File System Simulation: Hierarchical Performance Measurement and Modeling

Hai Quang Nguyen
University of Arkansas, Fayetteville

FILE SYSTEM SIMULATION: HIERARCHICAL PERFORMANCE MEASUREMENT AND MODELING

FILE SYSTEM SIMULATION: HIERACHICAL PERFORMANCE MEASUREMENT AND MODELING

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Computer Engineering

By

Hai Quang Nguyen
Ho Chi Minh City University of Technology
Bachelor of Science in Computer Engineering, 1997
University of Arkansas
Master of Science in Computer System Engineering, 2002

August 2011
University of Arkansas

ABSTRACT

File systems are very important components in a computer system. File system simulation can help to predict the performance of new system designs. It offers the advantages of the flexibility of modeling and the cost and time savings of utilizing simulation instead of full implementation. Being able to predict end-to-end file system performance against a pre-defined workload can help system designers to make decisions that could affect their entire product line, involving several million dollars of investment.

This dissertation presents detailed simulation-based performance models of the Linux ext3 file system and the PVFS parallel file system. The models are developed using Colored Petri Nets. A performance study, using the models, shows that the obtained results are close to the expected behavior of the real file system. The model shows that file system parameters have significant impact on the performance of the I/O when compared to the parameters of the disk subsystem.

This dissertation is approved for recommendation
to the Graduate Council.

Dissertation Director:

____
Dr. Amy Apon

Dissertation Committee:

____
Dr. Craig W. Thompson

____
Dr. Dale R. Thompson

____
Dr. Fred Limp

ACKNOWLEDGMENTS

Sir Isaac Newton once said: “If I have seen further than others, it is because I have stood on the shoulders of giants.” This dissertation could not have been possible without the great deal of collaboration, guidance, and friendship I have been blessed with over the years. I owe my gratitude to all who have supported me on my journey to and through graduate school.

I would like to express sincere appreciation to my advisor, Dr. Amy Apon, for her guidance, understanding, patience and support during my graduate study at the University of Arkansas. I cannot thank her enough for her willingness to go above and beyond to ensure the success of her students. It has always been a privilege to study under her tutelage. For everything you have done for me, Dr. Apon, I thank you.

I would like to thank my committee members, Dr. Dale Thompson, Dr. Fred Limp and Dr. Craig Thompson. I had the good fortune of having a stellar dissertation committee. I am extremely grateful for their expert guidance and suggestions concerning my research directions.

I would like to express my gratitude to Dr. Larry Dowdy of the Department of Electrical Engineering and Computer Science at Vanderbilt University for his feedback and for sharing his invaluable knowledge and insights into file system behavior and performance modeling, and to Dr. Laurence Dilday of the Departments of Writing and Communication at the University of Central Arkansas for his editorial assistance.

I have been blessed with many dedicated and capable good friends and colleagues of whose wisdom have been translated into much of the work represented herein. Phil Carn at

Argonne National Laboratory and David Metheny and Bart Taylor at Acxiom Corporation have shared their precious parallel file system knowledge and have provided valuable insight into the internal design of PVFS and PVFS2. I would like to thank Frank and Sharon Bode whose many instances of support have helped me stay sane through these difficult years. I am also thankful to Sharon for carefully reading and commenting on countless revisions of this manuscript. I also want to thank Linh Ngo, Hung Tran and Hung Bui who have shared their life with me through graduate school. I greatly value their friendship, and I deeply appreciate their belief in me.

I would like to thank the Department of Computer Science and Computer Engineering at the University of Arkansas. In particular, I would like to thank Susan Huskey for her support during my graduate study at the University of Arkansas and since.

I am grateful to the National Science Foundation for providing financial support for the work in this dissertation. This dissertation was supported in part by NSF Grant No. 0421099 and NSF Grant No. 0722625.

Finally, and most importantly, I would like to thank my family. None of this would have been possible without their love and patience. I thank my wife, Tien. Her support, encouragement, quiet patience, unyielding devotion and unwavering love have been with me through good times and bad, in sickness and in health. I would like to thank my parents for their faith in me. Their encouragement and support gave me the strength to become who I wanted to be.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

This dissertation promotes simulation modeling, using Colored Petri Nets as a tool for the evaluation of file systems performance with different architectures, including local and parallel file systems.

1.1 Problem definition

Today's scientific data-intensive research applications place very high demands on storage systems in both performance and capacity [1, 2] with much attention paid to large-scale resource sharing and allocation. Even though big and powerful mainframe systems are still being built and deployed, data-intensive clusters of computers utilizing cloud-computing technology are growing at an impressive rate. Although storage systems are well-established research areas [3-5], modern storage system development still lags behind processor technologies. When comparing the latency before data are available and the bandwidth of data transferring with the data processing rate of state of the art data-intensive clusters utilizing the most current processor technology, storage systems cannot keep up. As a result, storage systems are often considered the bottlenecks of many data-intensive applications. In other words, the overall system performance is adversely affected by the time required for accessing data from secondary storage components. Many applications spend a significant amount of run time in I/O wait as opposed to actual processing, computing and transforming data. In order to meet the storage capacity and performance demands of these applications, storage research has pushed aggressively on multiple fronts.

High- performance magnetic disks have become so inexpensive that users are finding new, previously unaffordable uses for storage. Multi-tier storage systems have included more layers of different types of magnetic storage. This introduces even better transition with more granularities between the high, but expensive performance of solid-state storage and the low, but cheap performance of the tape library. As a consequence, operational personnel costs for storage management and performance tuning now dominate over capital costs of the equipment over its useful lifetime [6]. The utilization of cloud computing and the concept of resource-on-demand push the issues even further, causing the management of storage performance and capacity to become even more challenging.

Because of the critical role of the storage system in overall system performance, choosing and integrating a storage component is usually a difficult challenge for a system designer. Typically, a storage configuration fits best with a certain type of application due to the I/O access pattern. Therefore, by choosing the right storage configuration, one can maximize the most important performance metrics for the targeted application. For example, scientific applications such as geospatial analysis and modeling, high density survey, and digital photogrammetry often use a sequential access pattern over very large data sets. This type of I/O pattern performs well with storage configurations using parallel or distributed file systems over highly redundant disk array subsystems. However, the same type of storage configuration will perform badly when utilized by business-reporting applications that randomly process a large quantity of medium or small files. Moreover, the whole system, once built, is so complex; it is not easy to make modifications or improvements to core components like storage systems. The consequence of a poor decision could significantly increase the cost to operate and maintain the system throughout its lifetime. As a result, many techniques have been developed to assist with such decisions. At the laboratory bench, real system interactions can be studied utilizing prototype or test-built systems. These experiments could provide accurate system behaviors and performances. However, a typical storage system used in a high performance data-intensive environment consists of many components, and they usually are not cheap. In addition to the initial investment, hardware deployment is time consuming as well. As a consequence, experiments with real hardware are sometimes not very attractive, even for big companies or research groups and especially when only proofs of concepts are needed. At the drawing board, analytical techniques or computer simulation models can be used in conjunction with models of different workloads to evaluate the expected consequences of a proposed device. Under this technique, the system is simulated in enough detail to evaluate the performance and behavioral response of the storage system. This technique offers the dual advantage that any or all of the individual system components can be speculative and hypothetical; in addition to looking at next year's storage device, components such as the processors and memory can be scaled up according to expected trends to simulate the overall system that will be available next year. Unfortunately, substantial effort is required to build and maintain a complete machine simulator, both in terms of correctly executing programs and correctly accounting for time. Additionally, such simulators usually run much slower than real systems, which limit the overall scope of what architectural designs can be considered. Among several storage architectures, the three most common ones are Direct-Attached Storage (DAS), Network-Attached Storage (NAS), and Storage Area

Networks (SANs). These storage architectures prove to be able to provide a shared, adaptable, and high-performance storage system for data-intensive applications. The performance of each of these classes of storage architectures has a strong impact on the overall performance of the system. An accurate, well-developed simulation modeling environment could allow researchers to fine tune both the performance and the workload of network storage architecture.

Perhaps more so than in the past, now is a particularly pertinent time for needing this sort of evaluation technology. The gap between high-end and low-end storage hardware is significant enough to make system designers rethink their design strategy toward application-specific storage. Large and non-critical data sets will be put on consumer-grade, low-cost, high-capacity storages while small but very fast, high-end multi-disk arrays will be used for mission critical and database operations. Both product divisions are areas in which the performance implications and the impact of new system configurations of file systems and storage devices could be readily examined under file system simulation models.

1.2 Thesis statement

Current approaches to storage system evaluation using hardware are limited in that they are expensive and usually take time to deploy. In order to sufficiently perform system-level evaluation of system designs or architectural decisions, system designers need to consider both budgets and timelines, which proves to be quite a challenge. In response to these limitations, this dissertation advocates that simulation models for file systems are feasible and enable end-to-end performance experimentation while supporting both local file systems as well as parallel file systems.

1.3 Contributions of this dissertation

This dissertation advances four primary contributions:

1. It presents and validates the model of file system simulation for both local file systems and parallel file systems.
2. It demonstrates the feasibility of file system simulation in the context of end-to-end complex system performance evaluation.
3. It describes a general architecture for a file system simulation model and mechanisms for making it flexible enough to simulate an existing complex parallel file system. The architecture can also be used as a framework to develop other local file systems and parallel file systems.
4. It details concrete examples of the use of storage and file system simulation for explorations of system configurations and functionalities.

1.4 Overview of this dissertation

The text of this dissertation is presented in three conceptual parts, corresponding roughly to background and motivations, the local file system simulation model, and the parallel file system simulation model.

Chapter 2 discusses the background of the modern storage subsystem, storage improvement in general, and the role of file system in overall system-level evaluations.

Chapter 3 presents measurement studies of the local file system candidate—the third extended file system, or ext3—as well as the parallel study candidate—the parallel virtual file system, or PVFS.

Chapter 4 and Chapter 5 discuss the design and implementation of a simulation model for the ext3 file system and the performance evaluation of that system using the model.

Chapter 6 and Chapter 7 discuss the design and implementation of a simulation model for the PVFS file system and the performance studies of that system using the model.

Chapter 8 concludes by looking to the future of file system simulation.

1.4.1 File system simulation in system-level evaluation

The file system is a very important component in a computer system, yet file system simulation is rarely used for performance evaluation of new system designs. We argue for more frequent use, noting that file system simulation offers significant advantages: the flexibility of simulation and the cost and time savings when utilizing simulation.

The applications for a file system simulation model are many. End-to-end file system performance measurement for existing workload is one example of such applications. Modern scientific and business applications could be divided into multiple categories; each has every specific type of workload. While a system designer could certainly use one type of file system for all of his or her applications and customers, the performance results are usually not adequate. This is particularly true when network file systems are involved. There are many commercial network file systems as well as open-source ones. They are all designed to support a broad range of applications; however, each of them has certain strengths and weaknesses. Being able to measure end-to-end file system performances against pre-defined workloads could help system designers to make decisions that could affect their entire product line and be worth several million dollars of investment.

Another application for a file system simulation model is bottle-neck analysis. In a complex high-performance parallel file system, identifying where the bottle-necks are that need to be upgraded or expanded could be very difficult. Components in such a system could be well integrated into the whole system and very hard and expensive to upgrade. It is in a system designer's best interest to identify the correct bottle-necks of the system to upgrade. File system simulation models could be used to analyze the I/O flow and identify bottle-necks. It could also be used for 'what-if' type of provisioning analysis.

1.4.2 Modeling using Colored Petri Nets

K. Jensen proposed an extended version of classical Petri Net called Colored Petri Net [7]. Colored Petri Net, or CPN, is a graphical-oriented language for design, specification, simulation and verification of systems. This language is particularly well suited to illustrating and simulating systems in which communication, synchronization between components, and resource sharing are primary concerns [8]. This offers a flexible framework that is well adapted for the analysis of I/O flow performance. CPNTools [9] is utilized for simulation and analysis.

In addition to places, transitions and tokens, the concepts of colors, guards and expressions are introduced, so that computed data values can be carried by the tokens. For simplicity, the color of a token can be described as a data type. It defines the types of data that can be carried by tokens. Each place in the net is also assigned a data type, and can only hold tokens of its assigned type. For example, Figure 1 presents a place of type integer. This place contains an initial token with a value of 1.

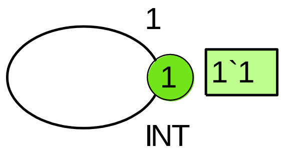


Figure 1: Integer type place

Each arc in the net is also assigned an expression that evaluates to a certain data type. The data type of the arc must match the color set of the place attached to the arc. For example, Figure 2 presents an arc connected to a place with integer type. The arc has an expression that is an integer variable named i.

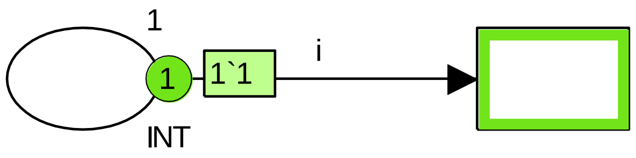


Figure 2: Arc with integer variable

A transition in the net has similar functionality to a subroutine in a program. Incoming arcs define input parameters to the transition. Outgoing arcs are results from the transition. A transition can have a guard which is a Boolean expression. Guards are used for testing the input arcs enabling conditions or restrictions. A transition can also have a code segment. Code segments are executed when tokens go through the transitions. Figure 3 presents a transition with a guard and a code segment. The guard restricts the value of the input tokens to less than 10. The code segments produce two output results. One is equal to the input plus 1. The other is equal to the input plus 2.

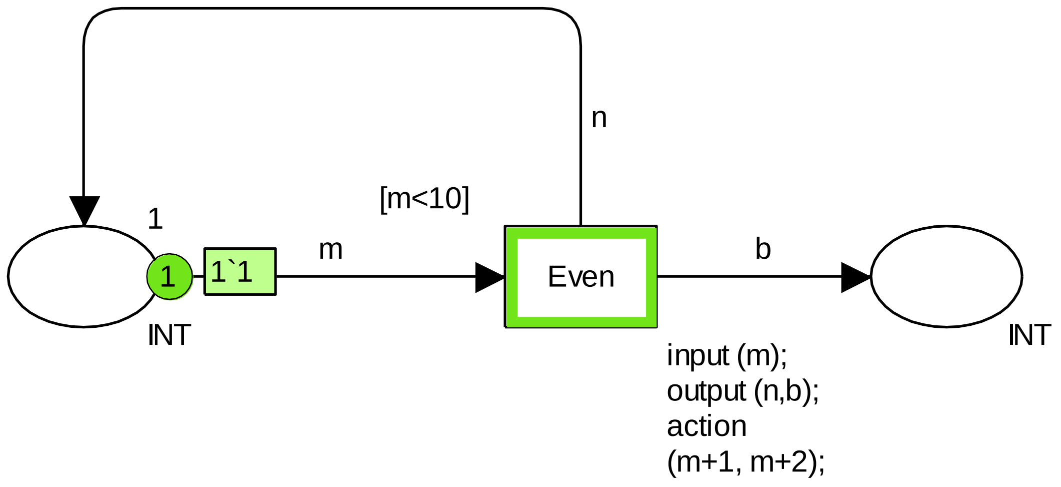


Figure 3: Transition in a Colored Petri Net

These concepts prove to be incredibly powerful since tokens can now carry information that is simple or complex. This feature is used extensively to carry time stamps with tokens flowing within the simulation model.

1.4.3 Design of a simulation model for local file systems

Linux, as an open-source operating system, offers a very flexible system that supports a large number of file systems, including journaling file systems, clustering file systems, and cryptographic file systems. The architecture for a file system under Linux was designed with an abstract layer to support a large variety of file systems on a large variety of storage devices. The application, when using the abstract layer function, is completely unaware of the true file system types or the storage medium. In this clean and well-designed layer system, an upper component often hides the details of the lower components and presents more unified and simpler information to the layer above it. The ext3 file system has been chosen to be the study candidate for this simulation model. Ext3 is a standard file system on every Linux distribution. It was released and officially supported by

Red Hat since 2001 [10]. Ext4, the successor of ext3, was introduced into Red Hat Enterprise Linux very recently as a technology review. It takes time for industry to adopt and migrate to a new file system. For the time being and in the near future, ext3 will continue to be deployed and utilized in industrial settings.

The model provides simulation implementations for most components in the ext3 file system, such as data pre-fetching, buffer cache, and data journaling. Components are designed to follow the implementation in Linux closely to preserve the performance characteristics of the file system. The simulation model is organized into two main I/O operations: data reading and data writing. Since the Linux abstract layer hides the details of the lower components, instead of having to model the lower hardware layer of storage devices in detail, the model uses a more simple queuing approach. Without going into the device drivers level, a direct attach disk drive is similar to a SAN storage array. This design helps the simulation model to be more flexible and to support multiple types of storage devices.

The ext3 file system simulation model is validated using different types of workload to make sure the model behaves similarly to the real file system under different situations. A synthetic workload, such as a sequential I/O access pattern and a random I/O access pattern, is generated and used in both the simulation model and the real system. In addition to the synthetic workload, I/O traces are collected from production systems and used to study the behavior of the simulation model under real-world situations.

1.4.4 Design of a simulation model for parallel file systems

While local file systems play a very important role in a modern datacenter, scientific and business applications put forth many challenges to computer system designers. Two pinnacles of those challenges are processing power and storage capacity. Parallel file systems are being developed to fill the gap between data accessing speed from secondary storage and the processing speed of a high-performance cluster.

Parallel file systems bring many advanced features to high-performance computing. Two major ones are providing unified name space across multiple machines (or nodes) and providing parallel accessing to storage devices. These two characteristics create a many-to-many relationship between high-performance clusters and storage devices, enabling high enough I/O throughput to satisfy state-of-the-art clusters. These characteristics, however, also bring another important component into file system architecture—the network. To implement a simulation model for a parallel file system, an end-to-end performance model of the network is developed.

There are many existing parallel file systems. Each of them has a different approach to how data and metadata are managed and allocated. The Parallel Virtual File System, or PVFS, is a powerful open-source parallel file system. It has been well received by both the academic and industrial worlds. Although there are other commercial parallel file systems on the market, their closed, proprietary source code becomes a significant challenge to the development of a simulation model. PVFS has been chosen to be the study candidate for the parallel file system simulation model. The architecture of the file system is presented in Figure 4.

At a basic level, PVFS uses the standard Linux ext3 file system as its foundation. By utilizing multiple ext3 file systems working in parallel with completely separate metadata operations, PVFS can achieve massive data bandwidth. These ext3 file systems are located on several I/O server machines. Each individual ext3 file system is governed by a PVFS I/O daemon. Many daemons could be stacked on a single physical machine to increase the size of the PVFS file system, thus lessening the number of physical machines required.

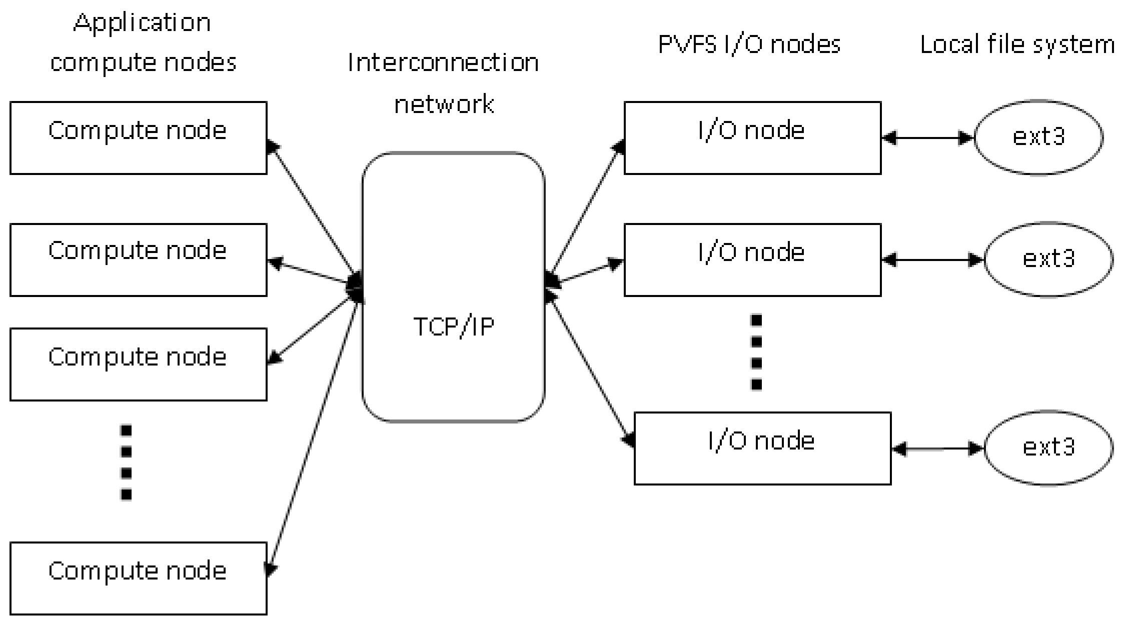


Figure 4: PVFS file system architecture

Similar to the ext3 file system model, the simulation model for PVFS is also divided into data reading operations and data writing operations. Each operation consists of three major parts: the client, the network interconnection, and the server. Components are designed to follow the implementation of the actual PVFS closely to preserve the performance characteristics of the file system. When a file system access request from an application arrives at the PVFS client, it will be divided into multiple small chunks of data of a certain size and distributed round-robin fashion into different payloads. The number of payloads is equal to the number of I/O daemon (IOD) of the PVFS file system. These payloads are delivered to the I/O daemons over the network using the network model. Each I/O daemon processes the payload requests and responds back to the PVFS clients individually. The I/O daemon storage device is simulated using the ext3 simulation model.

1.5 Summary

File system simulation offers the opportunity to investigate novel uses of different types of file systems and, to some degree, different storage subsystems in computer systems. This permits forays into the space of hypothetical system configurations without the difficulties of developing and supporting extensively prototype systems only for evaluation purposes. This dissertation demonstrates that there is a current and pressing need for a file system evaluation technique. It also emphasizes that it is feasible to design and construct a file system simulator and to use a simulation model for interesting systems-level performance experimentations. The file system performance model is divided into a local file system model and a parallel file system model. For the local file system model, ext3 is the study candidate. For the parallel file system model, PVFS is the study candidate. The performance model is developed using a Colored Petri Nets modeling method and is implemented using CPN-Tools. A resource model for the interconnection network is also developed. To evaluate both the local file system model and the parallel file system model, end-to-end performance validation is performed using different types of workload, including synthetic workload and I/O traces collected from production systems.

Chapter 2

BACKGROUND AND LITERATURE

2.1 Introduction

This chapter presents the background study of related work. Research in the storage simulation and modeling areas is examined. Work in other areas of storage development, such as general storage technology, multi-tier storage, and virtual storage, are also studied to have a thorough understanding while implementing storage component models.

2.2 Related work

This section is organized into three main areas. The first area surveys published models that capture the behavior and performance of storage devices. The second area surveys published network modeling studies. The last area examines developments in storage and file system technologies, the knowledge of which is essential when designing storage component models.

2.2.1 Storage modeling and simulation surveys

Much work has gone into the development of storage device models. This section presents major published studies and provides a better understanding of storage modeling research. It also presents methods and techniques have been used to model storage devices.

Among recent storage modeling research, DiskSim, which has been made publicly available to the research community, is one of the best-known disk simulation systems. Described by J.S. Bucy and G.R. Ganger [11], Disksim was developed to support research on several aspects of the storage subsystem architecture. By providing modules that simulate disks, intermediate controllers, buses, device drivers, request schedulers, disk block caches, and disk array data organization, Disksim is an efficient, accurate, and highly configurable disk system simulator that can simulate modern disk drives in great detail and has been validated against production disks with high accuracy. DiskSim has also been utilized in many subsequent studies as a foundation.

J.L. Griffin et al. [12] describe a prototype called the Memulator that appears to the system to be a real storage component with service times similar to the component it is simulating. This prototype produces service times within 2% of those computed by its component simulator for over 99% of requests. Memulator was used for performance measurements on a modern Linux system equipped with a micro-electro-mechanical system (MEMS)-based storage device and a modern Linux system equipped with a disk whose firmware had been modified. Griffin also uses timing-accurate storage emulation to experiment with nonexistent storage components to explore the interactions between modified computer systems and expanded storage device functionality, and to study storage-based intrusion detection systems [13]. He demonstrates the incorporation of intrusion detection capabilities into processing-enhanced disk drives.

Maghraoui et al. [14] presents a method of modeling a Flash device and building a Flash simulator. The authors capitalize on the throughput behavior of the Flash disk with no rotary components and develop a linear model for the Flash device. Benchmarks results show the throughput of the simulation model is within 7% error range compared to a real Flash disk. The authors also argue that one can simulate Flash-based solid-state drives

(SSDs) without having to simulate every minor detail and internal organization of a Flash device.

Wang and Kaeli [15] offer ParIOSim, a validated execution-driven simulator for network storage systems. Their simulator provides a flexible simulation environment for performing storage optimizations. This simulator can also be used to accurately predict the performance of parallel I/O applications as a function of the underlying storage architecture. They compared the performance of ParIOsim with the performance of an actual parallel system to demonstrate the accuracy of the tool and provided results from running a parallel I/O benchmark application over different storage architectures.

To understand and optimize database performance, a good model of the storage structure is needed. This is a difficult task because the fundamental role of conceptual-to-internal mapping in database management system (DBMS) implementations previously went unrecognized. Batory [16] presents a model of physical databases, called the transformation model, that makes conceptual-to-internal mappings explicit. The author shows that by exposing such mappings, it is possible to model the storage architectures (i.e., the storage structures and mappings) of many commercial DBMSs in a precise, systematic, and comprehendible way. The transformation model also helps bridge the gap between physical database theory and practice. The author further believes that the model also reveals the possibility of a technology to automate the development of physical database software.

Gomaa [17] introduces a hybrid modeling technique that combined two different modeling techniques, regression and simulation, to model virtual storage computer systems. This technique uses simulation to model in detail a task's arrival, its entering and dropping from the multiprogramming set, and its termination. The regression techniques are used to model the rest of the system in much less detail. The authors also describe the application of the method to model an IBM VM/370 system.

Coffman and Reiman [18] present a model of the storage resource. It is a basic model of the space-time requirements of jobs in a computer system and a number of its variations analyzed by means of diffusion approximations. Using the usual heavy-traffic assumptions, the result of their analysis allows the effects of limitation on both storage capacity and processing rates to be quantified.

Jacobson and Lazowska [19] offer an approximate solution technique for queueing network models that includes the simultaneous or overlapped possession of resources. This issue arises in many computer system contexts and has a significant effect on system performance. The key idea behind this method is to partition queuing delay according to which of the simultaneous help resources is responsible. This approach provides a unified, practical treatment of a diverse set of problems.

In large multimedia document archives, a major fraction of data may be stored in a tertiary storage library to reduce cost. Kraiss and Weikum [20] present an integrated approach to the vertical data migration between the tertiary, secondary, and primary storage. To predict the expected number of accesses to a document within a specified time horizon, the integrated migration policy is based on a continuous-time Markov-chain model. The parameters of this model, the probabilities of co-accessing certain documents, and the interaction times between successive accesses are dynamically estimated and adjusted to evolving workload patterns by keeping online statistics. The authors discuss a prototype system, which uses the integrated policy for vertical data migration. The Markov-chain model is also being used in the system for the scheduling of volume exchanges in the tertiary storage library. The authors also present initial results using simulation experiments with Web-server-like synthetic workloads. The results show significant gains in terms of client-response time. The experiments also show that the overhead of the statistical bookkeeping and the computations for the access predictions is affordable.

Network-attached storage devices improve I/O performance by separating control and data paths and eliminating host intervention during data transfer. Devices are attached to a high-speed network for data transfer and to a slower network for control messages. Hierarchical mass storage systems use disks to cache the most recently used files and tapes (robotic and manually mounted) to store the bulk of the files in the file system. Menasce, Pentakalos, and Yesha [21] explain how queuing network models can be used to assess the performance of hierarchical mass storage systems that use network-attached storage devices. The analytic model, validated through simulation, is used to analyze many different scenarios.

Zhaobin et al. [22] show that Stochastic Petri Net (SPN) models can be used to analyze the performance of hybrid I/O Data Grid storage systems. The authors discuss their implementation of a typical storage system SPN modeling. Based on aggregate I/O, they also simplify the complexity of the model. From case studies, it is shown that the priority schedule can be adjusted by changing the ratio of file I/O and multimedia I/O. Their work can be used to study complex and irregular I/O patterns of Data Grids applications.

Molero et al. [23] present the model and design of a very flexible and easy to use SAN simulator. They also discuss the basic modeling mechanisms, the main input parameters, and the output performance variables of the simulator. The tool can use both real-world I/O traces and synthetic I/O traffic, messages, faults in links and switches, virtual channels, different routing algorithms, etc. The authors reveal the preliminary simulation results of using I/O traces. They show that the storage network increases self-similarity of the traffic received by servers. A different model for SAN devices is presented by Routrey et al. [24]. The tool can be used to simulate many different management modules, including large scale multi-vendor heterogeneous SAN for enterprise. Moreover, the simulation model can also be used for what-if analysis of an enterprise information technology (IT) environment before any changes. Instead of developing separate tools, Staley et al. [25] use Optimized Network Engineering Tools (OPNET) as a discrete event simulator to model and simulate Fibre Channel-based SAN devices. The model is utilized to determine the scaling and stability issues of a large fabric as the size of the storage area network grows. The authors also present preliminary simulations. Focusing on the performance of I/O interaction of host servers and storage subsystems via the SAN fabric in a storage area network, Aizikowitz et al. offer a study using performance modeling [26]. The work describes a component-based simulation performance model, which supports a rich variety of both existing and future storage subsystems. It allows some basic network configurations and addresses many major I/O aspects of the server operating system. The simulation model has many flexible features, such as easy parameter modifications, configuration adjustments, architecture manipulations, and experimentation.

The fully distributed nature of peer-to-peer storage architecture allows it to have many interesting features, such as global scalability, self-configuration, dynamic adaptation, fault tolerance and anonymity. Hung-Chang et al. [27] study the memory architecture of the peer-to-peer storage systems, especially the effects of caches and directories on their performance. The authors describe an abstract model, called the distributed shared memory (DSM) model. It is used to capture the essence of the peer-to-peer storage architecture from the memory perspective. The performance of three peer-to-peer storage system models under different memory pressures, network sizes, and failure degrees is evaluated via simulation.

DeRosa et al. describe the design and implementation of Vesper, an instructional disk drive simulator with a high degree of performance realism [28]. While providing timing statistics close to that of real disk drives, the simulator can still retain its simplicity. The authors present their method to provide hardware abstractions that are simple, yet capable of capturing device interactions with major performance impacts. Users of the simulator can explore the performance consequences of various system designs without the cumbersome aspects of the real hardware interface.

Ali et al. [29] offer the design and development of a modeling and simulation prototype for the final assembly of hard disk drives with dynamic and static behavior. The prototype can develop intelligent dynamic machine knowledge. It can also capture dynamic activities with fuzzy systems. The model is highly flexible, fast and capable of self-development. It can help improve the system performance significantly. The authors show that modeling and simulation tools can be used to implement and integrate highly-automated systems for industrial processes.

2.2.2 Network modeling and simulation surveys

Network modeling is a very large area of research. However, since the network components are essential components of the parallel file system models, it is important to understand network modeling. Due to the amount of research that has been done in this area, only a selection of studies that are closely related to designing the network component is examined. The similarity between modeling network within parallel file systems and modeling network within full-system simulation environments suggests that a good understanding of this area will be very beneficial. The full-system simulation environment provides complex interactions between applications and systems. However, the simulation durations are usually very short and require fast simulation turnaround time. Several papers address and develop methods to overcome this problem while ensuring representative results [30-32]. Standard network tools are used to extract simplified models that are statistically validated and, at the same time, compatible with a full-system simulation environment. Different models are proposed with different accuracy-*versus*-speed ratios that compute network latency times according to the estimated traffic and measure those times on a real-world parallel scientific application.

While well-known packet-based simulators, such as ns-2, still play an important role in network simulation, the more recently-proposed hybrid systems model for data communication networks shows promise in achieving performance characteristics comparable to fluid models while retaining the accuracy of discrete models. The key to this method is that the averaging occurs over short time intervals to continuously approximate discrete variables, such as congestion window and queue size. Therefore, discrete events, such as the occurrence of a drop and the consequent reaction by congestion control, can still be captured [33, 34]. This modeling framework, thus, fills a gap between purely packet-level and purely fluid-based models. Observations show, in networks with large per-flow bandwidths, simulations using hybrid models require significantly less computational resources than ns-2 simulations.

Also using the hybrid systems paradigm, Kavimandan et al. [35] present several models of Transmission Control Protocol (TCP) behavior and the analysis/simulation of data communication networks based on these models. An important distinguishing feature of this study is a faithful accounting of link propagation delays which have been ignored in previous work for the sake of simplicity. The simulation results are consistent with well-known packet-based simulators such as ns-2, thus demonstrating the accuracy of the hybrid model.

The hybrid network model can also allow flows to interact with fluid flows within each network queue as Liu et al. note [36]. Therefore, it is possible to dynamically change the composition of traffic flows to allow the simulation to keep up with real time. Experiment results in the paper show that the model provides a good prediction of the network behavior. Parallel processing methods can also be used to improve performance when simulating large-scale networks [37]. Liu et al. demonstrate the benefit of the parallel hybrid model through a series of simulation experiments of a large-scale network, consisting of over 170,000 hosts and 1.6 million traffic flows on a small parallel cluster.

Verdickt et al. [38] propose a framework that allows modeling of both the software and the network components separately, using the modeling languages and tools most suited to those system aspects. The framework can produce a single set of performance evaluations for the entire system. The key benefit of the model is that it helps in evaluating the network latency and its relation to the application behavior when assessing the performance of a distributed system. The authors also offer a case study to illustrate the use of the framework and its efficacy in predicting system performance.

Yu et al. [39] present an approximate scheme to model and analyze switch networks with phase-type and bursty traffic, and they describe a traffic aggregation technique to deal with such traffic, including splitting and merging. To aggregate bursty traffic, the paper suggests a closed-form solution for two bursty traffics and a recursive algorithm derived in terms of the buffer size and number of inputs of a switch for more bursty traffics. The numerical and simulation results in the paper show that the proposed scheme achieves satisfactory accuracy and computational efficiency.

Kassa et al. [40] offer a simple, fast, and detailed analytical model of the TCP, which is the dominant transport protocol for the end-to-end control of information transfer. The model assumes that only basic network parameters, such as the network topology, the number of TCP connections for large file transfers, link capacity, distance between network nodes, and router buffer sizes, are known. The paper presents performance metrics obtained by using TCP and network sub-models and solving them, using a fixed-point algorithm.

Comparing against ns-2 simulations, the results show that the model is accurate, simple and computationally efficient. Therefore, the model can rapidly analyze network topologies with several bottlenecks and obtain detailed performance metrics.

2.2.3 Storage technology surveys

To accurately develop and implement storage simulation components, in addition to storage and network modeling research, other storage technology areas also need to be studied.

2.2.3.1 Buffer cache and pre-fetching research

In the operating-system kernel, buffer cache is a major component that directly affects I/O performance. Buffer caches are commonly used to reduce the number of slow disk accesses or network messages. An accurately-designed buffer cache is a vital component of a file system simulation model. In order to improve the performance of I/O processing, a buffer cache should be managed with regard to both blocks and files. However, it is difficult to keep track of both blocks and files together since they are on different levels of the file system pipeline. Katakami et al. [41] propose an I/O buffer cache mechanism based on the frequency of file usage. This mechanism calculates the importance of each file. Then blocks of important files are stored in a protected space and given priority for caching. This mechanism provides an interesting replacement policy for the buffer cache. Other researchers also pay much attention to the hierarchy and replacement policy of the buffer cache [42, 43]. They focus on improving overall buffer cache efficiency and investigating multiple approaches to effectively managing multi-level buffer caches.

Spatial locality of cached blocks in the buffer cache is also looked at by researchers [44, 45]. The authors argue that spatial locality of cached blocks is largely ignored, and only temporal locality is considered in current system buffer cache management. Thus, disk performance for workloads without dominant sequential accesses can be seriously degraded. A method that exploits both temporal and spatial localities in the buffer cache management is proposed. The placement scheme significantly increases the effectiveness of I/O scheduling and pre-fetching for disk performance improvements. In addition to localities, the effect of pre-fetching is also studied. Despite the well-known interactions between pre-fetching and caching, almost all buffer cache replacement algorithms have been proposed and studied comparatively, without taking into account file system pre-fetching, which exists in all modern operating systems. Several works [46-48] show that pre-fetching can have a significant impact on the relative performance. These results demonstrate the importance of buffer caching research taking file system pre-fetching into consideration and comparing the actual disk I/Os and the execution time under different replacement algorithms. Pre-fetch throttling and data pinning schemes are also proposed to improve performance of I/O pre-fetching. The impact of compiler-directed I/O pre-fetching is also studied. A profiler and compiler-assisted adaptive I/O pre-fetching scheme targeting shared storage caches is proposed and experimentally evaluated. A slightly different approach is proposed by Subha et al. [49]. The buffer cache is divided into two units, the main cache unit and the pre-fetch unit. The sizes of both of the units are fixed. The total sizes of both of the units are a constant. Blocks are fetched in a one-block look-ahead pre-fetch principle. The block placement and replacement policies are defined. The replacement strategy depends on the most recently-accessed block and the defined miss counts of the blocks.

2.2.3.2 File system journal research

Journaling systems are used as temporary spaces to store changes to the file systems before they are written to disks. The journal mechanism reduces the amount of lost information and protects the integrity of the file systems in case of failure. It is a major component of a file system and can directly affect I/O performance. There are many journaling systems available. They have different sets of features and advantages. Prabhakaran et al. [50] analyze major journaling systems and their evolution, including Linux ext3, ReiserFS, Journal File System (JFS), and Windows New Technology File System (NTFS); in the process, they uncover many strengths and weaknesses of these journaling file systems. This paper illustrates in great detail the characteristics of the Linux ext3 journaling system.

Since different journaling file systems react differently when disasters happen, it is important to determine the reliabilities of them under critical conditions. Vijayan et al. propose a novel method to measure the robustness of journaling file systems under disk write failures [51]. The authors build models of how journaling file systems order disk writes under different journaling modes and use these models to inject write failures during file system updates. They apply their technique to the three important Linux journaling file systems: ext3, ReiserFS, and IBM JFS.

2.2.3.3 I/O workload research

In data intensive operations, workload characteristics are very important. Ensuring performance isolation and differentiating among workloads that share a storage infrastructure are basic requirements in consolidated data centers. Existing management tools rely on resource provisioning to meet performance goals; they require detailed knowledge of the system characteristics and the workloads. Provisioning is inherently slow to react to system and workload dynamics and, in the general case, it is not practical to provision for the worst case.

