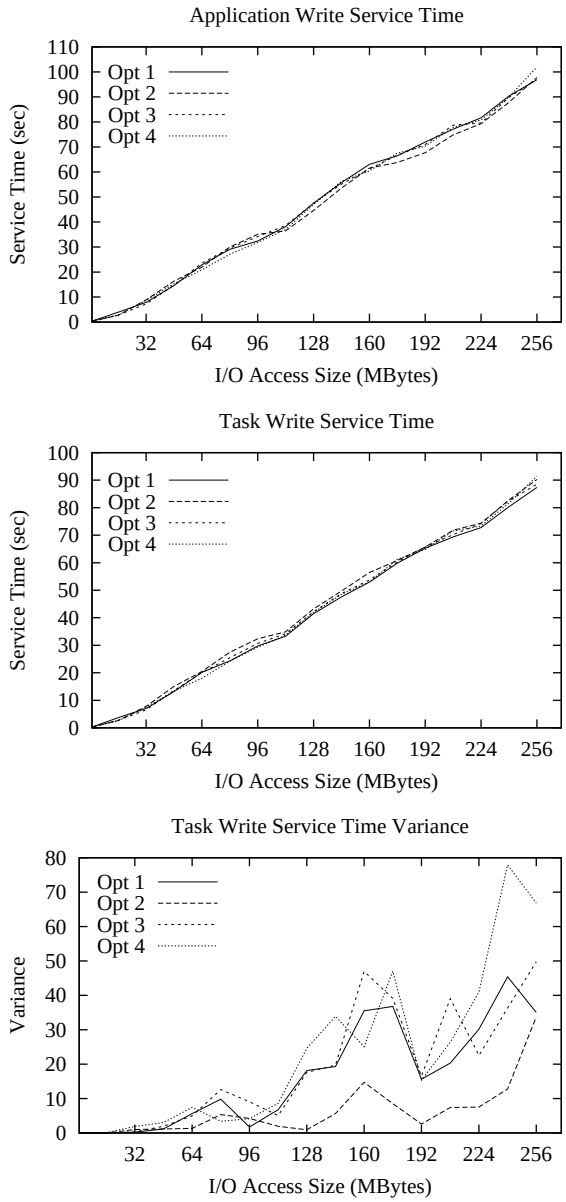


Figure 4.11: Random (16 Block) Write Performance

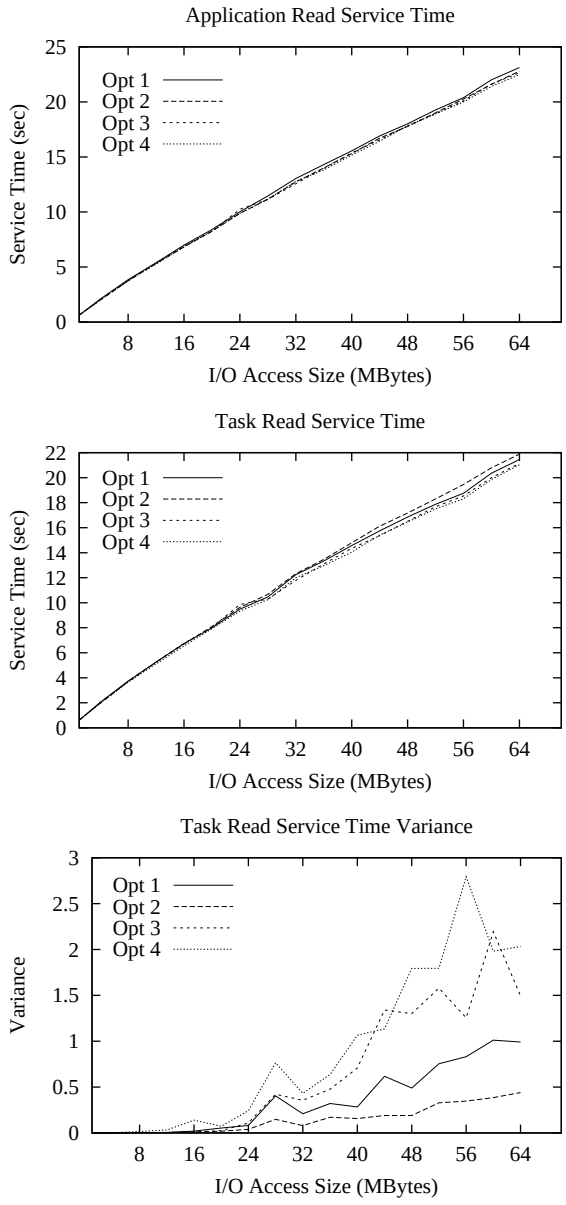


Figure 4.12: Random (16 Block) Read Performance, Small Uncached Accesses

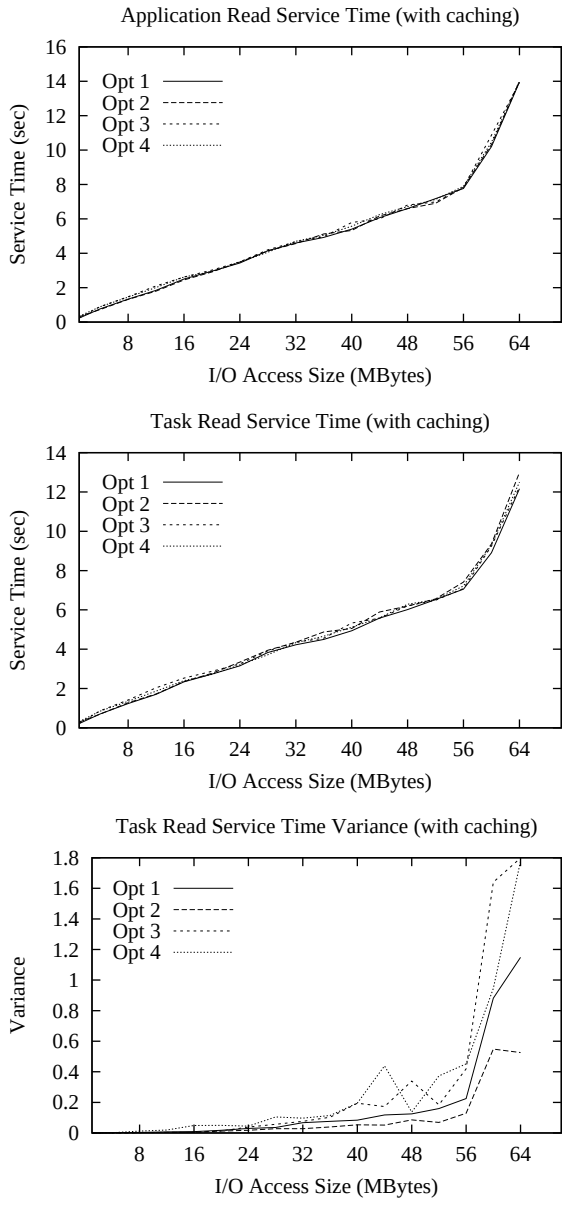


Figure 4.13: Random (16 Block) Read Performance, Small Cached Accesses

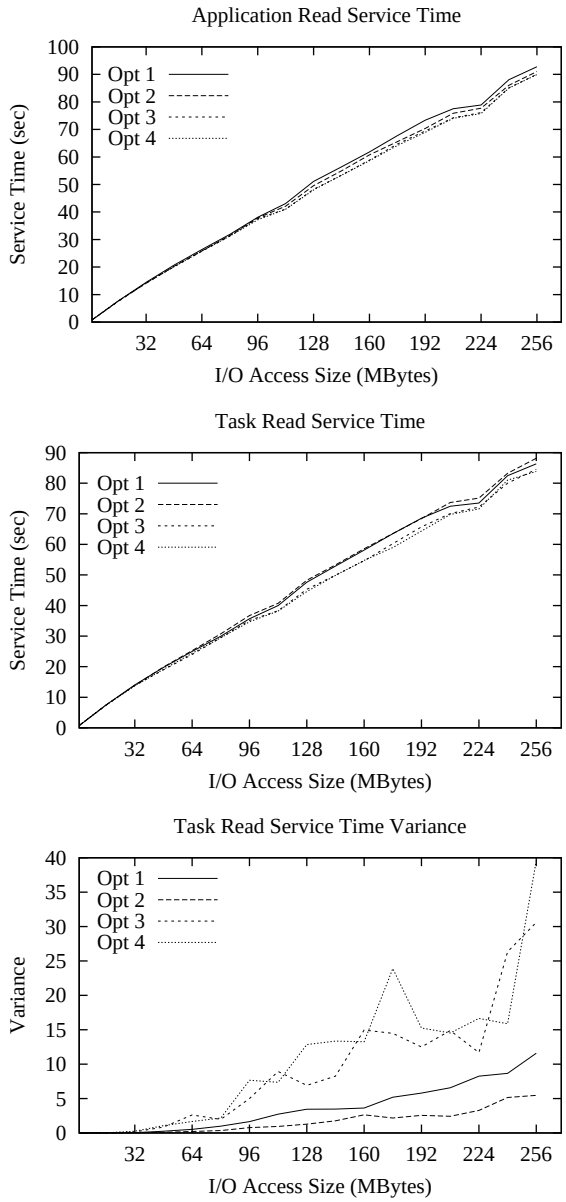


Figure 4.14: Random (32 Block) Read Performance

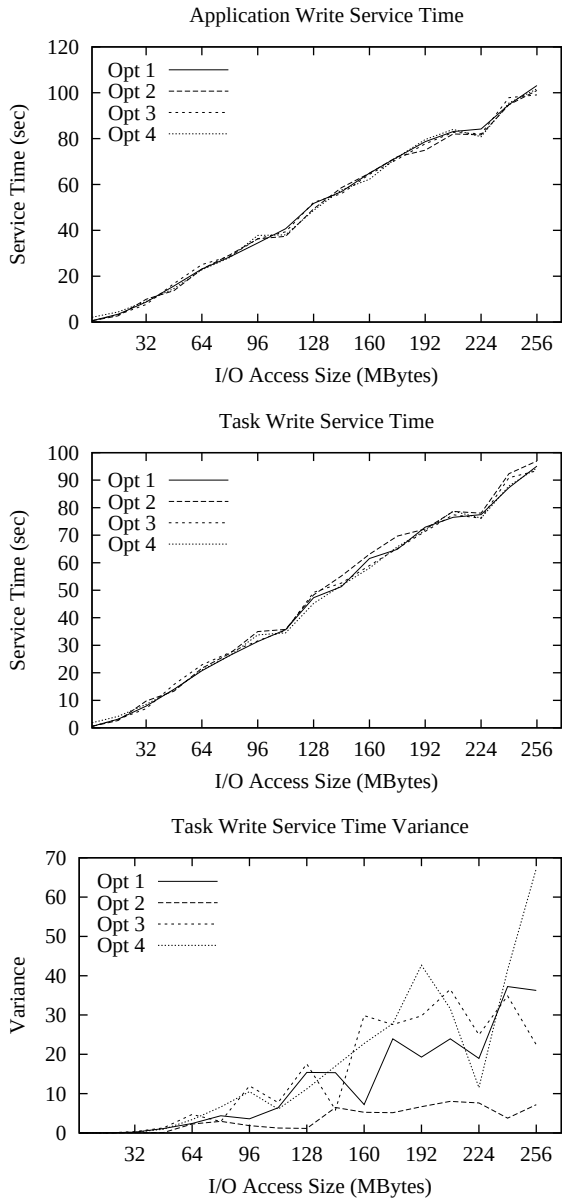


Figure 4.15: Random (32 Block) Write Performance

In Figures 4.16 and 4.17 we focus on read performance for small accesses with and without cached data. Again we see Opt 4 performing best for uncached files, with a 7% advantage over Opt 2, at the expense of high variance. On the other hand, if data is cached Opt 1 appears to perform best, outperforming Opt 4 by as much as 10%, with the second best variance.

4.7 Observations

As a whole we see that we are able to affect application service time in only a small number of cases, and in general our scheduling changes had little effect on write workloads. However, we see benefits to applying certain algorithms in all read cases with respect to task service time. In particular, for situations where uncached contiguous regions are being serviced, Opt 3 and Opt 4 show the best performance. On the other hand, for cases where significant fractions of data are cached we see that Opt 1 performs the best.

For our strided read workload we see that Opt 1 performs best as well. This is partially due to the implicit interleaving in the requests. However, it is likely that our predefined ordering of request data is also a factor. By this we mean that the order in which request data is returned to the requesting task by PVFS is always in order of monotonically increasing file byte offset. This constrains the order in which we can service the pieces that make up strided requests, and by doing so limits the ability of more disk-oriented algorithms to best access the disk.