Ghemawat, Gobioff, and Leung [52] present an approach to designing a file system by optimizing the file system toward a specific application workload and the system's technological environment. The Google file system has constant monitoring, error detection, fault tolerance, and automatic recovery. The file system was also designed to support files with very large size and to optimize appending file operations. It is widely deployed within Google as the storage platform for the generation and processing of data.

With a slightly different approach that is applicable to a wide range of storage systems and makes no assumptions about workload characteristics, Karlsson, Karamanolis, and Zhu [53] propose a software-only solution that ensures predictable performance for storage access. The authors use an online feedback loop with an adaptive controller that throttles storage access requests to ensure that the available system throughput is shared among workloads, according to their performance goals and their relative importance.

Iliya K. Georgiev and Ivo I. Georgiev [54] offer a method to reduce storage latency by taking advantage of the relative interconnectivity between data objects. The authors follow the locality-of-reference principle to partition interrelated data objects on close disk areas or network storage nodes. There are two primary parts of the study: a clustering algorithm that groups related objects together and a read-ahead group-caching algorithm, which will use the result of the first algorithm. Data objects that are associated together are clustered in the same group and can be read from disk and cached together. The proposed clustering and cache algorithms do not use floating point, allowing direct and fast implementation on a variety of disk controllers.

2.2.3.4 I/O scheduling research

Despite the many technological improvements that have been made, disk speed still lags behind when compared to processor and memory speed. Disk-scheduling algorithms have, for the most part, been experimental in the past. To help lessen this performance bottleneck, Andrews, Bender, and Zhang [55] propose a disk-scheduling algorithm that appears to give higher throughput than previously existing head-scheduling algorithms. The authors state that their goal is to schedule the disk head, so that it services all the requests in the shortest time possible using a 3/2-approximation algorithm. They consider a special case in which the disk-scheduling problem is related to the special case of the asymmetric traveling-salesman problem. In this particular case, optimal tour could be found in polynomial time and could be approximate for the disk-scheduling problem.

Periodic real-time I/O scheduling for continuous data streams and the effect of scheduling on communication performance are investigated by Altilar and Paker [56] who examine periodic real-time scheduling, assuming that the application is communication constrained where input and output data sizes are not equal.

Since I/O scheduling helps improve performance between data resources operating at different speeds, it stands to reason that I/O scheduling research can provide good solutions for tape systems. Much work has been done in this area [57-59]. New scheduling algorithms and schemes for the replacement and replication of hot data are developed and provide improvements over a wide variety of workload characteristics. The issues of scheduling across multiple tapes or disks, instead of only one or two media, are also investigated. An efficient algorithm for single-drive libraries that produce optimal schedules is developed. The scheduling problem for multiple drives is shown to be NP-complete, and an efficient and effective heuristic algorithm is used. The performance characteristic of tertiary storage is also considered with respect to efficient retrieval of data stored in tertiary storage devices with multiple platters. I/O scheduling algorithms use different heuristic methods for scheduling to reduce the latency involved in retrieving data.

2.2.3.5 Tertiary storage and hierarchy storage research

Tertiary storage has become more and more popular despite the fact that tape library is a very old technology. The rapid development of online applications results in increasing on-line access to massive amounts of data. Advanced database applications are also reaching the limit for a disk storage system in terms of both cost and scalability. Large-scale storage systems, using only magnetic disks with their high cost and low storage density, can be impractical or too costly for many applications. Cheaper and denser tertiary storage systems are being integrated into the storage hierarchies of applications. Applications utilizing tertiary storage include multimedia databases, data warehouses, scientific databases, and digital libraries. Much research has been done by the database research community to optimize the performance of tertiary storage. A major part of a tertiary storage system is tape libraries that have unusual performance characteristics. Although the system could archive at a very high transfer rate, the access latencies can reach several minutes. The trend of tertiary storage is presented in detail one report [60, 61] which summarizes the current state of the art in tertiary storage systems. Their analysis of product data indicates that in contrast to disk technology, tertiary storage products have significant variability in terms of data transfer rates as well as other performance figures. With the new technology, tape storage is two orders of magnitude more efficient than disk in terms of cost per terabyte and physical volume per terabyte. This certainly is very attractive for the development of databases with very large volumes of data. A system that seamlessly combines both disk storage and tape storage would be of great value. However, the biggest problem with tape storage is that the random access latency of tape is three to four orders of magnitude slower than disk. Thus, this problem of access latency has to be resolved before a system of online tape bulk storage could be utilized. Many studies have been done to find a solution to this problem [62-64]. Detailed measurement of tertiary storage systems, which included several tape drives and robotic storage libraries, such as Digital Linear Tape (DLT) 4000, DLT 7000, Ampex 310, IBM 3590, 4mm Digital Audio Tape (DAT), and the Sony Digital Tape Format (DTF) drive are presented. This mixture of equipment includes high and low performance drives, serpentine and helical scan drives, and cartridge and cassette tapes, and gives a big picture of different aspects of tertiary storage systems, providing a better understanding of issues related to utilizing tertiary storage. Algorithms to reduce the latency are presented. The results show that the algorithms could improve the latency of random access to tape significantly. A disk-tape join algorithm is also discussed. The algorithm has three phases: hashing, merging, and probing. Each phase is designed, so that join results can be produced. The authors show that the algorithm is very efficient.

A different approach to the problem that uses optimal data placement strategies is also examined [65, 66]. Traditionally, this approach is used for disks and disk arrays. Tertiary libraries have been neglected, even though tertiary storage remains three orders of magnitude slower than secondary storage. This issue is addressed by deriving an optimal placement algorithm for disk libraries and tape libraries. The authors also look at different scheduling algorithms, different configurations of disk libraries, and different tape library technologies, and show how these impact the placement strategy. The special attributes of stored data that have an impact on the optimal placement are also discussed.

A high-level Application Programming Interface (API) for the user is also looked at. No et al. [67] propose a system that combines the advantages of both file I/O and databases. By using various I/O optimizations available in Message Passing Interface (MPI) I/O, such as collective I/O and noncontiguous requests that are transparent, the user can write and read their data with the performance of parallel file I/O without having to go into the details of actually optimizing their file I/O.

The database research community is not the only one interested in optimizing tertiary storage. With the improvement in communication technology, multimedia information systems have become an important component of many application domains from library information systems to entertainment technology. The storage system must support several data types, such as text, image, video, and graphics defined as multimedia objects, which need to be synchronized and to meet some timing requirements. Berson et al. [68] propose an approach to eliminate the problem of frequent disruptions in multimedia data accessing by de-clustering multimedia objects across multiple disk drives. This method uses the aggregate bandwidth of several disks to support the continuous data retrieval. The issue where data is being continuously uploaded from tertiary storage for display purposes is also being investigated. Triantafillou and Papadakis [69] present a method which can fully utilize and reserve the tape drive bandwidth, therefore allowing the secondary storage to serve more requests without increasing the memory space on the host.

Unlike other researchers who utilize software solutions entirely for their hierarchy storage system, Wilkes et al. [70] propose a two-level storage hierarchy implemented inside a single disk-array controller. The technology is used to automatically and transparently manage migration of data blocks between the two levels of the storage hierarchy when access patterns change. The system is very easy to use, has full redundancy, and is suitable for many different workloads. The system could also adapt to dynamic workload changes and perform very well while being able to keep the amount of front-end random access memory (RAM) cache and number of spindles relatively small.

Data latency is not the only problem with tertiary storage systems. Cache replacement and cache filtering could potentially cause some problems as well. Investigated in two papers [71, 72], the authors introduce a special cache replacement algorithm to maximize efficiency. They define a utility function for ranking the candidate objects for eviction and then offer an efficient algorithm for computing the replacement policy based on this function. They have evaluated the system, using simulation with a wide range of workloads. They also compare their policy with traditional replacement policies, such as least frequently used (LFU) and least recently used (LRU), using simulations of both synthetic and real workloads of file accesses to tertiary storage. Also working on improving this bottleneck, Shoshani et al. [73, 74] describe an architecture designed to optimize the use of a disk cache and thus minimize the number of files read from tape. The authors use a specialized index to locate the relevant data on tapes and coordinate file caching over multiple queries. They also include the results of various tests that demonstrate the benefits and efficiency gained from using the system. The authors also note a method to identify the bundling of files before caching. After the file bundles are identified, a scheduler is set up to schedule bundle caching in such a way that files shared between bundles are not removed from the cache unnecessarily.

2.2.3.6 Storage virtualization research

In an enterprise storage system, the total capacity could reach a petabyte, and the number of disk drives and storage devices could reach tens of thousands. These huge numbers of storage devices could serve thousands of host computers. A system of this scale could be very difficult to design. There are so many choices to be made, and the interactions between them are not always predictable. Storage system provisioning is tedious and complicated to do by hand which usually leads to solutions that are grossly over-provisioned, substantially under-performing, or, in the worst case, both. Mass storage systems and cluster storage systems are designed with high performance RAID clusters, robotic tape libraries, or a combination of both. It is challenging to provide a cluster storage system, which becomes more and more popular for data-intensive applications with its ability to expand, its high availability and scalability, and its strong consistency. To solve this configuration nightmare, much work has done in this area [75-79]. The system can be optimized at design time, using declarative specifications of application requirements and device capabilities. Another approach is to provide storage systems that can self-organize. Such systems utilize several techniques, including adaptive storage management methods, elastic page allocation in multi-size paging architecture, partial analysis controls, partial swapping, and adaptive pre-paging. Experimental systems could manage hundreds of terabytes of virtual storage. A self-organizing storage cluster could also exploit data location schemes to dynamically and rapidly adapt to configuration changes, improving availability and manageability. When there is a change in server resources caused by failure or recovery, the system will dynamically add or remove those servers. The system is adaptive, self-tuning, and able to provide nearly uniform performance across all servers.

The idea of creating virtual storage systems by combining several commodity disk drives or unused storage pools is also being investigated by several researchers [80-86]. They believe that the storage densities of this type of storage organization could match or exceed tape libraries while maintaining the performance of disk arrays. The I/O performance is achieved by using multiple techniques to aggregate desktop network bandwidth and local I/O bandwidth, such as data striping, a combination of peer-to-peer storage and network file system, or low-level Internet Small Computer System Interface (iSCSI) protocol. Different approaches for the bandwidth problem, such as allocating disk bandwidth based on lexicographic minimization [87], efficiency-aware real-time disk-scheduling algorithm [88], or virtual storage controller [89] are also receiving attention. In addition to achieving large bandwidth, sophisticated management software is also developed to make the collection of disks appear to a client as only one storage system. Virtual storage systems can tolerate and recover from disk, server, and network failure. This capability proves to be very beneficial in distributed environments.

The idea of virtual storage is also being utilized in tertiary storage systems [90] and virtual machines [91]. When used in tertiary storage systems, virtual storage allows the system to hide the details of robotic libraries and media characteristics behind a uniform, random access, block-oriented interface. It also allows the system to avoid media mount operations for writes, giving write performance similar to that of secondary storage. When used in virtual machines (VM), virtual storage provides on-demand cross-domain access to VM state. It enables private file system channels for VM instantiation and supports user-level and write-back disk caches. It also leverages application-specific meta-data associated with files to expedite data transfers.

2.3 Summary

This chapter discusses related work and surveys research in the storage modeling and simulation area. Previous work in the network modeling and simulation areas is also examined. These areas of research help to develop the network simulation component in the PVFS file system simulation model. In addition to storage and network modeling topics, this part also looks at several studies in other areas of storage technologies, such as buffer cache, journaling system, I/O scheduling, tertiary and hierarchical storage, and storage virtualization. Work done in these areas helps accurately develop storage components in the file system simulation model.

C h a p t e r 3

PERFORMANCE MEASUREMENTS AND WORKLOAD STUDY

3.1 Introduction

Before implementing the system simulation model, measurement studies need to be done to determine the performance characteristics of the targeted systems, which are ext3 and PVFS. Ext3 is the candidate for the local file system simulation model, and PVFS is the candidate for the parallel file system simulation model.

3.2 Local file system performance study

The objective of the performance measurement study is to analyze the behavior of the proposed ext3 file system. By studying the ext3 file system performance, we can better understand the level of detail needed for the simulation model.

3.2.1 Experimental setup

Performance measurement experiments are executed on production computers (Dell PowerEdge 1850) with the hardware configurations shown in Table 1. The test computers are set up to have a single drive with no redundant array of independent disks (RAID), two single drives with RAID 0 configuration, or connections to a SAN, depending on the experiment. The test computers are located in an isolated environment with dedicated resources to minimize extra factors affecting performance study. The primary I/O testing suite used in the following experiments is iozone [92].

Table 1: Test system configuration

Processors	Dual Intel Xeon processors at 2.8GHz
Front side bus	533MHz
Cache	512KB L2 cache
Chipset	ServerWorks GC LE
Memory	4GB DDR-2 400 SDRAM
Drive controller	Embedded dual channel Ultra320 SCSI
RAID controller	PERC 4/Di
Hard drives	Fujitsu MAT3147NC 147GB 10,000 rpm
	Seagate ST3146707LC 146GB 10,000 rpm
External array	EMC Clariion CX700
HBA card	Qlogic 2340

3.2.2 I/O performance study with different file sizes and block sizes

In a real-world environment, a day to day workload could consist of many different file sizes, and the I/O operations could use many different block sizes. The purpose of this measurement study is to determine the suitable workload configuration for the model. First, sequential I/O read performance is examined using a set of small to large size files (from 4Kbytes to 1Gbyte). The results for the sequential I/O read measurement experiments are presented in Figure 5. Figure 5 is a 3-dimensional graph in which the z-axis represents throughput of the test file system. The x-axis and y-axis represent test block size and test file size respectively. Figure 5 shows that file sizes do not affect sequential I/O read performance.

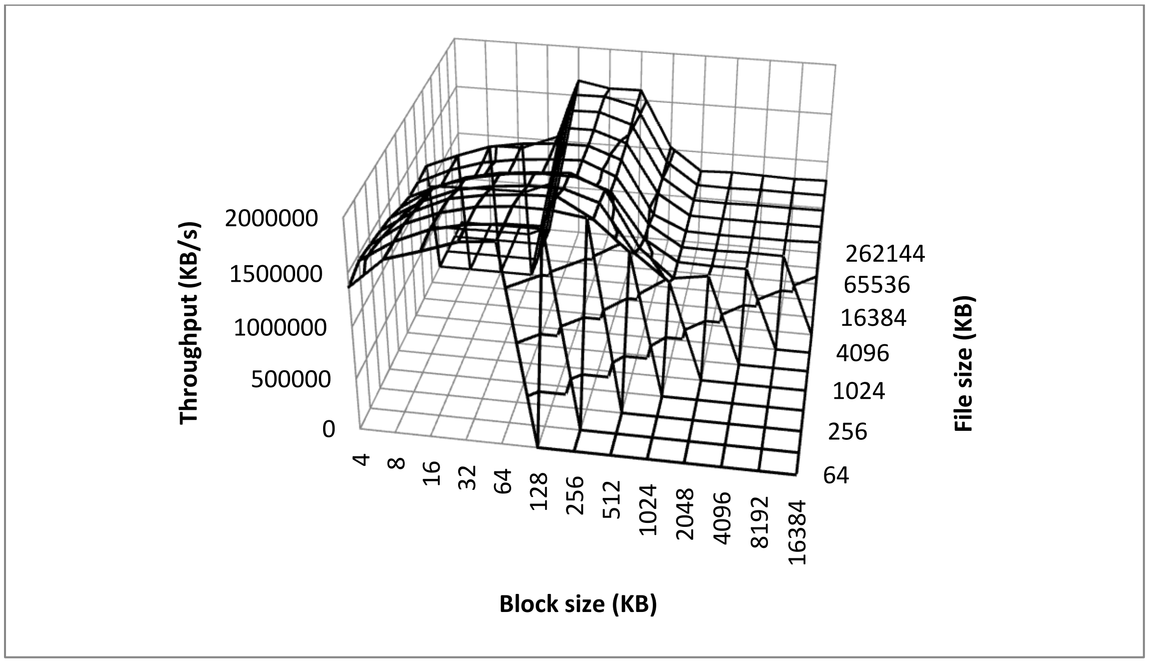


Figure 5: Sequential I/O read performance

For better viewing, Figure 6 presents a closer look at a section of the experiment results.

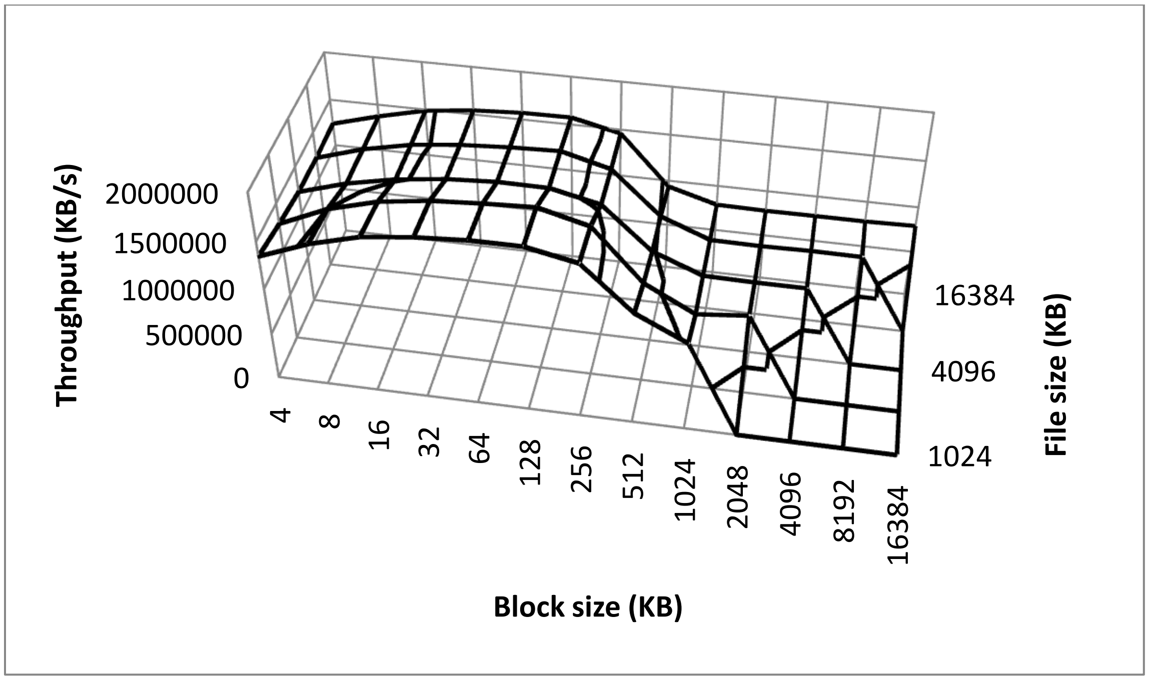


Figure 6: Detail view - sequential I/O read performance

Two observations can be made from the measurements in Figure 5 and Figure 6. First, because reading is sequential and kernel cache effects are minimized, the I/O read performance is not affected by file size. There is an exception to this, and it will be presented at the end of this section. Secondly, the I/O read performance starts to drop after operation block size reaches around 64Kbytes. More details of this drop in performance are described in section 3.2.3.

Similar performance behavior can also be observed for I/O write operations. The write performance, using a similar set of files and block sizes, varies the same way as with read performance. The results for the sequential I/O write measurement experiments are presented in Figure 7. Figure 7 is also a 3-dimentional graph similar to Figure 6.

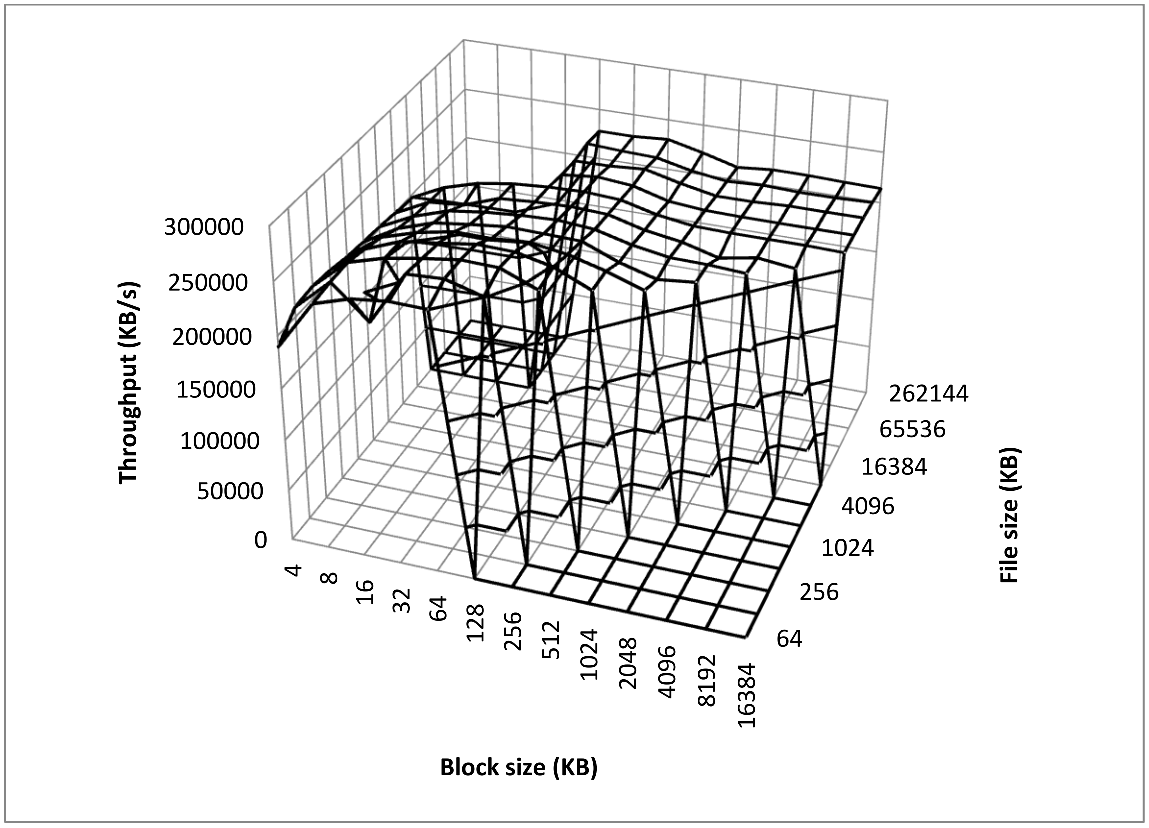


Figure 7: Sequential I/O write performance

For better viewing, a section of the experiment results is displayed in detail in Figure 8.

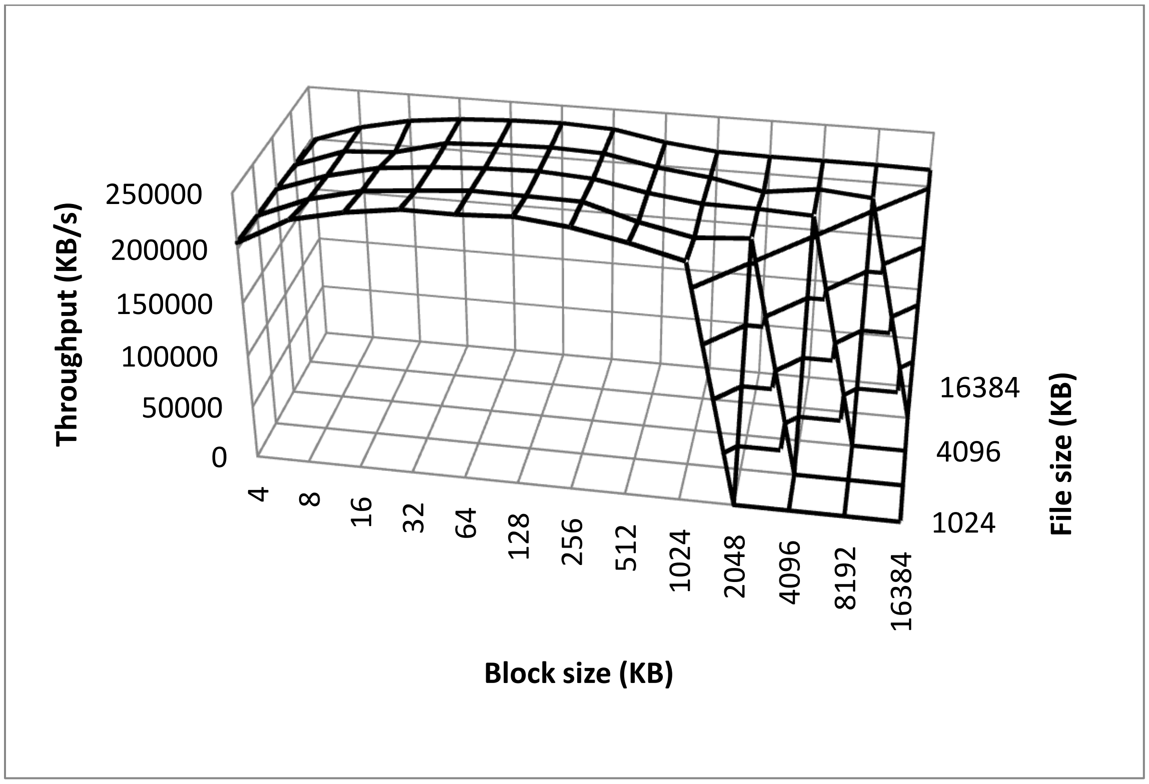


Figure 8: Detail view- sequential I/O write performance

Figure 7 and Figure 8 show that file sizes also do not affect I/O write performance. Overall, in both read and write measurements, results show that file size does not affect I/O performance. For the sequential workloads used for these measurement experiments, it is the block size of the I/O operation that affects the I/O performance. This statement holds true until the file size reaches the physical memory capacity of the machine. If the file size reaches the memory capacity, memory reclaiming is triggered and swapping also occurs. The memory reclaiming and swapping process causes disk thrashing, leading to a very large I/O performance degradation [93].

Figure 9 shows sequential I/O read performance as the file size is allowed to increase to the physical memory capacity of the machine. In these experiments, the test machine has 4Gbytes of physical memory.

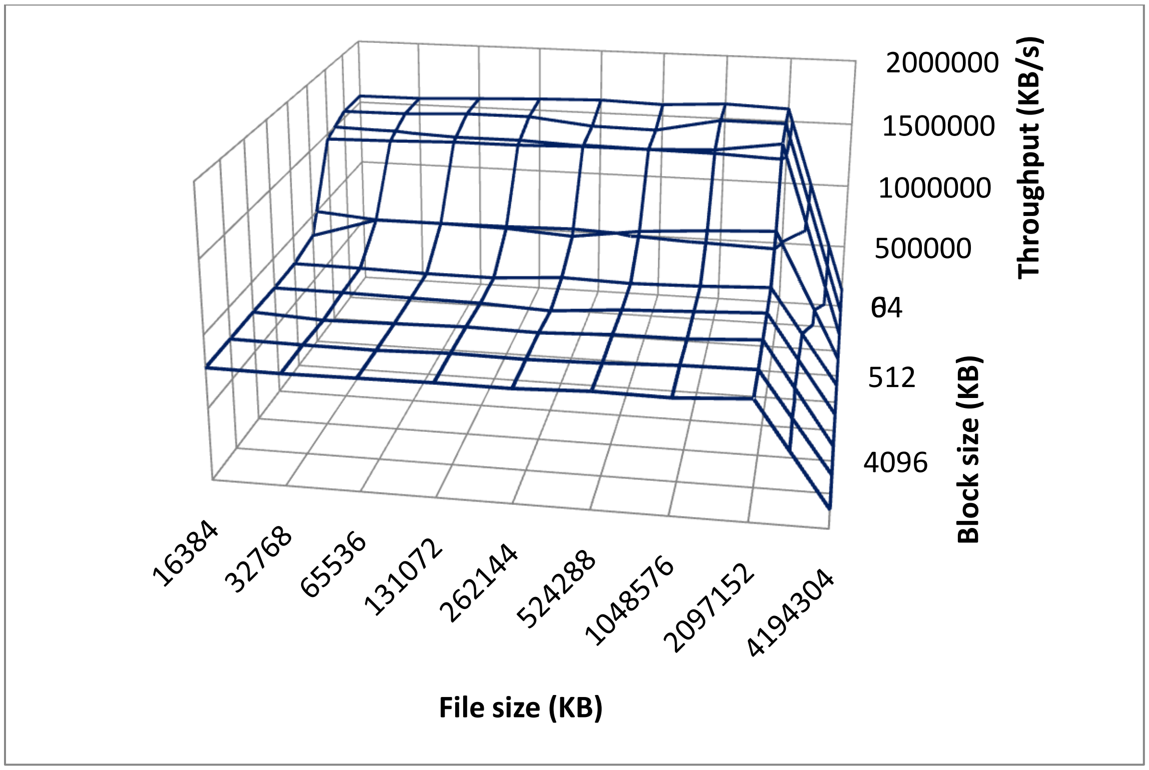


Figure 9: Physical memory capacity and I/O performance

Sequential write performance shares the same characteristic. However, under Linux, a threshold (dirty ratio) is usually in place to synchronously flush data to disk. This threshold is configurable via the Linux kernel parameters. If this threshold is set equal to total physical memory capacity of the machine, sequential write will behave the same as sequential read presented above. In Figure 10 the dirty threshold is set to the default value put forth by Red Hat (~512MB for the test machine). This shows that the dirty ratio threshold affects file write performance.

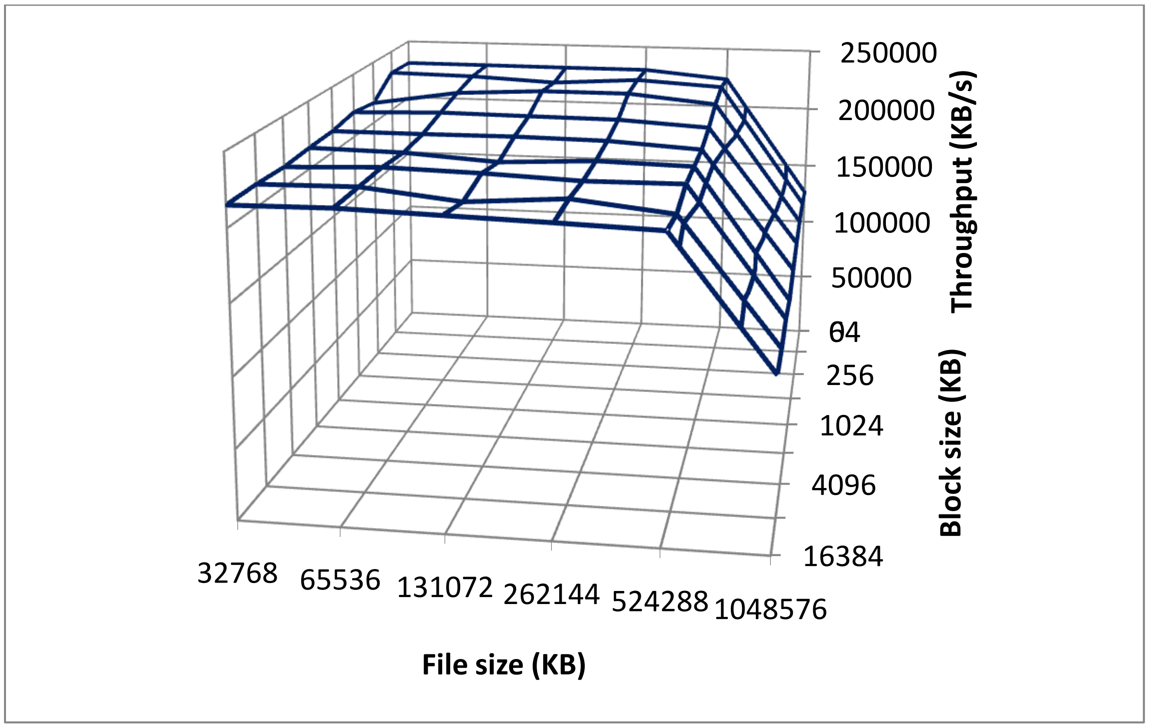


Figure 10: Dirty ratio parameter and I/O performance

For random I/O, because of the nature of the I/O pattern, a set of random I/O requests are used to study performance instead of trying to read in a whole file using random requests. Therefore, in the case of random I/O, file size is not a concern.

Generally, an application is designed to avoid processing, all at once, files that are bigger than its physical memory capacity without breaking them into smaller chunks, since doing so will degrade the performance of the system. From that assumption, 512Mbytes is selected to be the standard file size for all models in the performance study. This file size is large enough to study the performance of the model, yet small enough for the simulation to run within a reasonable time.

3.2.3 I/O performance behavior of ext3 file system

This section describes the measured performance behavior of an ext3 file system. Figure 11 shows the I/O read performance of the ext3 file system that is measured with different hardware sub-systems. The measurements in Figure 11 illustrate that the ext3 file system hides the performance characteristics of the hardware storage sub-systems very well. The performance curve shapes are very similar in spite of hardware sub-system differences.

Figure 11 also shows that when the block size reaches 64Kbytes, the performance of the file system starts to drop. Figure 12 shows that the I/O write performance exhibits a similar behavior, but not as dramatic.

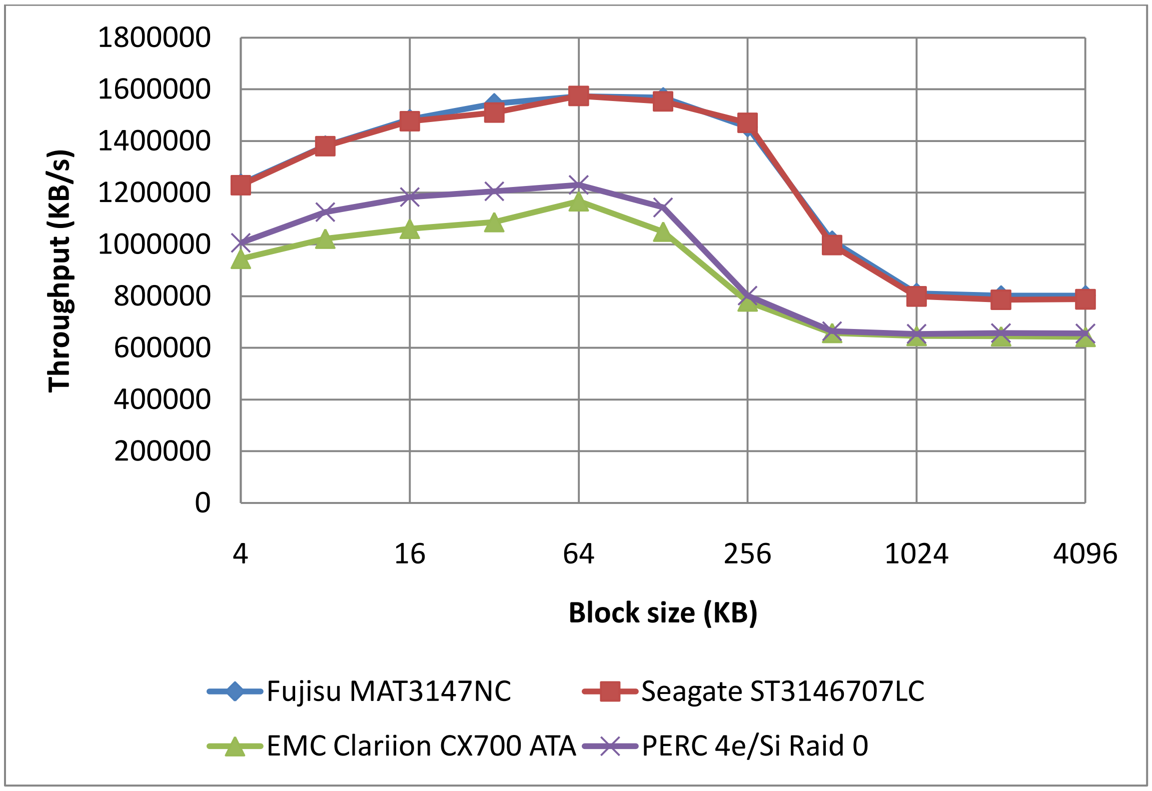


Figure 11: I/O read performance with different hardware

To find the root cause behind this drop, kernel tracing was performed, and operation response times were carefully profiled along the I/O path. The two kernel functions, copy_to_user and copy_from_user, exhibit interesting response times. Figure 13 shows the average response times of copy_to_user and copy_from_user functions as I/O block size increases. This is also the reason why the drop in I/O write performance is not as dramatic as the performance drop in the I/O read case. The response time of the copy_to_user function, when compared to the overall I/O read time, is much more significant than the same response time of the copy_from_user function when compared to the overall I/O write time.

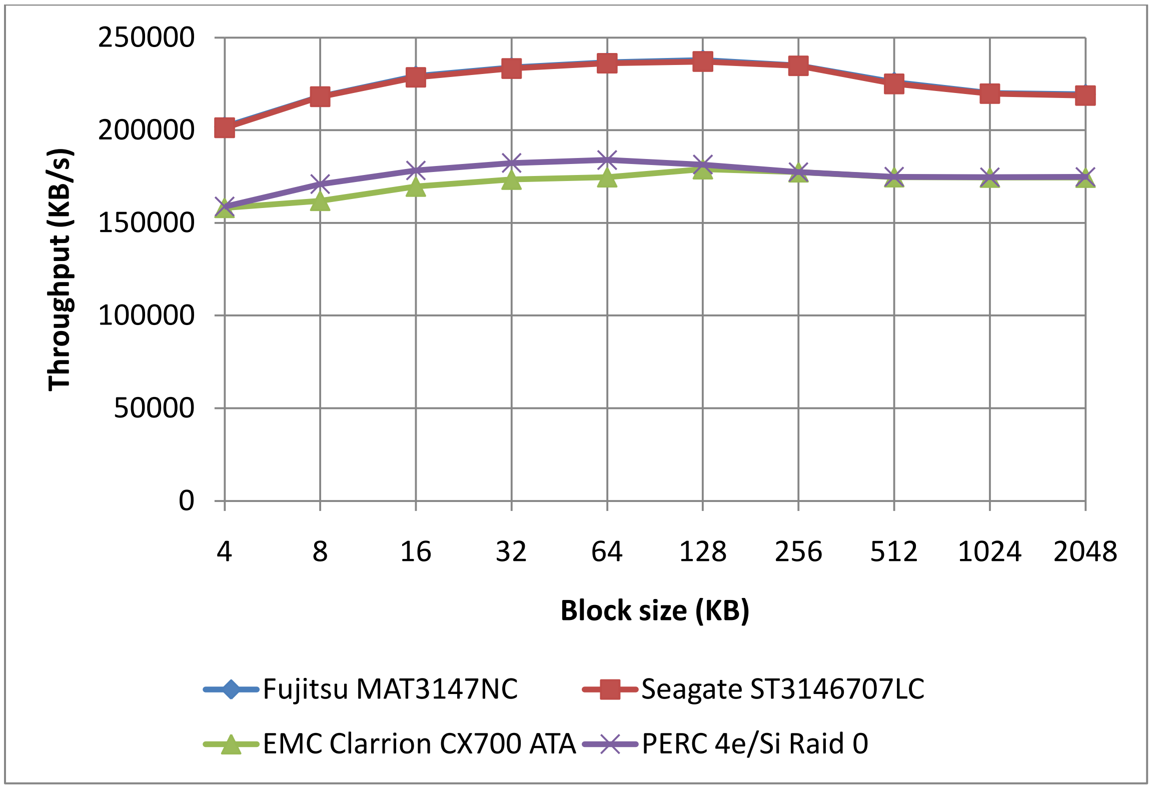


Figure 12: I/O write performance with different hardware

Examination of the kernel code for the two functions shows no evidence from the functions' codes to support this performance behavior [94]. On the other hand, the shape of the performance curve suggests that this performance behavior may be caused by constraints in system resources. To investigate system resource utilization, low-level profiling of the test system was performed using oProfile [95] while running the I/O experiments.