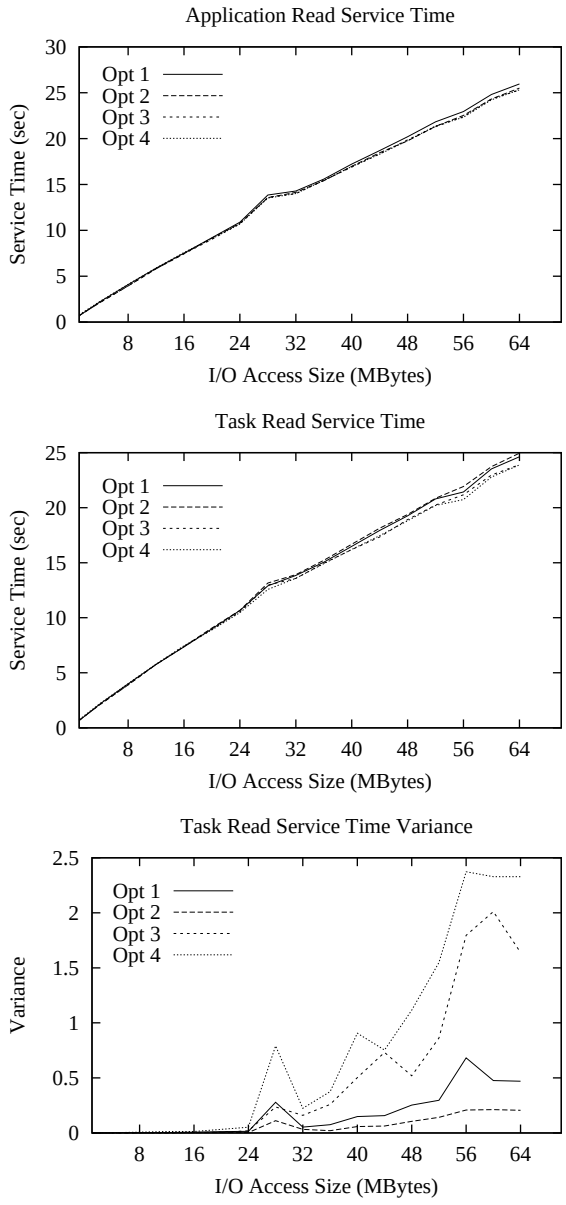


Figure 4.16: Random (32 Block) Read Performance, Small Uncached Accesses

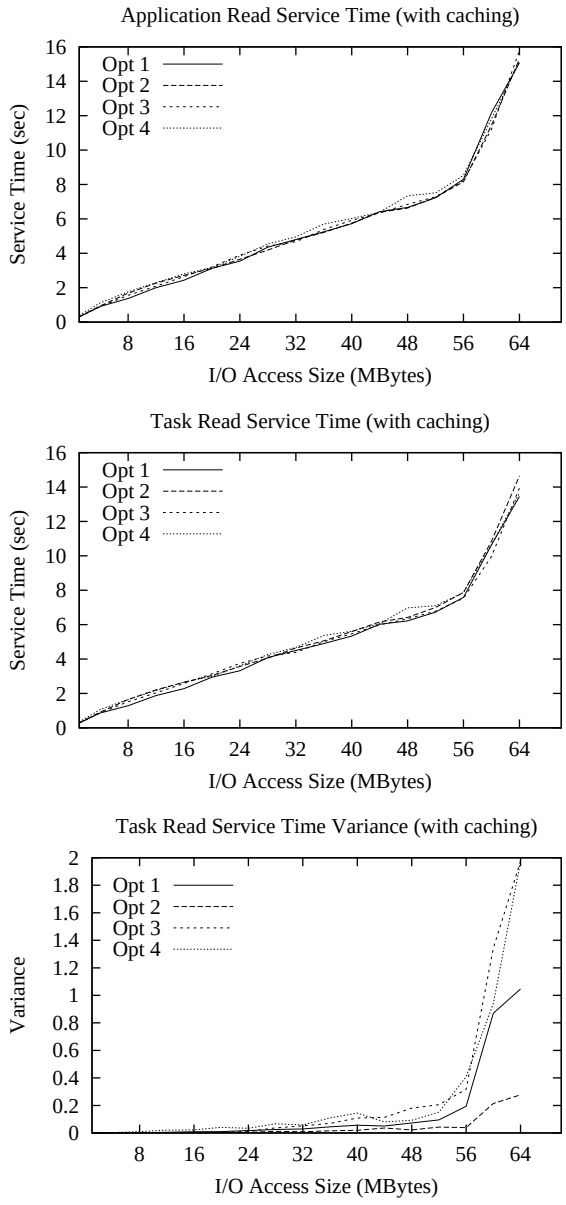


Figure 4.17: Random (32 Block) Read Performance, Small Cached Accesses

Chapter 5

Reactive Scheduling

In Chapter 4 we discussed the performance of our test system utilizing four different scheduling algorithms. These algorithms are one component of our reactive scheduling implementation. The second of these components is a system model. This model will be used to predict which scheduling algorithm is the appropriate choice given a specific system state and workload in progress. In this chapter we will develop a two-part empirical model which will serve as the system model in our reactive scheduling implementation. First a simple system model is used to characterize behavior of the system for various workloads. This model is then tuned to match the behavior of the system when algorithm effects are included. Model parameters are calculated for the specific system in order to most accurately estimate behavior.

We will develop a model for read-only traffic which will allow us to correctly select an algorithm based on the workload, the system state, and the available algorithms. We will constrain our discussion to accesses to a single file, and we will observe file access locations. However, this same description could be used to cover physical blocks on a

single device. It is our intent that the model be extendable to include both better modeling of specific components and new choices for algorithms. For this reason the model is split into two components, the *baseline model* and the *algorithm efficiency functions*. The baseline model allows us to estimate the performance of the system given workload parameters without algorithm-specific effects included. The algorithm efficiency functions allow us to calculate the effects of a given algorithm on the system performance given a specific workload and system state. By separating the behavior of the system as a whole from the effects of the algorithms on this behavior, we reduce the complexity of modeling the system as a whole.

Once this model has been developed we will cover the implementation of RS in PVFS. In Section 5.5 the particulars of our selection implementation for PVFS and the implications of adding this component to a real system. Following this we will cover the performance of PVFS using RS to automatically select scheduling algorithms in Section 5.6. In Section 5.7 we will discuss the effectiveness of RS in this environment. Finally, in 5.8, we cover capabilities that might aid in more effectively applying the RS approach.

5.1 Baseline Model

Our original intention in this work was to develop a system model which was based on modeling the components that make up the system, including network devices, disks, and cache. This model was to include such factors as caching and dirty buffer flushing, and is described in Appendix A. Unfortunately due to the complexity of the system with which

we are working we were unable to map this model to what we were seeing in tests of the actual system.

This failure led us to take an alternative approach to modeling the system, instead applying an empirical approach. We performed the workload study shown in Chapter 4, and using this data we built a model that we felt matched the behavior of the system as a whole, including both the hardware components (eg. disks and network devices) and the software involved, particularly the parallel file system. This led us to the model presented here, which is the result of observing the system as a whole for a number of workloads. This model relies on a set of parameter values, selected by the system administrator based on observed system behavior, to operate correctly. These parameters are selected using a series of benchmark tests, and a discussion of selecting these values is given in Appendix B. Once these parameters are selected, the model will accurately predict performance for the given system for a number of workloads. If any system components are changed, including but not limited to processor, disk, available memory, and network components, these parameters must be recalculated. If the model is to be used on another system, these parameters must also be recalculated.

This model will estimate task service time for read requests and assumes we are using a round-robin disk-ordered servicing of the requests as in Opt 2. The parameters for the model are listed in Table 5.1 and will all be discussed here. The basic equation describing the system can be written as:

$$T_{taskread} = T_{over} + T_{ioread} \tag{5.1}$$

The term T_{over} is a constant value describing the overhead of servicing a request. This is time spent performing operations other than I/O; processing the request, sending acknowledgements, and so on. This term could be a function of the workload in a more complicated model. The second term, T_{ioread}, represents the time spent performing disk and network I/O in order to service a request. This term is a function of a number of parameters.

Before we discuss the calculation of T_{ioread}, it is important to understand the parameters we will use to describe workloads presented to the system. We found that for the tested workloads we needed four parameters to accurately describe workloads in progress:

- total size of requests, S_{req}
- number of requests, N_{req}
- range of request locations, R_{req}
- total number of disjoint regions in requests, D_{req}

The first two of these values describe the total number of requests in service and the total size of the data requested by them. The range will be the number of bytes included in the smallest contiguous region which would include all the bytes requested at some point in time. In our simple, single file case this is calculated by subtracting the lowest byte position (in the file) of any one request from the highest byte position of any one request, and will be denoted R_{req}. We will also be interested in the number of disjoint regions that make up our requests. For example, in our strided test cases there were sixteen disjoint regions for each request. We will denote the total number of disjoint regions in the request set as D_{req}.

We also noted that there appeared to be two effective bandwidths seen in our tests; one rate occurred when caching was effective while a second occurred when it was not. We were able to characterize the I/O behavior with four parameters:

- I/O bandwidth when caching is effective, B_{drc}
- I/O bandwidth when caching is not effective B_{dr}
- Effective size of cache, S_{cache}
- Cache drop-off coefficient, C_{cache}

Given these parameters we can then characterize T_{ioread} with:

$$T_{ioread} = \begin{cases} \frac{S_{req}}{B_{drc}} & \forall\ S_{req} \leq S_{cache} \\ \frac{S_{req}}{B_{dr}} & \forall\ S_{req} \geq S_{cache}(1 + C_{cache}) \\ B_{dr}(\frac{S_{req}-S_{cache}}{C_{cache}S_{cache}}) + B_{drc}(1 - \frac{S_{req}-S_{cache}}{C_{cache}S_{cache}}) & \text{otherwise} \end{cases} \quad (5.2)$$

The C_{cache} and S_{cache} values allow us to match the observed transition from cached to non-cached I/O performance.

This allows us to calculate a task service time given our I/O characterization parameters and a total size of requests. This is reasonable due to our assumption that we will service in round-robin fashion; with this type of scheduling the number of requests is not as important as the total size because we will be servicing them all simultaneously, so service will complete for all requests around the same time.

This characterization of behavior does not take non-idealities in the workload into account, and these must be acknowledged for our model to hold for a range of workloads.

We will add two new factors to help account for this:

$$T_{taskread} = T_{over} + E_{req} E_{dist} T_{ioread} \tag{5.3}$$

The first new factor is E_{req}, which will take into account the efficiency of the requests. In our case this will cover the inefficiency of using requests that specify disjoint regions. We aren't saying that these requests are "bad"; they are simply less efficient to service than contiguous requests. They are still likely to be more efficient than using multiple requests to access the disjoint regions. The second factor we introduce is E_{dist}, which takes into account the efficiency of the distribution of requests. This will enable us to take sparse data access into account. Adding in these two factors enables us to more accurately tune the model to match observed behavior.

Equations 5.4 and 5.5 show the calculation of these new factors, which are based on our workload characteristic values and four parameters. C_{req} and C_{dist} control the degree to which the particular inefficiency affects performance, while P_{req} and P_{dist} help us describe the type of relationship (eg. linear).

$$E_{req} = \frac{1}{(1 - C_{req}) + C_{req}(\frac{N_{req}}{D_{req}})^{P_{req}}} \forall\, 0 \geq C_{req} \leq 1 \tag{5.4}$$

$$E_{dist} = \frac{1}{(1 - C_{dist}) + C_{dist}(\frac{N_{req}}{D_{req}})^{P_{dist}}} \forall\, 0 \geq C_{req} \leq 1 \tag{5.5}$$

From our observations of random block access we found that the performance degradation caused by increasing sparseness of data was not a linear effect. This was the motivation for the addition of the exponential parameter P_{dist} to the equation describing E_{dist}. For

Table 5.1: Model Variables

Predetermined Constants	
B_{dr}	Bandwidth of I/O system when cached data is unavailable
B_{drc}	Bandwidth of I/O system when cached data is available
C_{dist}	Coefficient determining effect of sparse data distribution
P_{dist}	Exponent component for sparse data distribution function
C_{req}	Coefficient determining effect of strided requests
P_{req}	Exponent component for sparse data distribution function
C_{cache}	Coefficient determining range of effect of caching
T_{over}	Service time spent not performing I/O
System State Values	
D_{req}	Number of disjoint regions in requests in service
N_{req}	Number of requests in service
R_{req}	Range of file positions in requests in service
S_{req}	Total size of requests in service
S_{cache}	Memory available for caching
Derived Values	
E_{dist}	Efficiency of access based on data distribution
E_{req}	Efficiency of access based on request characteristics
T_{ioread}	Expected time to service a read request for a task

completeness this parameter was also added to the equation relating to disjoint requests, E_{req}, although our testing was not sufficient to determine if the increasing the number of disjoint regions had a linear or exponential effect on performance.

5.2 Matching Model to Data

As mentioned earlier we will attempt to match our baseline to the performance of Opt 2, our round-robin disk-ordered scheduling algorithm. Opt 2 provides the most consistent

performance of the four algorithms, and its behavior is the easiest to characterize. Our focus in calculating constants for our baseline model is on accurate prediction over a wide range of sizes.

We first calculate T_{over}, B_{dr}, and B_{drc} using the data from our single block tests, which we will consider to be the "ideal" workload case. We then calculate C_{req} using data from the strided tests and C_{dist} and P_{dist} using data from the two sets of random block tests. We assume that P_{req} is one, since we only have one set of data points from which to calculate both C_{req} and P_{req}.

Table 5.2 summarizes the values we calculated for our parameters. It is interesting to note that B_{dr} does in fact closely match the observed disk bandwidth of the test system, and B_{drc} is approximately what we would expect given the bandwidth of our network.

The C_{dist} and P_{dist} values indicate that there is a nonlinear relationship between performance drop-off and the ratio of requested data to requested data range. This is intuitive; if data is spread across a wider range of file locations it is likely to also be spread across a wider range of disk blocks, which will make caching less effective, as read ahead is less likely to have read the next desired block, and disk access less efficient, as the disk head must seek further and the on-disk cache is less likely to hold our data.

Figures 5.1 and 5.2 show a comparison between our model, assuming no caching, and our data from tests where cache was flushed between test runs. Figure 5.1 focuses on small accesses, while Figure 5.2 shows results for larger ones. For single block and strided accesses it is very accurate, but for small random block I/O we underestimate service time somewhat. This is due to the limited number of variables in our model and our desire to match to a wide range of values; in order to match well in the caching and large accesses

Table 5.2: Parameter Values

B_{dr}	4.0 Mbytes/sec
B_{drc}	9.2 Mbytes/sec
C_{dist}	0.9
P_{dist}	0.10
C_{req}	0.13
P_{req}	1.0
C_{cache}	0.65
T_{over}	0.02 sec

cases some accuracy was sacrificed in these cases. However, as we will see in the following sections, this will not affect the ability of the model to predict the appropriate scheduling algorithm.

In Figures 5.3 and 5.4 we see the model compared to observed performance with caching in effect for both small and large accesses. The model closely matches in all tested cases, although there is some deviation around the point at which request sizes begin to exceed cache size. We are limited in our ability to match to all workloads in this case by the use of a single set of parameters to model the transition into heavy disk access, but the deviation is not great and will be taken into account in the next section.

5.3 Algorithm Efficiency Functions

Where our simple model is able to effectively account for caching effects and the effects of non-ideal workloads on system performance, it would take a much more complicated system model to account for changes in scheduling that result from the application of our alternative algorithms. This is because the order in which requests are serviced and utiliza-

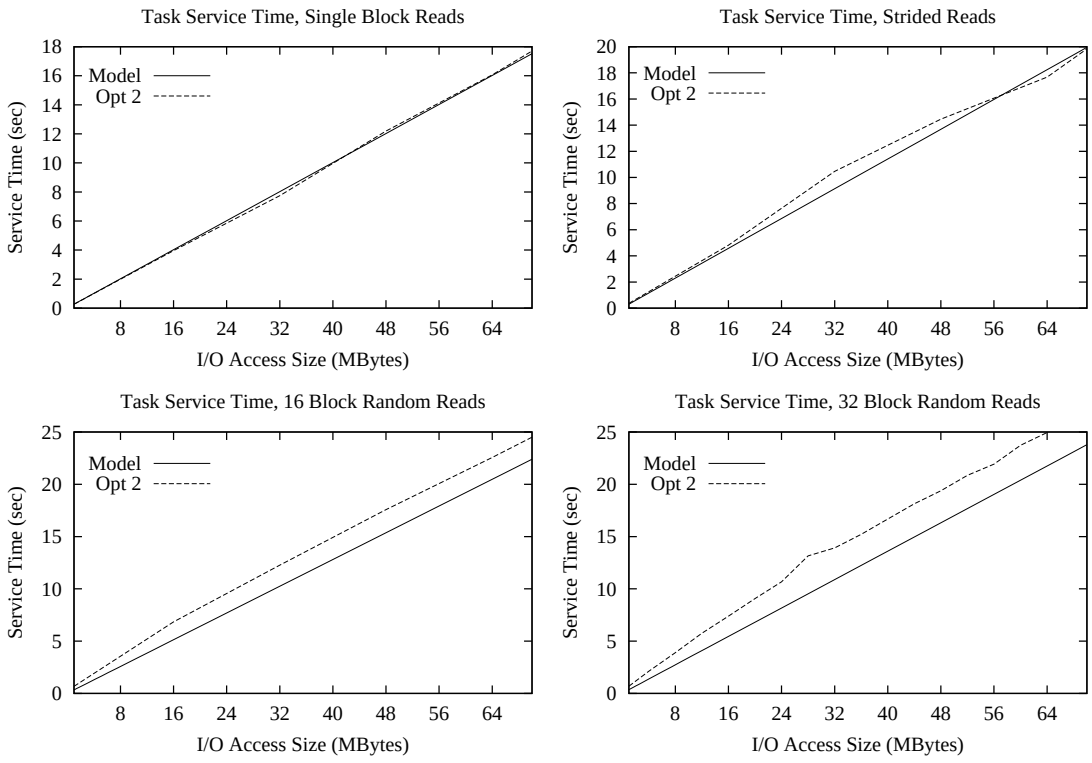


Figure 5.1: Comparing Opt 2 without Caching to Baseline Model for Small Accesses

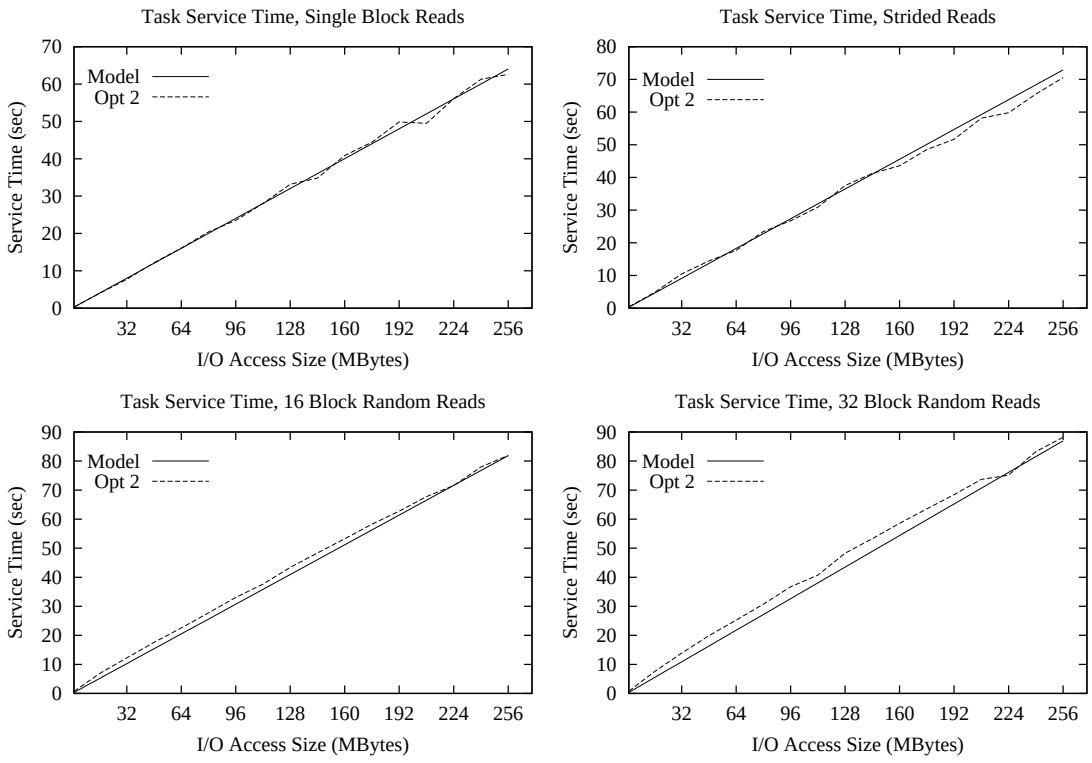


Figure 5.2: Comparing Opt 2 without Caching to Baseline Model for Large Accesses

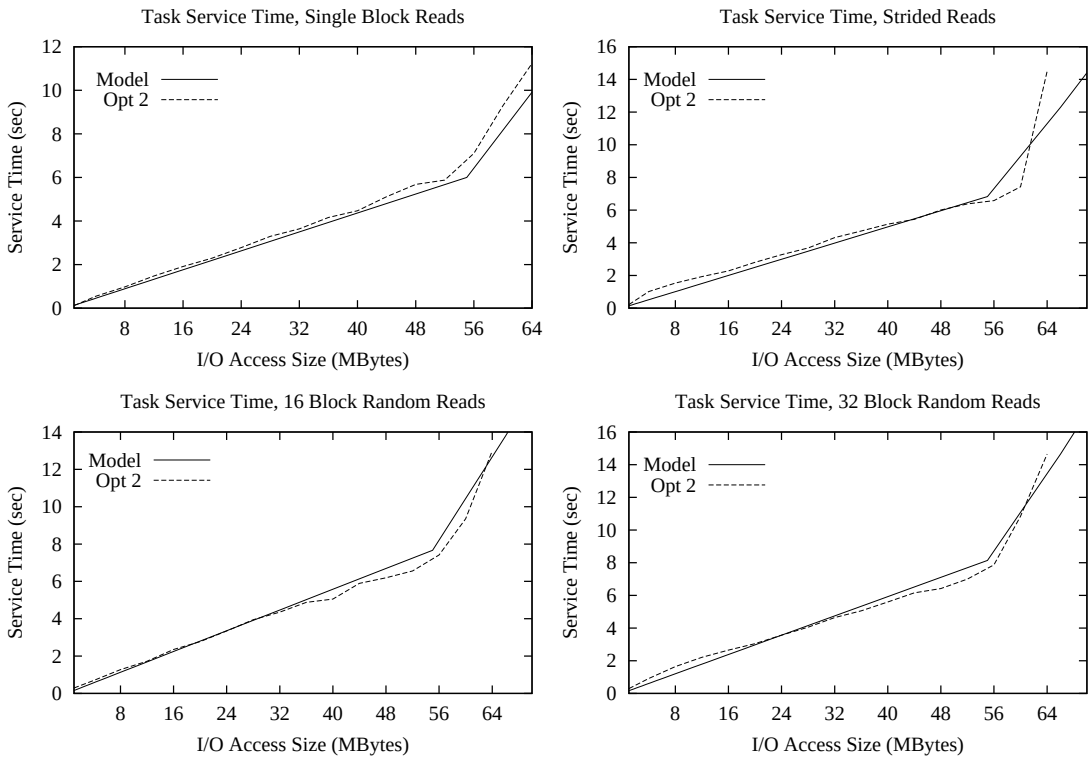


Figure 5.3: Comparing Opt 2 with Caching to Baseline Model for Small Accesses

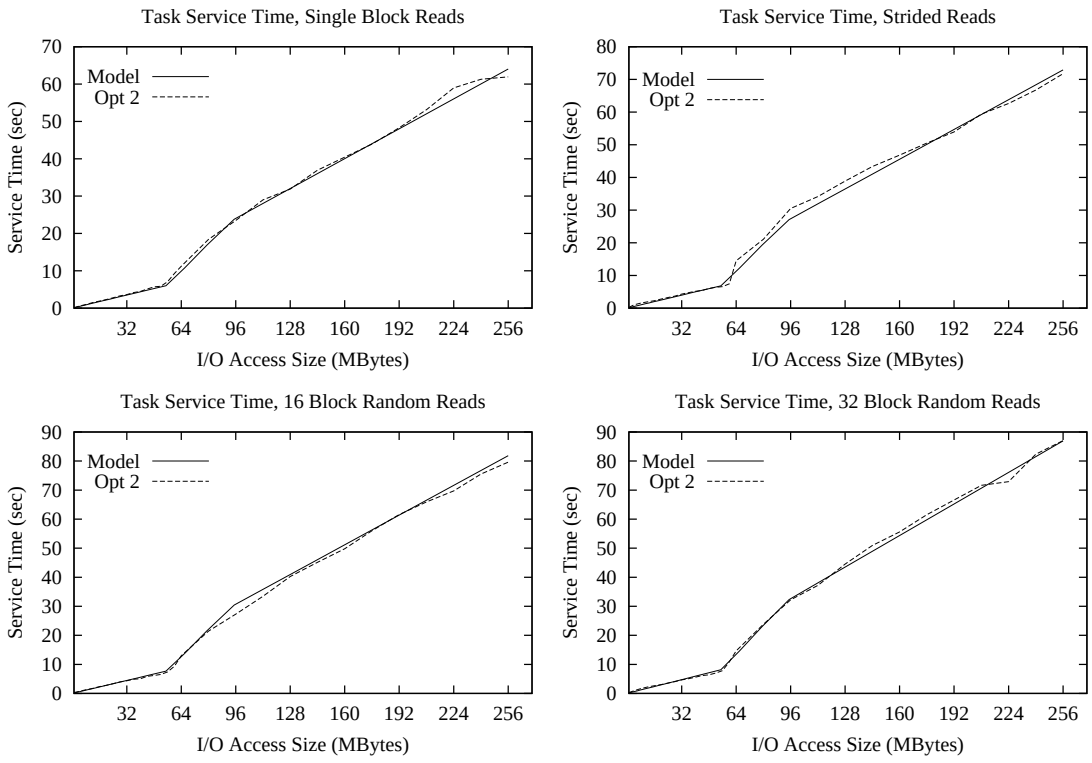


Figure 5.4: Comparing Opt 2 with Caching to Baseline Model for Large Accesses

Table 5.3: Optimization Efficiency Values

	Uncached			Cached		
Optimization	Ideal	Sparse	Disjoint	Ideal	Sparse	Disjoint
Opt 1	0.99	0.97	0.95	1.05	0.96	0.90
Opt 2	1.00	1.00	1.00	1.00	1.00	1.00
Opt 3	0.90	0.95	0.99	0.86	0.97	1.02
Opt 4	0.71	0.94	1.05	0.83	0.99	0.92

tion of cache become less easily predicted. Because of this we apply rule-based changes to our function based on choice of algorithm.