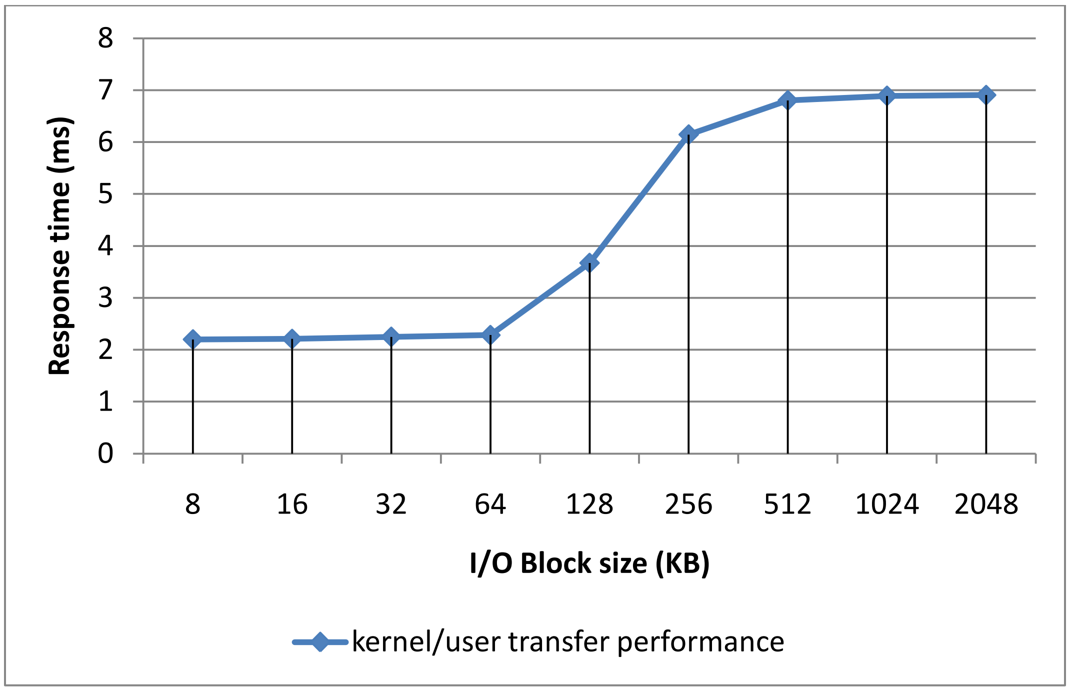


Figure 13: Average response time of data transferring between kernel and user space

The results of L2 cache behavior of the copy_to_user and copy_from_user functions obtained during I/O benchmark testing are shown in Figure 14. Those measurements show that the L2 cache misses increase after the block size reaches 64Kbytes and become very noticeable after the block size of 128Kbytes. The L2 cache misses continue to increase even after the block size goes beyond 1024Kbytes.

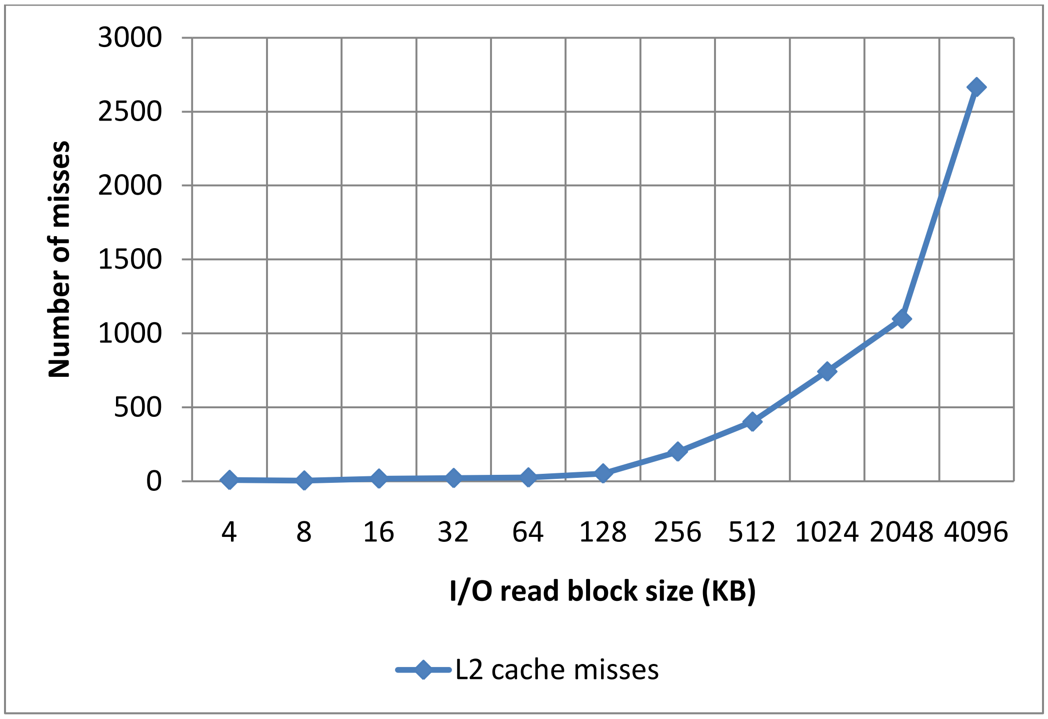


Figure 14: L2 cache misses during I/O with different block sizes

When the block size is increased beyond 1024Kbytes, the usefulness of the L2 cache in copying data from kernel space to user space is completely negated, and the response time levels off, as shown in Figure 13.

Table 2: CPU L2 cache sizes vs. performance drop off points

CPU L2 cache size (KB)	I/O Block size where performance starts to drop (KB)
512	64
1024	128
2048	256
4096	512

The I/O block sizes where L2 cache misses become significant are important for the model. Additional measurements using different central processing units (CPUs) of the same model (Intel Xeon 2.8 GHz) with different L2 cache size configurations were performed. The measurements in Table 2 show that when the I/O block size reaches about 1/8 of the total L2 cache size, copy_to_user and copy_from_user performance starts to drop.

3.3 PVFS file system performance study

The objective of the performance measurement study is to analyze the behavior of the proposed PVFS file system. By studying the PVFS file system performance; we can better understand the level of detail needed for the simulation model.

3.3.1 Experimental setup

Performance measurement experiments are executed on the PVFS cluster with the I/O servers (Dell PowerEdge 1850) configured as shown in Table 3. The I/O servers are set up to have 5 drives with RAID 5 configuration. The PVFS cluster is located in an isolated environment with dedicated resources to minimize extra factors that affect performance study. The primary I/O testing suite used in the following experiments is iozone [92].

Table 3: PVFS test cluster machine configuration

Processors	Dual Intel Xeon processors at 2.8GHz
Front side bus	533MHz
Cache	512KB L2 cache
Chipset	ServerWorks GC LE
Memory	4GB DDR-2 400 SDRAM
Drive controller	Embedded dual channel Ultra320 SCSI
Hard drives	Fujitsu MAT3147NC 147GB 10,000 rpm
	Seagate ST3146707LC 146GB 10,000 rpm

The PVFS cluster is built using the configuration presented in Figure 15. The cluster has a total of 4 I/O servers with the total capacity of approximately 2 Tbytes. The PVFS cluster can provide a decent space for testing and enough I/O servers to run performance evaluation.

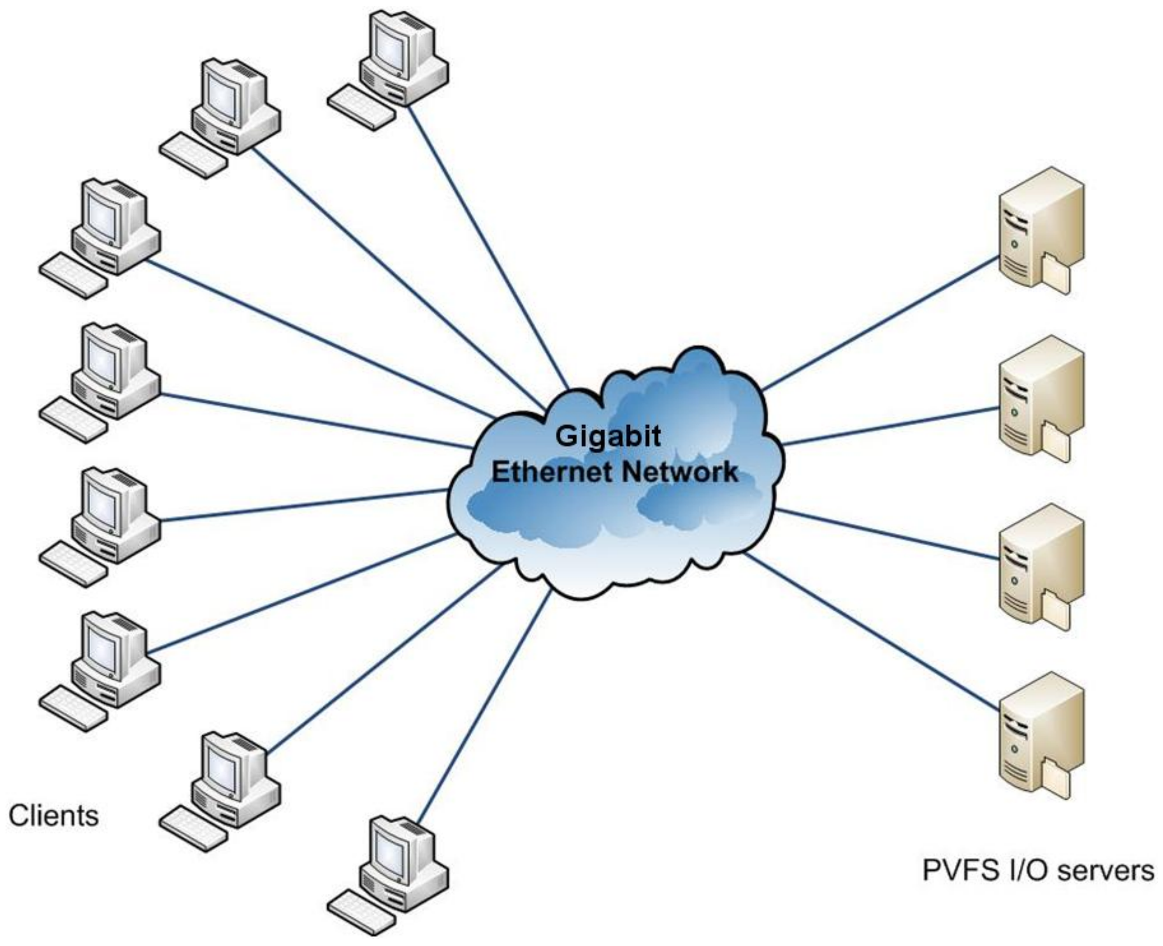


Figure 15: PVFS test cluster architecture

3.3.2 I/O workload study

Designed to achieve massive performance by parallelizing I/O accesses, PVFS, like any other parallel file system, works best with large files, using sequential access with large block size. Knowing this, applications running on PVFS file systems are configured to take advantage of this behavior as much as possible. Using a very large I/O buffer, an application sequentially accesses the file system, using large block sizes of up to 100 Mbytes. Observing I/O workload on multiple PVFS file systems in a Shared Production environment with approximately 276 Tbytes of total capacity, the breakdown of I/O workload percentage is shown in Table 4.

Table 4: PVFS I/O workload breakdown

Pure random I/O	0.00028%
Mix random I/O and sequential I/O	2.047%
Large block size sequential I/O	97.952%

The pure random I/O accesses are very small in comparison to the sequential accesses. In order to study the I/O pattern of the sequential accesses, I/O traces are captured on the PVFS file system. The captured I/O pattern is presented in Figure 16.

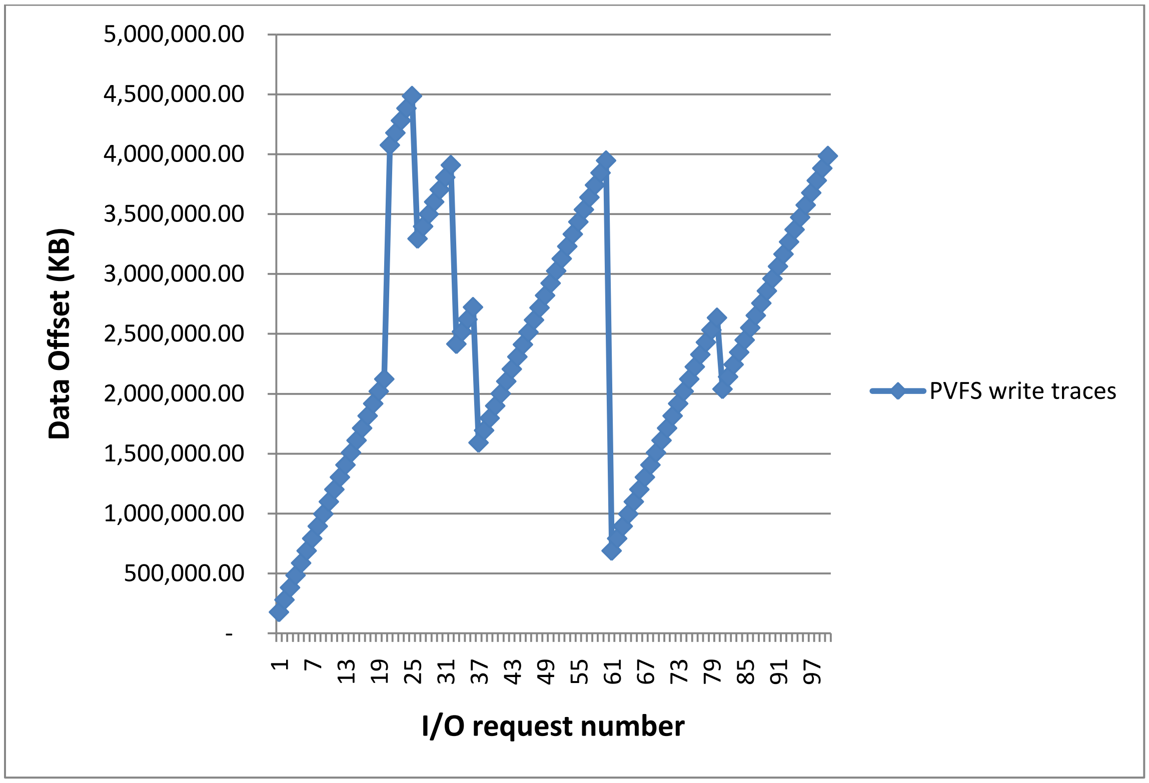


Figure 16: PVFS captured I/O traces

The I/O traces are a combination of multiple sequential I/O accesses, starting at different sections of the data. Due to the large block size, the slopes of the access pattern charts are very steep.

From the real-world workload breakdown, it is clear that pure random I/O occupies a very small amount of workload on a parallel file system. Of course, I/O access pattern on a file system is greatly dependent on the user of the file system. However, PVFS obviously was not designed for a small files, random I/O workload. If one should choose to use PVFS for such a workload, the performance of the parallel file system will be greatly degraded. For that reason, only sequential I/O workload is studied and evaluated in the simulation model.

3.3.3 I/O performance study with different file sizes and block sizes

Similar to the local disk experiment, this measurement study is to observe the I/O behavior of the PVFS file system when file size and block size are changing. First, sequential I/O read performance is examined, using a set of small to large size files (from 4Kbytes to 1Gbytes).

The results for the sequential I/O read measurement experiments are presented in Figure 17. These measurements show that, similar to the local file system, the I/O read performance is not affected by file size.

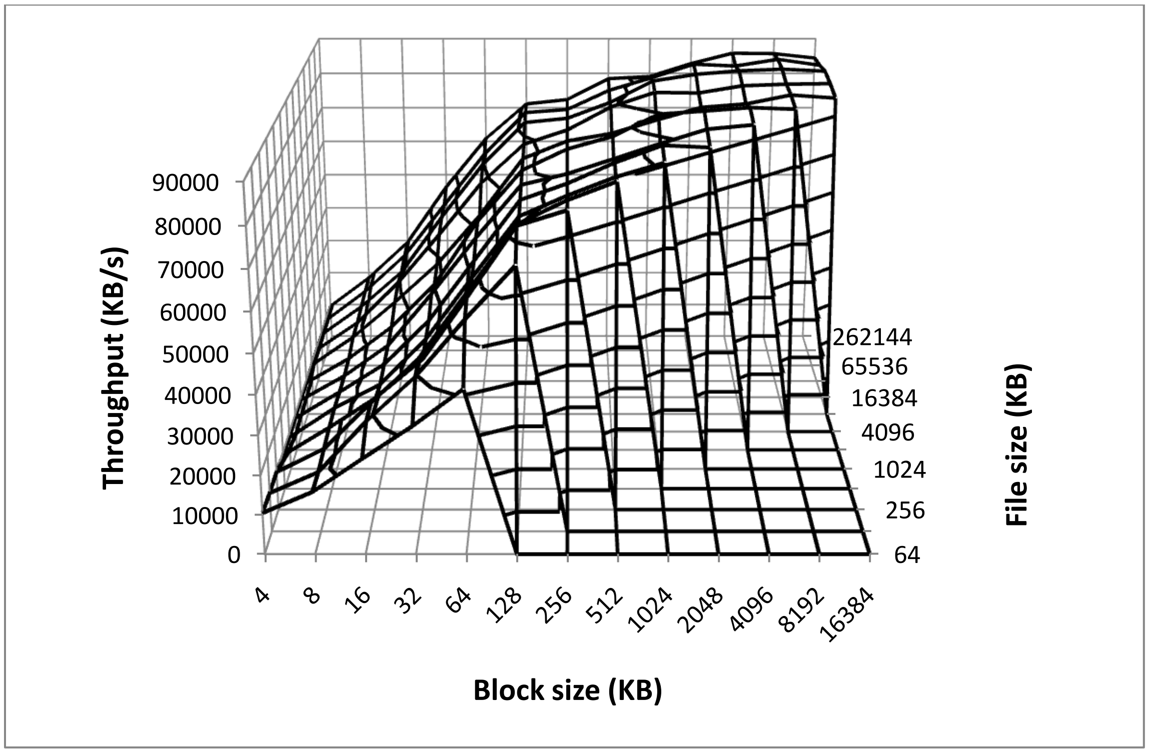


Figure 17: PVFS sequential I/O read performance

The write performance uses a similar set of files, and the block sizes vary the same way as with read performance. However, I/O write performance shows slight differences. The I/O write throughputs at small file sizes are less than I/O throughput at larger file sizes. This observation shows that the I/O performances are not at peak level until file size is equal or greater than 2Mbytes. The nature of PVFS is what causes this performance behavior. PVFS is a parallel file system. Files stored in a PVFS file system are divided into multiple stripes and are distributed across multiple I/O servers. By striping file contents across multiple servers, a client machine can access several pieces of file data at the same time. For a small file, this mechanism creates some overhead which causes the I/O performance to become lower until the file size is large enough to receive full advantage from the workload parallelization as shown in Figure 18. The results for the sequential I/O write measurement experiments are presented in Figure 18.

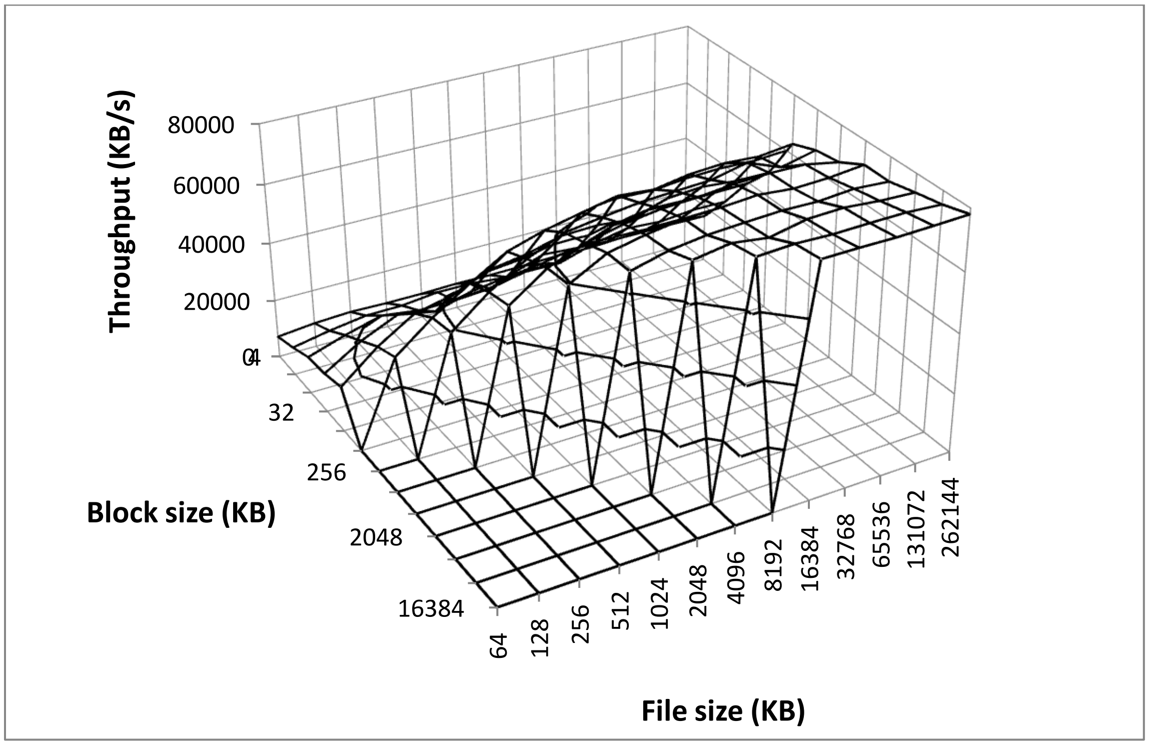


Figure 18: PVFS sequential I/O write performance

According to the read and write measurement results, after file sizes become large enough, PVFS I/O performance does not change when the file size changes to a greater amount. Reaching this stable level, I/O performance is now affected by the block size of the I/O operations instead. Of course, similar to local file system behavior, the I/O performance drops sharply when the file size reaches the physical memory capacity of the machine. This behavior is caused by memory reclaiming and swapping, which in turn causes disk thrashing, leading to a very large I/O performance degradation.

3.4 Summary

This chapter presents background performance studies of a local file system – the Linux ext3 file system. Performance characteristics of the ext3 file system are presented. The relations between file sizes, I/O block sizes, and file system performance are investigated. The performance behaviors of the ext3 file system are also carefully examined. The relation between CPU L2 cache and the I/O read and write behavior is also pointed out. The real-world I/O access pattern and production workload on PVFS file systems are also studied in this part. The performance measurements of the PVFS file system are also presented.

C h a p t e r   4

DESIGN OF A SIMULATION MODEL FOR LOCAL FILE SYSTEM

4.1 Introduction

The simulation model for the local file system is the most basic foundation for file system modeling. It mimics the behavior of a local file system over a block device. It interfaces with higher-level software, such as applications or parallel file system servers, and provides the response time associated with each I/O request. This chapter discusses the design of a simulation model for a local file system. The file system model is expected to be able to provide end-to-end file system performance against a pre-defined workload. System designers could use the model to evaluate file system performance in different scenarios and to perform bottle-neck analysis. It could also be used for 'what-if' type of provisioning analysis. The implementation of the simulation model is presented in a top down fashion, from application level down to the hard disk level, and each level is described using Colored Petri Nets.

4.2 Assumptions and model limitations

A complex scientific or business application may have both I/O reads and I/O writes at the same time. However, a typical I/O pattern often seen is a large read operation followed by computing which is then followed by a large I/O write. Many times, the phase of execution where the application is reading is separated from the phase where the application is writing. With that in mind, the simulation model is divided into an I/O read model and an I/O write model. These are simulated separately to simplify the multiple conditions when simulating the file system.

The ext3 journaling mechanism has three modes of operation: write back, ordered and journal modes. Write back mode and ordered mode are quite similar except that ordered mode guarantees that data is flushed to disk before the metadata is written to disk. Journal mode, however, is very different as it writes both data and metadata into the journal. The default mode for ext3 under Linux is ordered mode as it has good protection and performance. The model is designed to work with all three modes. However, since write back mode and ordered mode are similar, only the default —ordered mode— and full data journal mode are examined in detail.

Although no data flushing is needed at the end of the benchmark before the data file is closed, for stability and validity of the performance result, an fsync() command is enforced to flush all dirty data to disk at the end of the benchmark and simulation. The performance study of the ext3 file system, which is discussed in detail in Chapter 3, shows that the Linux file system does a very good job at hiding the performance characteristics of the lower level hardware sub-system. As a result of this, simple queuing models are used for I/O scheduler and disk sub-system model.

4.3 File read model implementation

From the application standpoint, reading a file basically divides the file into smaller manageable blocks and uses the fread function to read blocks into memory.

```
while (!feof(file_handle)) {
    bytes_read = fread(buffer, block_size, number_of_block, file_handle);
}
```

The model for this operation is simple. A loop breaks the needed file into multiple blocks of read requests and passes the list to the fread simulation module. The result of this operation is an array of data passed back from fread after reading it from the disk. The Petri Net for this operation is presented in Figure 19.

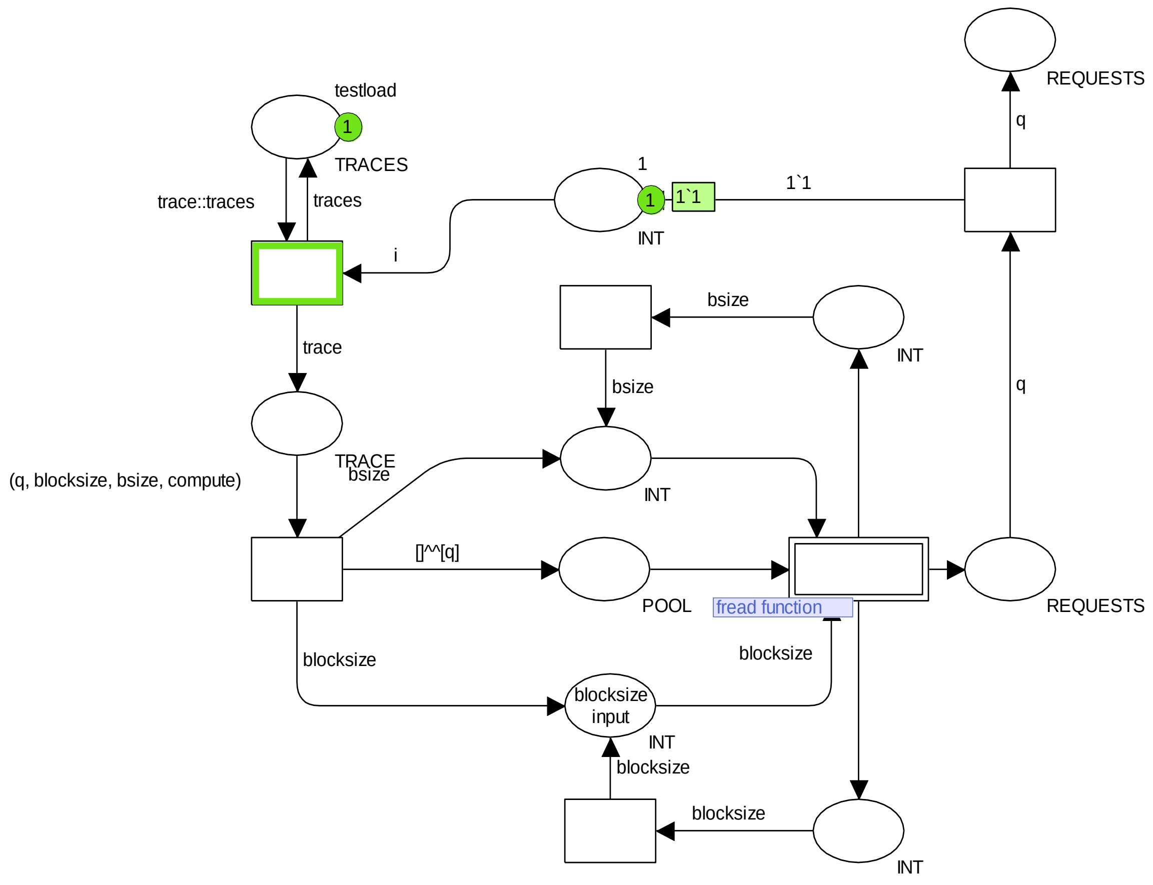


Figure 19: High level application read model

The implementation of the fread function in standard C library could be described as dividing the block of read requests into a list of single read requests and passing this list to the read system call to carry out the actual read from the disk. The result of fread is an array of data gather from the read system call, and this array is returned back to the application.

The Petri Net implementation of the fread function is presented in Figure 20.

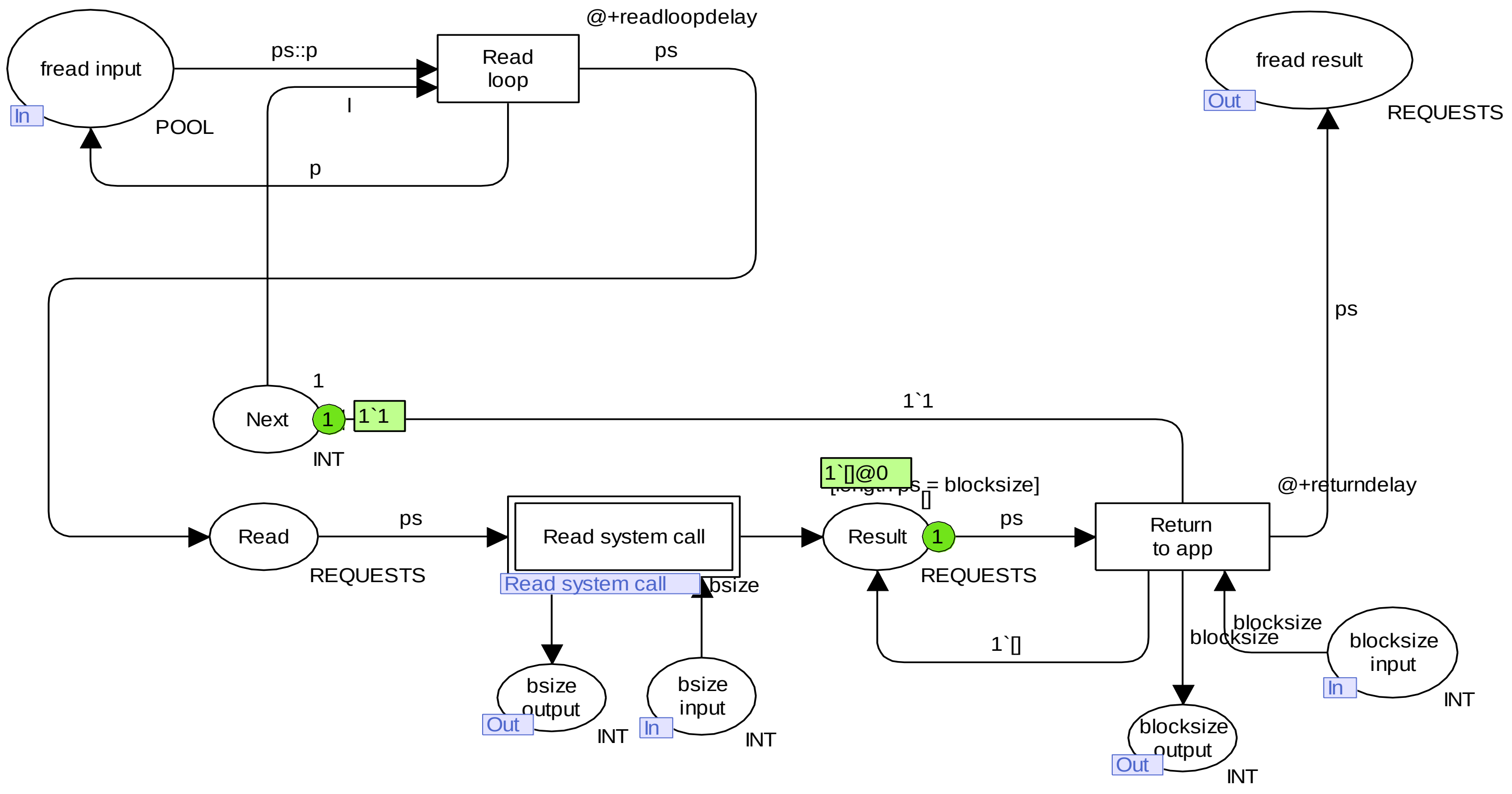


Figure 20: The fread function model

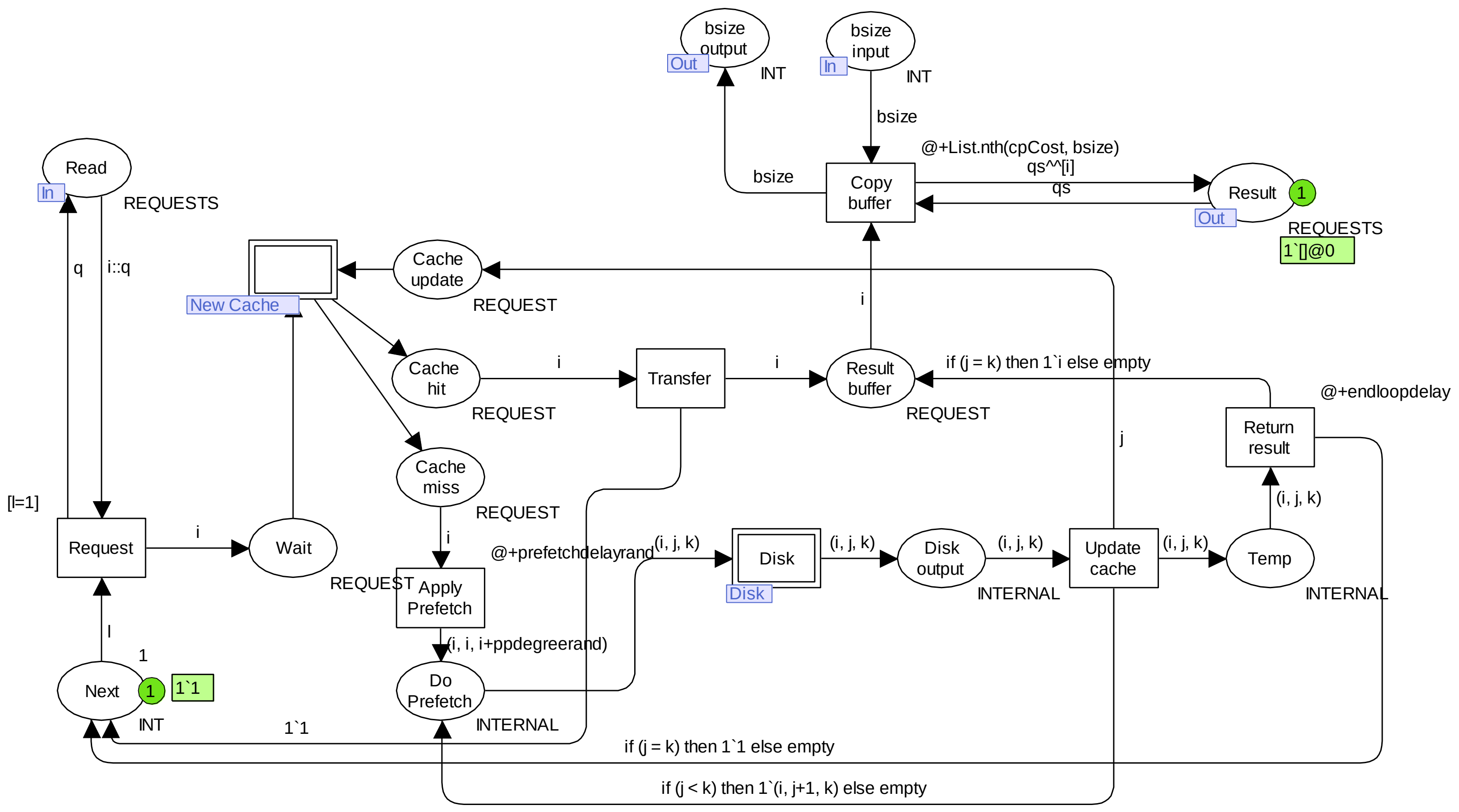


Figure 21: The generic_file_read model

In kernel space, the read system call is mapped to the function generic_file_read. The Petri Net implementation of the generic_file_read function is presented in Figure 21.

The Petri Net shown in Figure 21 is designed to have separate components that can be easily changed or improved in future work, including the cache component and the disk component. The functionality of this net follows the flow of the generic_file_read function closely. It accepts I/O read requests as input, and then compares against the page cache to see if the page was previously retrieved. If the page exists in cache, it is returned to the application immediately. The time to do this page copy, including the L2 cache effect shown in the top right section of Figure 21, is implemented, using a mathematical formula presented in Section 4.5.

If a page does not exist in cache, it is read into page cache, using a pre-fetch mechanism. The kernel attempts to pre-fetch a pre-defined value number of pages into cache. This pre-defined value is a kernel parameter and can be changed using the /proc file system. If the read pattern is random, the pre-fetch mechanism will reduce the number of read-ahead pages to a minimum number. This number is also a kernel parameter and can be changed using the /proc file system.