First we add an additional coefficient to our $T_{taskread}$ function:

$$T_{taskread} = T_{over} + E_{opt}E_{req}E_{dist}T_{ioread} \tag{5.6}$$

This coefficient will represent the effect of our algorithms on performance and will vary with workload and algorithm. Since our baseline model was tuned to match Opt 2, E_{opt} will be 1 for all cases where Opt 2 is in use. Table 5.3 presents the values we calculated based on our previous workload study. Workloads are characterized as being "ideal", "sparse", or "disjoint" based on the workload parameters. If $S_{req} < S_{cache}$ we use the coefficients in the cached columns and use the uncached ones otherwise.

5.4 Comparing Final Model to Data

In this section we compare our complete model to the data collected in our workload tests. Our goal is to establish that the model will correctly predict the fastest performing algo-

rithm for the majority, and ideally all, tested cases. We present data for cache-enabled behavior only; results are similar for the no cache case.

Figure 5.5 shows a side-by-side comparison of model output to observed workload behavior including caching for the range of tested sizes. The data is presented separately to increase readability; the same line type is used in each graph for the corresponding data and model output.

From comparing the graphs we see that the model can be used to predict the appropriate algorithm (shows the same algorithm giving the minimum service time) for all large accesses, with one notable exception. In the case of strided reads, the model incorrectly predicts that Opt 1 would be best for all sizes, when in fact there is a region between 64 Mbytes and 160 Mbytes for which Opt 4 would be better.

In Figure 5.6 we focus on small access sizes. Graphs from the workload section are reproduced here to allow for direct comparison. We see that the model again can be used to accurately predict the correct scheduling algorithm for most cases. One notable exception is strided accesses of around 24 Mbytes, where Opt 4 in practice performed slightly better than the predicted Opt 1. This fluctuation in the observed behavior is likely inconsistent, however, and in any case the performance difference at that point is minimal.

Overall this simple model provides an accurate estimation of system performance over a wide range of access sizes and workloads. The baseline model consists of a concise set of intuitive equations and parameters which can be used to model performance given a round-robin scheduling system. On top of this a set of rule-based modifiers is used to describe the effects of various algorithms on system performance.

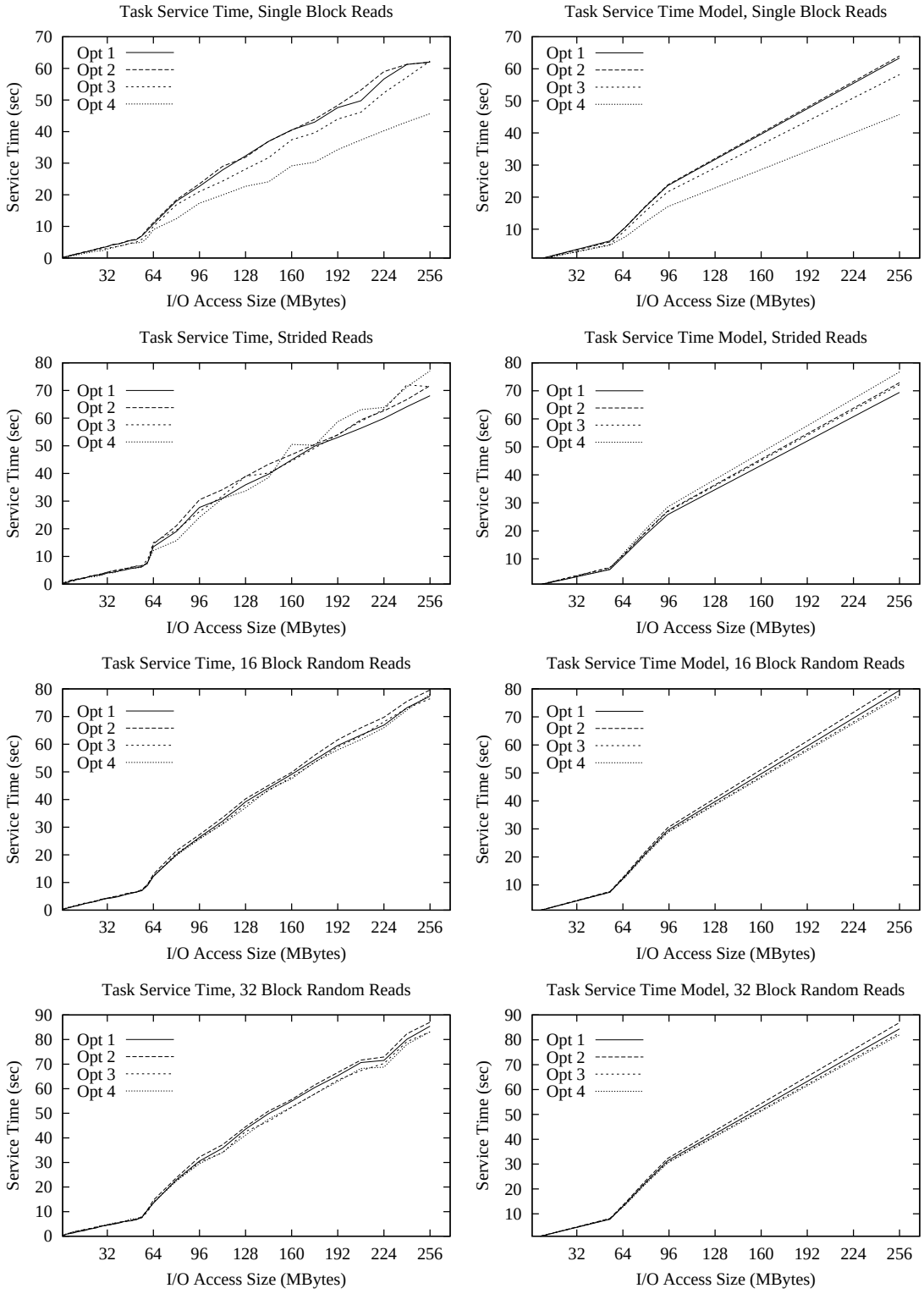


Figure 5.5: Comparing Observed Behavior to Model

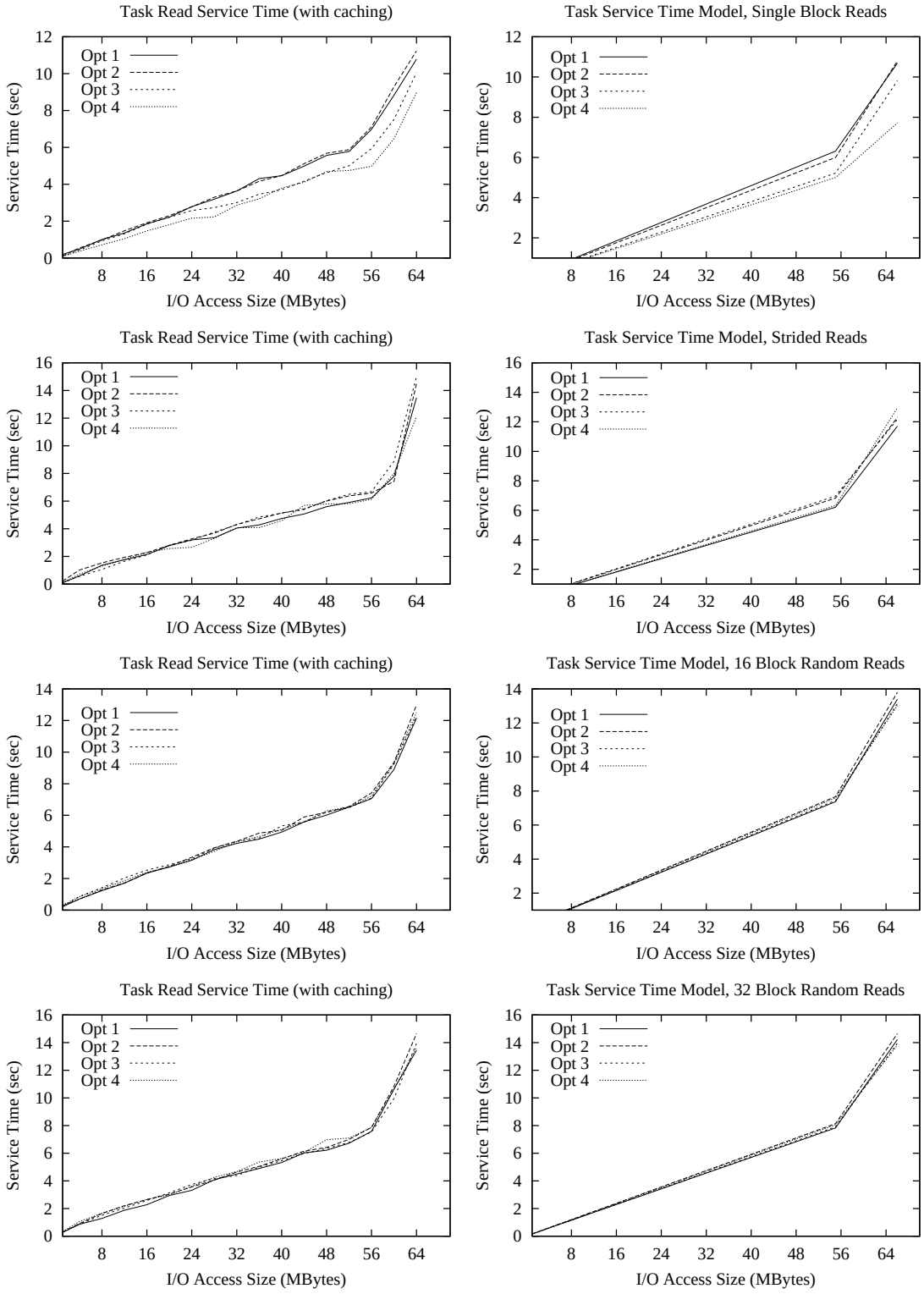


Figure 5.6: Comparing Observed Behavior to Model for Small Accesses

5.5 Implementing Reactive Scheduling with PVFS

As mentioned previously, the reactive scheduling (RS) system consists of three components. Two of these, the system model and the set of scheduling algorithms, have already been discussed. The remaining component is the selection mechanism. This component must supply the model with the necessary system state and predetermined constants and use the model to then select the best scheduling algorithm.

As a user space process, the PVFS I/O daemon is limited in its access to knowledge of system state outside its process space. This is not a problem with respect to workload information; the number, size, and characteristics of requests in service are all readily available. However, it is impossible for this process to reliably know what data is in cache. This limits our ability to implement the RS approach, as available cached data is an important input to our model. We will address this issue in our study in two ways. First, we will supply the system with accurate knowledge in a set of tests in order to ascertain the system's ability to operate in this ideal case. Following this we will make a worst case assumption that for all cases no cached data is available. This is a practical assumption to make for many systems and workloads considering our inability to gather accurate information on cache state. The system is re-tested under this assumption.

A separate concern is in accurately determining the number of tasks requesting, or about to request, data at any given time. Recall that our model takes as input the number of requests that make up a parallel I/O operation and also a total data size. However, requests from independent tasks do not arrive simultaneously in the PVFS system, which results in a period of time near the start of a collection of requests where the current request

statistics do not truly represent the operation about to take place. Likewise as requests are completed our state information, if it is based solely on requests in service, does not accurately describe the set of operations that are being completed.

A better indicator of the number of tasks likely to perform I/O at some point in time is needed as an input. Recognizing that our workloads are parallel ones, we note that the number of tasks holding a file open is a good indicator of the number of tasks likely to perform I/O simultaneously. Based on this assumption we perform a small number of calculations using the available workload data to predict accurate model values. Let us define a number of observed values:

- N_{file}, the number of tasks holding a given file open
- N_{req_o}, the number of observed requests
- S_{req_o}, the total size of observed requests
- D_{req_o}, the number of disjoint regions in observed requests
- R_{req_o}, the total range of file positions in observed requests

These observed values are used to calculate the model parameters as follows:

$$N_{req} = N_{file} \tag{5.7}$$

$$S_{req} = S_{req_o} \frac{N_{file}}{N_{req_o}} \tag{5.8}$$

$$D_{req} = D_{req_o} \frac{N_{file}}{N_{req_o}} \tag{5.9}$$

$$R_{req} = max\{R_{req_o}, S_{req_o}\} \tag{5.10}$$

These values are passed to the model in order to perform the necessary calculations. An expected task service time is calculated for each of the algorithms, and the one returning the smallest service time is chosen. This process is repeated on each new request.

5.6 Results of Utilizing Reactive Scheduling

In this section we detail the results of tests utilizing reactive scheduling in PVFS. The test applications and system are the same used in Chapter 4.

In our first set of tests we fixed the reported available cache at 56 Mbytes, which was the value we used in our model testing in Chapter 5. We then repeated our original workload tests allowing data to remain in cache between runs. The results are shown in Figures 5.7 and 5.8 separately for small and large accesses compared to the best performance seen in our workload study. Here we see that indeed our RS implementation is able to adapt to the previously studied workloads, providing performance comparable to the best seen in the study. The only exception seen is in the case of small 32 block random accesses. After studying this effect, we determined that the request sizes were so small that the model was reporting a zero value for the time for all algorithms and was defaulting to Opt 4, an implementation detail. This effect could be eliminated by choosing to default to a more practical algorithm for this situation, such as Opt 1, which is more likely to be a good choice when small accesses are encountered.

In our second set of tests we fixed the reported available cache at 0 Mbytes and repeated our workload tests, flushing cache between runs. The results are shown in Figure 5.9, again

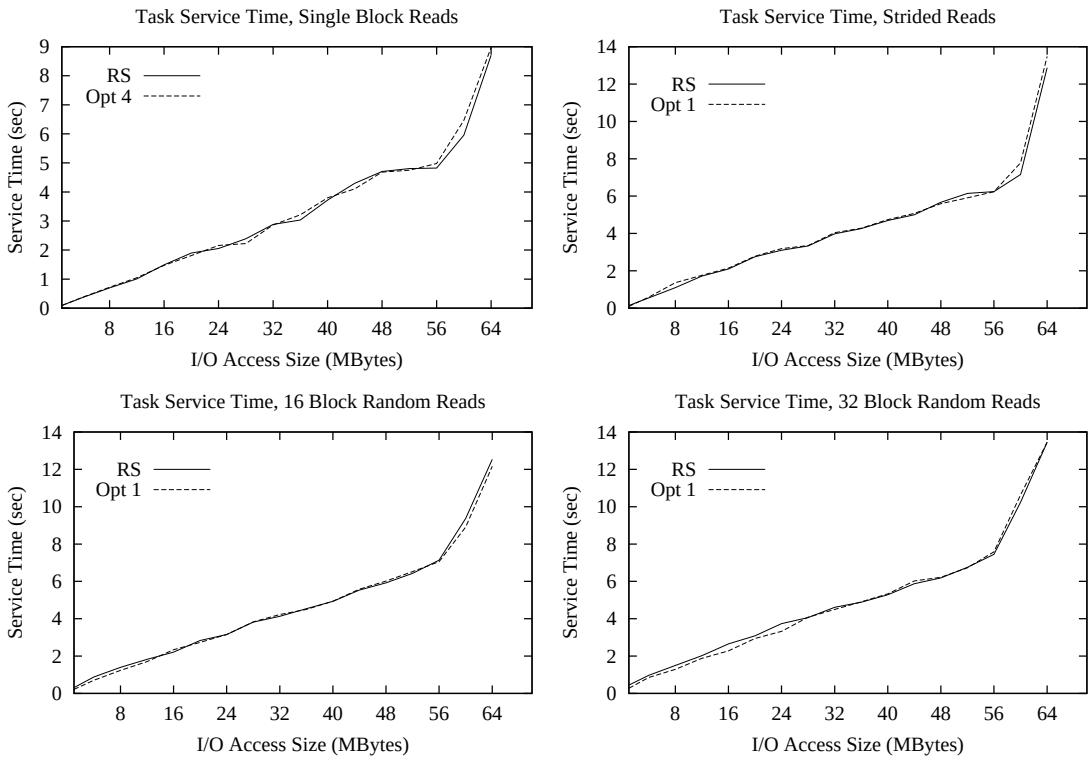


Figure 5.7: RS Performance with Caching for Small Accesses

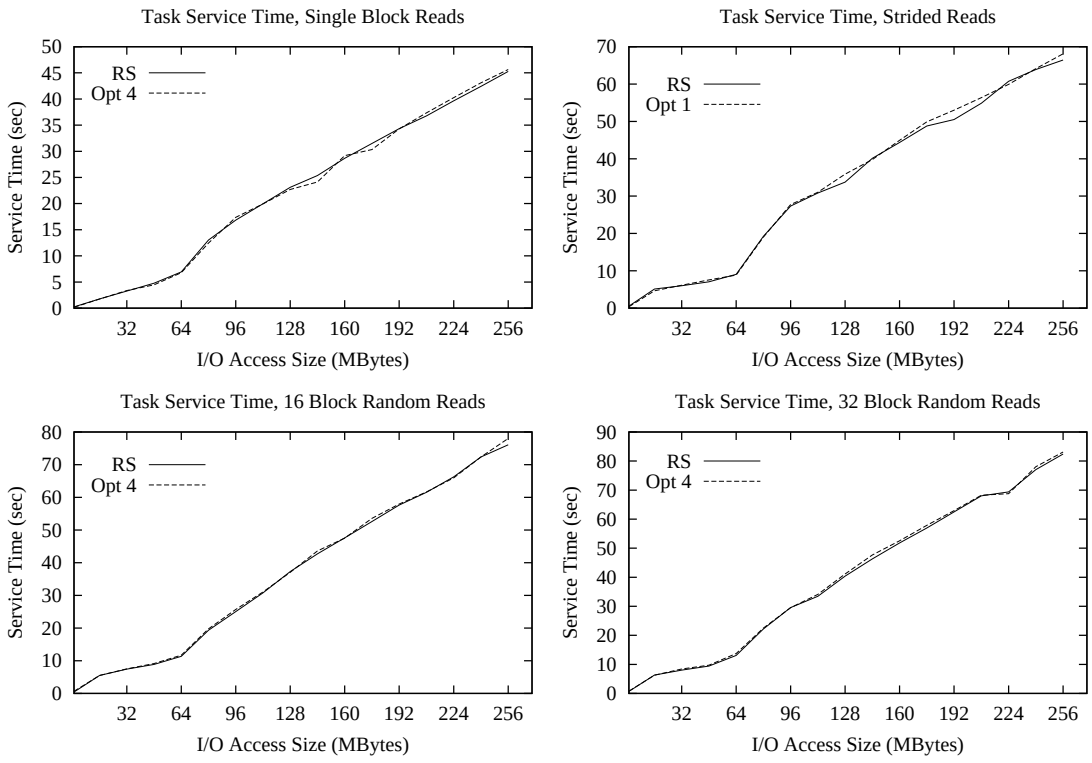


Figure 5.8: RS Performance with Caching for Large Accesses

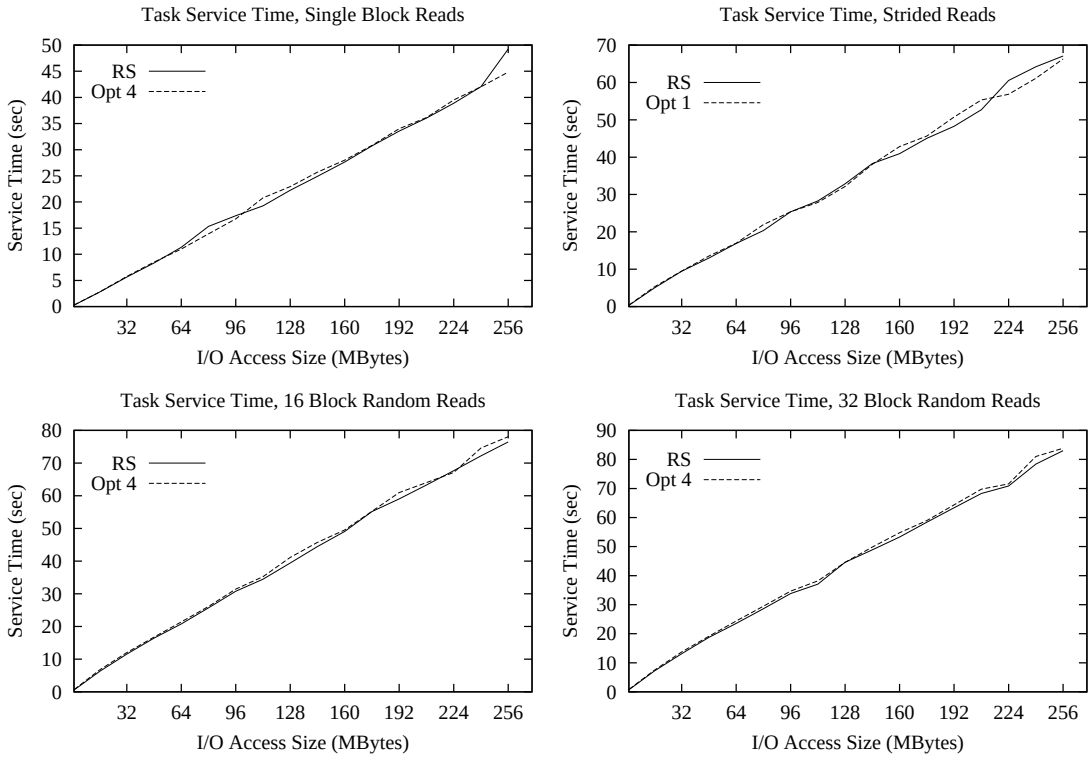


Figure 5.9: RS Performance without Caching for Small and Large Accesses

compared to the best algorithm seen in the workload study. Again RS is able to make the correct algorithm selection.

In our final set of tests we fixed the reported available cache at 0 Mbytes and reran our workload tests, allowing cached data to remain between runs. This simulates performance in a real environment with cache active but cache utilization data unavailable. The performance of RS under these conditions is compared to the best case run for each of the four workloads in Figures 5.10 and 5.11 for small and large accesses respectively. Here we see that for large sizes we perform comparably, but for small random block accesses we perform less well. This is because Opt 1 is the best performing algorithm for cached random

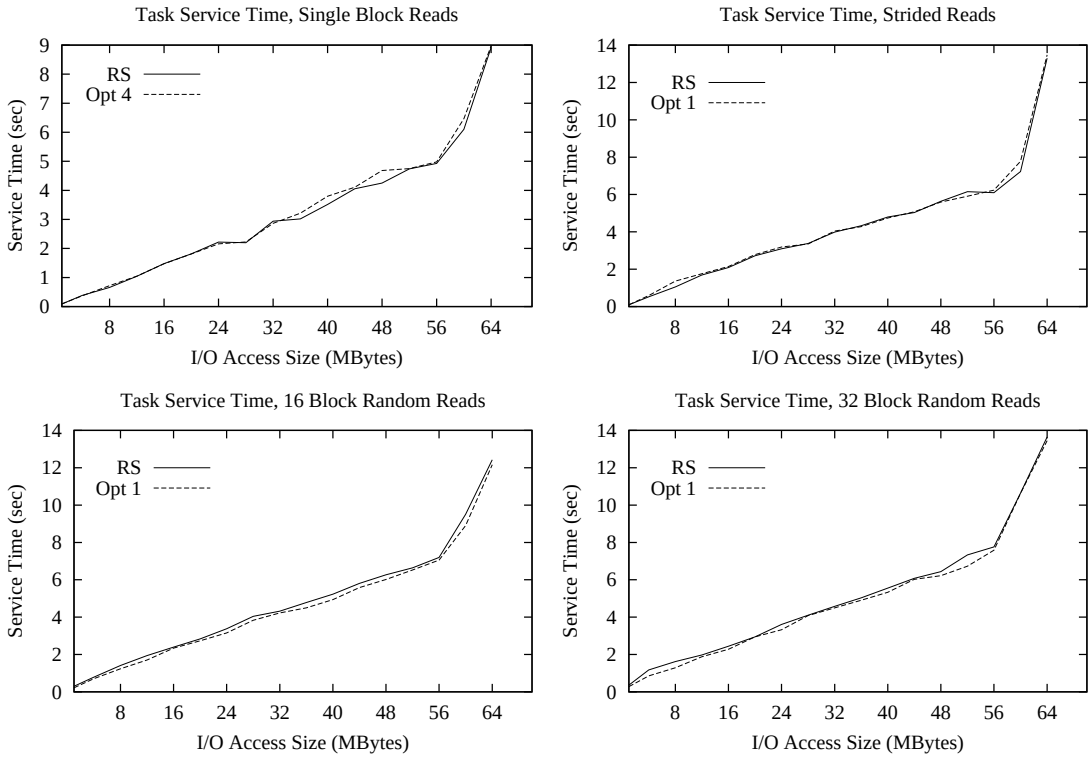


Figure 5.10: RS Performance for Small Accesses Assuming No Cache

blocks, while Opt 4 is the best for uncached data (which is what our selection mechanism thinks is the state).

5.7 Conclusions

The model presented combines a simple behavioral model of the system under different workloads with a set of heuristics that account for the effects of varying scheduling algorithms. We have encapsulated important workload characteristics, especially sparse data access and disjoint requests, into the model as well. This model accurately predicts system

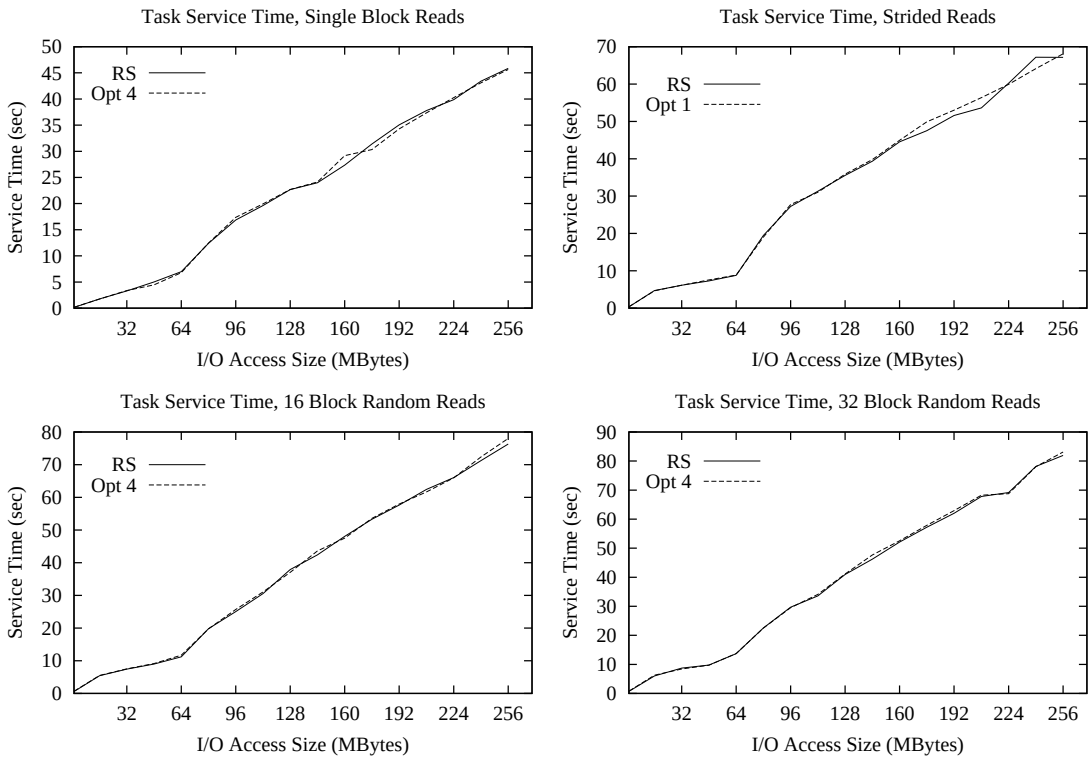


Figure 5.11: RS Performance for Large Accesses Assuming No Cache

performance for the workloads tested and thus provides a solid basis for our RS implementation.