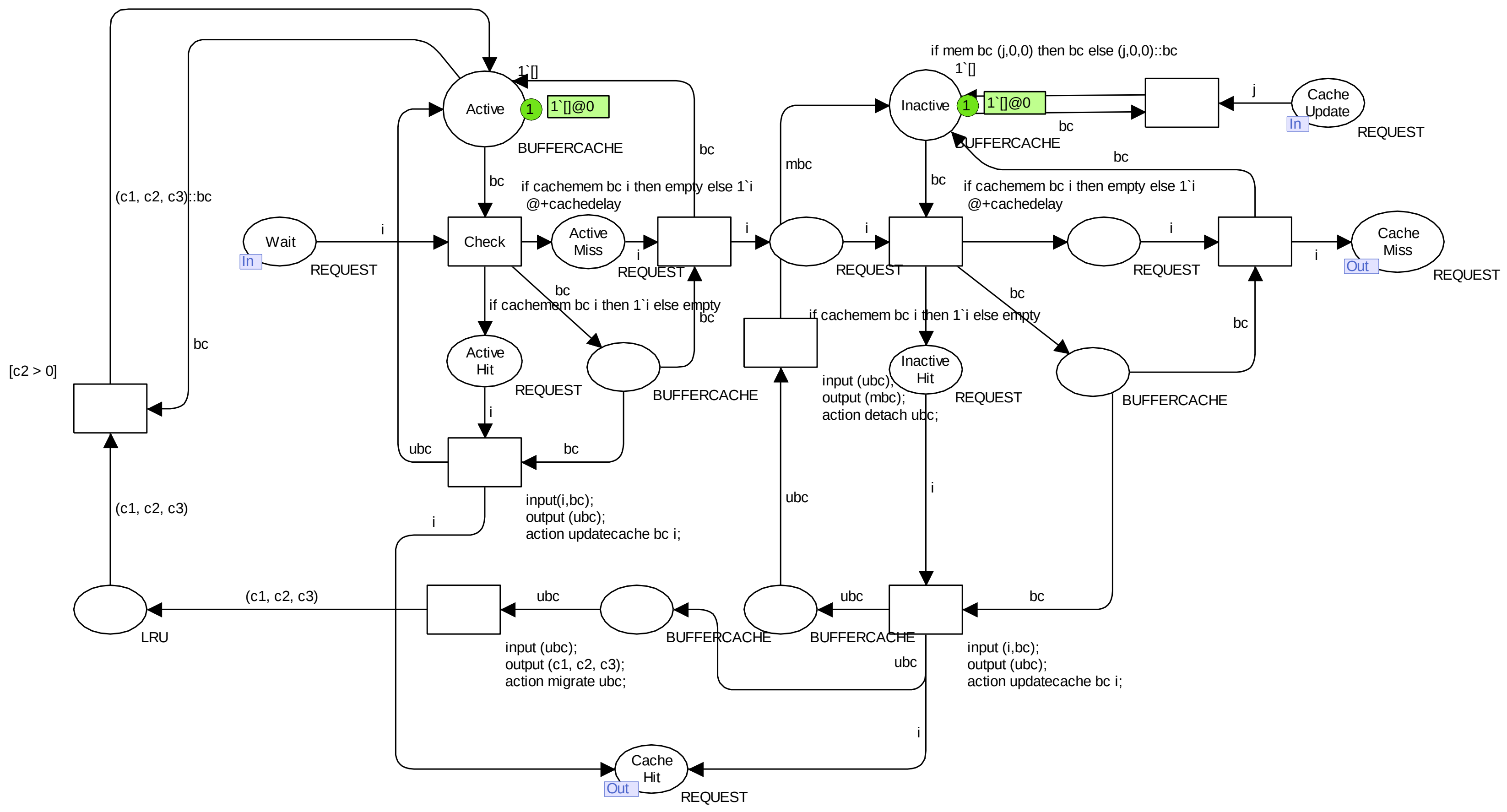


Figure 22: Buffer cache component model

The Petri Net model of the Linux buffer cache component is presented in Figure 22. It contains two queues of memory pages: an active queue and an inactive queue. Each entry of these queues also has two status flags. When a page is introduced into the buffer cache, it is added into the inactive queue with both flags set to 0. When the page is accessed the first time, one flag is set to 1, but the page still does not change queue. If the page is accessed a second time, the second flag is set to 1, and the page is moved to the active queue. If enough time has passed from the last time the page was accessed, it is moved back to the inactive queue. When the system runs out of memory, the memory reclaiming process reclaims pages in the inactive queue first. The model has two outputs “cache hit” and “cache miss”. The I/O scheduler and the disk component are implemented using a simple queuing model and are shown in Figure 23.

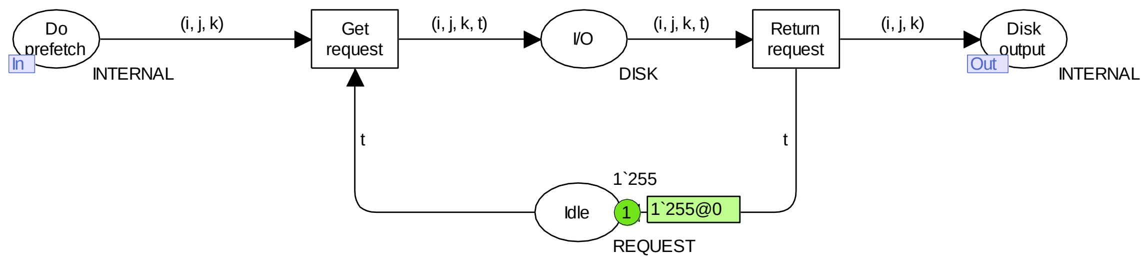


Figure 23: Disk component model

4.4 File write model implementation

From the application perspective, the file write model and the file read model are very similar. They both partition a file into multiple smaller blocks and pass them to the fread function or the fwrite function. The difference between file read and file write is what is being transferred.

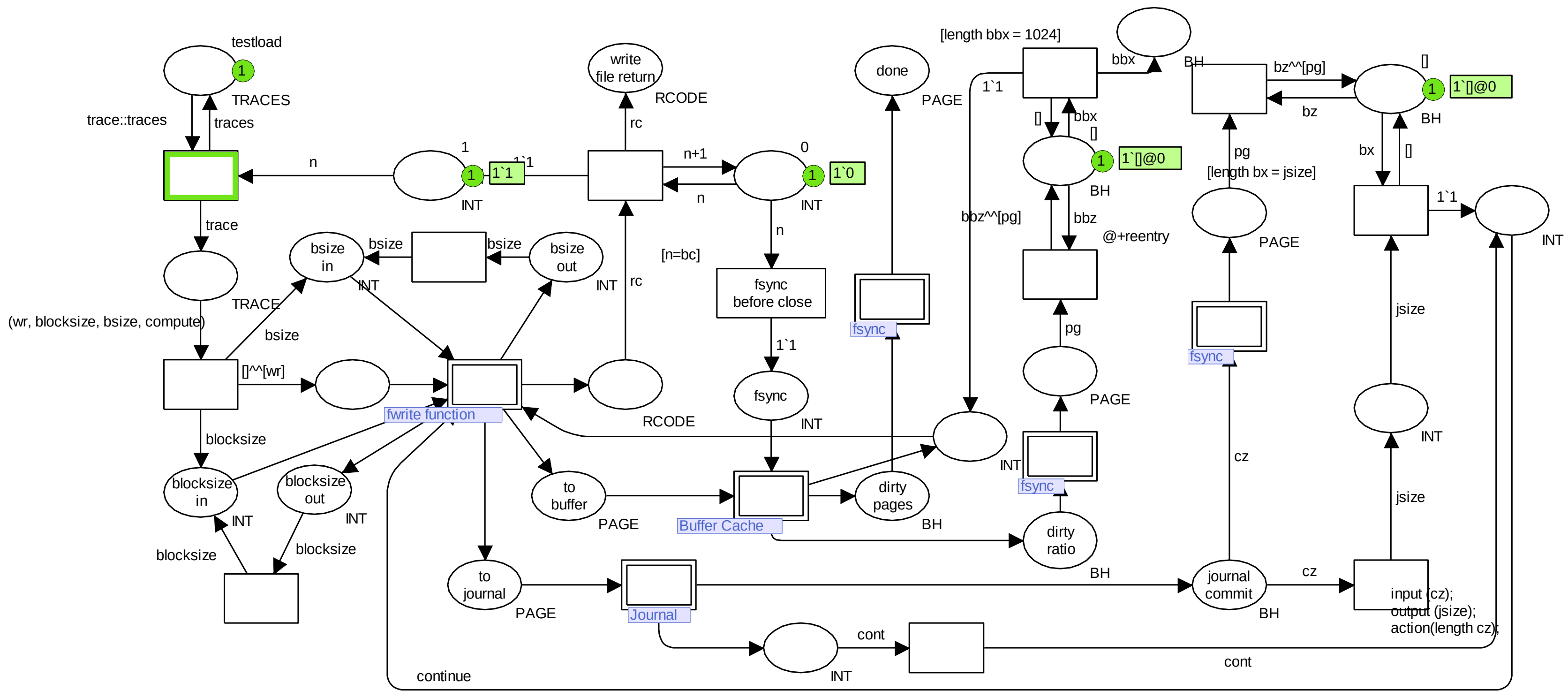


Figure 24: High level I/O write model

For a file read operation, the application passes a list of requests to the lower levels and expects an array of data in return. For a write operation, the application passes an array of data to the lower levels and waits for a set of return codes to ensure that the operation completes successfully. After receiving return codes, the application can continue its operations. The Petri Net implementation of the write operation is presented in Figure 24.

The implementation of the fwrite function is similar to the fread function with the exception of having a buffer of data passed to the write function call. The Petri net implementation for the fwrite function is presented in Figure 25.

The data, however, may or may not be written to disk right away. If the application specifies that the write operation is synchronous, the data is written to disk before fwrite returns to the application. If the application uses the asynchronous write operation, the actual data is kept in memory and will be written to disk at a later time. This delayed write operation is implemented and used in most modern UNIX systems. The operating system (OS) relies on a sync mechanism to flush the data in memory to disk at certain conditions such as low available memory, periodic timer trigger, dirty pages ratio kernel configuration, and force flushing using the fsync() function.

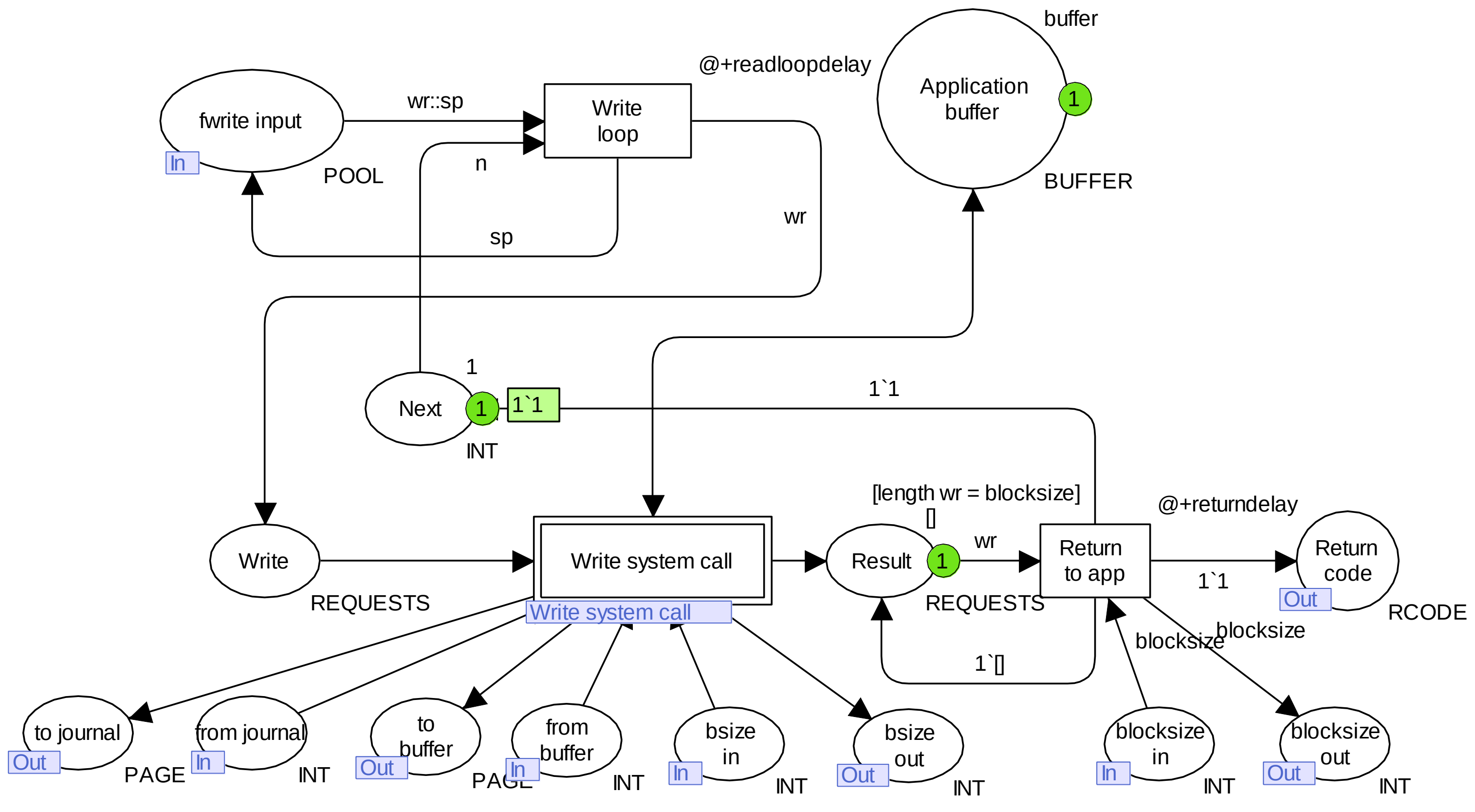


Figure 25: fwrite function model

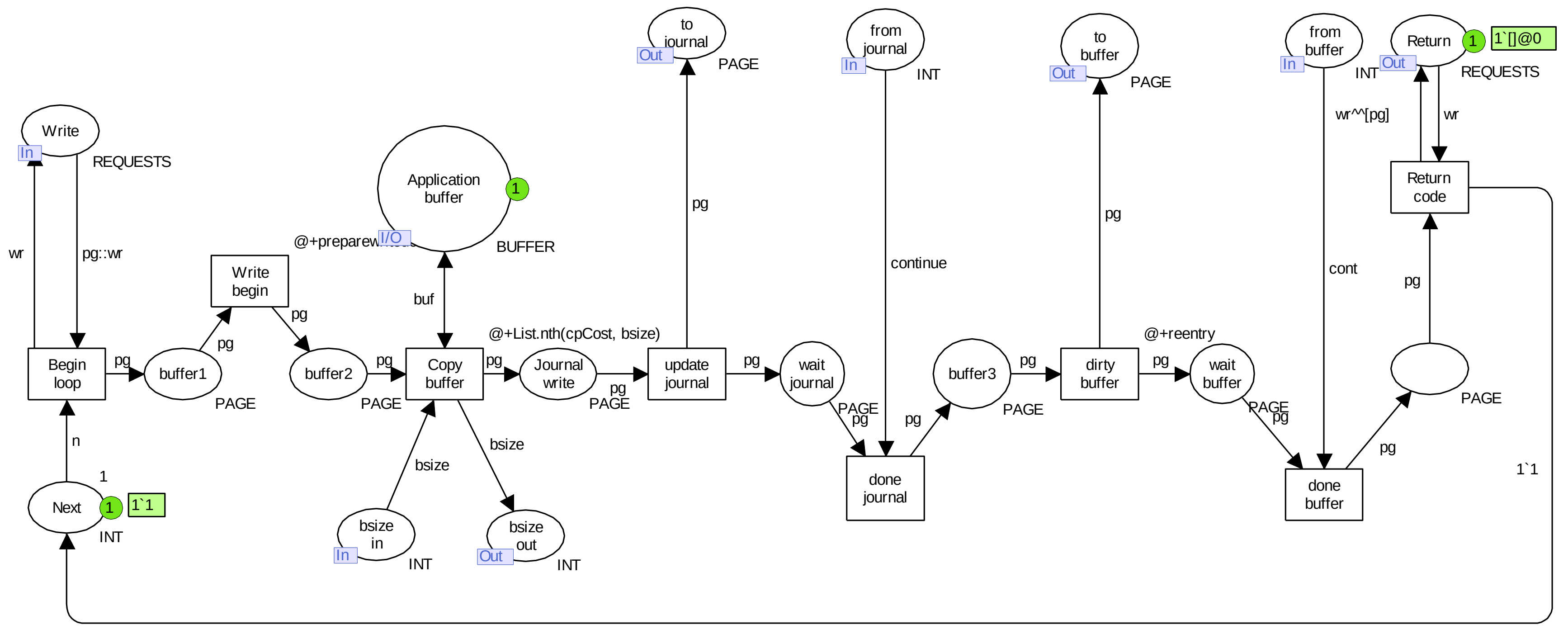


Figure 26: generic_file_write model

The Petri Net implementation of the write system call is presented in Figure 26. The "Write begin" process prepares the system, through tasks such as allocation of memory and journal tracking, for the data from the user space. Then the array of data is copied to kernel space from user space memory and combined into full pages. The kernel call for this copy has an interesting performance behavior that is similar to the call to copy data from kernel space to user space and is implemented using the formula presented in Section 4.5. The "Commit write" process, implemented in Petri Net by several smaller processes, such as "update journal" and "dirty buffer," posts changes to the journal, marks the data dirty in the buffer cache, and submits journal changes. "Commit write," however, does not write the data to disk.

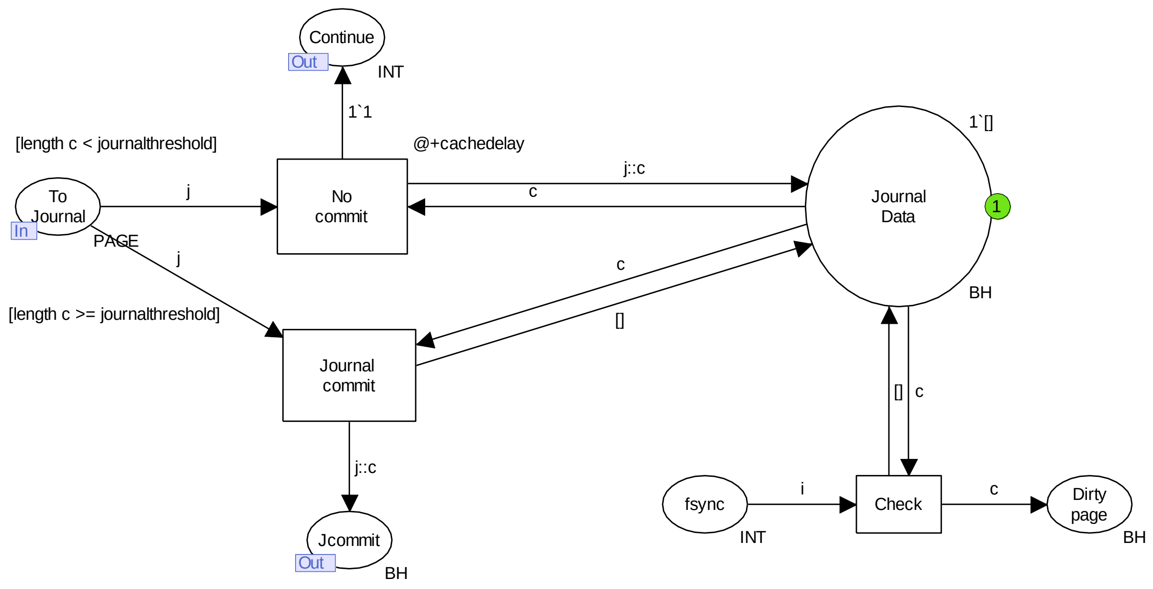


Figure 27: Journal component model

The Petri Net implementation of the journal is presented in Figure 27. Data is flushed to the disk, using a different mechanism which is triggered by several different conditions. A periodic timer triggers the data flush at a pre-determined moment. The data flush is also triggered when the amount of dirty data in the buffer cache reaches a certain threshold. Low memory availability also triggers the data flush. Finally, the data flush can be manually triggered by the fsync() function. The Petri Net implementation of the data flushing is presented in Figure 28.

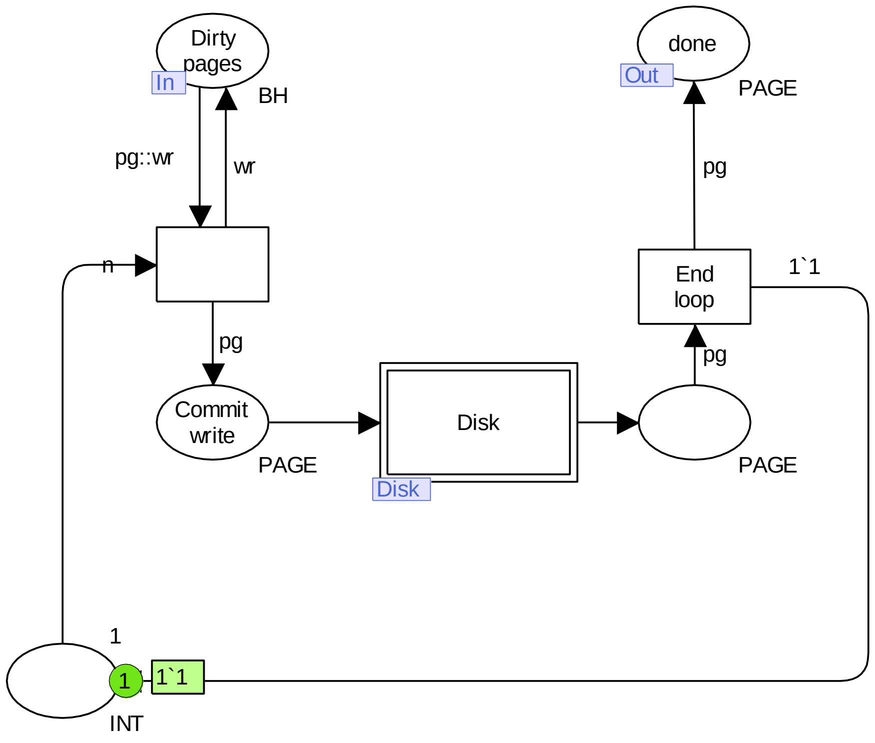


Figure 28: Data flushing component model

The write system call implementation also uses a disk component very similar to the disk component in the read system call. The Petri Net implementation of the component is shown in Figure 29.

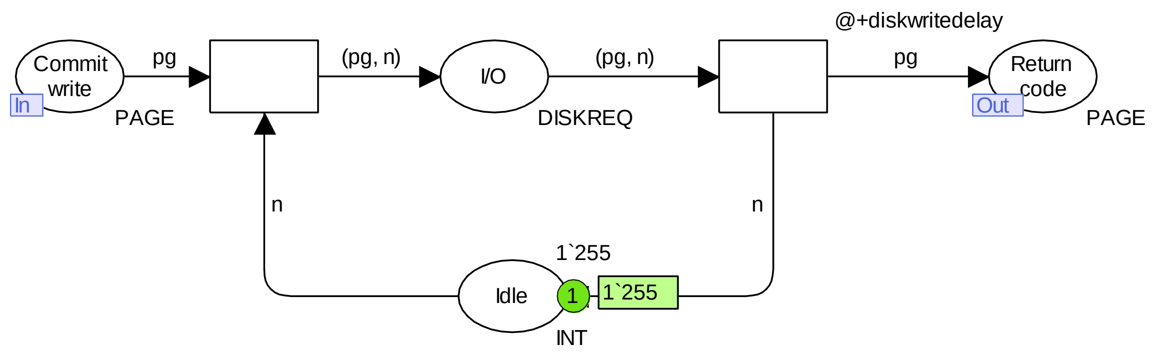


Figure 29: Disk component model

4.5 L2 cache effect model

The measurements in Chapter 3 determine the I/O block size where the performance starts to drop. The performance drops when the average response times of the copy_to_user and copy_from_user functions start to increase significantly. For modeling purposes, this is called $S_{threshold}$. This value is the amount of L2 cache available for copying data from kernel space to user space. When the I/O block size becomes bigger than this value, data is copied at a much slower speed. The response time of the copy function, when using L2 cache, is T_{L2}. The response time of the copy function when not using L2 cache is labeled T_{memory}. The kernel page size is labeled S_{page}. The default value for page size is 4096 bytes. Data movement in the kernel is done using pages. The total amount of data needed to be transferred is labeled S_{total}.

There are two cases. If the I/O block size is less than S$_{\text{threshold}}$, the average response time is

$$t = T_{L2}$$

If the I/O block size is greater than S$_{\text{threshold}}$, the average response time is

$$t(x) = \frac{\left(S_{threshold} / S_{page} \right) \cdot T_{L2} + x \cdot T_{memory}}{\left(S_{threshold} / S_{page} + x \right)}$$

With

$$x = \left(S_{total} / S_{page} \right) - \left(S_{threshold} / S_{page} \right)$$

x is the number of pages needed to be transferred and which does not fit within available L2 cache. When x becomes very large, t approaches T$_{\text{memory}}$.

$$\lim_{x \to \infty} t(x) = \lim_{x \to \infty} \frac{\left(S_{threshold} / S_{page} \right) \cdot T_{L2} + x \cdot T_{memory}}{\left(S_{threshold} / S_{page} + x \right)}$$
$$= T_{memory}$$

So the response time of the copy functions can be modeled using a step function with t(x) as defined above

$$t(x) = \begin{cases} T_{L2} & if \quad x \le 0 \\ t(x) & if \quad x > 0 \end{cases}$$

Figure 30 shows a comparison of this model for the L2 cache effect as compared to the measured data from Figure 13. Figure 30 shows that the response time for the copy_to_user function is very close to the model calculation in most cases, and that the trend of the effect of L2 cache on copy_to_user performance is captured well by the model.

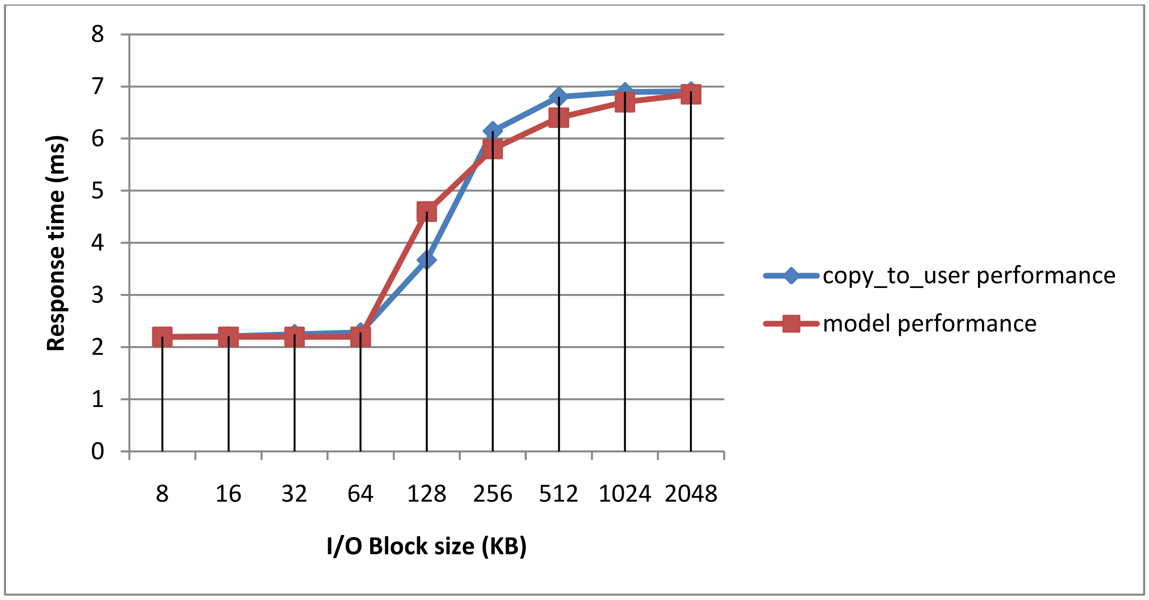


Figure 30: L2 cache model validation

4.6 Summary

This chapter presents a set of detailed and hierarchical performance models of the Linux ext3 file system, using Colored Petri Nets. Studies of the file system read and write operations, including buffering and caching effect, are performed. A model for the L2 cache behavior captures the behavior of the L2 cache and is used directly in the full model. Both file read and file write, including buffering effect and caching effect, are modeled. In future work, this performance model will be extended to model the successor of the ext3 file system, ext4. A new detailed I/O scheduler model will be implemented. The ext3 model will be utilized as a basic foundation to model distributed file systems and parallel file systems.

Chapter 5

LOCAL FILE SYSTEM SIMULATION MODEL PERFORMANCE VALIDATION

5.1 Introduction

This chapter discusses the performance validation of the simulation model for a local file system. Several performance experiments are performed, using different types of workload. The simulation performance results are compared to the real-world performance measurements to study the accuracy of the simulation model.

5.2 Validation setup

In order to validate the entire Petri Net file system model against real-world data, the model hardware parameters, such as memory delay, execution speed, function overhead, and disk speed, are measured directly from the machines where the real experiments take place, using kernel traces. This machine is configured with a single SCSI drive Seagate ST3146707LC. The tracing mechanism used is Ftrace. Ftrace is a powerful kernel-tracing method and has been a part of the mainline kernel since version 2.6.27. Ftrace supports the ability to perform function-graph tracing, which tracks both function entry and function exit as well as providing function duration.

To reduce the simulation time for the L2 cache effect model, the values of the response function are calculated, using the developed model for a very wide range of block sizes, and recorded into a table. The values of the function's constants ($S_{threshold}$, S_{page}, T_{L2}, T_{memory}) are measured from the test system. The Petri Net model (Figure 21, top right corner, and Figure 26, center) uses this table in the transition called Buffer Copy to produce the response time for the data copy from kernel space to user space.

5.3 Synthetic sequential workload

Simulations of sequential workload are run several times, and the average results are used to compare with iozone benchmark results running on the test system. The simulation experiments are run, using a set of synthetic I/O requests and simulating sequential I/O. The I/O requests are grouped into similar block size configurations of the izone benchmark. Data- write operations in this section are asynchronous. The file system journal mode used in this section is ordered mode. The result of the I/O read performance model is presented in Figure 31. The errors bars are set at 10%.

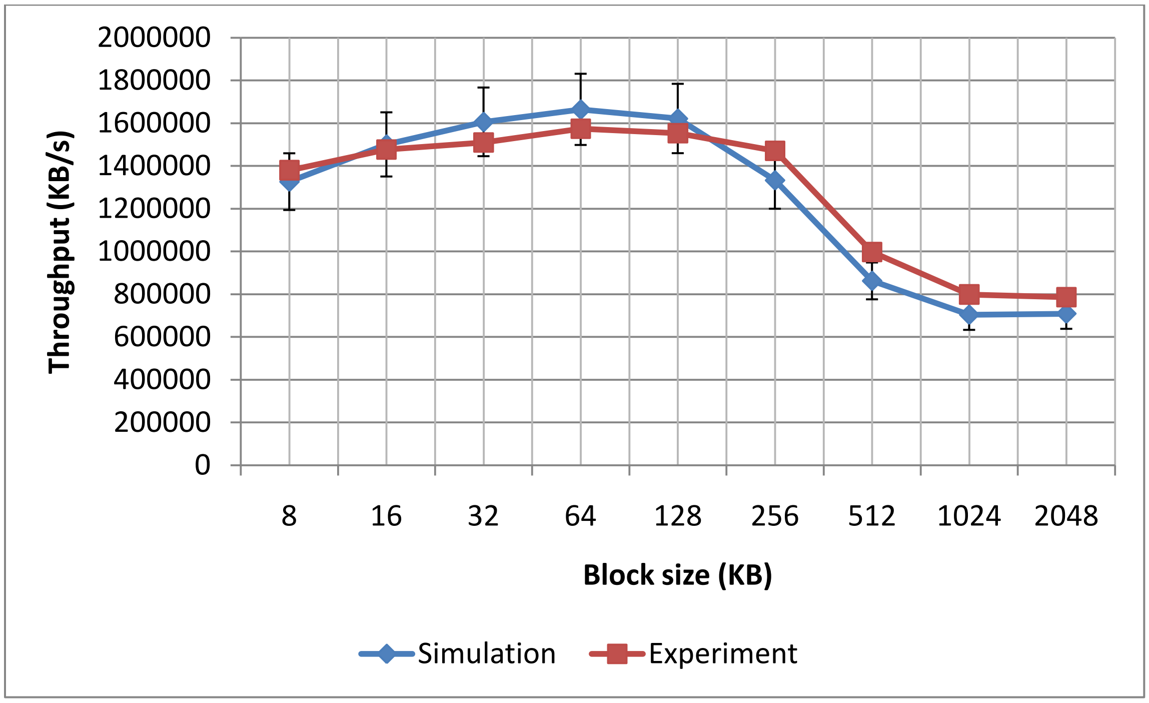


Figure 31: Sequential I/O read performance validation

Most data points fall within a 10% error range or very close to that range, which is a highly accurate result of an end-to-end model of a system as complex as the Linux ext3 file system. Figure 31 shows that the Petri Net model result captures the trend of file system performance well, showing behavior very similar to the real file system.

I/O write performance experiments are performed in the exact same manner. Measured data from the actual kernel I/O path are inserted into the Petri Net model. The write simulations are run multiple times, and the average results are compared with the real file system data. The result of the I/O write Petri Net model is presented in Figure 32. The error bars are also set at 10%.

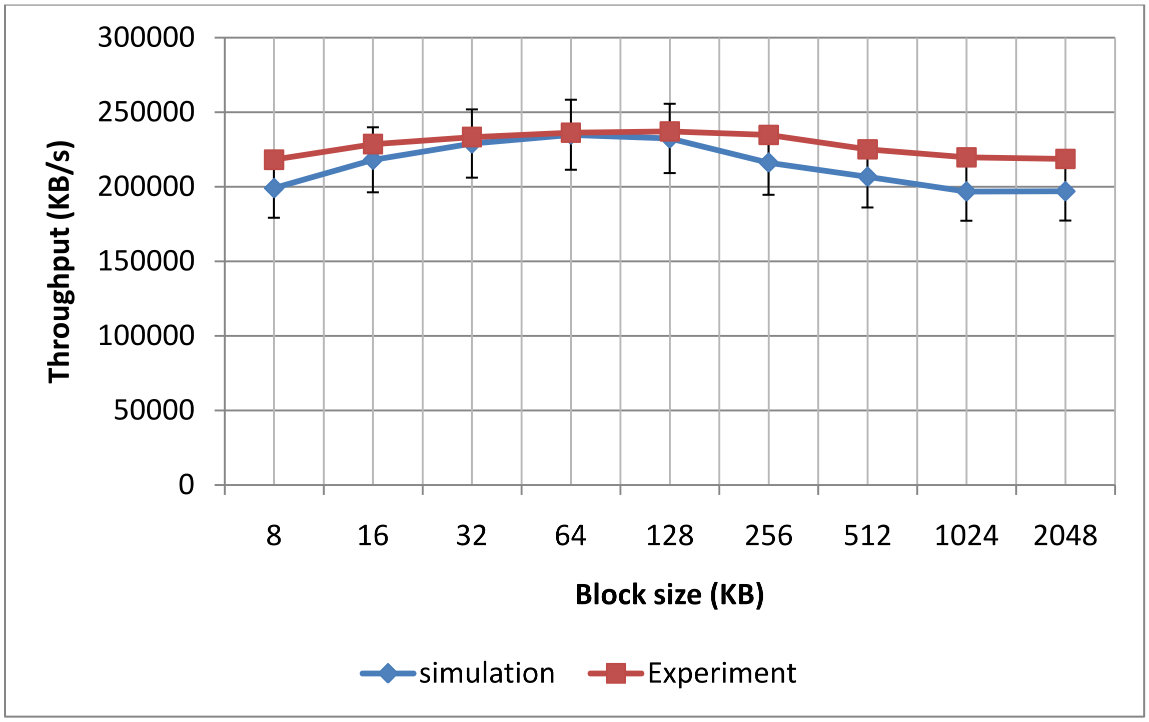


Figure 32: Sequential I/O write performance validation

The write result is even better than the I/O read performance result. Figure 32 shows that all data points fall within or very close to a 10% error range. In the case of the write performance, the Petri Net models the simulation consistently and underestimates the performance of the actual file system throughput, again by less than 10%. Thus, the model is a very effective tool for predicting the expected performance of the real file system with sequential workload. It is useful to designers of new data-intensive computing systems and for capacity planning of existing systems [96].

5.4 Synthetic random workload

Simulations with random workload are also run several times, and the average results are used to compare with the real-world results. The simulation experiments are run, using synthetic I/O requests and simulating random I/O with very small block size to minimize the sequential characteristic of the workload. The same set of synthetic I/O requests is also used to feed the iozone benchmark to produce performance results on the test system. Data-write operations in this section are asynchronous, and the file system journal mode used in this section is ordered mode. The I/O access pattern of the workload is presented in Figure 33. The Y axis represents the location of the I/O request. The X axis represents the order in which the I/O requests occur. Figure 33 presents the randomness of the workload very clearly.

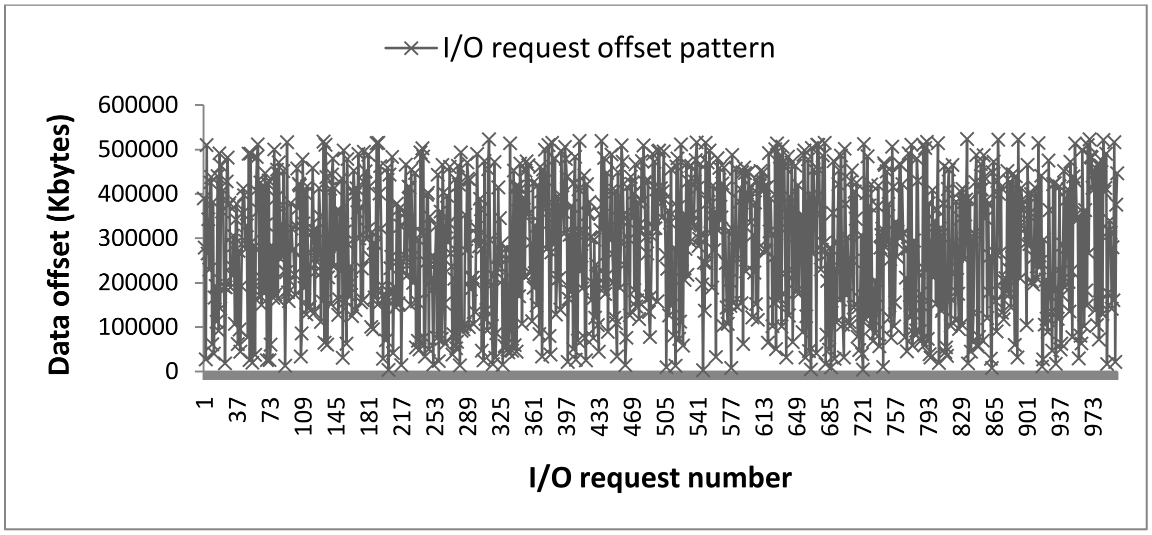


Figure 33: Synthetic random workload pattern

The random I/O read performance results are presented in Table 5.