Building on the system model and scheduling algorithms presented previously, we implemented a selection mechanism to complete our RS implementation for PVFS. In this case it was necessary for the selection mechanism to perform some estimation of model parameters based on system state.

We have shown that, provided accurate cache values, the RS system is able to apply the appropriate algorithm for the tested workloads. Furthermore, since these workloads cover a range of important characteristics, including disjoint access and sparse requests, we can expect that performance would be optimal for many similar workloads.

Without accurate cache information we must make conservative assumptions, such as assuming that we will not ever access data already in cache. This assumption will result in optimal behavior for large requests, as we know the system can accurately choose algorithms for this range.

From our workload study we realize that there is anywhere from a 6% to a 60% difference in file system performance simply due to scheduling algorithm selection. By applying RS, we are able to avoid situations where a single algorithm would fail and thus monopolize on the advantages of specific algorithms. This is done automatically within the system. This means that end users will see task service times drop without any changes to the system components or their applications. Since the RS system uses general workload characteristics as its basis for operation, it should operate across a wide variety of workloads. In fact, the ability of the RS system to adapt to changing workloads means that it can adapt

even while collections of requests are in progress, increasing the potential for performance gains.

5.8 Future Work

One obviously useful capability in a production implementation would be the ability to query the system for cache information on a by-file or by-disk block basis. This would allow us to obtain this needed data at run time, improving our accuracy.

The use of physical block locations rather than logical file ones when scheduling would likely result in improved performance as well, particularly if we also allowed requested data ordering to be chosen dynamically (as opposed to the fixed logical ordering imposed in the current system).

Finally, combining RS with a hint mechanism would further improve the accuracy of the inputs passed to the model, assuming we could trust the hints. Hints could provide exact values for S_{req}, N_{req}, D_{req}, and R_{req} for a parallel operation directly.

Chapter 6

Conclusions

As processor speeds and cluster sizes increase, it becomes ever more important for parallel I/O systems to extract maximum performance from their underlying hardware. Reactive Scheduling is one piece of the performance puzzle; it provides a novel technique for improving performance in parallel I/O systems. RS is complementary to existing optimization methods such as those that concentrate on caching, prefetching, and writeback strategies.

First we presented the Parallel Virtual File System, a parallel file system build by us from the ground up for use in parallel I/O research in cluster environments. PVFS has grown from a research toy into a popular tool for parallel computing and is in use at a number of facilities around the world. PVFS is rapidly becoming the de facto standard for parallel file systems on Linux clusters, and continued development should strengthen this position.

Next we examined the performance of a set of workloads on PVFS using a collection of scheduling algorithms. We noted that the "best" algorithm for any given situation depended on the goal of the system (eg. fastest performance, lowest variance), the state of the system

(eg. available cache), and the workload in progress. The only other scheduling algorithm study performed in a parallel I/O environment to our knowledge was using disk-directed I/O (DDIO), and they found DDIO to be optimal in all tested cases, making this an interesting result. We quantified which algorithm performed best for each of our workloads for conditions where caching was available and for conditions where it was not. We also noted two particular workload characteristics, the presence of requests with multiple disjoint regions and sparse data access, which had noticeable impact on system performance. These two characteristics are explicitly integrated into our system model.

Using our workload study as a basis we designed and tuned a system model to represent system performance. We started with a simple behavioral model which represented system performance using a round-robin scheduling algorithm and incorporated workload and system effects. To account for the complexities of algorithm changes, we used a set of heuristics to predict expected performance. We compared the model output to collected data and showed it to be accurate. This model is used in our RS implementation to predict the appropriate scheduling algorithm. This approach is different from those used in other adaptive parallel I/O systems, who instead have used more complicated Markov models or neural networks.

Finally we brought the file system, model, and algorithms together into a single RS implementation within PVFS. This implementation correctly predicted the best scheduling algorithm for all our tested workloads, in real time, given one supplied input – cache availability – which was not available accurately in the PVFS environment. Without this value we were able to operate optimally for almost all cases by assuming that no cache was available. In our best case, as shown in Figure 6.1, the RS approach provided a 24-

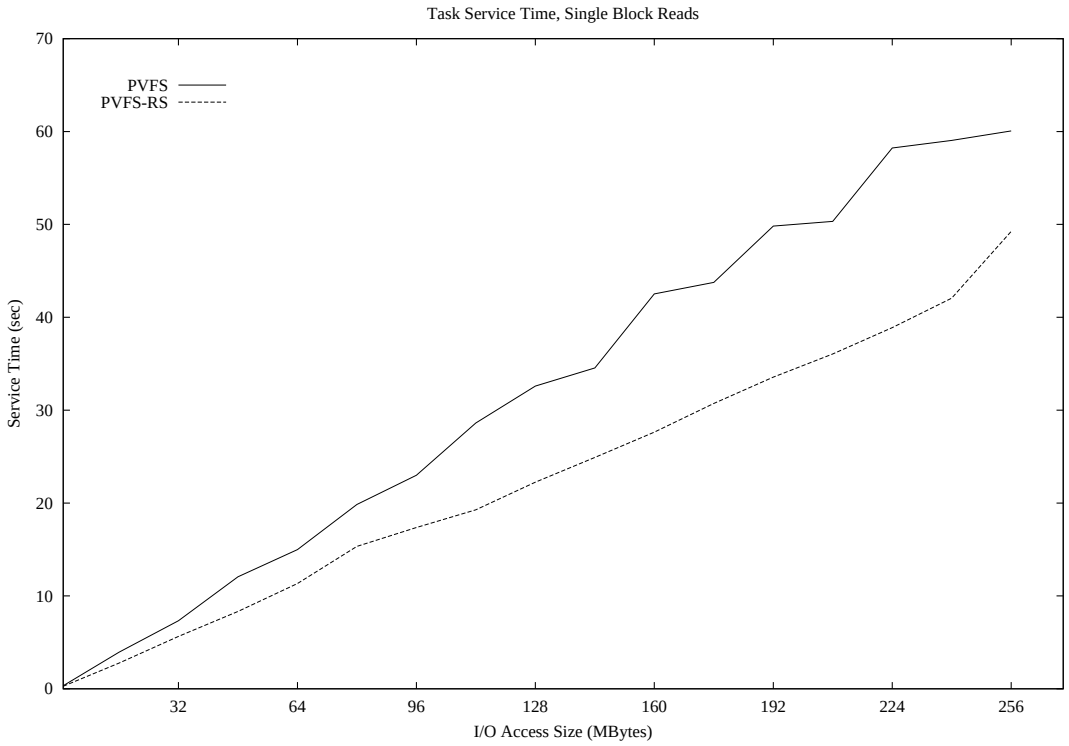


Figure 6.1: Best Case RS Performance

60% reduction in mean service times over the original PVFS system for requested sizes of 1 Mbyte and larger. This is due to the correct application of disk-oriented scheduling techniques for block accesses.

This work combines theoretical modeling and scheduling with practical issues of real time operation and data availability to create a solution to an important problem – providing high performance parallel I/O. RS is a new technique for performance improvement in this environment; real-time scheduling algorithm selection has not been shown to work or even researched before. The solution should be applicable to a variety of such systems; indeed, the model can be tuned to match a number of networking and disk technologies, and the heuristic system for accounting for scheduling algorithms can be tuned to match

new algorithms. Additionally we enumerated a number of workload characteristics which partially determine system performance, and pointed out that these characteristics have an impact on algorithm selection as well.

The issue of workloads and their effect on algorithm selection, while not the central point of this work, is a ripe area of study in parallel I/O. This work serves only as a starting point, and further research in this area will undoubtedly help further quantify the characteristics of workloads which effect algorithm selection and the range of workloads for which given algorithms are appropriate. On the practical side, increasing the knowledge of system state in PVFS would allow for more accurate prediction. This would be an excellent area of study, and it would likely lead to a more reliable RS implementation.

APPENDICES

Appendix A

Original System Model

In this chapter we discuss a more thorough system model which might serve as the basis of a more effective system model in the future. The model presented here is a straight forward one which considers the system from the point of view of the I/O server and includes disk, network, and memory components. It should suffice for many simple network and storage topologies, but it is by no means meant to model all possible systems.

We begin in Section A.1 by developing a simple model for single requests which utilizes constant values for disk and network performance. In Section A.2 we extend the model to account for multiple requests. In Section A.3 we relax our assumption of constant values for resource performance and build equations which allow us to calculate expected performance based on workload.

A.1 Single Request Model

In a parallel file system, two resources are usually the potential bottlenecks: network devices and disks. We will denote the expected value of raw network bandwidth seen by the file system server as B_{net}, the expected value of raw disk write bandwidth as B_{dw}, and the expected value of raw disk read bandwidth as B_{dr}. We will assume a normal distribution for these variables. Given these values we can roughly estimate the effective bandwidth to the file system, B_{write}:

$$B_{write} = \min\{B_{net}, B_{dw}\} \quad \text{(A.1)}$$

$$B_{read} = \min\{B_{net}, B_{dr}\} \quad \text{(A.2)}$$

Here we are simply saying that the effective bandwidth is the minimum of the system's effective bandwidth to either of the resources. This assumes that we can completely overlap communication and I/O. Assuming a request size S_{req} we can similarly estimate the time for a single write or read data transfer, T_{wio} and T_{rio} respectively, with:

$$T_{wio} = \frac{S_{req}}{B_{write}} \quad \text{(A.3)}$$

$$T_{rio} = \frac{S_{req}}{B_{read}} \quad \text{(A.4)}$$

The *setup time* to transfer the request and prepare for data transfer, T_{ws} or T_{rs} for writes or reads respectively, is also a factor. We will define new times which include both this

setup time and the data I/O time, T_{wx} and T_{rx}, for read transfers and write transfers:

$$T_{wx} = T_{ws} + T_{wio} \tag{A.5}$$

$$T_{rx} = T_{rs} + T_{rio} \tag{A.6}$$

To obtain a more accurate model, a term should be included to adjust for the fraction of time that disk and network I/O actually occur simultaneously, or *overlap* time. In our case we will indicate this as a fractional value which will be denoted as O_{write} for write transfers and O_{read} for read ones. We will discuss the estimation of these values later in this section.

$$T_{wio} = \frac{O_{write}S_{req}}{\min\{B_{net}, B_{dw}\}} + \frac{(1-O_{write})S_{req}}{B_{net}} + \frac{(1-O_{write})S_{req}}{B_{dw}} \tag{A.7}$$

$$= \frac{O_{write}S_{req}}{\min\{B_{net}, B_{dw}\}} + \frac{(1-O_{write})(B_{net}+B_{dw})S_{req}}{B_{net}B_{dw}} \tag{A.8}$$

$$T_{rio} = \frac{O_{read}S_{req}}{\min\{B_{net}, B_{dr}\}} + \frac{(1-O_{read})(B_{net}+B_{dr})S_{req}}{B_{net}B_{dr}} \tag{A.9}$$

This says that for some transfer request of size S_{req}, given some fractional overlap of network and disk transfer for writes or reads, we can calculate the expected time for the I/O portion of the transfer as a function of the effective bandwidths of network and disk.

This overlap between communication and disk I/O is related to the current state of available memory because there must be memory available to hold buffers in order for overlap to be possible. We can categorize this memory into three distinct types of interest to us:

- free, unallocated memory (or memory holding useless file data), M_{free}

- dirty buffers, M_{dirty}
- clean buffers holding useful file data, M_{clean}

Free memory is useful for both reads and writes, allowing for further prefetching in the case of the former and buffering of data in the latter. Dirty buffers are in some sense of more detriment than not having free memory at all; these buffers will often need to be written during the time we are performing our data transfer. Clean buffers with file data are especially useful in that they can be used to avoid disk reads in some cases, but can be converted to free memory for use in buffering writes as well. In this discussion we will use the terms buffers and cache interchangeably to refer to physical memory used for holding file data.

We now define two new sizes, S_{disk} and S_{cache}, which correspond to the data accessed via disk and cache respectively. In all cases $S_{req} = S_{disk} + S_{cache}$. For writes the available cache is $M_{free} + M_{clean}$, so:

$$S_{cachewr} = \min\{S_{req}, (M_{free} + M_{clean})\} \tag{A.10}$$

For reads the available cache data is based on a fraction of the number of buffers holding file data:

$$S_{cacherd} = \min\{S_{req}, hit \times (M_{clean} + M_{dirty})\} \tag{A.11}$$

where hit is a fraction indicating the probability a given byte is cached.

These defined we can look at our overlap for read and write transfers. We assume that overlap behavior is dominated by available cache:

$$O_{write} = \min\{\frac{S_{cachewr}}{S_{req}}, 1\} \quad \text{(A.12)}$$

$$O_{read} = \min\{\frac{S_{cacherd}}{S_{req}}, 1\} \quad \text{(A.13)}$$

At some point in time dirty buffers must be written to disk. In many systems a daemon such as "bdflush" is responsible for this activity. We will model this behavior with two parameters, an amount of dirty buffers flushed periodically S_{bf}, and an interval between writes T_{bf}. What this really does is lower our effective disk bandwidth by stealing time from our active disk requests. We can then calculate an expected amount of time we will spend flushing buffers during a request:

$$T_{wf} = \frac{1}{B_{dw}} \min\{M_{dirty}, \frac{T_{wio}}{T_{bf}} S_{bf}\} \quad \text{(A.14)}$$

$$T_{rf} = \frac{1}{B_{dr}} \min\{M_{dirty}, \frac{T_{rio}}{T_{bf}} S_{bf}\} \quad \text{(A.15)}$$

These simply take into account the average number of times the flushing would occur and the size of the data that is written each time, with a maximum amount of data, M_{dirty}, written out due to dirty buffers during the time I/O transfer is taking place. We will assume that this time is not overlapped in our equations.

Now we can revisit our transfer time equations, including this new term:

$$T_{wx} = T_{ws} + T_{wf} + T_{wio} \quad \text{(A.16)}$$

$$T_{rx} = T_{rs} + T_{rf} + T_{rio} \quad \text{(A.17)}$$

Now we have an estimate of read and write transfer times for single requests, taking into account caching and dirty buffer write-back effects.

A.2 Multiple Request Model

In the case where multiple transfers of the same type are occurring simultaneously, the most obvious effect will be the increase in start up time, which will increase linearly with the number of connections N assuming independent requests and a fixed startup time. If we consider S_{req} in the previous requests to be the aggregate size, then our new equation becomes simply:

$$T_{wx} = NT_{ws} + T_{wf} + T_{wio} \quad \text{(A.18)}$$

$$T_{rx} = NT_{rs} + T_{rf} + T_{rio} \quad \text{(A.19)}$$

So now we have an expression for the time required to service N requests totalling S_{req} bytes of data, taking into account the time to receive and parse the requests, the amount of memory available which will determine overlap, and the amount of dirty blocks which will be written out during the transfer. We will not extend our model to simultaneous read and write traffic in this work.

A.3 Improved Resource Performance Modeling

Given the model of our system described above, we can begin to examine the relative performance of our resources and thus predict what resources will limit performance. Again we will begin by assuming that network, disk read, and disk write bandwidths are normally distributed with expected values B_{net}, B_{dr}, and B_{dw} respectively.

We can define an expected network transmit time T_{net}:

$$T_{net} = \frac{S_{req}}{B_{net}} \tag{A.20}$$

Likewise we can predict the disk transfer times for writing and reading, taking into account the availability of cache and the overhead of any flushing:

$$T_{dw} = \frac{(1 - O_{write})S_{req}}{B_{dw}} + T_{wf} \tag{A.21}$$

$$T_{dr} = \frac{(1 - O_{read})S_{req}}{B_{dr}} + T_{rf} \tag{A.22}$$

Now we have the basic components that we need in order to determine which resource will be a bottleneck in our system by directly comparing the expected service times.

The importance of acknowledging the roles of B_{net}, B_{dw}, and B_{dr} in overall performance is that these values are not actually constant and are interrelated. This is true for a number of reasons, including CPU utilization and access characteristics. Traditionally the application of scheduling algorithms has resulted in improvements in effective bandwidth in numerous situations. However, in this model there are multiple resources, and attempt-

Table A.1: Network Model Parameters

S_{maxnet}	Size at which network performance reaches maximum bandwidth
S_{minnet}	Size at which network performance reaches minimum bandwidth
B_{maxnet}	Maximum network bandwidth
B_{minnet}	Minimum network bandwidth

ing to create optimal schedules for all of them would be a complex endeavor. Instead we will optimize for the slowest resource, which is most likely to improve our throughput.

Now let us consider the disk and network bandwidth at some point in time, which will be denoted as b_{net}, b_{dw}, and b_{dr}. We will approximate them based on parameters of the transfer and knowledge of the system architecture and state. Other architectures might require different equations to represent the behavior of these resources, and certainly more complex, more accurate models could be developed than the ones here.

For our network model we will assume that the performance will improve from some minimum to some peak value based solely on the size of the request(s). We will use four parameters to characterize network performance, which are described in Table A.1 and shown in Figure A.1a.

$$b_{net} = \begin{cases} B_{minnet} & \forall\ S_{req} \leq S_{minnet} \\ \frac{(S_{req}-S_{minnet})(B_{maxnet}-B_{minnet})}{(S_{maxnet}-S_{minnet})} + B_{minnet} & \forall\ S_{minnet} < S_{req} < S_{maxnet} \\ B_{maxnet} & \forall\ S_{req} \geq S_{maxnet} \end{cases} \quad \text{(A.23)}$$

Our disk model will also utilize the total request size in determining performance. Performance will be modeled as degrading beyond a certain aggregate size down to a minimum performance. The four parameters used in our disk model are described in Table A.2 and

Table A.2: Disk Model Parameters

$S_{maxdisk}$	Size at which disk performance reaches maximum bandwidth
$S_{mindisk}$	Size at which disk performance reaches minimum bandwidth
$B_{maxdisk}$	Maximum disk bandwidth
$B_{mindisk}$	Minimum disk bandwidth

shown in Figure A.1b.

$$b_{disk} = \begin{cases} B_{maxdisk} & \forall\ S_{req} \leq S_{mindisk} \\ \dfrac{(S_{req} - S_{mindisk})\,(B_{mindisk} - B_{maxdisk})}{(S_{maxdisk} - S_{mindisk})} + B_{mindisk} & \forall\ S_{mindisk} < S_{req} < S_{maxdisk} \\ B_{mindisk} & \forall\ S_{req} \geq S_{maxdisk} \end{cases} \quad \text{(A.24)}$$

Assume at some time t we have:

- N independent requests, all either reads or writes,
- S_{req} total data size, and
- M_{free}, M_{clean}, and M_{dirty} units of free, clean, and dirty buffers respectively.

Using the equations derived previously in this chapter we obtain expected disk and network bandwidth values. Using these values we select one of a number of transfer algorithms. The algorithms are ordered based on their concentration on a certain resource. The ratio of disk to network performance is used to select one algorithm. This operation is repeated as necessary, either as new requests are received, as requests are serviced, or as time passes.

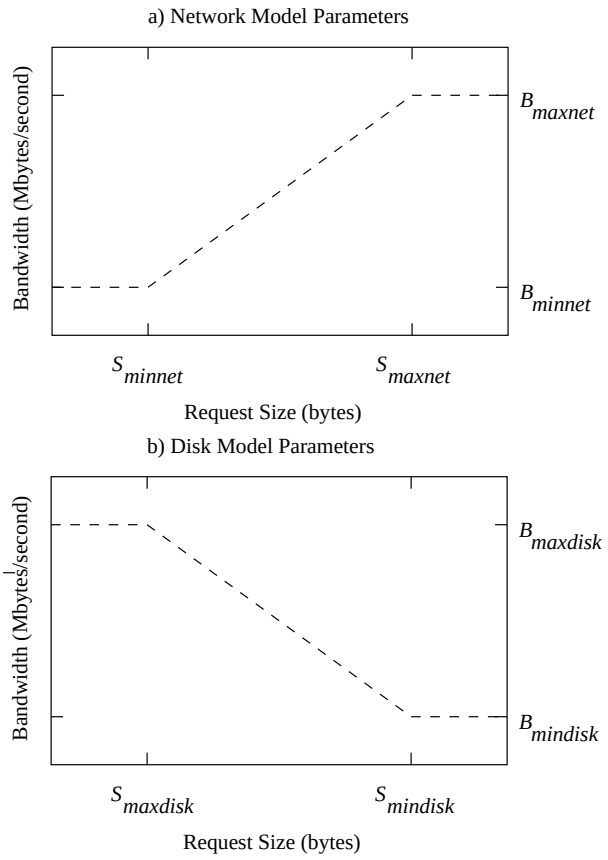


Figure A.1: Model Parameters

Table A.3: Model Variables

Predetermined Constants	
S_{bf}	Amount of data written on one pass of dirty buffer flushing
T_{bf}	Time between dirty buffer flush operations
T_{ws}	Setup time for a write operation
T_{rs}	Setup time for a read operation
$S_{maxdisk}$	Size at which disk performance reaches maximum bandwidth
$S_{mindisk}$	Size at which disk performance reaches minimum bandwidth
$B_{maxdisk}$	Maximum disk bandwidth
$B_{mindisk}$	Minimum disk bandwidth
S_{maxnet}	Size at which network performance reaches maximum bandwidth
S_{minnet}	Size at which network performance reaches minimum bandwidth
B_{maxnet}	Maximum network bandwidth
B_{minnet}	Minimum network bandwidth
System State Values	
M_{free}	Unused memory
M_{clean}	Memory holding buffers which do not need to be written
M_{dirty}	Memory holding buffers which need to be written to disk
S_{req}	Total size of requests in service
N_{req}	Number of requests in service

A.4 System Model Implementation

In order to implement our system model we must integrate system state information, estimated disk and network performance values, and the equations we previously derived. Table A.3 summarizes the values used in this process.

The calculation of T_{net} is as follows in this model:

$$T_{net} = \frac{S_{req}}{b_{net}} \tag{A.25}$$

This function does not include a component representing the time to acknowledge requests; this is because this time impacts both disk and network performance equally. The value of b_{net} is found using the equations presented in Section A.3.

For a pure write workload we estimate T_{dw} with:

$$
\begin{aligned}
T_{dw} &= \frac{(1-O_{write})S_{req}}{b_{dw}} + T_{wf} && \text{(A.26)}\\
&= \frac{(1-O_{write})S_{req}}{b_{dw}} + \frac{1}{b_{dw}}\min\{M_{dirty}, \frac{T_{wio}}{T_{bf}}S_{bf}\} && \text{(A.27)}\\
&= \frac{1}{b_{dw}}\left((1-O_{write})S_{req} + \min\left\{M_{dirty}, \frac{S_{req}S_{bf}}{T_{bf}\min\{b_{net}, b_{dw}\}}\right\}\right) && \text{(A.28)}\\
&= \frac{1}{b_{dw}}\left(\left(1-\min\left\{\frac{S_{cachewr}}{S_{req}}, 1\right\}\right)S_{req} + \min\left\{M_{dirty}, \frac{S_{req}S_{bf}}{T_{bf}\min\{b_{net}, b_{dw}\}}\right\}\right) && \text{(A.29)}\\
&= \frac{1}{b_{dw}}\left((S_{req} - \min\{S_{cachewr}, S_{req}\}) + \min\left\{M_{dirty}, \frac{S_{req}S_{bf}}{T_{bf}\min\{b_{net}, b_{dw}\}}\right\}\right) && \text{(A.30)}\\
&= \begin{array}{l}\frac{1}{b_{dw}}\left(S_{req} - \min\left\{\min\left\{S_{req}, M_{free}+M_{clean}\right\}, S_{req}\right\}\right.\\ \left.+\min\left\{M_{dirty}, \frac{S_{req}S_{bf}}{T_{bf}\min\{b_{net}, b_{dw}\}}\right\}\right)\end{array} && \text{(A.31)}\\
&= \frac{1}{b_{dw}}\left(S_{req} - \min\{S_{req}, M_{free}+M_{clean}\} + \min\left\{M_{dirty}, \frac{S_{req}S_{bf}}{T_{bf}\min\{b_{net}, b_{dw}\}}\right\}\right) && \text{(A.32)}
\end{aligned}
$$

The first two terms within the parenthesis represent the amount of data that will not fit in cache, while the final term within the parenthesis represents the amount of data that will be written due to dirty buffer flushing.

For a pure read workload:

$$
\begin{aligned}
T_{dr} &= \frac{(1-O_{read})S_{req}}{b_{dr}} + T_{rf} && \text{(A.33)}\\
&= \frac{(1-O_{read})S_{req}}{b_{dr}} + \frac{1}{b_{dr}}\min\{M_{dirty}, \frac{T_{rio}}{T_{bf}}S_{bf}\} && \text{(A.34)}\\
&= \frac{1}{b_{dr}}\left((1-O_{read})S_{req} + \min\{M_{dirty}, \frac{S_{req}S_{bf}}{B_{read}T_{bf}}\}\right) && \text{(A.35)}
\end{aligned}
$$

$$= \frac{1}{b_{dr}} \left((1 - O_{read}) S_{req} + \min \left\{ M_{dirty}, \frac{S_{req} S_{bf}}{T_{bf} \min \{b_{net}, b_{dr}\}} \right\} \right) \quad \text{(A.36)}$$

Once these calculations have been made, we can estimate the time to complete a set of transfers by calculating T_{wx}, which can now be written as:

$$T_{wx} = NT_{ws} + \max\{T_{net}, T_{dw}\} \quad \text{(A.37)}$$

Appendix B

Model Parameter Values

In this chapter we discuss how one might select the parameters used by our system model. This discussion will be broken into three parts. First we will discuss how to find the baseline parameters; those parameters which define the ideal behavior of the system using Opt 2. Next we will discuss how to calculate the workload effect parameters which account for disjoint access and sparse data access. Finally we will discuss how to build the table of algorithm efficiency values to account for varying scheduling algorithms.