Table 5: Random I/O read performance validation

Random I/O Read performance result	
Block size (Kbytes)	8
Simulation throughput (KB/s)	757,791.04
Measure throughput (KB/s)	631,162.80
Error	20%

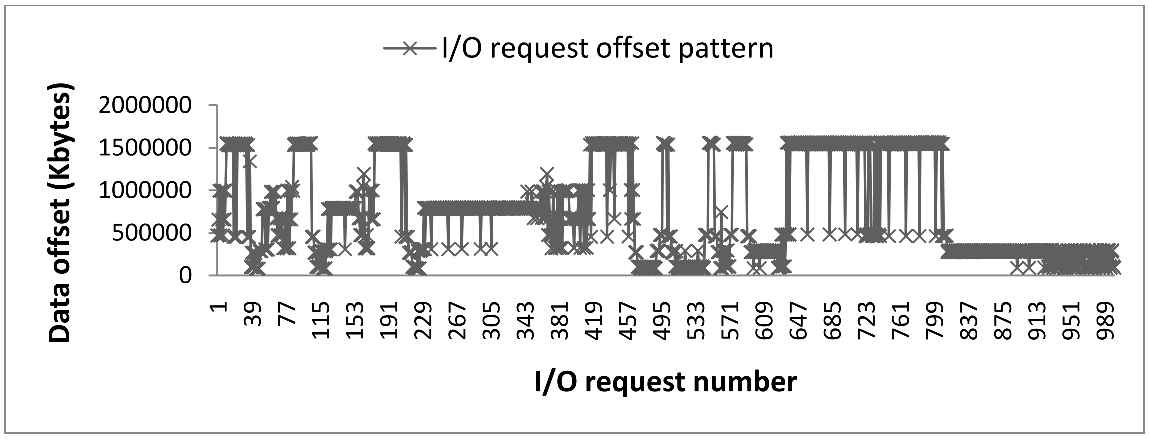
The same set of synthetic random I/O requests is also used in the I/O write experiment. The performance results of random I/O write are presented in Table 6. The result of random I/O write simulation is not as good as random I/O read simulation and will be addressed in future work.

Table 6: Random I/O write performance
validation

Random I/O Write performance result	
Block size (Kbytes)	8
Simulation throughput (KB/s)	199,004.98
Measure throughput (KB/s)	147,995.60
Error	34%

5.5 Captured I/O traces from production systems

Synthetic workloads are very useful for system performance study. However, they do not always reflect the real workload in a system under real-world conditions. A captured I/O request trace can provide a closer presentation of real-world workloads. I/O traces are captured from live production systems to use in this experiment. Figure 34 presents the I/O read requests pattern of the first captured trace.



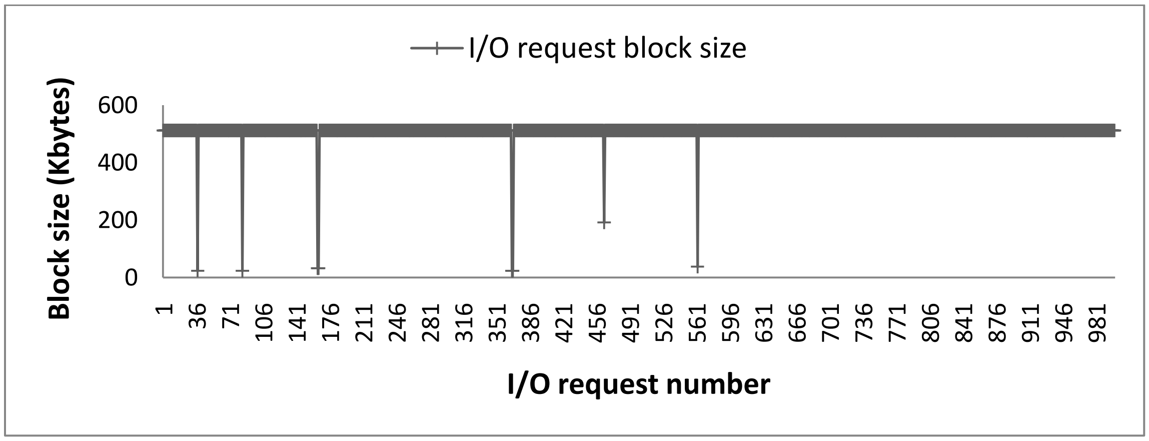
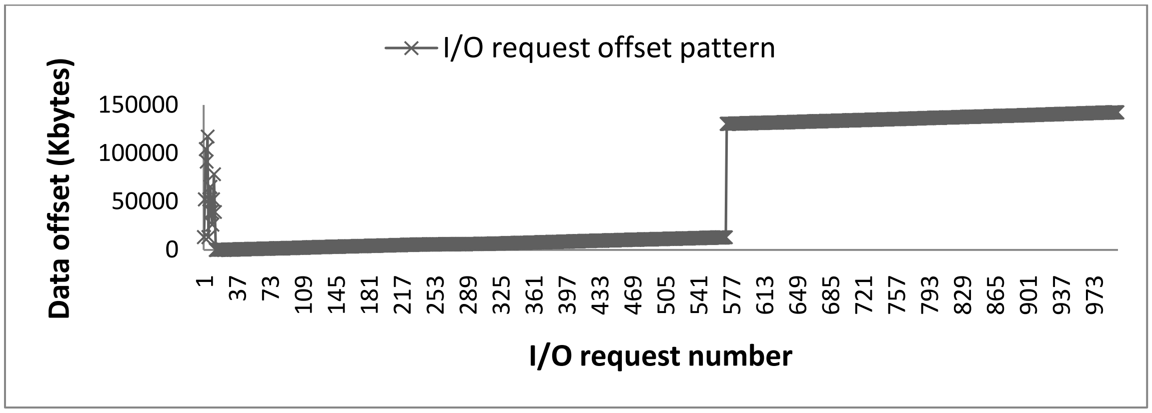


Figure 34: I/O read pattern of the first trace

The I/O pattern shows less randomness in I/O read activities. The large block size of the I/O reads gives the workload a mixed characteristic of both sequential I/O and random I/O.

Figure 35 shows the I/O write request pattern of the first captured trace.



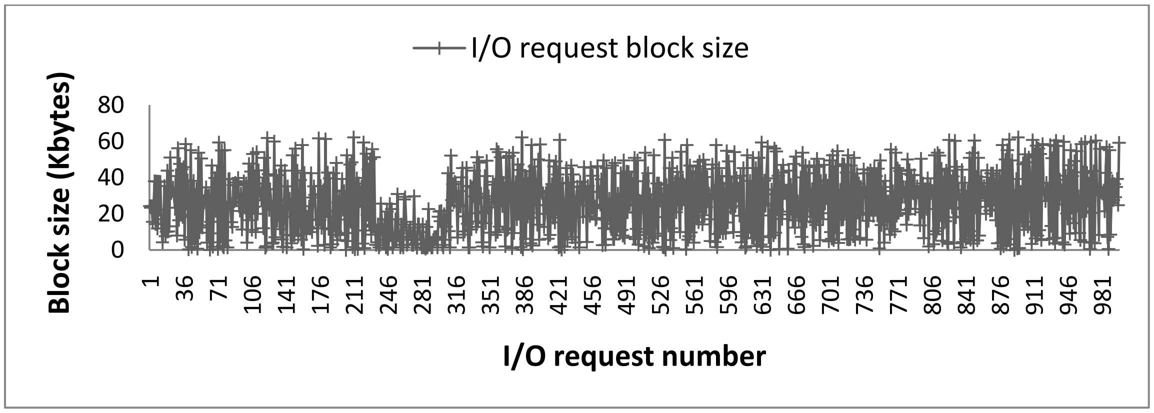
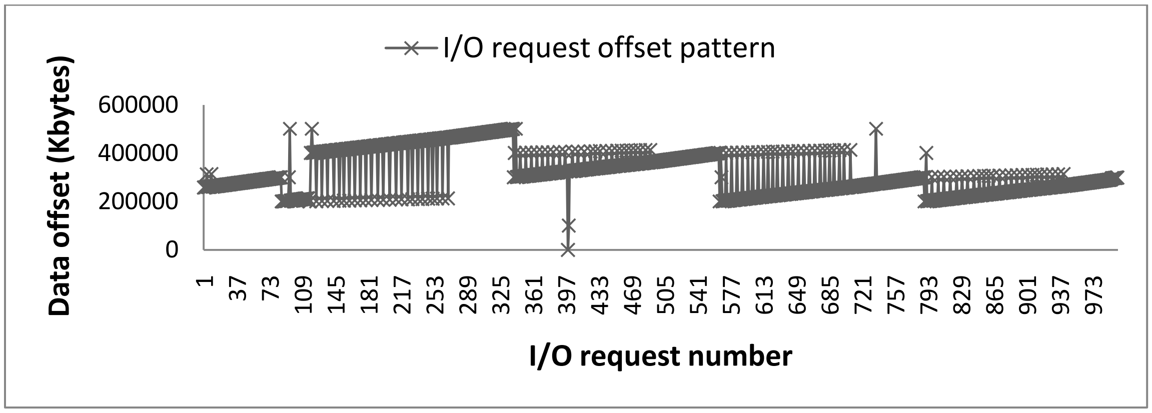


Figure 35: I/O write pattern of the first trace

In this trace, the I/O write requests are random at the beginning of the trace, but eventually become sequential in the latter part of the trace. The block sizes of the I/O write, however, change quite randomly.

Figure 36 presents the I/O read requests from the second captured trace. The I/O read pattern in this trace has less randomness than the previous trace. This I/O pattern also shows several mixtures of random accesses and sequential accesses.



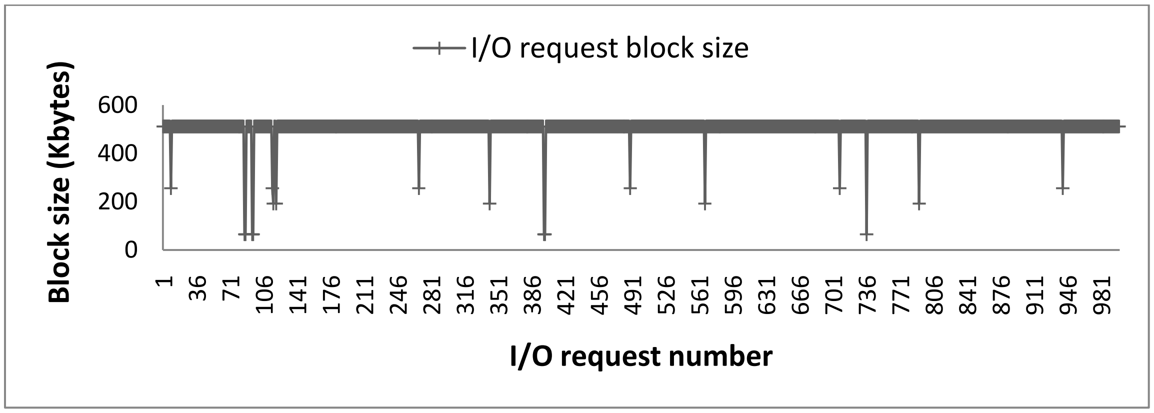
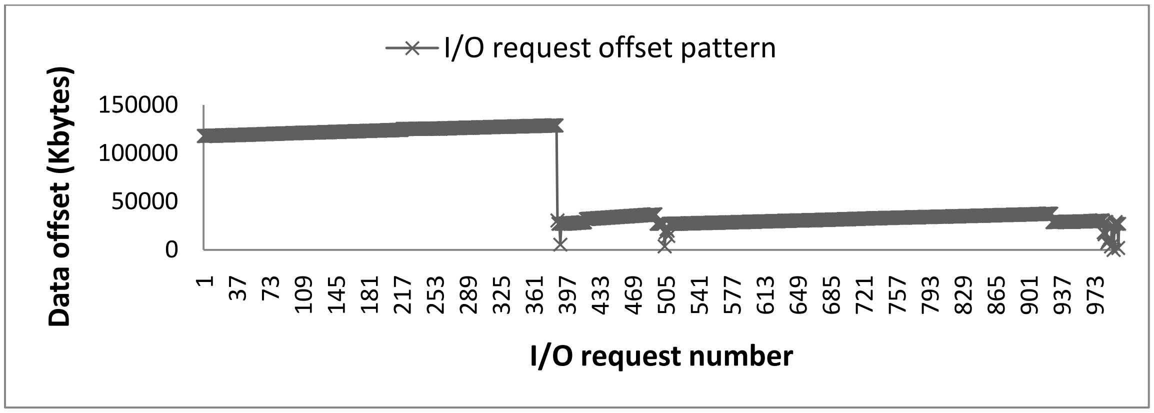


Figure 36: I/O read pattern of the second trace

Figure 37 shows the I/O write request from the second captured trace. The I/O write pattern in this trace is also a combination of sequential write and random write. The block sizes of the I/O write are also greatly varied through the duration of the trace.



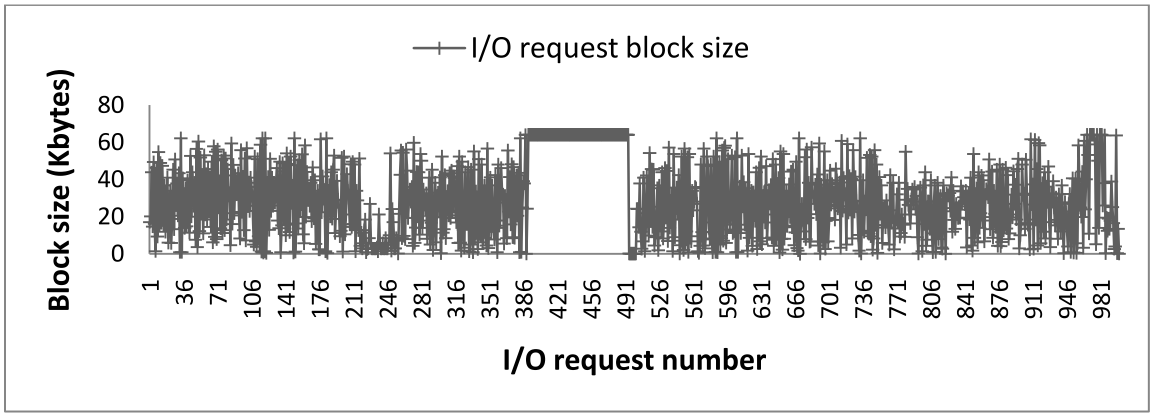


Figure 37: I/O write pattern of the second trace

The two captured I/O read traces from Figure 34 and Figure 36 are fed into the model and the iozone benchmark to produce the I/O read performance comparison. Data-write operations in this section are asynchronous. The file system journal mode used in this section is ordered mode. Similar to the previous performance studies, simulations are run several times and produce the average result. The I/O performances are higher than previous experiments due to caching effect. Table 7 presents the I/O read performance results.

Table 7: Captured traces I/O read validation

I/O Read performance result	Trace 1	Trace 2
Simulation Throughput (KB/s)	873,238.11	876,237.20
Measure throughput (KB/s)	991,969.14	1,008,167.15
Error	12%	13%

The two I/O write traces from Figure 35 and Figure 37 are also fed into the model and the iozone benchmark to produce the I/O write performance. Table 8 shows the I/O write performance result.

Table 8: Captured traces I/O write validation

I/O Write performance result	Trace 1	Trace 2
Simulation throughput (KB/s)	146,644.10	146,813.74
Measure throughput (KB/s)	207,203	180,783.2
Errors percent	29%	19%

5.6 The impact of the dirty-ratio kernel parameter

The kernel parameter—dirty ratio—which is discussed in Chapter 3 influences the I/O write performance behavior that the model should exhibit correctly. In order to validate this behavior, an experiment is performed, using a test file with a larger size than the default value of the dirty-ratio threshold setting on the system (~512MB). Figure 38 shows the comparison between the measure from the actual system and the simulation result of the model. The error bars are set to 10%, similar to previous experiments.

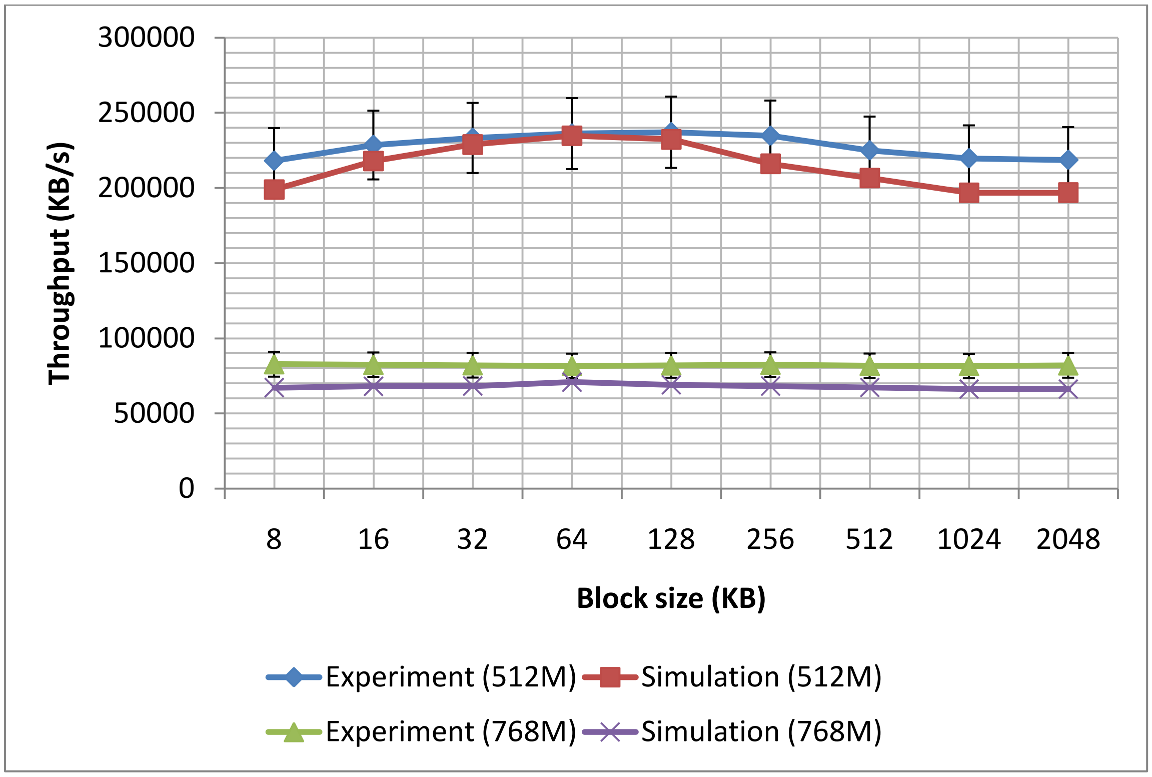


Figure 38: The impact of dirty ratio parameter

The simulation results are close to the measurements from the actual system. The errors fall between 10% and 20% for all data points. Similar to the sequential write experiment, the model consistently underestimates the performance of the actual system for both file sizes.

5.7 Full data journal mode write performance

In previous validation experiments, from section 5.3 to section 5.6, the file system is operating under ordered journaling mode. As stated in Chapter 3, the performance differences of write-back journaling mode and ordered journaling mode are small. Full data journal mode, however, is a completely different case. Unlike ordered journal mode or write-back journal mode, full-data journal mode writes data as well as metadata to the journal, which is located on the disk. As a result of this, the same data are actually written to the disk twice. As data and metadata are being written into the journal, the amount of free space allocated for the journal become smaller. When the journal free space reaches a threshold, a journal checkpoint happens. The exact amount of journal free space that triggers a checkpoint is not derived in a straightforward manner, as Prabhakaran notes [50]. Journal checkpointing occurs when the amount of journal free space is between ½ and ¼ of the journal size. For the validation experiments in this section, we use a threshold equals to approximately ½ of the journal size as it seems to produce best results.

Using the same process described in section 5.3, the first validation experiment uses a synthetic sequential workload. Simulations are run several times, and the average results are used to compare with iozone benchmark results, running on the test system. The I/O requests are grouped into similar block size configurations of the izone benchmark. The result of the I/O read performance model is presented in Figure 39. The errors bars are set at 10%.

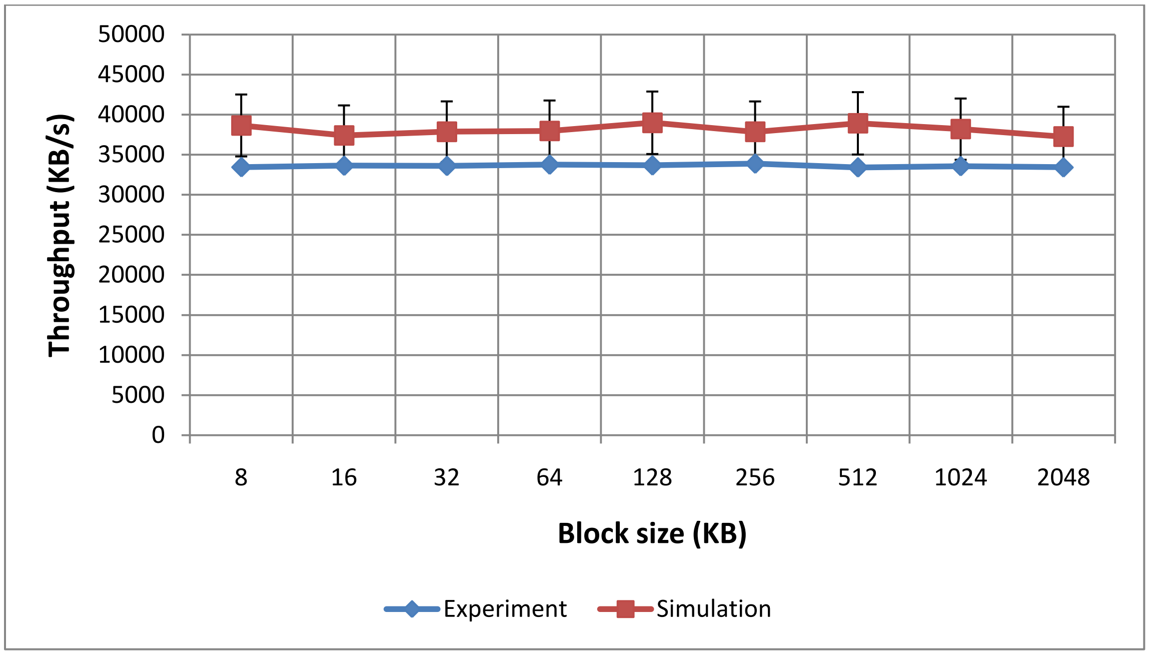


Figure 39: Sequential I/O write validation – full data journal mode

The errors between simulation data and real-world measurement data are close to 10%. The performance impact of the full-data journal mode is quite clear. The shapes of the performance curves are different from the shapes of performance curves in section 5.3. The effect of L2 cache still exists. However, because the response time of the file system is slow, the effect is not noticeable any longer.

Following the same order previously presented, an experiment similar to the experiment in section 5.4 is performed. The simulation experiments are run, using synthetic I/O requests and simulating random I/O with very small block size to minimize the sequential characteristic of the workload. The result of the experiment is presented in Table 9.

Table 9: Random I/O write validation - full data journal mode

Random I/O Write performance result	
Block size (Kbytes)	8
Simulation throughput (KB/s)	30016.45127
Measure throughput (KB/s)	26470.4
Error	13%

The last experiment is similar to the experiment in section 5.5. The two I/O traces are fed into the model and the iozone benchmark to produce the I/O performance comparison. Like previous performance studies, simulations are run several times and produce the average result. The result of the experiment is presented in Table 10.

Table 10: Captured traces I/O write validation - full data journal mode

I/O Write performance result	Trace 1	Trace 2
Simulation throughput (KB/s)	51417.04	63660.53
Measure throughput (KB/s)	46470.4	56593
Errors percent	11%	12%

5.8 Synchronous write performance

In previous validation experiments, up to this section, I/O write operations all use asynchronous write mode. It provides the best performance for the system, and under normal circumstances, is the default operating mode for Linux I/O write operations. However, synchronous write mode is still being used occasionally in situations where data needs to be written to disk after each write request. In this operating mode, the real system, as well as the model, issues a data synchronization at the end of the write request. Because data synchronization is done at the end of every write request, the file system journal mode does not have any effect.

The same process described in section 5.3 is used. Simulations are run, using a synthetic sequential workload several times, and the average results are used to compare with iozone benchmark results, running on the test system. The I/O requests are grouped into similar block-size configurations of the izone benchmark. The file system journal mode is ordered mode. The result of the I/O write performance model is presented in Figure 40. The errors bars are set at 10%.

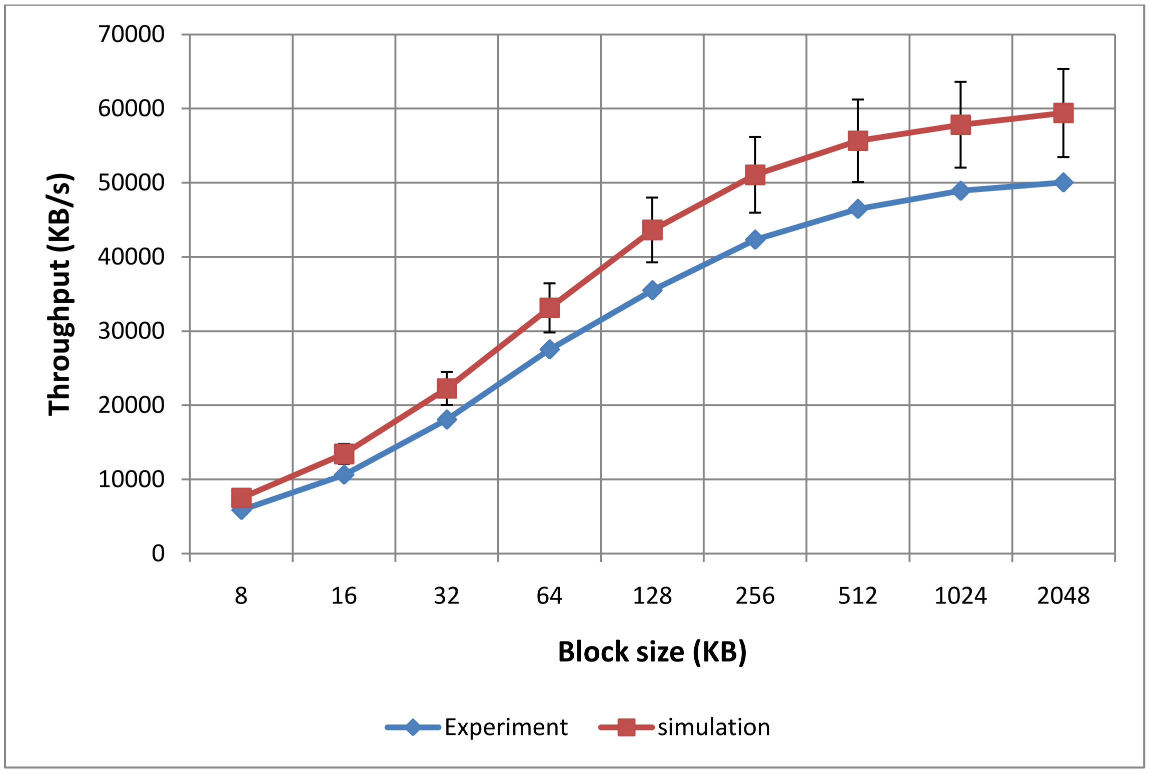


Figure 40: Sequential I/O write validation -
synchronous write

The simulation results are very good, even though the errors are bigger than 10% at multiple data points. The performance impact of the synchronous write mode is also very clear. The shapes of the performance curves are different from the shapes of performance curves in section 5.3. Because of the slow response time of the file system, the L2 cache effect is also insignificant in this experiment.

The next experiment is similar to the experiment in section 5.4. The simulation experiments are run, using synthetic I/O requests and simulating random I/O with very small block size to minimize the sequential characteristic of the workload. The result of the experiment is presented in Table 11.

Table 11: Random I/O write validation - synchronous write

Random I/O Write performance result	
Block size (Kbytes)	8
Simulation throughput (KB/s)	6689.75
Measure throughput (KB/s)	5388.8
Error	24%

The last experiment is similar to the experiment in section 5.5. The two I/O traces are fed into the model and iozone benchmark to produce the I/O performance comparison. Like previous performance studies, simulations are run several times and produce the average result. The result of the experiment is presented in Table 12.

Table 12: Captured traces I/O write validation - synchronous write

I/O Write performance result	Trace 1	Trace 2
Simulation throughput (KB/s)	18,196.46	25,633.22
Measure throughput (KB/s)	14,856.8	20,628.2
Errors percent	22%	24%

5.9 Summary

This chapter presents a set of detailed performance validation experiments of the Linux ext3 file system model. To validate the performance behavior of the file system model, several types of workload are utilized. A synthetic sequential workload is generated to study the simulation model behavior and to compare the model with real file system performance. A random synthetic workload is also generated to study the behavior of the simulation model when random accessing is involved. In addition to synthetic workload, I/O traces captured from production systems are also utilized to study the performance behavior of the simulation model in a real-world environment.

The validation experiments are run under both ordered journal mode and full data journal mode. The results for ordered journal mode are very good. For sequential file read and file write, the simulation performances are within 10% of the real file system in most cases. For random file read, the simulation performances are within 20% of the real file system. For random file write, the simulation performances differ less than 35% of the real file system. For I/O traces captured from live systems, the simulation performances differ less than 20% in most cases. An additional performance factor— dirty ratio threshold—is also modeled and validated. The results for full-data journal mode are very good. In all experiments for this mode, the errors are less than 15%. In good cases, the errors are between 10% and 12%.

Synchronous I/O write operation is also validated. The results are very good, as the errors are less than 10% in many cases. However, for random synthetic workload and captured I/O traces workload, the errors are approximately 24%.

Chapter 6

DESIGN OF A SIMULATION MODEL FOR PARALLEL FILE SYSTEM

6.1 Introduction

The first and foremost goal for a parallel file system is to achieve massive I/O throughput. This is done by providing access to multiple I/O resources in parallel. PVFS as well as many other parallel file systems implements this by utilizing multiple connected local file systems as foundation. The simulation model for the parallel file system is developed using similar concept. It utilizes multiple connected local file system simulation models as its foundation. It interfaces with higher level applications and provides them the response time associated with each I/O request. This chapter discusses the design of a simulation model for PVFS – a parallel file system. The implementation of the simulation model is presented in a top down fashion, from application level down to the local file system level, and each level is described using Colored Petri Nets.

6.2 Assumptions and model limitations

Similar to the local file system simulation model, the parallel simulation is also divided into an I/O read model and an I/O write model. Read operations and write operations are simulated separately to simplify the multiple conditions when simulating the file system.

A key difference between a parallel file system and a local file system is the network component. Parallel file systems use network to simultaneously access multiple local file system at the same time. A parallel file system simulation model must contain a network model. Although the network simulation model is an important component in the parallel file system simulation model, it only serves as a transport from the client model to the server model. The network model does not need to model every network operations in detail. Instead, a resource model is used to simulate network end-to-end performance.

A PVFS cluster has a certain number of I/O servers. This number is determined at the time the cluster is built. After the cluster goes into production, the number of I/O servers is relatively fixed. Although, under a certain circumstance, I/O servers can be added or removed from the cluster, but this procedure usually cause the original data on the cluster to be destroyed. For the simulation model, the PVFS cluster has 4 I/O servers. In real-world situation, A 4 I/O servers cluster could house approximately 4 Tbytes of data.

6.3 File read model implementation

From the application standpoint, reading a file from a parallel file system is no different than reading a file from a local file system. The way an application reads a file is similar to the following illustration.

```
while (!feof(file_handle)) {
    bytes_read = fread(buffer, block_size, number_of_block, file_handle);
}
```

From this level, the operation is divided into three main components: the client component, the network component and the server component.

6.3.1 File read model client component

At the top level, the model is simple. A loop breaks the needed file into multiple blocks of read requests and passes the list to the client simulation component. The client component processes the data then passes them on to the network component. The result of the read operation is an array of data passed back from the network model. The Petri net for this operation is presented in Figure 41.

The implementation of the client component could be described as dividing the block of read requests into a list of payloads and passing this list to the network component to send over the network to the server component. The number of payloads depends on the number of I/O servers in the file system. The Petri net implementation of the client component is presented in Figure 42.

Payloads are created by striping request data into multiple chunks according to the file system's stripe depth parameter. Stripe depth in PVFS usually is 64 Kbytes. The distribution of data chunks in a payload is done using round-robin mechanism. The Petri Net implementation of the payload creation process is presented in Figure 43.

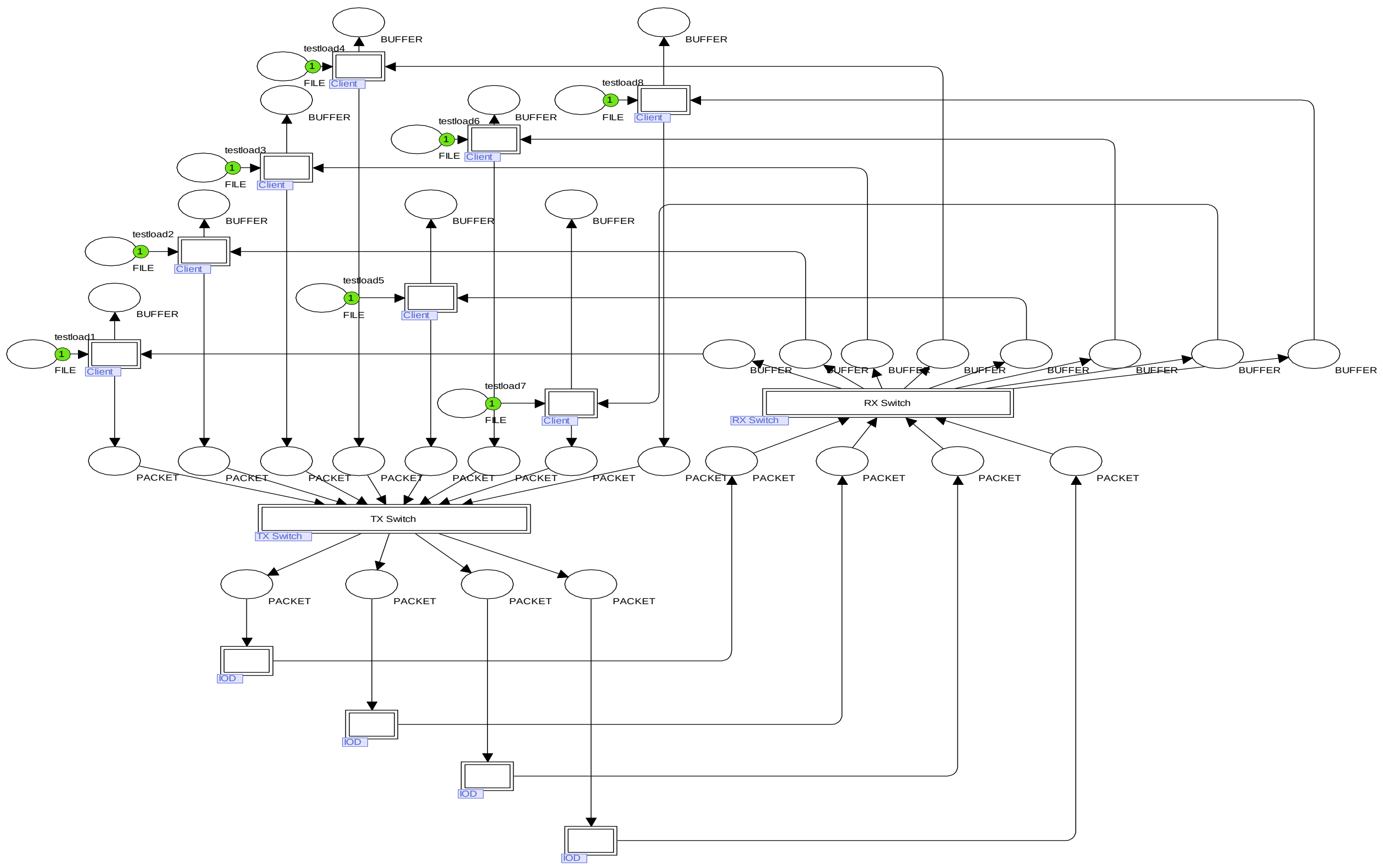


Figure 41: High level PVFS application read model

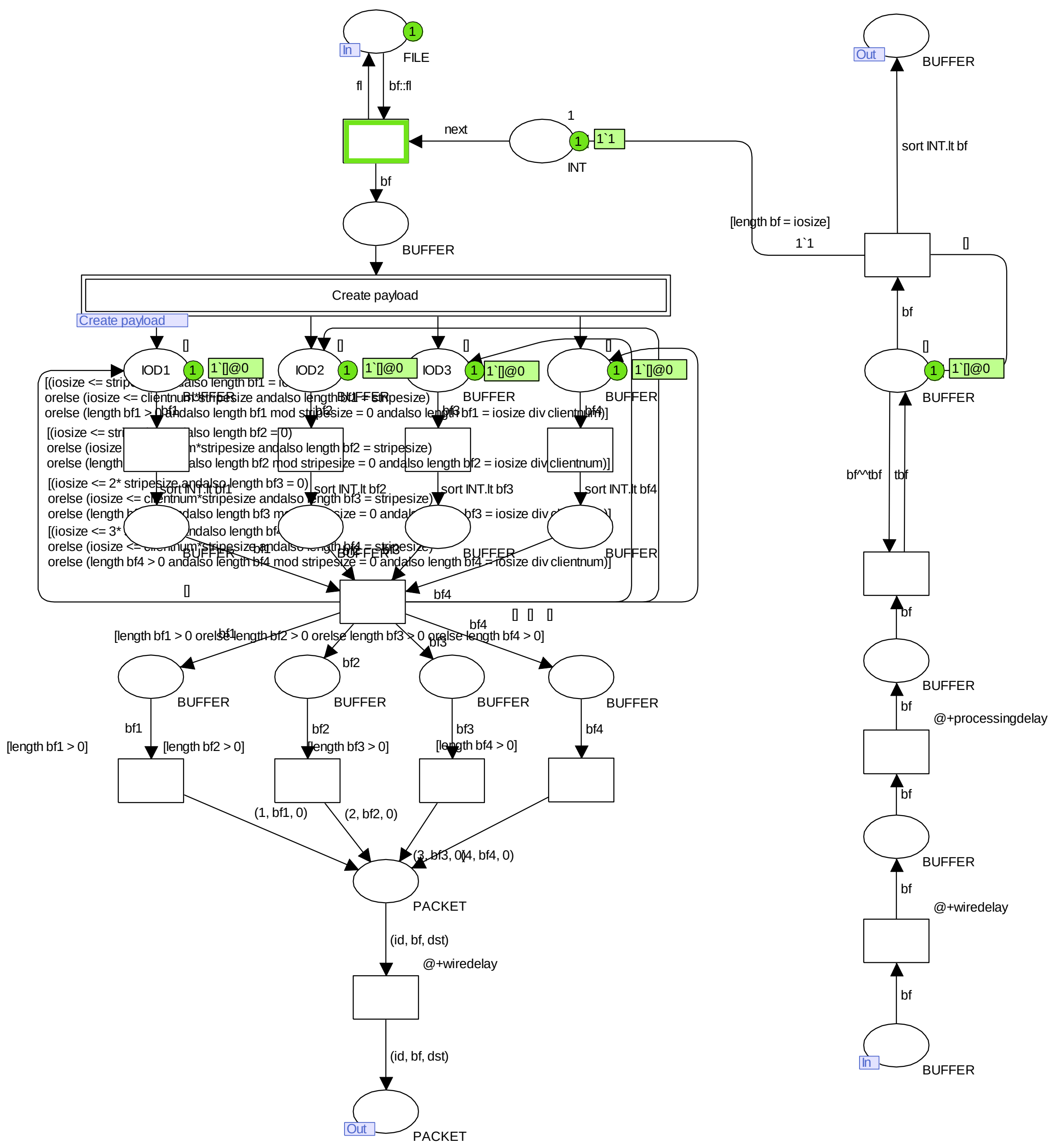


Figure 42: PVFS client component model for file read

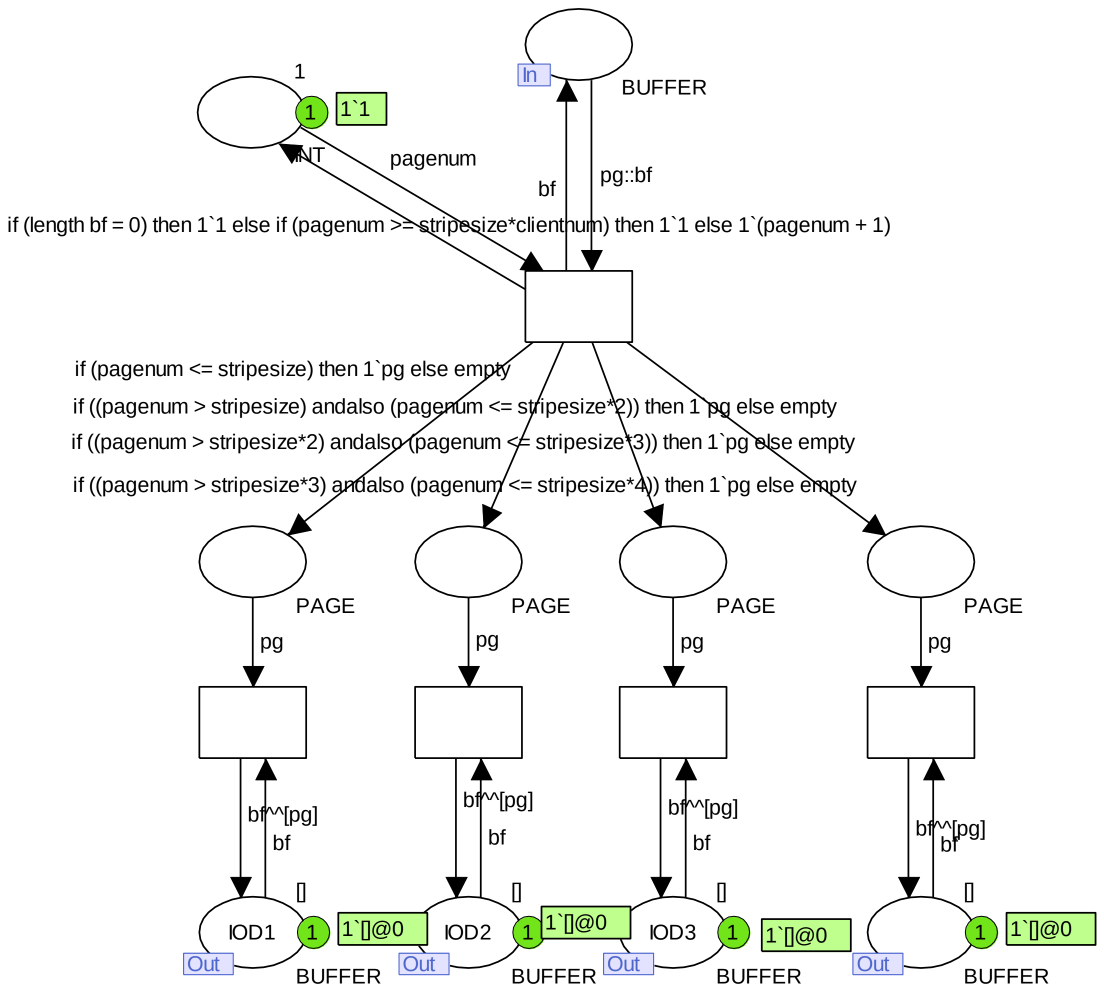

1
1
1`1
INT
pagenum
In
BUFFER
bf
pg::bf
if (length bf = 0) then 1`1 else if (pagenum >= stripesize*clientnum) then 1`1 else 1`(pagenum + 1)
if (pagenum <= stripesize) then 1`pg else empty
if ((pagenum > stripesize) andalso (pagenum <= stripesize*2)) then 1`pg else empty
if ((pagenum > stripesize*2) andalso (pagenum <= stripesize*3)) then 1`pg else empty
if ((pagenum > stripesize*3) andalso (pagenum <= stripesize*4)) then 1`pg else empty
PAGE
pg
bf^^[pg]
bf
[]
IOD1
IOD2
IOD3
1
1`[]@0
Out
BUFFER

Figure 43: Payload creation component model for PVFS file read

After the payloads are created, the client component prepares the packet before sending them to the network component. This process represents the network stack on the client computer. While this process could be considered a part of the network component, it uses physical resources on the client machine and thus is more closely related to the client component. Taking the payloads and building network packet around this data, the client component adds the network identifications of the I/O servers to the data. The network component will later use this information to deliver the packet to the correct I/O server. For an I/O read operation, the client component only sends read requests to the servers. Read requests are very small and will not need to be broken down into smaller fragments. After the network packets are created, they are sent to the network device buffer.

In addition to sending read requests to the I/O servers, the client component also receives data being sent back from the I/O servers. From the network device receiving buffer, the client component gathers the network packets. It assembles the data from these network packets received from different I/O servers into the needed result and sends it back to the application.

6.3.2 File read model network component

The network component provides the transportation for the data packets from the client to the I/O servers. Since only end-to-end performance characteristics of the network component are needed, the network component will not model switches and routers in detail. Instead, the network component is designed using multiplexer model. The client packets are examined and routed to the correct I/O servers.

When the result data are sending back to the clients, a similar mechanism is used. The server component, depends on the result data, will send data packets back to the original requested client. The network component examines the packet and route them to the correct clients. The Petri Net models of the sending and the receiving network components for PVFS file read operation are presented in Figure 44 and Figure 45.

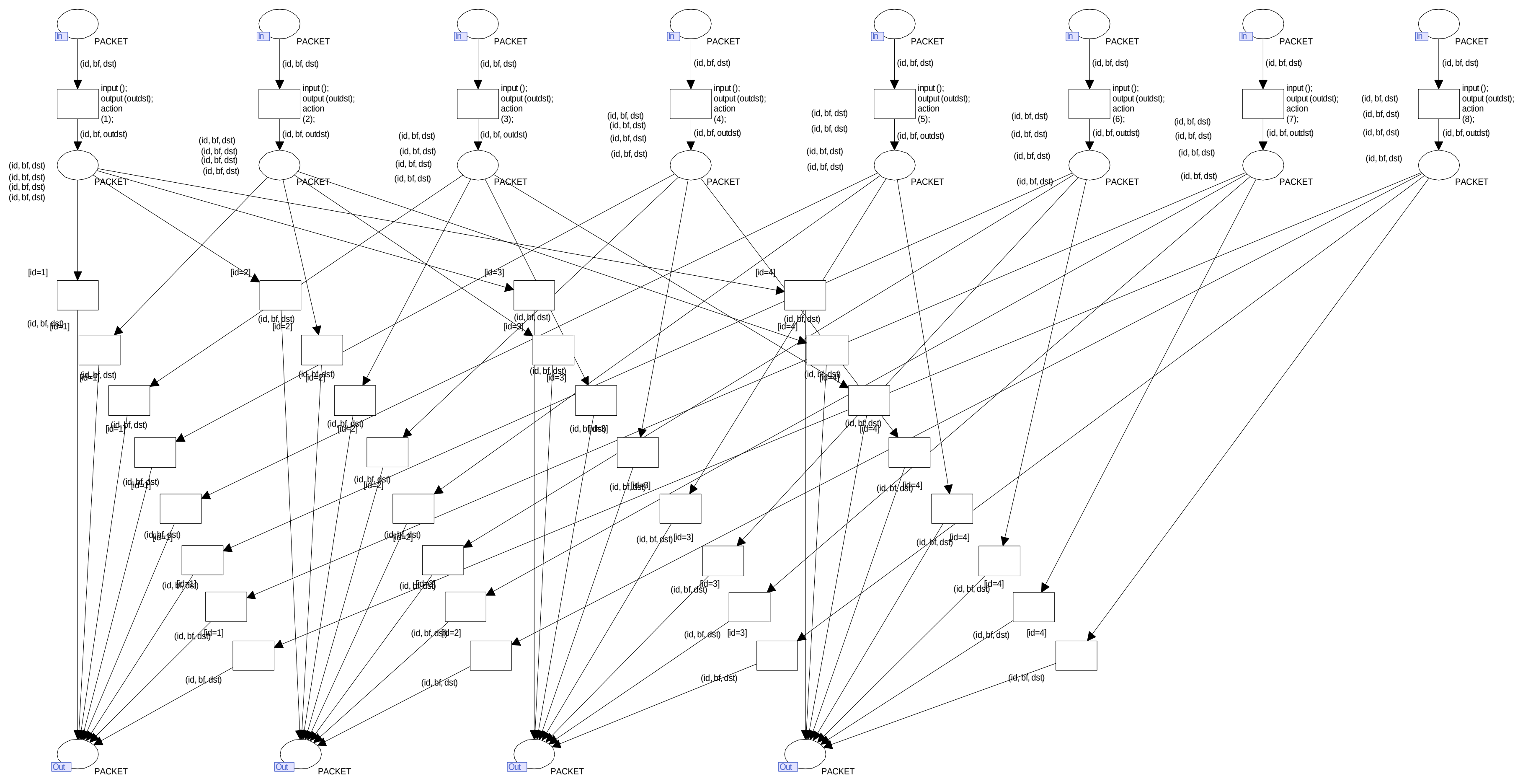


Figure 44: Sending network component for PVFS file read

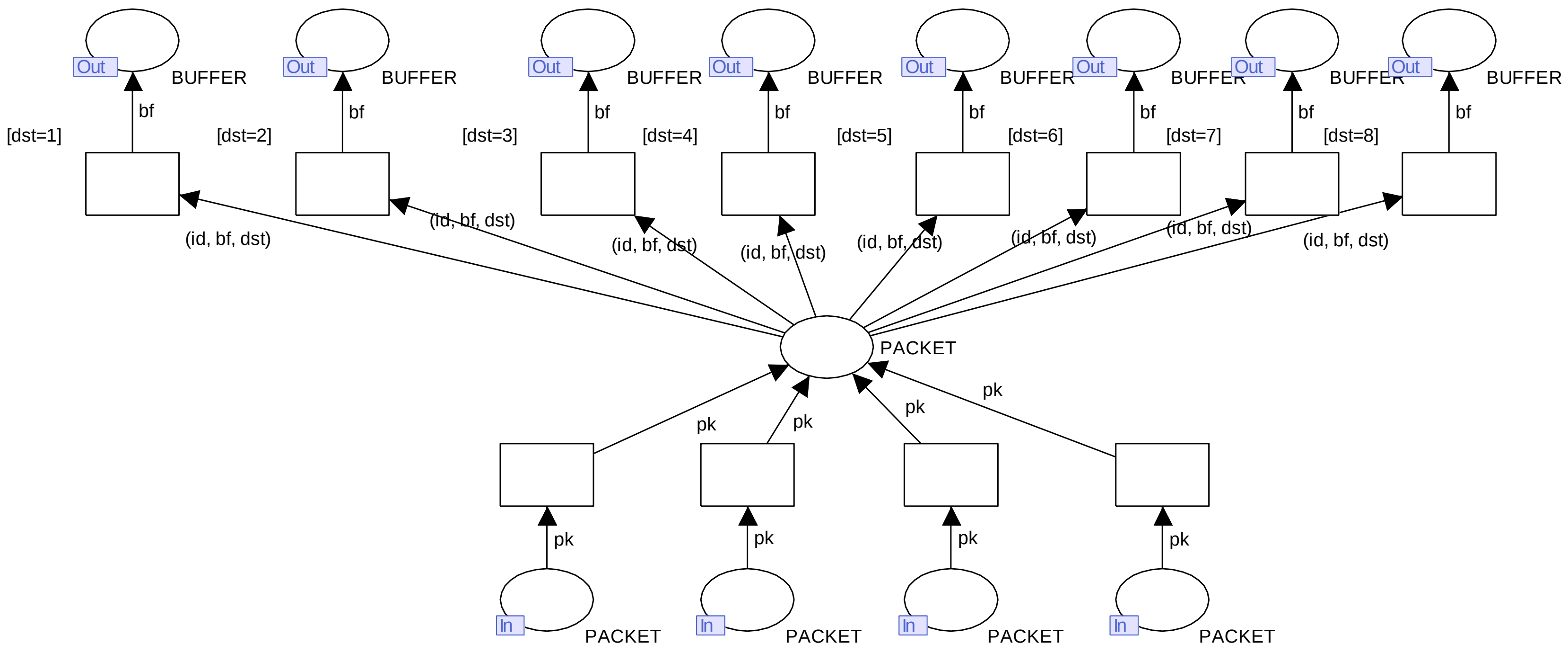


Figure 45: Receiving network component for PVFS file read

6.3.3 File read model server component

I/O servers are where the actual I/O operations are performed. Each PVFS file system has multiple I/O servers that work independently in parallel to provide large I/O bandwidth that local file system could never achieve.

Each I/O server, similarly to the client side, has a network layer to process network packets from the network component. A network packet, after arriving at the I/O server, is examined and categorized into different receive buffers, using a first-come-first-served (FCFS) mechanism. This process is designed following the same implementation in the real system. Each client has its own receive buffer.

The server component, following a FCFS order, takes read requests from the receive buffers and sends them to the local file system. The requests are sending in chunk of 64 Kbytes, which is the PVFS default stripe depth. If the PVFS file system is built with a different stripe depth, this chunk size is changed. The local file system on the I/O server performs a sequential read operation. Since the I/O server component takes read request from the receive buffers using FCFS order, the read request chunks are mixed together. The next chunk of read requests may not be from the same client as the chunk before it. Two different clients rarely try to read the same file at the same location. This causes the read requests stream sending to the local file system to have a very special pattern. This pattern is multiple session of sequential read requests. Each session may start at a random location. The Petri Net model for the server component for PVFS file read operation is presented in Figure 46.

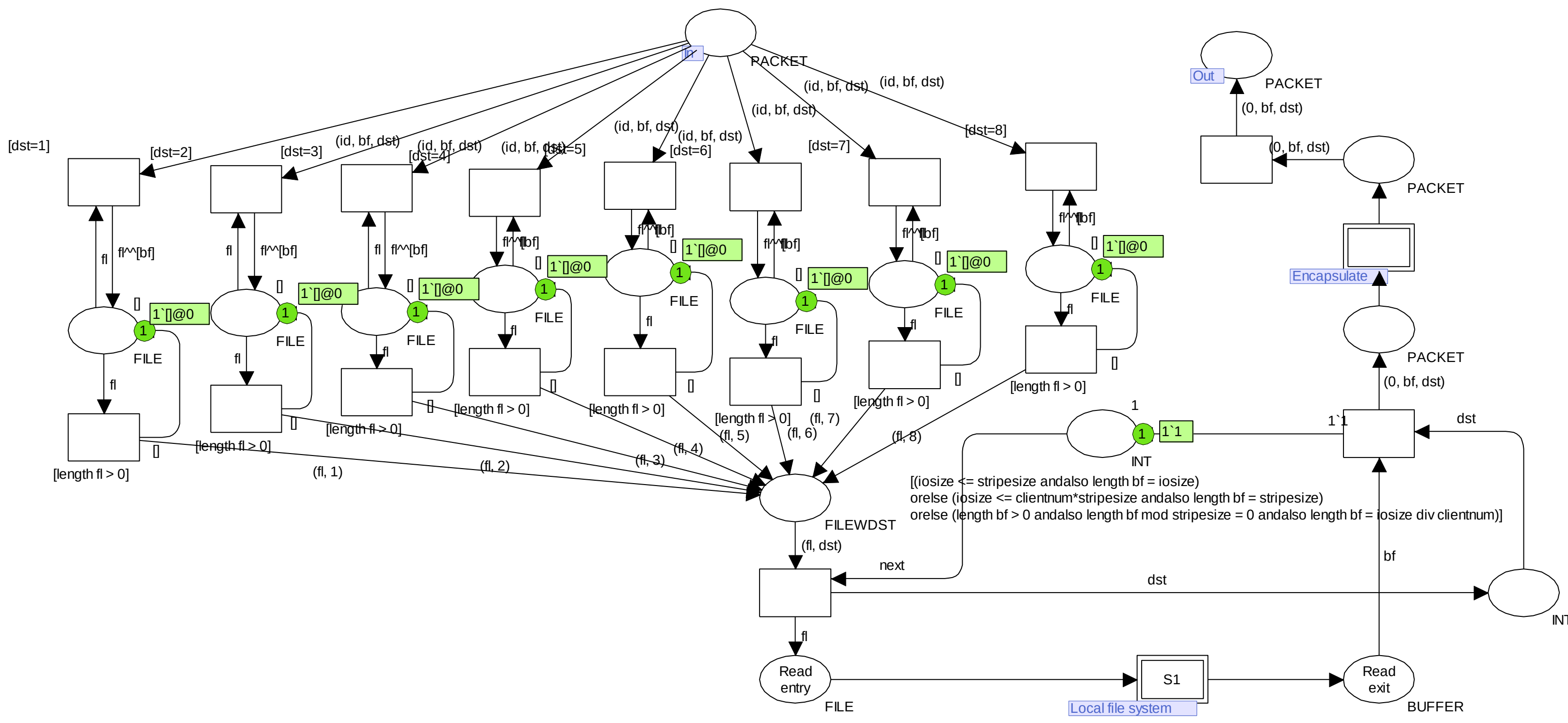


Figure 46: Server component for PVFS file read

After read requests passing through the local file system component, it returns the result data read from disk. At this step, the I/O server component sends these data through a network packet creation process similar to the client component. However, when the client component send the read requests over the network, the size of these read requests are relatively small and can fit within a standard frame. The result data, however, do not. They need to be divided into multiple segments before they are attached the headers and network addresses. The Petri Net model for dividing data into segments is presented in Figure 47.

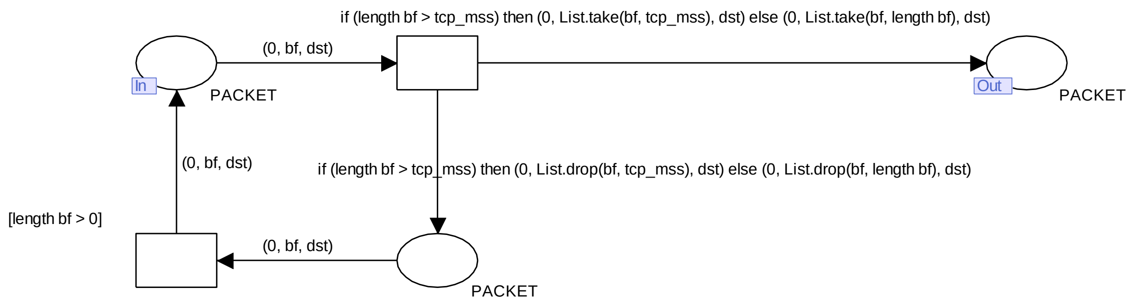

if (length bf > tcp_mss) then (0, List.take(bf, tcp_mss), dst) else (0, List.take(bf, length bf), dst)
(0, bf, dst)
In
PACKET
Out
PACKET
(0, bf, dst)
if (length bf > tcp_mss) then (0, List.drop(bf, tcp_mss), dst) else (0, List.drop(bf, length bf), dst)
[length bf > 0]
(0, bf, dst)
PACKET

Figure 47: Data segmentation component for
PVFS file read

The segment size of a packet is limited by the MTU of the network. Usually, in a Gigabit Ethernet network, the MTU is set to 1500. This means that a network packet maximum size is 1500 bytes.

6.4 File write model implementation

From the application standpoint, writing a file to a parallel file system is no different than writing a file to a local file system. The way an application writes a file is similar to the following illustration.

bytes_write = fwrite(buffer, block_size, number_of_block, file_handle);

The top level model is very similar to the I/O read model. The operation is divided into three main components: the client component, the network component and the server component. The Petri Net implementation of the top level model is presented in Figure 48.

6.4.1 File write model client component

The top level of the file write model client component is simple. The file data needed to be written to disk are broken into multiple blocks of write requests. These write requests are passed to the client simulation component. The client component will process the data then send the packaged data to the network component. The result of the write operation is a series of return codes received from the network model.

The implementation of the client component for file write operation is quite similar to the client component of the file read operation. However, write requests not only contain requests to write data to disk but also contain the actual data needed to be written. The client component needs to divide these blocks of data into multiple payloads. The number of actual payloads is determined by the number of I/O servers in the system. The Petri Net model for the PVFS client component is presented in Figure 49.

Payloads are created by striping request data into multiple chunks according to the file system's stripe depth parameter. Stripe depth in PVFS usually is 64 Kbytes. The distribution of data chunks in a payload is done using round-robin mechanism. The Petri Net implementation of the payload creation process is presented in Figure 50.

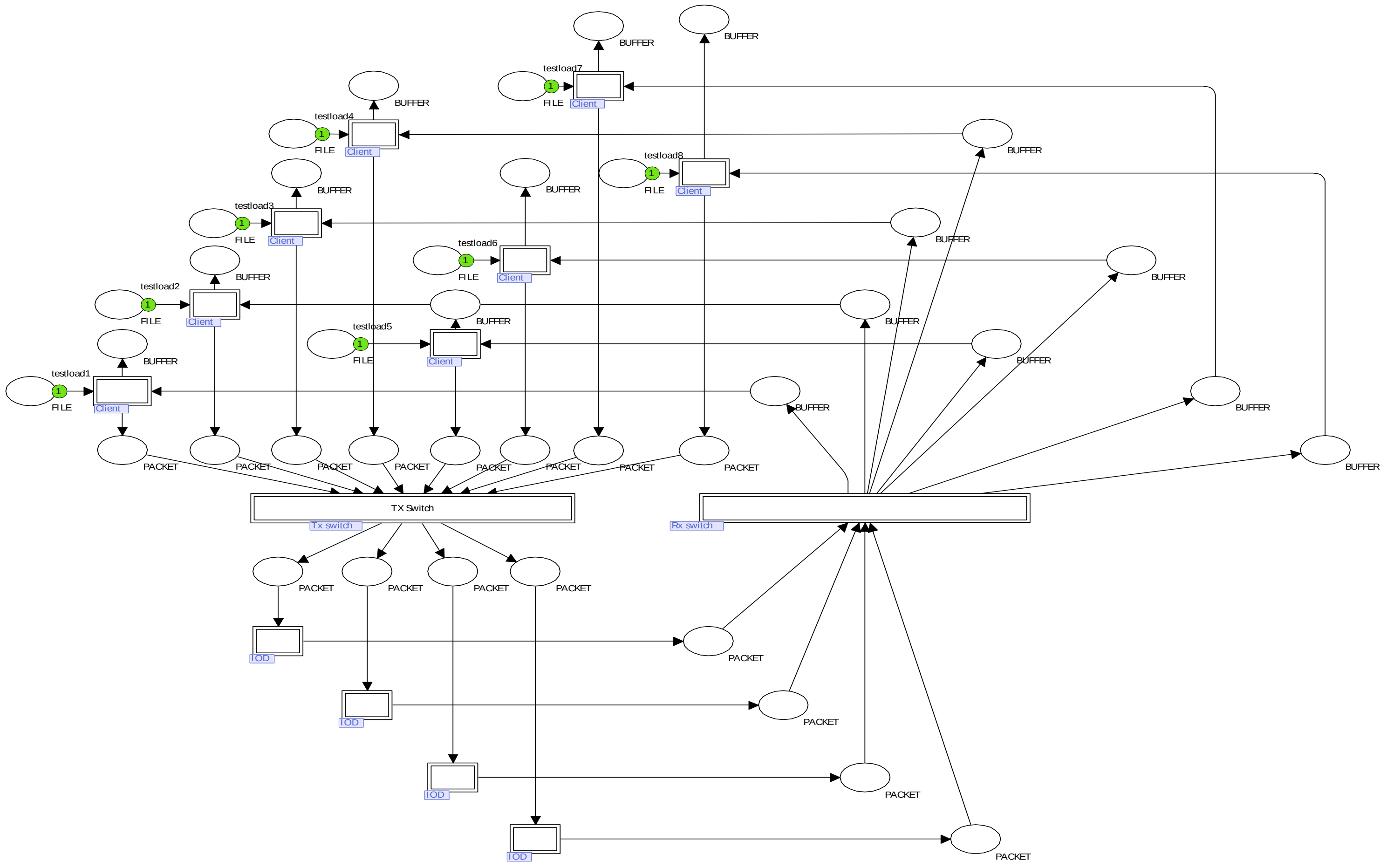


Figure 48: High level PVFS application write model

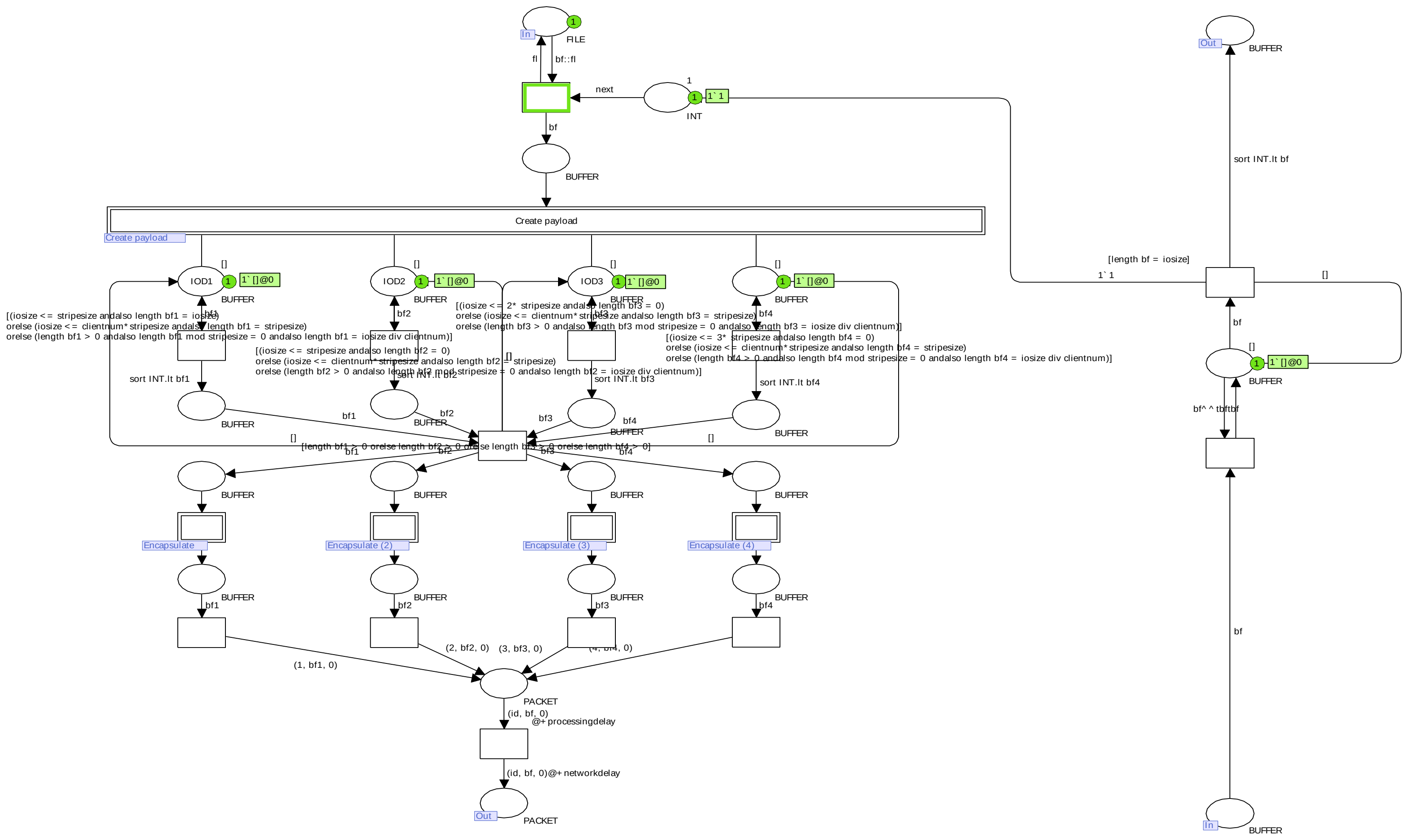

In
1
FILE
fl
bf::fl
next
1
1
1`1
INT
bf
BUFFER
Create payload
Create payload
[]
IOD1
1
1`[]@0
BUFFER
[]
IOD2
1
1`[]@0
BUFFER
[]
IOD3
1
1`[]@0
BUFFER
[]
1
1`[]@0
BUFFER
bf1
[(iosize <= stripesize andalso length bf1 = iosize)
orelse (iosize <= clientnum* stripesize andalso length bf1 = stripesize)
orelse (length bf1 > 0 andalso length bf1 mod stripesize = 0 andalso length bf1 = iosize div clientnum)]
bf2
[(iosize <= stripesize andalso length bf2 = 0)
orelse (iosize <= clientnum* stripesize andalso length bf2 = stripesize)
orelse (length bf2 > 0 andalso length bf2 mod stripesize = 0 andalso length bf2 = iosize div clientnum)]
bf3
[(iosize <= 2* stripesize andalso length bf3 = 0)
orelse (iosize <= clientnum* stripesize andalso length bf3 = stripesize)
orelse (length bf3 > 0 andalso length bf3 mod stripesize = 0 andalso length bf3 = iosize div clientnum)]
bf4
[(iosize <= 3* stripesize andalso length bf4 = 0)
orelse (iosize <= clientnum* stripesize andalso length bf4 = stripesize)
orelse (length bf4 > 0 andalso length bf4 mod stripesize = 0 andalso length bf4 = iosize div clientnum)]
sort INT.lt bf1
sort INT.lt bf2
sort INT.lt bf3
sort INT.lt bf4
BUFFER
BUFFER
BUFFER
BUFFER
bf1
bf2
bf3
bf4
[]
[]
[]
[length bf1 > 0 orelse length bf2 > 0 orelse length bf3 > 0 orelse length bf4 > 0]
bf1
bf2
bf3
bf4
BUFFER
BUFFER
BUFFER
BUFFER
Encapsulate
Encapsulate (2)
Encapsulate (3)
Encapsulate (4)
BUFFER
BUFFER
BUFFER
BUFFER
bf1
bf2
bf3
bf4
(1, bf1, 0)
(2, bf2, 0)
(3, bf3, 0)
(4, bf4, 0)
PACKET
(id, bf, 0)
@+ processingdelay
(id, bf, 0)@+ networkdelay
Out
PACKET
Out
BUFFER
sort INT.lt bf
[length bf = iosize]
1`1
[]
bf
[]
1
1`[]@0
BUFFER
bf^^tbf
tbf
bf
In
BUFFER

Figure 49: PVFS client component for file write

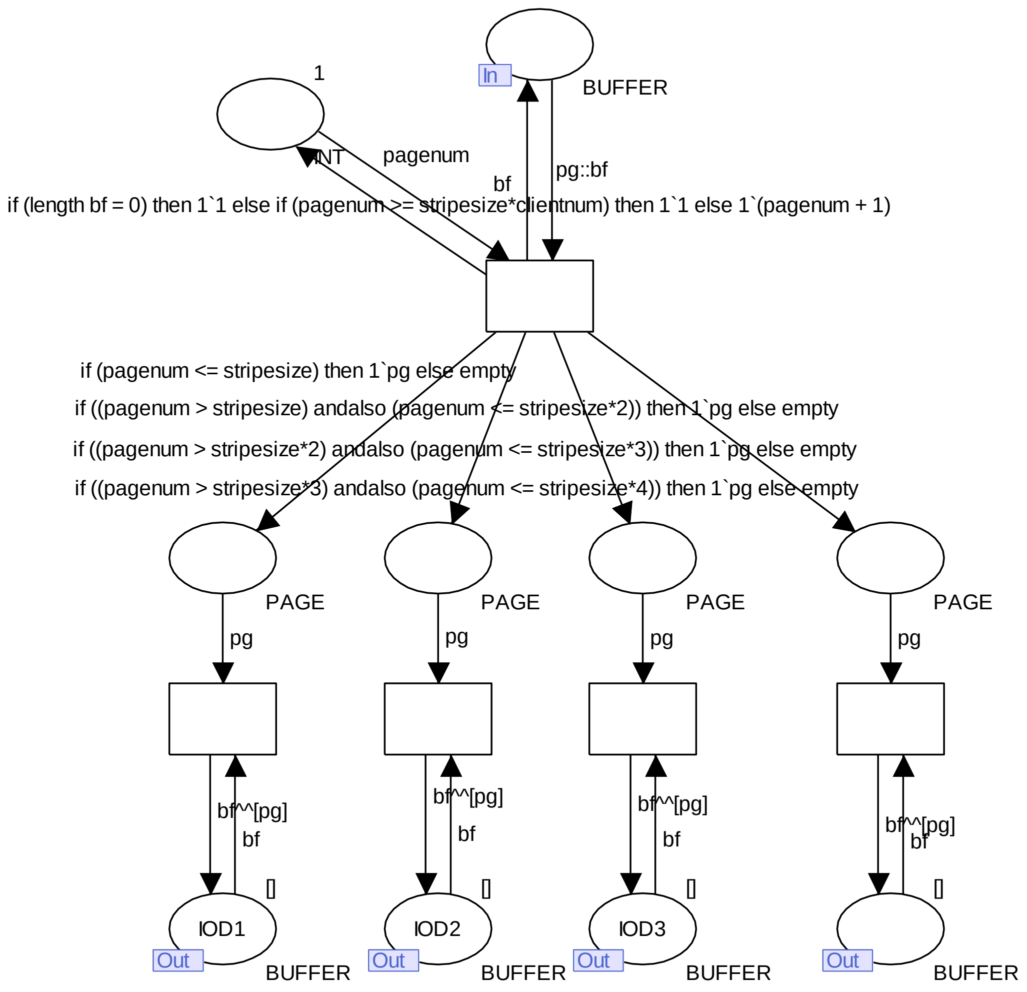

1
In
BUFFER
INT
pagenum
bf
pg::bf
if (length bf = 0) then 1`1 else if (pagenum >= stripesize*clientnum) then 1`1 else 1`(pagenum + 1)
if (pagenum <= stripesize) then 1`pg else empty
if ((pagenum > stripesize) andalso (pagenum <= stripesize*2)) then 1`pg else empty
if ((pagenum > stripesize*2) andalso (pagenum <= stripesize*3)) then 1`pg else empty
if ((pagenum > stripesize*3) andalso (pagenum <= stripesize*4)) then 1`pg else empty
PAGE
pg
bf^^[pg]
bf
[]
IOD1
IOD2
IOD3
Out
BUFFER

Figure 50: Payload creation component for PVFS file write

After creating the payloads, the client component attaches network addresses and control information to the payloads to create network packets. Since the packet size depends on the MTU of the network, the client component has to split the payloads into multiple segments. The Petri Net model for dividing data into segments is presented in Figure 51.

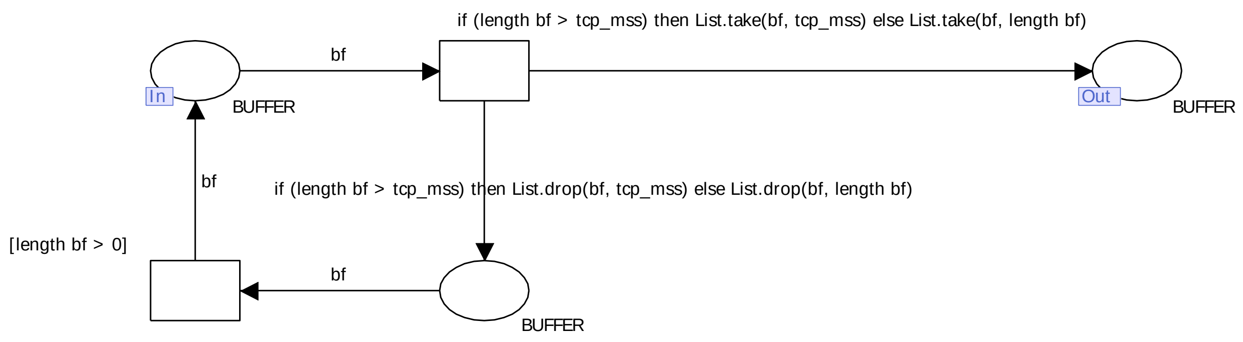

if (length bf > tcp_mss) then List.take(bf, tcp_mss) else List.take(bf, length bf)
bf
In
BUFFER
Out
BUFFER
bf
if (length bf > tcp_mss) then List.drop(bf, tcp_mss) else List.drop(bf, length bf)
[length bf > 0]
bf
BUFFER

Figure 51: Data segmentation component for PVFS file write

Typically, the MTU is set to 1500 in a Gigabit Ethernet network, so the packet size for data sending from clients to I/O servers is at the maximum size of 1500 bytes.

6.4.2 File write model network component

The network component model in the file write operation is very similar to the network component model in the file read operation. There are only some slight differences in the model due to the data flow of the operation being different. The network packets from the client component are examined, the destination addresses are checked and the packets are routed to the correct receiver. The network component provides the transportation for the packets and also simulates the wire-delay on the network medium. The Petri Net model for the sending and the receiving network component for PVFS file write are presented in Figure 52 and Figure 53.