The process for selecting these parameters is an informal one; no methodology for precisely calculating these values was developed as part of this work. This text serves only as a guide for selecting reasonable values.

In order to calculate these parameters, a large number of workload tests must be performed on the installed PVFS system. We utilize the three test programs discussed earlier in the text which create contiguous, strided, and random block access patterns. We used a range from 128 Kbytes per I/O node to 256 Mbytes per I/O node, with approximately fifty data points for each test series. We chose 128 Kbytes somewhat arbitrarily; we simply

wanted a fairly small size to start with. The upper bound was chosen as four times the size of physical memory, which allowed us to see how accesses that greatly exceed cache size are handled by the system. We repeated each run three times and used the average of these runs as data points when finding our parameter values.

The basic approach to finding the parameter values was simple. First we used our smallest data point to calculate any constant. Then we attempted to match the slope of the remaining data points via visual inspection; in other words we plotted the output of our model given a set of parameters along side the actual data points and tweaked the parameters to best match the data points.

B.1 Calculating Baseline Parameters

The first set of parameters we will calculate will be T_{over}, B_{drc}, B_{dr}, S_{cache}, and C_{cache}. These parameters define how, under ideal conditions, the system performs for both cached and uncached data.

First we utilize our single block workload test program with Opt 2 turned on. We ensure that all cached data is flushed on I/O nodes between test runs by reading a large data file between runs. We run tests over a range of values and perform a best fit to obtain T_{over} and B_{dr}, which are our fixed overhead value and our uncached bandwidth values. Note that for this workload case:

$$T_{taskread} = T_{over} + \frac{S_{req}}{B_{dr}} \tag{B.1}$$

Next we repeat the test allowing caching to occur. We observe the data set size at which caching appears to lose effectiveness; this is S_{cache}. We use the data points below this point and perform a best fit to obtain B_{drc}. For this region:

$$T_{taskread} = T_{over} + \frac{S_{req}}{B_{drc}} \tag{B.2}$$

Finally we note the data set size at which the effects of caching seem to have disappeared entirely. Denoting this point $S_{nocache}$ we can then calculate C_{cache}:

$$S_{nocache} = S_{cache}(1 + C_{cache}) \tag{B.3}$$

$$C_{cache} = \frac{S_{nocache}}{S_{cache}} - 1 \tag{B.4}$$

B.2 Integrating Workload Effects

Next we will select the four parameters which model workload effects: C_{req}, P_{req}, C_{dist}, and P_{dist}. To do this we must first perform additional testing.

First we utilize our strided workload test program, again using Opt 2. This workload isolates the effect of disjoint requests from that of sparse access. We test using both cached and uncached data. This data will be used to select C_{req} and P_{req}. There is no simple method for selecting these values. For this work we assumed that P_{req} was 1.0 and fit C_{req} using end cases as a starting point. Note for the uncached cases that:

$$T_{taskread} = T_{over} + \frac{1}{(1 - C_{req}) + C_{req}(\frac{N_{req}}{D_{req}})^{P_{req}}} \frac{S_{req}}{B_{dr}} \tag{B.5}$$

C_{req} varies from 0 to 1, with a value of 0 indicating that performance is not affected by this parameter. By plotting the observed data alongside equation output we were able to obtain our final values.

Selecting C_{dist} and P_{dist} was performed similarly, only in this case the random block workload was used to gather data. By utilizing the random block workload we isolate the effect of sparse access from that of disjoint requests. We set our algorithm to Opt 2 and perform tests for two numbers of blocks. This gives us enough data to estimate both values. Again the presence of a piece-wise function and an exponential value makes it difficult to algorithmic-ally calculate these values, so we begin by matching the end cases and by substitution arrive at reasonable values.

B.3 Accounting for Scheduling Algorithms

Finally we must build the table of values used to model the effect of a given scheduling algorithm. From our observations it appears we can model this effect with a simple coefficient for each case. We re-run our workload tests using all the available optimizations and by matching end cases for uncached tests obtain our values for the uncached case. Using data points near S_{cache} we then obtain values for the cached case.

Bibliography

[1] Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, and Joel Saltz. Jovian: A framework for optimizing parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 10–20, Mississippi State, MS, October 1994. IEEE Computer Society Press.

[2] Rajesh Bordawekar, Alok Choudhary, and Juan Miguel Del Rosario. An experimental performance evaluation of Touchstone Delta Concurrent File System. In *Proceedings of the 7th ACM International Conference on Supercomputing*, pages 367–376. ACM Press, 1993.

[3] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, Portland, OR, 1993. IEEE Computer Society Press.

[4] Tim Bray. Bonnie file system benchmark. `http://www.textuality.com/bonnie/`.

[5] R. A. A. Bruce, S. R. Chapple, N. B. MacDonald, and A. S. Trew. CHIMP and PUL: Support for portable parallel programming. Technical Report EPCC-TR93-07,

Edinburgh Parallel Computing Center, March 1993.

[6] Karen Castagnera, Doreen Cheng, Rod Fatoohi, Edward Hook, Bill Kramer, Craig Manning, John Musch, Charles Niggley, William Saphir, Douglas Sheppard, Merritt Smith, Ian Stockdale, Shaun Welch, Rita Williams, and David Yip. Clustered workstations and their potential role as high speed compute processors. Technical Report RNS-94-003, NAS Systems Division, NASA Ames Research Center, April 1994.

[7] Peter M. Chen and David A. Patterson. Maximizing performance in a striped disk array. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 322–331, 1990.

[8] Chiba city, the Argonne scalable cluster. `http://www.mcs.anl.gov/chiba/`.

[9] Alok Choudhary, Rajesh Bordawekar, Michael Harry, Rakesh Krishnaiyer, Ravi Ponnusamy, Tarvinder Singh, and Rajeev Thakur. PASSION: parallel and scalable software for input-output. Technical Report SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University, September 1994.

[10] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.

[11] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, and Sandra Johnson Baylor. Parallel access to files in the Vesta file system. In *Proceedings of Supercomputing '93*, pages 472–481, Portland, OR, 1993. IEEE Computer Society Press.

[12] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, and Marc Snir. User-friendly and efficient parallel I/Os using the Vesta parallel file system. In *Transputers '94:*

Advanced Research and Industrial Applications, pages 23–38. IOS Press, September 1994.

[13] Peter F. Corbett, Jean-Pierre Prost, Chris Demetriou, Garth Gibson, Erik Reidel, Jim Zelenka, Yuqun Chen, Ed Felten, Kai Li, John Hartman, Larry Peterson, Brian Bershad, Alec Wolman, and Ruth Aydt. Proposal for a common parallel file system programming interface. `http://www.cs.arizona.edu/sio/api1.0.ps`, September 1996. Version 1.0.

[14] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[15] P. J. Denning. Effects of scheduling on file memory operations. In *AFIPS Spring Joint Computer Conference*, April 1967.

[16] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, and Chris Dodd. The virtual interface architecture. *IEEE Micro*, 18(2):66–76, March/April 1998.

[17] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[18] John L. Hennessy and David A. Patterson. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[19] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceed-*

ings of the 9th ACM International Conference on Supercomputing, pages 385–394, Barcelona, July 1995. ACM Press.

[20] IEEE/ANSI Std. 1003.1. Portable operating system interface (POSIX)–part 1: System application program interface (API) [C language], 1996 edition.

[21] David Kotz. Disk-directed I/O for MIMD multiprocessors. *ACM Transactions on Computer Systems*, 15(1):41–74, February 1997.

[22] David Kotz and Carla Schlatter Ellis. Caching and writeback policies in parallel file systems. *Journal of Parallel and Distributed Computing*, 17(1–2):140–145, January and February 1993.

[23] John Krystynak and Bill Nitzberg. Performance characteristics of the iPSC/860 and CM-2 I/O systems. In *Proceedings of the Seventh International Parallel Processing Symposium*, pages 837–841, Newport Beach, CA, 1993. IEEE Computer Society Press.

[24] F. Kurzweil. Small disk arrays – the emergine approach to high performance. In *presentation at COMPCON 88*, March 1988.

[25] W. B. Ligon and R. B. Ross. Implementation and performance of a parallel file system for high performance distributed applications. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 471–480. IEEE Computer Society Press, August 1996.

[26] Michael Litzkow, Todd Tannonbaum, Jim Basney, and Miron Livny. Checkpoint and migration of UNIX processes in the condor distributed processing system. Tech-

nical Report Computer Sciences Technical Report #1346, University of Wisconsin-Madison, April 1997.

[27] Susan J. LoVerso, Marshall Isman, Andy Nanopoulos, William Nesheim, Ewan D. Milne, and Richard Wheeler. *sfs*: A parallel file system for the CM-5. In *Proceedings of the 1993 Summer USENIX Technical Conference*, pages 291–305, 1993.

[28] Tara M. Madhyastha, Christopher L. Elford, and Daniel A. Reed. Optimizing input/output using adaptive file system policies. In *Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems*, pages II:493–514, September 1996.

[29] Tara M. Madhyastha, Garth A. Gibson, and Christos Faloutsos. Informed prefetching of collective input/output requests. In *Proceedings of the ACM/IEEE SC99 Conference*, November 1999.

[30] Tara M. Madhyastha and Daniel A. Reed. Intelligent, adaptive file system policy selection. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 172–179. IEEE Computer Society Press, October 1996.

[31] Tara M. Madhyastha and Daniel A. Reed. Input/output access pattern classification using hidden Markov models. In *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems*, pages 57–67, San Jose, CA, November 1997. ACM Press.

[32] Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface, July 1997. `http://www.mpi-forum.org/docs/docs.html`.

[33] Steven A. Moyer and V. S. Sunderam. A parallel I/O system for high-performance distributed computing. In *Proceedings of the IFIP WG10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems*, 1994.

[34] Steven A. Moyer and V. S. Sunderam. PIOUS: a scalable parallel I/O system for distributed computing environments. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 71–78, 1994.

[35] Myrinet software and documentation. `http://www.myri.com/scs`.

[36] Nils Nieuwejaar and David Kotz. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, May 1996. ACM Press.

[37] Nils Nieuwejaar and David Kotz. The Galley parallel file system. *Parallel Computing*, 23(4):447–476, June 1997.

[38] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael Best. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, October 1996.

[39] Bill Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992.

[40] David Patterson, Garth Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 109–116, Chicago, IL, June 1988. ACM Press.

[41] Daniel A. Reed, Ruth A. Aydt, Roger J. Noe, Philip C. Roth, Keith A. Shields, Bradley Schwartz, and Luis F. Tavera. Scalable performance analysis: The pablo performance analysis environment. In *Proceedings of the 1993 Scalable Parallel Libraries Conference*, October 1993.

[42] Daniel Ridge, Donald Becker, Phillip Merkey, and Thomas Sterling. Beowulf: Harnessing the power of parallelism in a pile-of-pcs. In *Proceedings of the 1997 IEEE Aerospace Conference*, 1997.

[43] ROMIO: A high-performance, portable MPI-IO implementation. `http://www.mcs.anl.gov/romio`.

[44] John Salmon and Michael Warren. Parallel out-of-core methods for N-body simulation. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[45] Scheduled transfer – API mappings. `http://www.hippi.org/cSTAPI.html`.

[46] Peter Scheuermann, Gerhard Weikum, and Peter Zabback. Data partitioning and load balancing in parallel disk systems. Technical Report 209, ETH Zurich, January 1994.

[47] P.H. Seaman, R. A. Lind, and T. L. Wilson. An analysis of auxiliary-storage activity. *IBM System Journal*, 5(3):158–170, 1966.

[48] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[49] K. E. Seamons and M. Winslett. An efficient abstract interface for multidimensional array I/O. In *Proceedings of Supercomputing '94*, pages 650–659, Washington, DC, November 1994. IEEE Computer Society Press.

[50] Huseyin Simitci, Daniel A. Reed, Ryan Fox, Mario Medina, James Oly, Nancy Tran, and Guoyi Wang. A framework for adaptive storage of input/output on computational grids. In *Proceedings of the Third Workshop on Runtime Systems for Parallel Programming*, 1999.

[51] Hal Stern. *Managing NFS and NIS*. O'Reilly & Associates, Inc., 1991.

[52] Hakan Taki and Gil Utard. MPI-IO on a parallel file system for cluster of workstations. In *Proceedings of the First IEEE International Workshop on Cluster Computing*, 1999.

[53] Test TCP. `ftp://ftp.arl.mil/pub/ttcp/`.

[54] Rajeev Thakur, William Gropp, and Ewing Lusk. An abstract-device interface for implementing portable parallel-I/O interfaces. In *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187. IEEE Computer Society Press, October 1996.

[55] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, May 1999.

[56] Uresh Vahalia. *UNIX Internals: The New Frontiers*. Prentice Hall, Upper Saddle River, NJ, 1996.