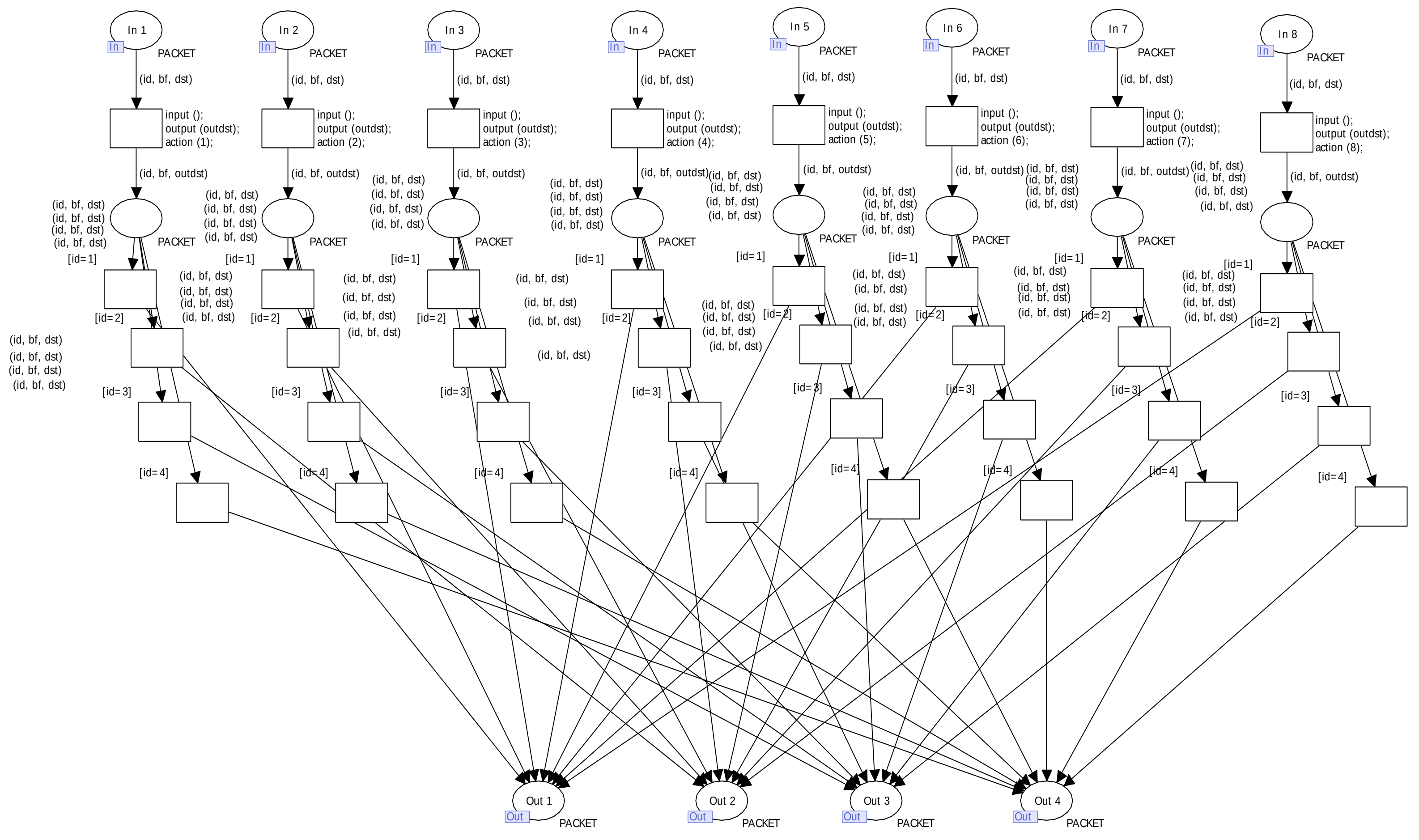


Figure 52: The sending network component for PVFS file write

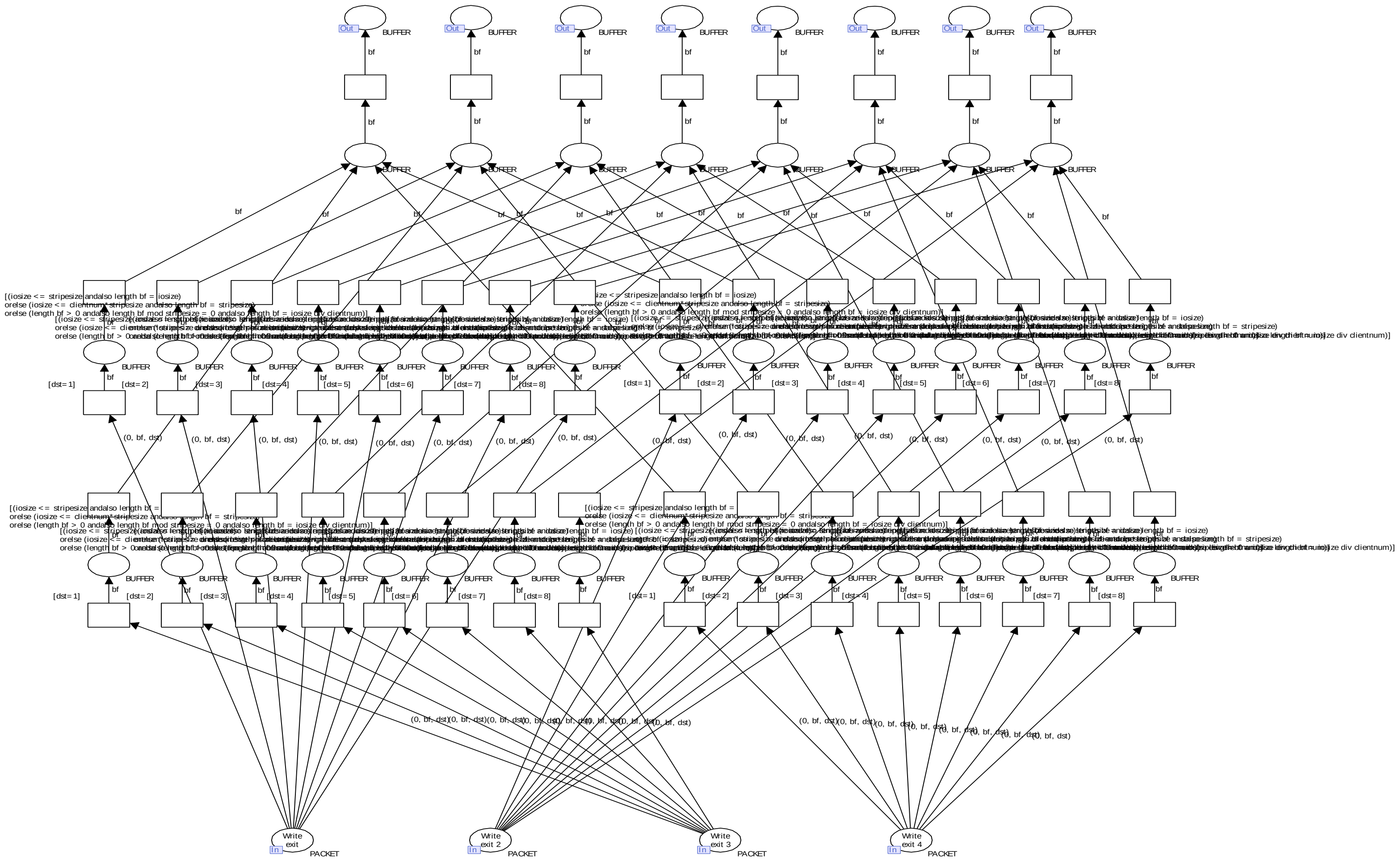

Out
BUFFER
bf
[(iosize <= stripesize andalso length bf = iosize)
orelse (iosize <= clientnum*stripesize andalso length bf = stripesize)
orelse (length bf > 0 andalso length bf mod stripesize = 0 andalso length bf = iosize div clientnum)]
[dst=1]
[dst=2]
[dst=3]
[dst=4]
[dst=5]
[dst=6]
[dst=7]
[dst=8]
(0, bf, dst)
Write exit
Write exit 2
Write exit 3
Write exit 4
In
PACKET

Figure 53: The receiving network component for PVFS file write

6.4.3 File write model server component

The file write server component is built upon the local write model. The local write model is the foundation of the file write server model. A network packet, after arriving at the I/O server, is processed and sent to the local file write model. The server creates a receive buffer for each client sending in requests. The server model examines the network packets and moves the request data into the correct buffers using FCFS mechanism. This process is designed to follow the same implementation in the real system.

Since each packet is limited by the maximum segmentation size of the network, the server component combines multiple packet data into the original request sent by the client. Unlike the file read server model, the file write server model does not attempt to combine the original request into 64Kbytes chunk. Instead the server model combines the fragmented data into the original request and sends it to the local file write model. Because of this, the block sizes of the write requests sent to the local file write model are not fixed. PVFS is relied on the delay write mechanism of the local file system to combine multiple different small write requests into big and sequential write requests. The local file system on the I/O server performs the write operation. Since the server model sends the write requests to the local file system model as it receives in a FCFS order, the block size of the write requests are quite random. Even though, the write requests could be in sequential order, the block sizes of the requests are not. This creates a special I/O access pattern. The Petri Net model for PVFS file write server model is presented in Figure 54.

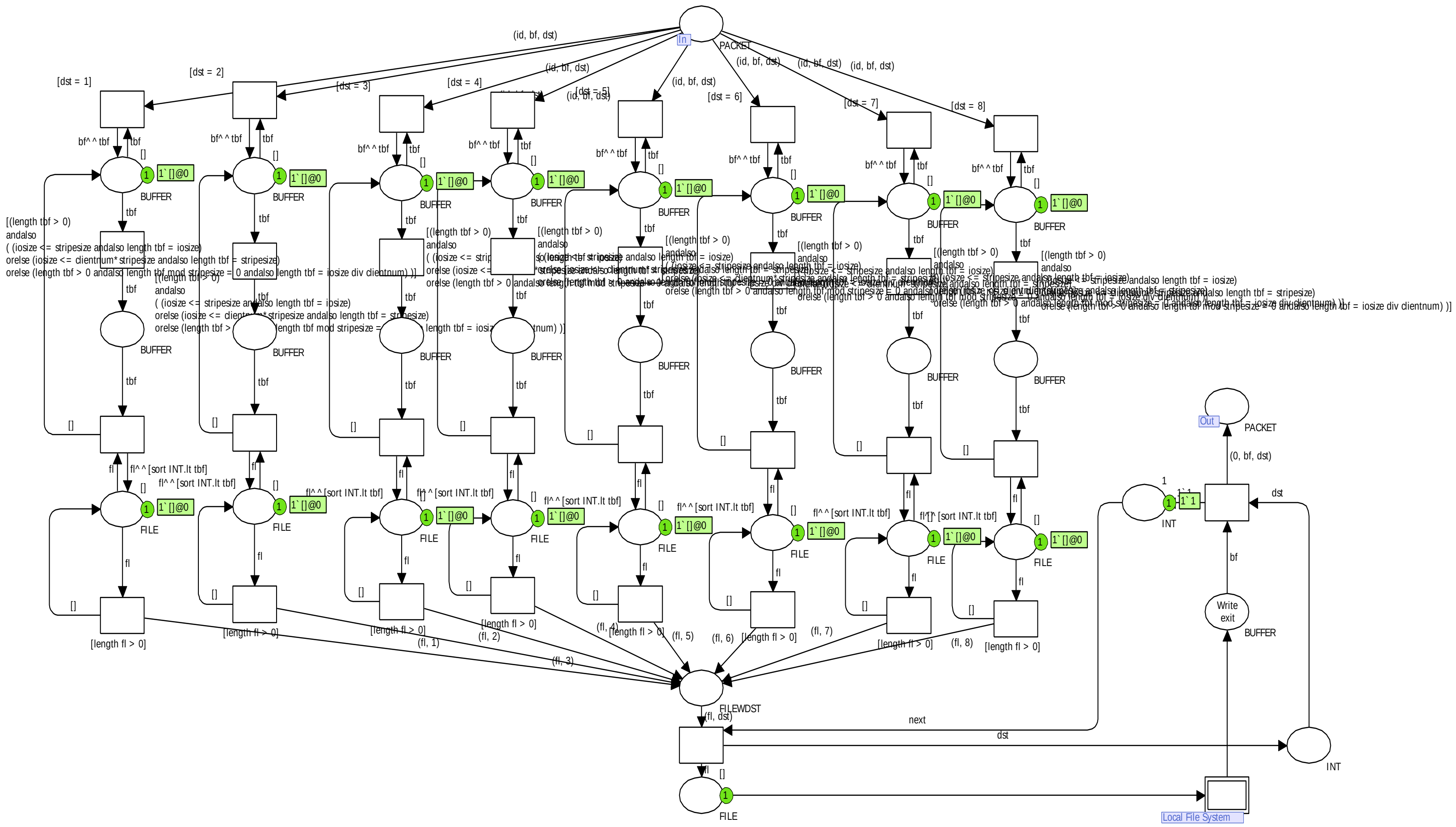


Figure 54: Server component for PVFS file write

After read requests pass through the local file system component, it returns the result data read from disk.

6.5 Summary

This chapter presents a set of detailed and hierarchical performance models of the PVFS file system using Colored Petri Nets. PVFS read operation and PVFS write operation are studied and their models are built. Each operation is divided into sub-components: client, network and server. The models of these components are presented. The client components are where the read requests and write requests from applications are received. The client components take these read requests and write requests and create several network packets. The network packets are sent to the servers using the network component. The server component built upon the local file system model processes the request data and performs actual I/O operations. The results of the I/O operations are sent to the clients using the network component.

The current PVFS model is setup to have eight clients and four servers. This is equal to a small size production file system. The model can be extended to have more clients and servers. The model currently uses TCP/IP protocol over a Gigabit Ethernet network. It can also be modified to simulate a different network protocol and different network hardware.

Chapter 7

PARALLEL FILE SYSTEM SIMULATION MODEL PERFORMANCE VALIDATION

7.1 Introduction

This chapter presents the performance validation of the simulation model for a PVFS file system. Because PVFS is a parallel file system, the number of clients accessing the file system at the same time is important. The file system is designed to provide a massive I/O bandwidth and throughput by allowing multiple I/O servers to work with multiple clients at the same time. The performance measurements are performed similarly to the way the local file system performance experiments are done.

7.2 Validation setup

In order to validate the entire Petri Net file system model against real-world data, the model hardware parameters, such as memory delay, execution speed, function overhead, and disk speed, are measured directly from the machines where the real experiments take place, using kernel traces. The same Ftrace mechanism as described in Chapter 5 is utilized. Since PVFS is a parallel file system, a network is involved. The performance parameters of the network stack on the client and server machines are also measured, using the Ftrace facility. Network performance parameters on the wire are recorded, using network monitoring tools, including ping, traceroute and packet sniffer. The performance validations are executed, starting with one client accessing the file system. The number of clients is increased until the number of clients equals eight. The PVFS file system model is implemented with four I/O servers. With eight clients (double the amount of servers) accessing the file system simultaneously, the file system level of stress is high enough to produce good performance results.

7.3 Performance validation experiments

Simulations are run several times, and the average results are used to compare with iozone benchmark results running on the test system. The simulation experiments are run using a set of synthetic I/O requests and simulating sequential I/O. The I/O requests are grouped into similar block-size configurations of the iozone benchmark.

7.3.1 Single client performance experiment

In this performance measurement, one client reads and writes to the PVFS file system. The result of the I/O read performance in the experiment is presented in Figure 55. The error bars are set at 20%.

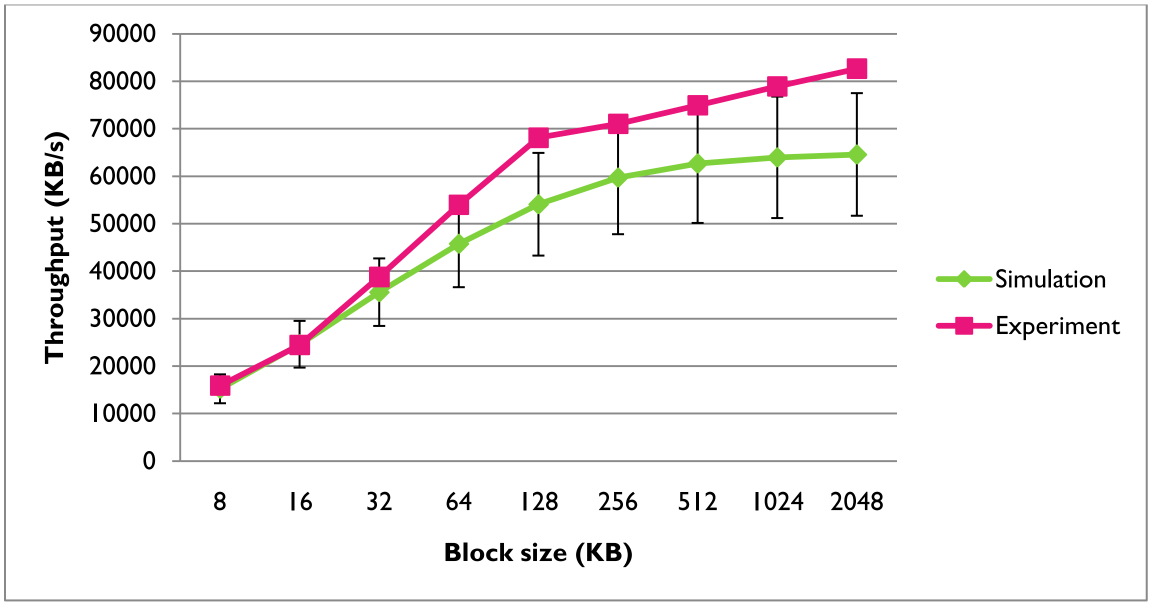


Figure 55: Single client I/O read validation

All points, except the last one, are within or very close to 20% of the real-world measurement. Even though the last data point is farther away than other data points, it is still a very good result, and the error is likely to come from measurement inaccuracy. The simulation data points are consistently lower than real-world data.

The result of the I/O write performance in the experiment is presented in Figure 56. The error bars are set at 20%.

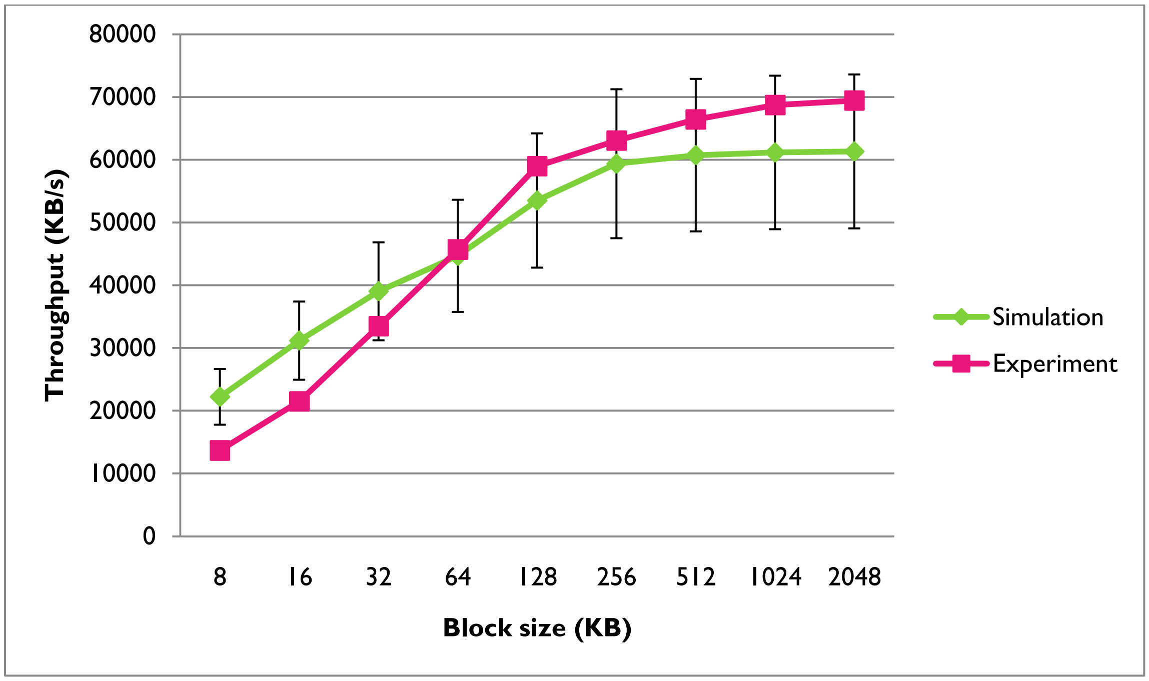


Figure 56: Single client I/O write validation

Like the I/O read result, the I/O write result is also very good. The majority of data points are within 20% of the real system measurement. Simulation data in this experiment are not consistently lower than real-world data like we have observed in the I/O read result.

At small block size, the simulation results are higher than real-world data, but at bigger block size, the simulation results become lower.

The reason for this performance behavior comes from the buffer design of the I/O server model. The I/O server has a receive buffer for every client sending requests to the server. Data are taken out of the buffers, using a first-come-first-served (FCFS) order. The receive buffers in the real server are implemented, using a linked-list data structure. The larger the buffer, the slower an item in the buffer can be accessed. Currently, the buffers of the simulation model are implemented to have a fixed operating cost. This means that the time it takes to access an item in the buffer stays the same, regardless of the size of the buffer.

The number of write requests needed to write a file when using a small block size is much larger than the number of write requests when using a large block size. In the simulation model, this does not change the time it takes to de-queue requests. This causes the simulation model to run faster than the real system at the small block sizes and slower than the real system at the large block sizes.

7.3.2 Two clients performance experiment

In this experiment, two clients read and write to the PVFS model. The result of the I/O read performance in the experiment is presented in Figure 57. The error bars are set at 20%.

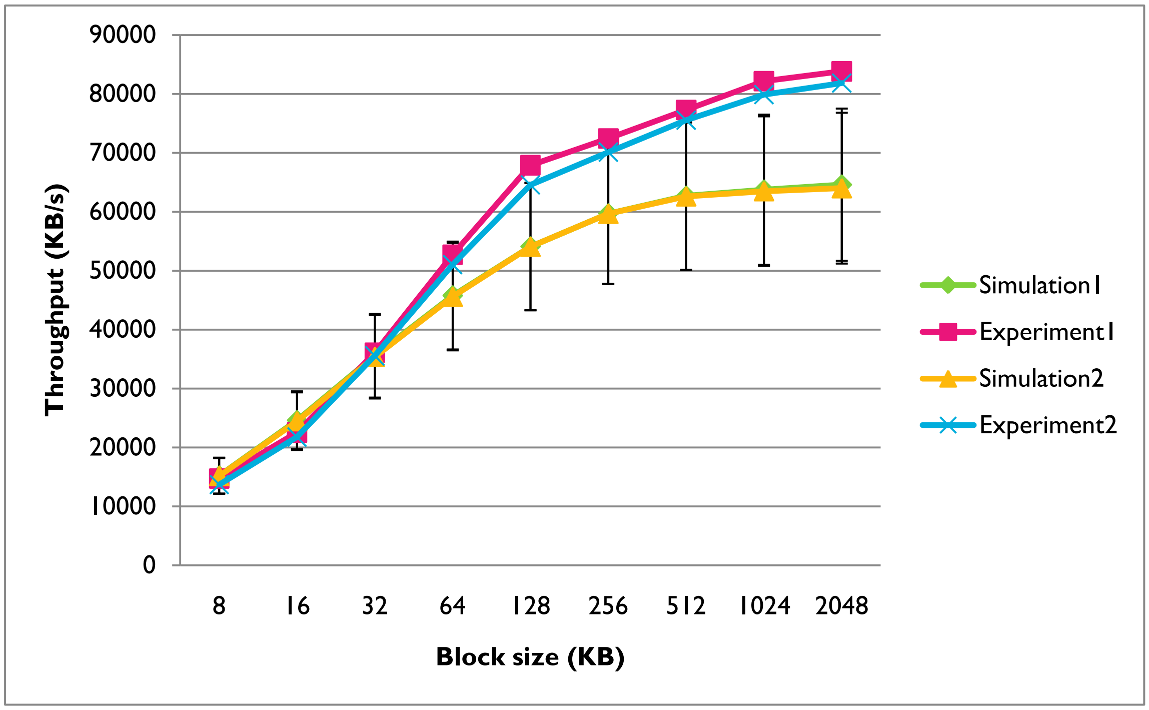


Figure 57: Two clients I/O read validation

The result is consistent with the I/O read result presented in the single client experiment. All data points, except two at the highest block size are within or close to 20% of the real-world data. The shapes of the performance curves are also similar to the single client result. The simulation data points are consistently lower than the real-world data. The performances of the clients show only slight differences. This shows the workload is balanced well in the PVFS file system, and the file system level of stress is still low.

The result of the I/O write performance in the experiment is presented in Figure 58. The error bars are set at 20%.

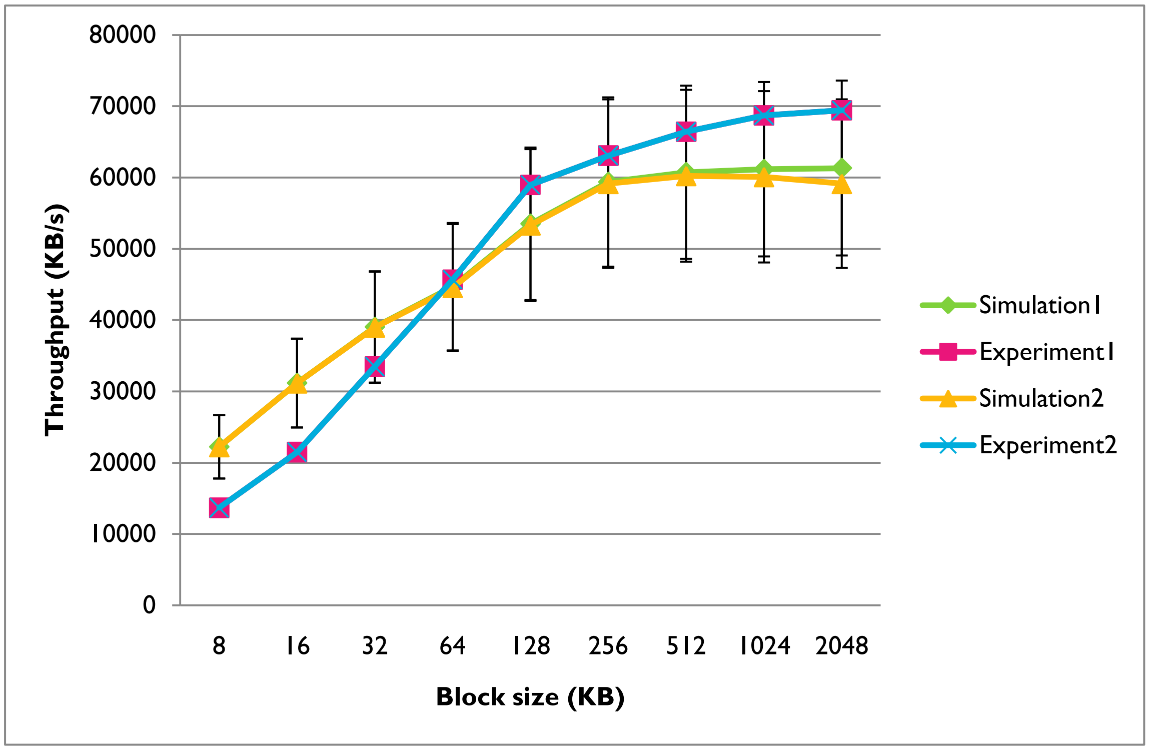


Figure 58: Two clients I/O write validation

I/O write also exhibits similar behavior as the single client experiment. All data points, except two at the small block sizes, are within 20% of the real-world data. The two exception data points are also very close to 20% of the real-world data. The performance curves are also similar to the single client experiment. The data points for small block-size simulation are higher than the real-world data, but the data points for bigger block-size simulation are lower than the real-world data.

7.3.3 Three clients performance experiment

In this experiment, three clients read and write to the PVFS model. The result of the I/O read performance in the experiment is presented in Figure 59. The error bars are set at 20%.

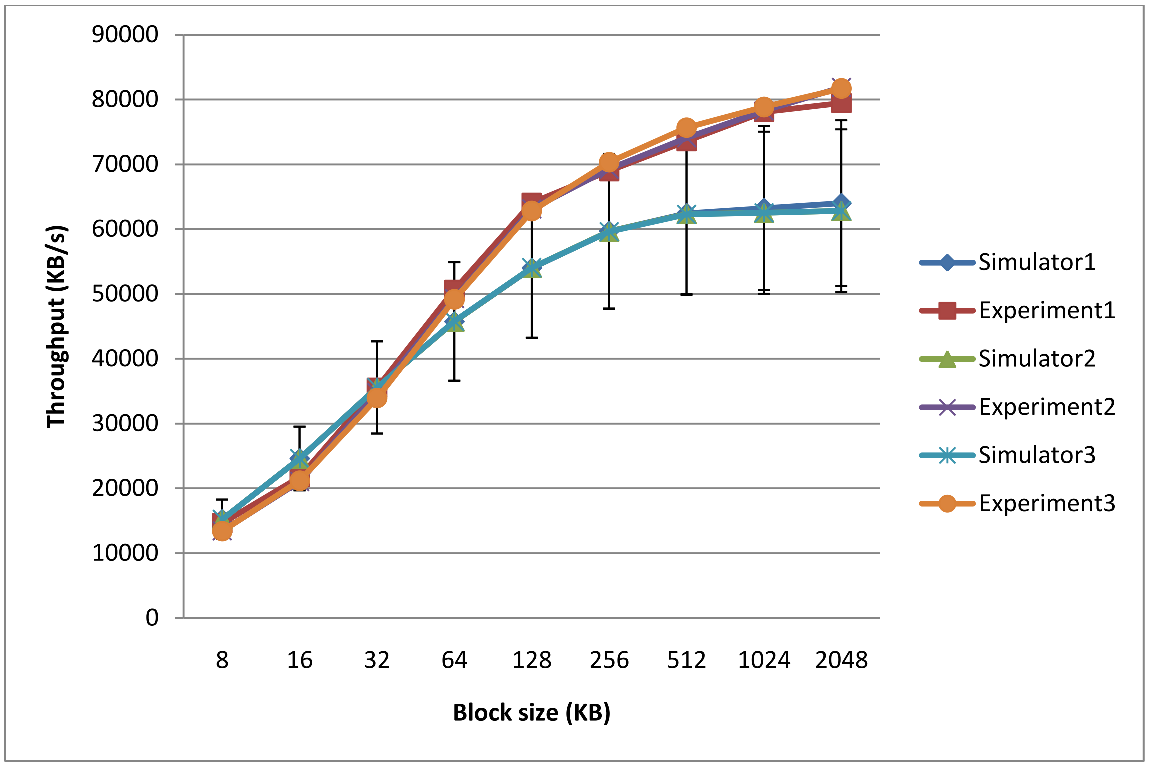


Figure 59: Three clients I/O read validation

In general, the performance behavior is similar to what we have observed so far. The simulation data points are also consistently lower than the real-world data points. The performance curves are also very close together. This shows the file system is responding well, and the stress level is not high enough to make a difference.

The result of the I/O write performance in the experiment is presented in Figure 60. The error bars are set at 20%.

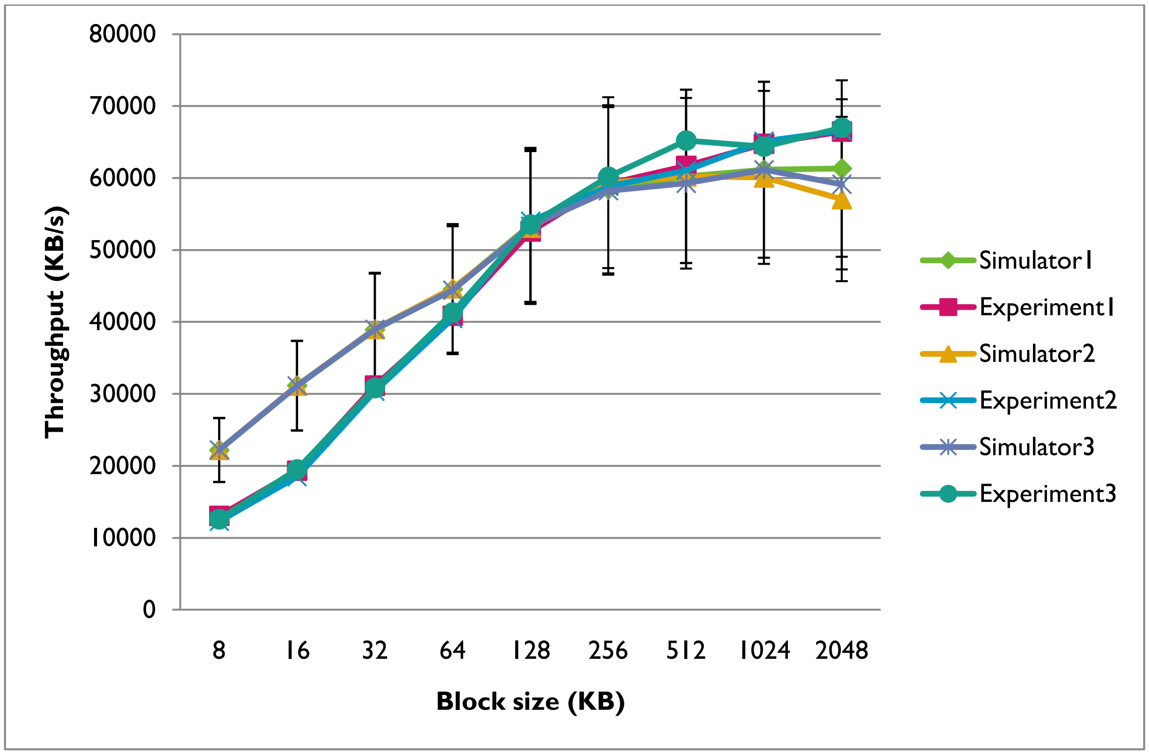


Figure 60: Three clients I/O write validation

The I/O write performance in the experiment confirms what was observed in the I/O read portion of the experiment. The file system stress level with three clients is still not high enough to make a difference in performance behavior. However, there are some slight differences from the previous I/O write performance chart at the bigger block sizes. These differences become more visible when the stress level becomes high enough. For the most part, data points are within 20% of the real-world data or very close.

7.3.4 Four clients performance experiment

In this experiment, four clients read and write to the PVFS model. The result of the I/O read performance in the experiment is presented in Figure 61. The error bars are set at 20%.

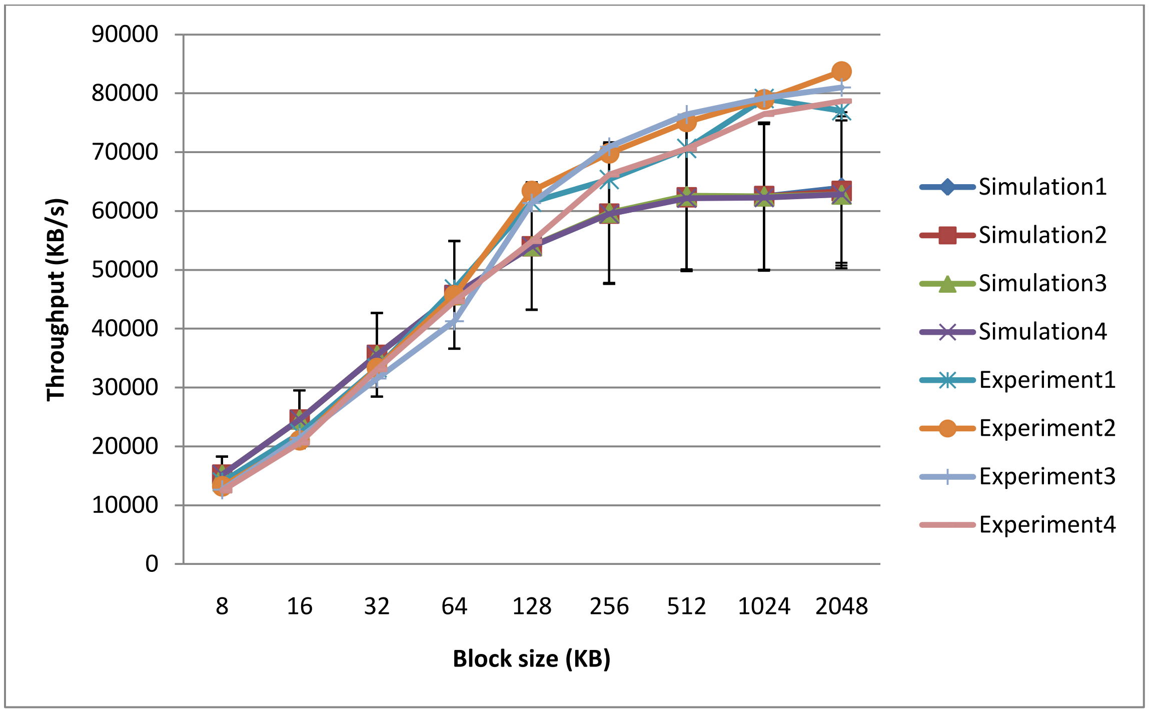


Figure 61: Four clients I/O read validation

With four clients accessing the PVFS file system at the same time, we start to notice variations within the data points, especially in the real-world data. The simulation data, however, are still very consistent. This is due to the simulation model having fewer factors affecting the result. The more clients accessing the PVFS file system, the more outside factors are introduced to the real-world data.

Even with the increasing variation of the data points, the experiment result is still very good. The performance behavior is still similar to what we have observed in previous experiments. The last two data points are not within 20% of the real-world data, but are still very close to them.

The result of the I/O write performance in the experiment is presented in Figure 62. The error bars are set at 20%.

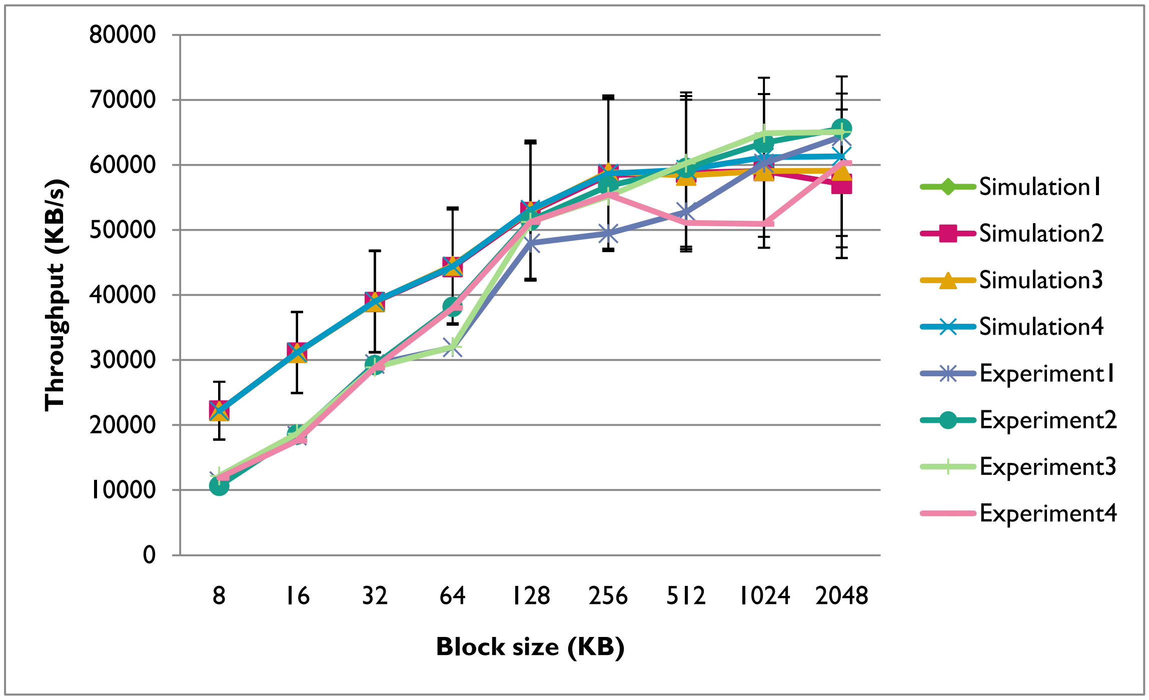


Figure 62: Four clients I/O write validation

The I/O write experiment result also has variations. The amount of variations is slightly more than in the I/O read experiment. In general, the performance behavior is slightly different to what we have previously observed. The simulation data points are higher than the real-world data points at small block sizes. The simulation data points are lower than the real-world data points at larger block sizes. The cross-over point is slightly shifted toward the larger block sizes.

The simulation data points are still within 20% of the real-world data points or close to them. The two data points at smallest block sizes are somewhat farther away from the real-world data points.

7.3.5 Five clients performance experiment

In this experiment, five clients read and write to the PVFS model. The result of the I/O read performance in the experiment is presented in Figure 63. The error bars are set at 20%.

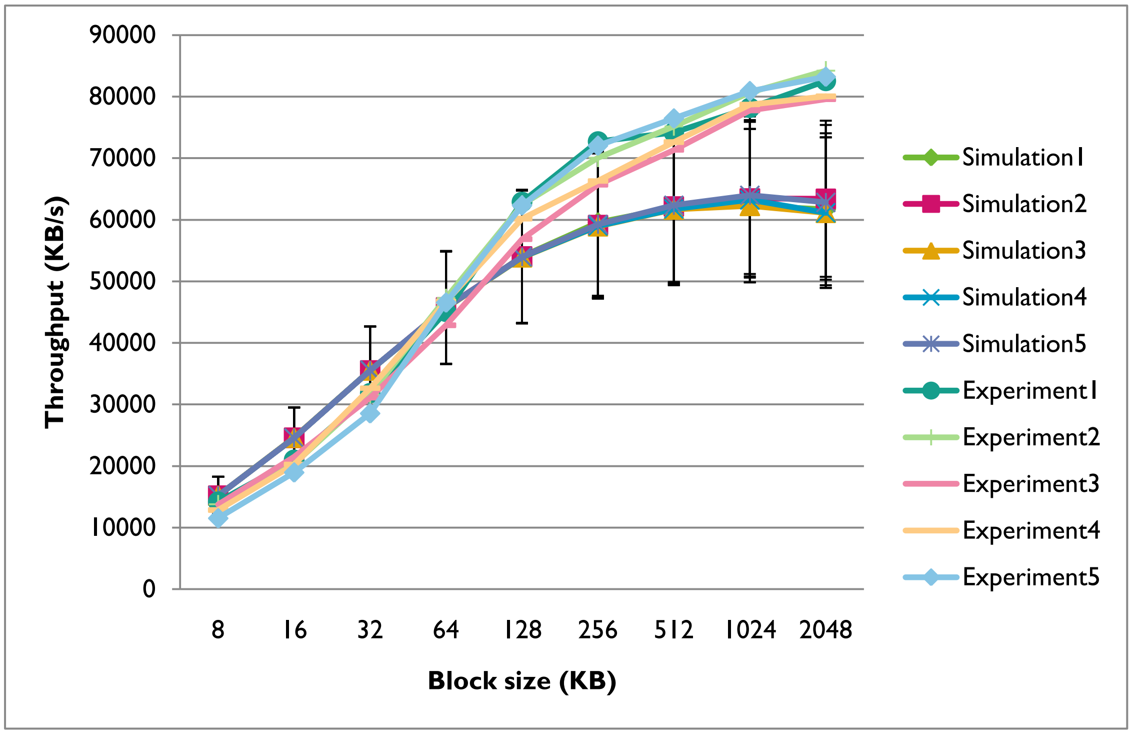


Figure 63: Five clients I/O read validation

The experiment result is still very consistent, even when five clients are reading the PVFS file system at the same time. The variations are there, but they do not badly affect the overall performance. Most data points, except the last two points at large block sizes, are within 20% of the real-world data.

The result of the I/O write performance in the experiment is presented in Figure 64. The error bars are set at 20%.

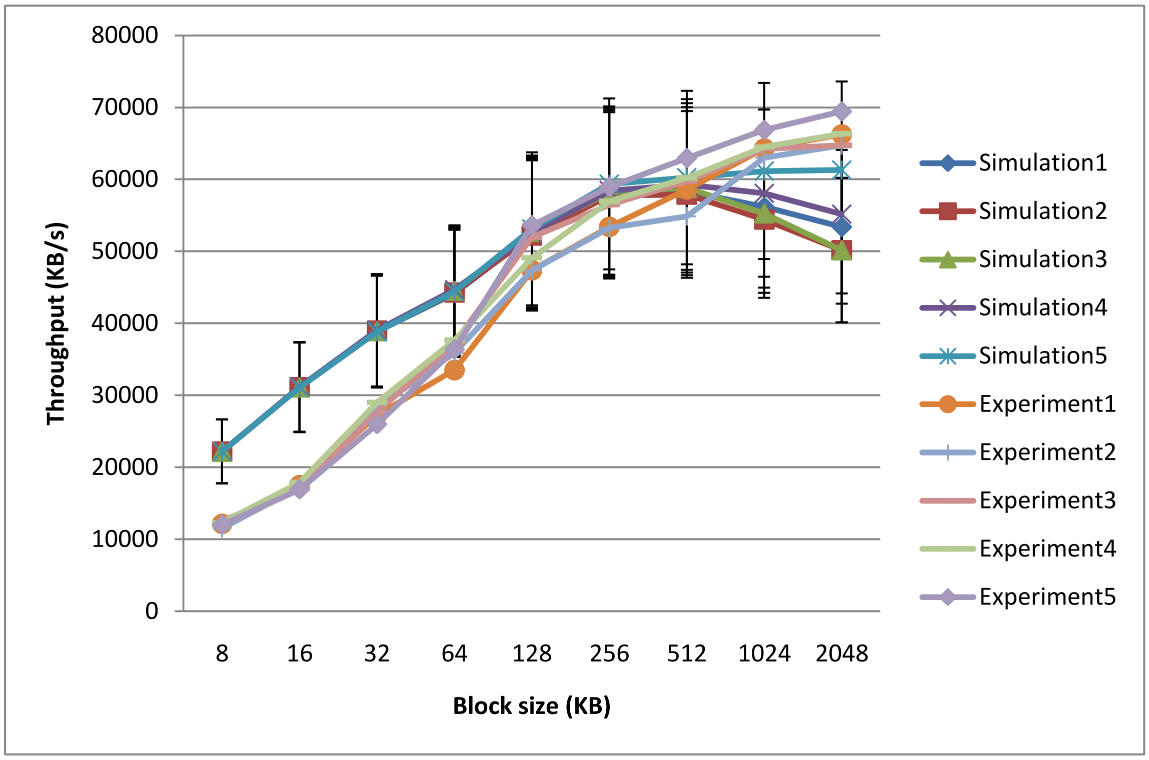


Figure 64: Five clients I/O write validation

Compared to the I/O read experiment, the I/O write experiment has more variations, and the effect of them on the overall performance is more visible. This is due to I/O write operations generating more stress on the file system than I/O read operations. In general, I/O write operations are slower and more resource intensive than I/O read operations.

The performance curves are still following the same trend. Simulation data points are higher than real-world data points at small block sizes and lower than real-world data at big block sizes. However, the stress on the file system has caused the error to become bigger, especially the data points at small block sizes. The gap between the simulation data points and the real-world data points has become significant. There are also more variations at the large block sizes than previously observed.

7.3.6 Six clients performance experiment

In this experiment, six clients read and write to the PVFS model. The result of the I/O read performance in the experiment is presented in Figure 65. The error bars are set at 20%.

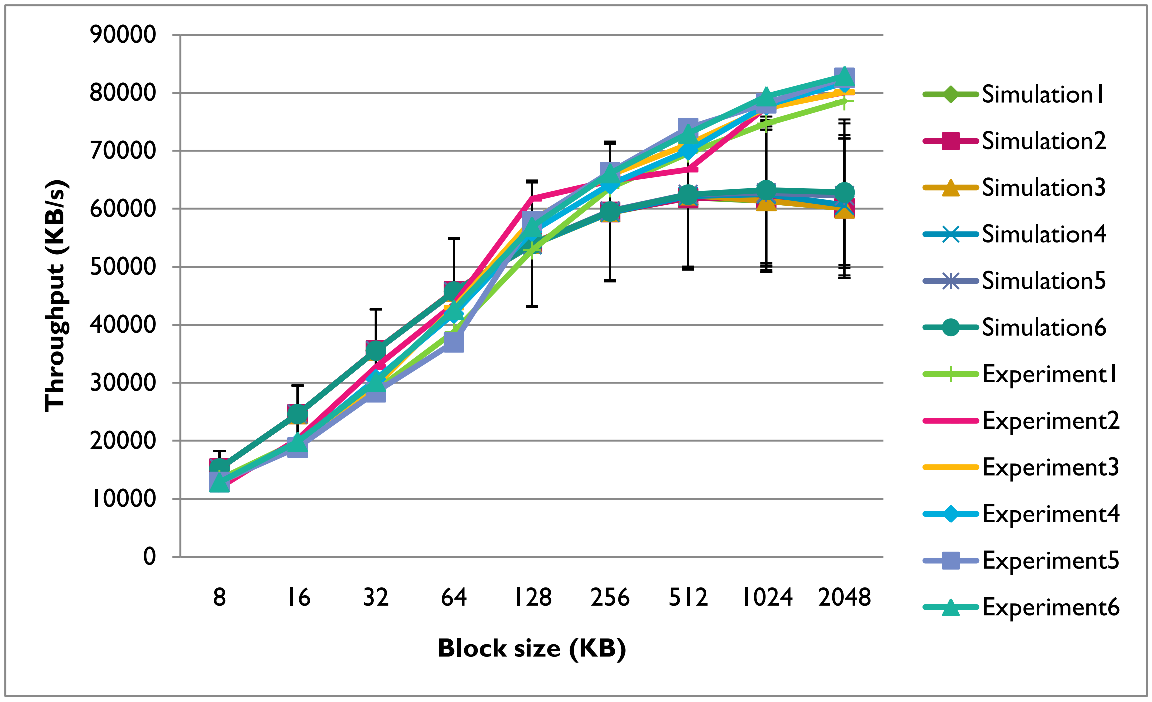


Figure 65: Six clients I/O read validation

Compared to the previous experiment, it is clear that the amount of variations increases consistently every time the number of clients increases. This supports the assumption, which seems to be obvious, that the level of stress on the file system increases when the number of clients, accessing the file system at the same time, increases.

However, the performance curves are still grouped together quite nicely. All data points, except the last two points, are still within 20% of the real-world data. In the next few experiments, we start to see significant changes in the performance behavior.

The result of the I/O write performance in the experiment is presented in Figure 66. The error bars are set at 20%.

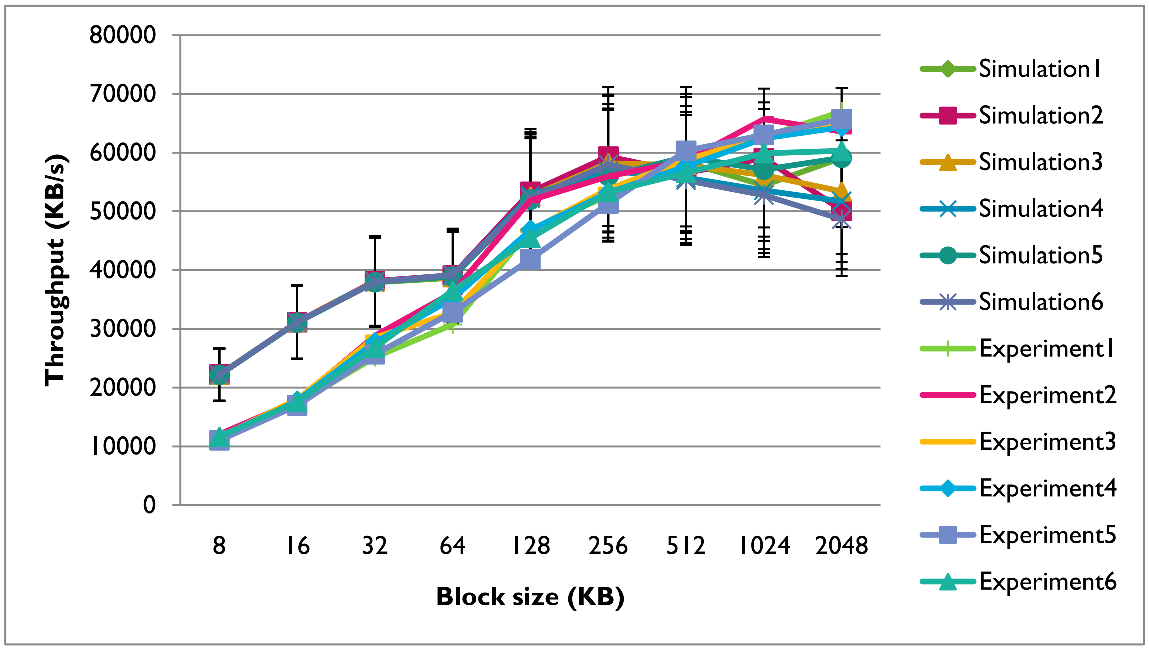


Figure 66: Six clients I/O write validation

The variations and the effects of the file-system stress level are very visible in this experiment. This shows that the file system stress level has become significant. At large block sizes, simulation data points are still within 20% of real-world data points. However, at small block sizes, the errors have become quite large. The performance curves are also not as smooth as before, even though they are still staying very close to each other.

7.3.7 Seven clients performance experiment

In this experiment, seven clients read and write to the PVFS model. The result of the I/O read performance in the experiment is presented in Figure 67. The error bars are set at 20%.

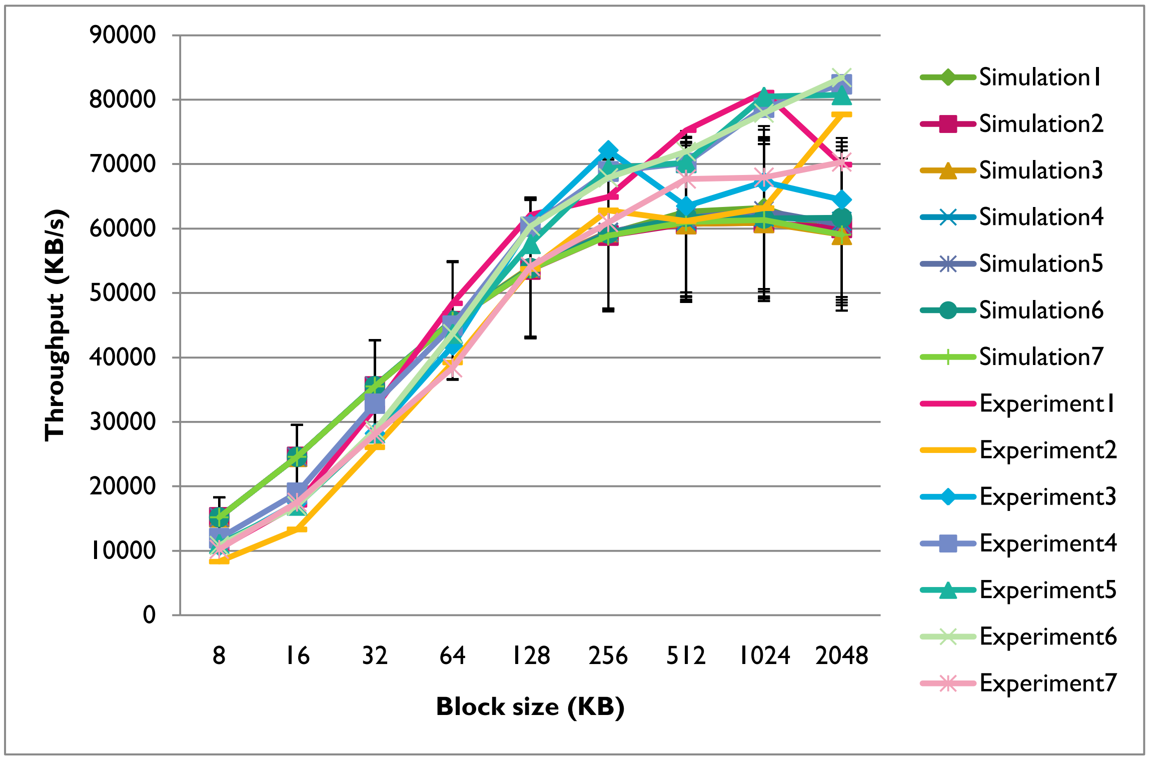


Figure 67: Seven clients I/O read validation

When seven clients are reading the PVFS file system at the same time, the workload has become high enough to visibly affect the file system performance behavior. Comparing to the previous experiment with six clients, this experiment shows much more variations and distortions. Simulation data points started to show outside of the 20% range, not only at the big block sizes, but also at the small block sizes.

The result of the I/O write performance in the experiment is presented in Figure 68. The error bars are set at 20%.

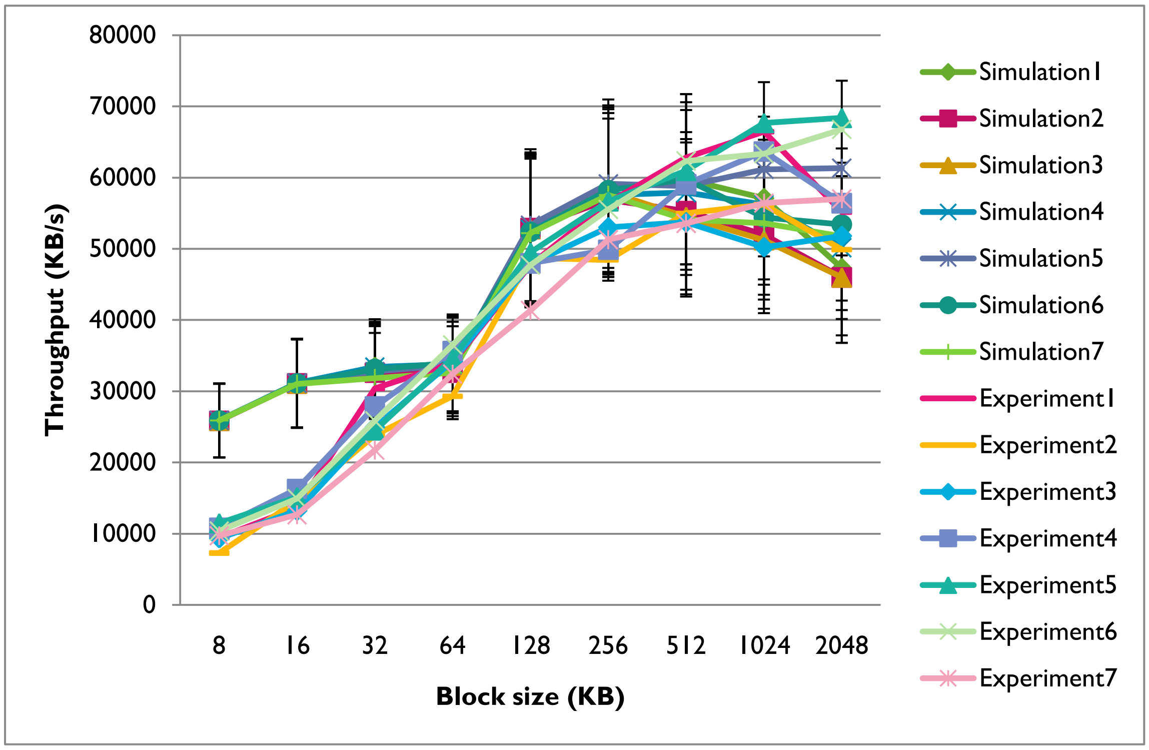


Figure 68: Seven clients I/O write validation

The variations and distortions are becoming even more visible in this experiment. However, similarly to previous experiments, the block sizes in the middle are the most stable. Data points of the middle block sizes are all stay within 20% of the real-world data points. Errors and distortions are happening at the small block sizes and large block sizes. At small block sizes, data points stay very close to each other. This allows the errors to be observed easily. At large block sizes, data points are more dispersed with large variations. It is harder to observe the error at the large block sizes.

7.3.8 Eight clients performance experiment

In this experiment, eight clients read and write to the PVFS model. The result of the I/O read performance in the experiment is presented in Figure 69. The error bars are set at 20%.

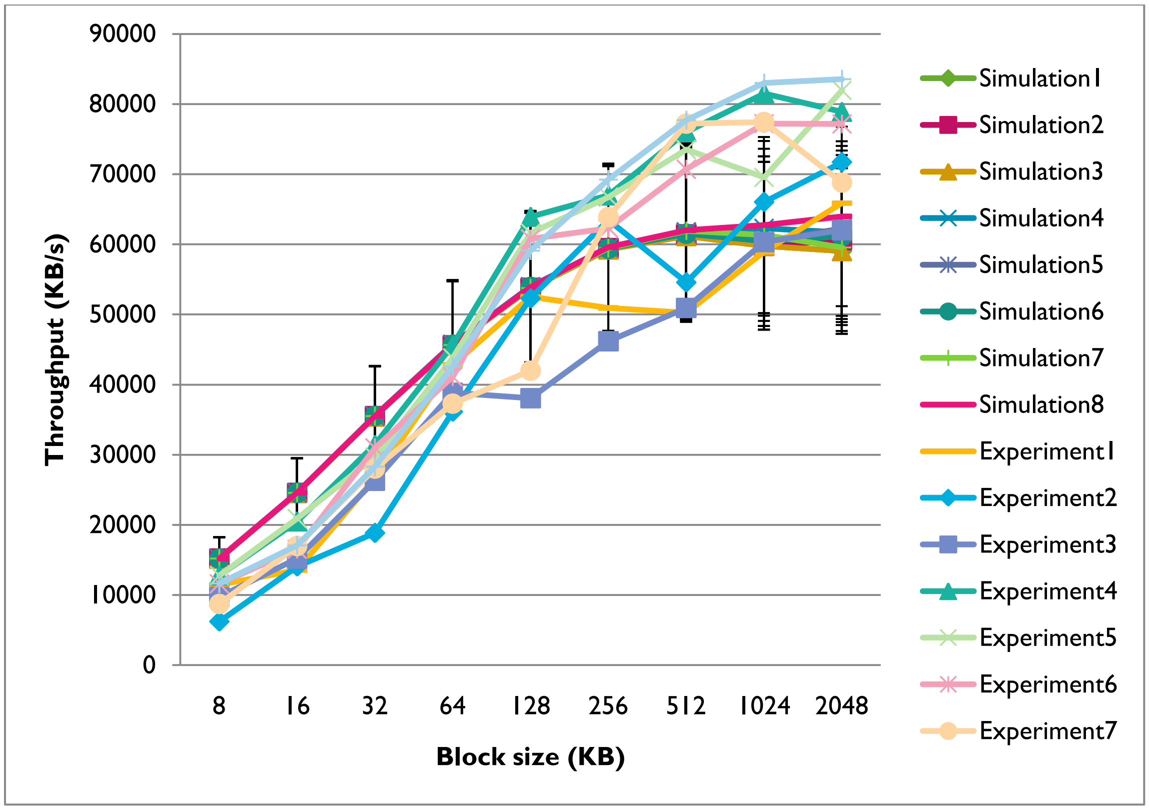


Figure 69: Eight clients I/O read validation

When the number of clients simultaneously reading the PVFS file system reaches 8 clients, we expect the stress level of the file system to be very high, and the experiment supports that expectation. At this level of stress, even the middle block sizes data points, which have stayed very stable until now, start to show variations and distortions. Many data points have now fallen well outside of the 20% error range. The biggest changes are at the big block sizes. As the number of client increases, the errors at the big block sizes also increase, especially at the largest block size.

As stated in the previous experiment, simulation data points are showing much less variations and distortions. This makes perfect sense, as the simulation model has much fewer outside factors. Simulation experiments are also performed under well-controlled and precise conditions. The result of the I/O write performance in the experiment is presented in Figure 70. The error bars are set at 20%.

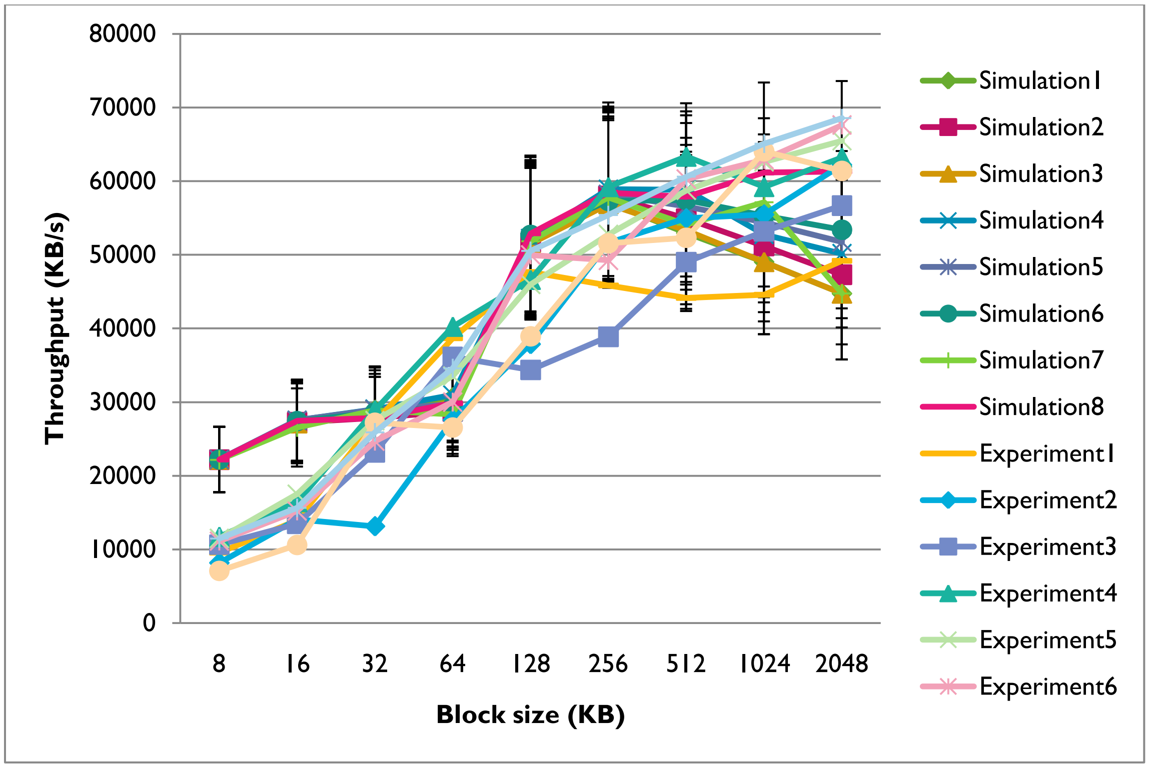


Figure 70: Eight clients I/O write validation

Even at eight clients writing to the PVFS file system at the same time, with the only exception at the 64Kbytes block size, the simulation performance behavior is still quite consistent with what was observed previously. In this experiment, many data points fall outside of the 20% error range; however, simulation data points still group together very well, especially at the small block sizes. Even though there are variations among simulation data points, the magnitude of errors at the small block size have stayed relatively the same since the beginning. The magnitude of errors at the large block sizes, however, increases when the number of clients simultaneously writing to the PVFS file system increases.

7.4 Summary

This chapter presents a set of detailed performance validation experiments of the simulation model of the PVFS file system. The workload for the parallel file system, as observed in Chapter 3, primarily consists of very-large-block-size sequential I/O. Therefore, the performance validation utilizes synthetic sequential I/O workload to study the simulation model and to compare with real-world data. Performance validations are set up with eight separate experiments. Each experiment uses a different number of clients accessing the PVFS file system. The number of clients is increased from one to eight. The last experiment uses eight clients, which is double the number of I/O servers, simultaneously accessing the PVFS file system. By increasing the number of clients from small to large, we observe the behavior of the simulation model when the stress level of the file system increases.

For the single client experiment, the simulation performances are within 20% of the real file system in most cases. When the number of clients increases, we observe the performance curves start to change, as the stress level of the file system increases. Up to three clients accessing the PVFS file system at the same time, the performance curves stay very close together. When the numbers of clients become equal to or larger than four clients, the variations and distortions become visible. The simulation data points group together much better than the real-world data points because the affecting factors are much less in the simulation environment. The magnitude of errors stays relatively the same at small block sizes. The errors become larger at the large block sizes when the number of client increases.

In general, the performance behavior is consistent throughout the performance validation process. The performance validation results are also very good, considering that this is a very complex environment, involving a parallel file system and multiple clients accessing simultaneously.

Chapter 8

CONCLUSION

We conclude this dissertation by summarizing the importance of file system simulation models, presenting some of the implications of this research, discussing what will be required for file system simulation models to achieve user acceptance in computer systems analysis, and identifying several promising avenues for continuing work.

8.1 The importance of the file system simulation models

Existing file system evaluation techniques have limitations and disadvantages in evaluating the role and performance of hypothetical file systems within complex computer environments. This dissertation describes the simulation models of the local and parallel file system and its role in providing alternative evaluation techniques in addition to existing ones. The file system simulation model enables end-to-end performance experiments of complex file systems, using different workloads which include real-system production workloads. This technique will provide an opportunity to analyze the interaction of different system components as well as different performance behavior introduced by the operating system.

8.2 Implications of this research

The file system simulation models offer the opportunity to investigate the performance behavior of different file systems in different type of storages in computer systems. It permits forays into the space of hypothetical file system functionalities without the difficulties of developing and supporting a prototype system or a proof of concept study. It also helps in eliminating the cost of purchasing and deploying actual hardware to build the actual system. This is especially relevant when considering the number of technologies available today and the recent trend toward the development of application-specific storage systems. Examples of these systems include, but are not limited to, audio and video recording and playback systems, scientific data processing, business data factory processing, and database housing, where support for application-specific features in individual system often play a key role in the success of the products in the market.

8.3 Keys to the acceptance of the file system simulation models

The benefits of the file system simulation models as an evaluation technique will not come without investments toward the development and maintenance of the simulation components. These investments include those of developing accurate and computationally inexpensive simulation models for storage devices and other components of the file systems. It also includes extending and creating a broader set of evaluation workloads that are more representative of the systems to be deployed or the existing systems which need to be analyzed.

For the file system simulation models to remain effective, new storage device models, new network models, and new operating system models need to continue to be created. Simulation experiments require validated or high-confidence component models in order to provide useful experimental results. This is not likely to be a problem, since the current simulation models are built with expansion and improvement in mind. Simulation components are designed to be as modular as possible, providing the flexibility and freedom to improve or replace. Depending on the type of component, in addition to the component architecture, the operating characteristics and performance parameters of the component also need to be captured. They include, but are not limited to, memory access time, instructions execution time, device seek time, and device access time. A physical device's attributes and characteristics can be obtained from the technical data of the device released by the manufacturer. Operating system component parameters can be gathered by profiling and monitoring tools as well as kernel traces. Looking to the future, Section 8.4 discusses possible advancements in the file system simulation models through improvements in existing components and explores new component implementation options.

Additionally, application-level workloads will need to be carefully developed in order to gain the full usefulness of the file system simulation models. Availability of such workloads could potentially lead to better characterization of real-system workloads and better benchmarks for storage systems. Even though well-accepted workloads exist, they are proprietary and belong to a few organizations. The lack of diverse and representative workloads for storage evaluation has been and continues to be a problem in the storage systems community [97, 98].

8.4 Opportunities for future work

In this section we discuss groups of improvements and developments for the simulation models centered on the themes of existing component improvement and new component implementation.

8.4.1 Improving existing simulation components

As demonstrated by the evaluations in this dissertation, the simulation models could produce very similar performance results to the real-world measurements. However, many components within the simulation models could still be improved to create even better result. An important component whose improvement benefits the simulation models greatly is the read-ahead mechanism. Usually, regular files are stored on disk in large groups of adjacent sectors, so that they can be retrieved quickly with few moves of the disk heads. Therefore, many disk accesses are sequential. Accordingly, read-ahead consists of reading several adjacent pages of data of a regular file or block device file before they are actually requested. In most cases, read-ahead significantly improves I/O read operation performance. Consequently, it improves system performance. An application, when sequentially reading a file, does not have to wait for the requested data because they are already available in memory. However, when the application accesses files randomly, read-ahead does not help improving performance. In the case of random I/O, it is actually detrimental because it not only wastes space in the page cache with useless information, but also spends more time to read them into memory. Therefore, the read-ahead component needs to reduce or stop read-ahead when it detects that the most recently I/O access is not sequential to the previous one. The current model component could be switched from sequential I/O access to random I/O access. However, it does not have all needed features currently implemented. The improved read-ahead component needs to implement the following features:

- Read-ahead may be gradually increased as long as the process keeps accessing the file sequentially.
- Read-ahead must be scaled down or even disabled when the current access is not sequential with respect to the previous one (random access).
- Read-ahead should be stopped when a process keeps accessing the same pages over and over again (only a small portion of the file is being used), or when almost all pages of the file are already in the page cache.

Another important simulation component to improve is the memory reclaiming mechanism. This mechanism is currently implemented partially in the page cache component. A more complete implementation of the memory reclaiming mechanism could help the model more accurately present the state of the I/O memory buffer.

Unfortunately, due to the empirical nature of the memory reclaiming design in Linux, its code changes very quickly. However, the general ideas and most major heuristic rules should continue to be valid. The design ideals of the memory reclaiming mechanism are:

- Pages in disk and memory caches not referenced by any process have priority. These pages are considered “harmless.” They should be reclaimed before pages belonging to processes in the user spaces. Also, non-dirty pages have higher priority than dirty pages because they do not have to be written to disk.
- Except locked pages, all pages of user space processes are reclaimable. The memory reclaiming process must be able to steal any page of a user space process, including anonymous pages. If a process has been sleeping for a long period of time, it will progressively lose all its page frames.

- If a page is shared by several processes, the memory reclaiming process clears all page table entries that refer to the page frame before reclaiming the page.
- The memory reclaiming process uses a Least Recently Used (LRU) replacement algorithm and two lists (active and inactive) to identify which pages to reclaim. If a page has not been accessed for a long time, the probability that it will be accessed in the near future is low, and it can be considered inactive page. On the other hand, if a page has been accessed recently, the probability that it will continue to be accessed is high, and it must be considered as active page. The reclaiming process will only reclaim inactive pages.

On the server component of the simulation models, the receiving buffer component also needs some improvements. Currently, the receiving buffer component is implemented, using a cost model with computational complexity of O(1) for inserting and searching incoming packets. In Linux, the implementations of the network receiving buffer models are usually a linked list with the computational complexity of O(n) for inserting and searching packets. This is the reason why the simulation models have slower performance than the real-world measurement when using small block size and faster performance than the real-world measurement when using big block size. Due to the flow nature of Petri Net, there are some difficulties in modifying the model from a constant cost model to a linear cost model. However, the change can reduce the errors of the simulation models when comparing to the real-world measurement.

8.4.2 Implementing new components

In addition to improving existing model components, implementing new model components is another direction to extend the capability of the simulation models. One interesting component that has not been implemented is the Linux I/O scheduler. The I/O scheduler controls the way I/O reads and writes are committed to disk. The goal of the I/O scheduler is to provide better optimization for different classes of workload by allowing the operating system to utilize many different scheduling mechanisms.

Each scheduling mechanism is designed to improve a certain aspect of the I/O operations. The techniques used by the scheduler to improve performance include, but are not limited to, merging request, elevator, and prioritization. Merging request is a technique where adjacent requests are merged together to reduce disk seeking. Elevator is a technique where requests are ordered, based on their physical location, and the requests are usually traversed in one direction from the closest location to the farthest or vice versa. Prioritization is a technique where the priorities of requests are manipulated to improve performance. There are currently four I/O schedulers available. They are the no-op scheduler, the anticipatory I/O scheduler (AS), the deadline scheduler and the complete fair queuing scheduler (CFQ).

The no-op scheduler is the simplest scheduling scheme. It only has the merging request technique implemented. All I/O requests are put into a simple first-in-first-out (FIFO) queue. Perhaps, the no-op scheduler works best with solid state devices that do not depend on mechanical movement to access data.

The anticipatory I/O scheduler is the former default scheduling scheme in the Linux kernel. It implements the merging request technique, the elevator technique and an anticipating read operation technique. Basically, it pauses for a short time (usually a few milliseconds) after a read operation in anticipation of another read request.

The deadline scheduler implements request merging and elevator queues. More importantly, it imposes a deadline on all operations to prevent resource starvation by maintaining two deadline queues, in addition to the elevator queues (both read and write). Deadline queues are basically sorted by their deadline, while the elevator queues are sorted by the sector number. The deadline scheduler decides which queue to use before processing any request. Read queues are given a higher priority, because processes usually block on read operations. After that, the deadline scheduler checks if the first request in the deadline queue has expired. If none of the requests in the deadline queue is close to expiration, the scheduler will process requests from the elevator queue.

The complete fair queuing (CFQ) scheduler also implements request merging and elevator queues. It additionally attempts to give all users of a particular device the same number of I/O requests over a particular time interval. CFQ categorizes incoming requests into synchronous type and asynchronous type. According to I/O priority of the requesting process, asynchronous requests are distributed into multiple priority queues, one queue per I/O priority. Each queue is assigned a time slice which depends on the I/O priority of the submitting process. The scheduler accesses these queues in a round-robin order. Synchronous requests are distributed into a number of per-process queues. The number of requests in a queue is also restricted, based on the I/O priority.

Obviously, depending on which scheduling scheme is in use, the I/O performance behavior of the system can have different characteristics. By implementing the I/O scheduler, the file system simulation models can accurately mimic the performance behavior of the actual file system and storage subsystems. The I/O scheduler is complex, but the current file system simulation models have many existing components that could be reused to make the implementation easier.

Another interesting component to implement is a simulation model for different network hardware. InfiniBand is a very good one with which to start, since there are PVFS modifications to operate successfully, using InfiniBand as the network hardware [99-101]. InfiniBand is a powerful network architecture, designed to support I/O connectivity for the Internet infrastructure. Uniquely providing both backplane solutions and also traditional networking interconnects, InfiniBand offers communication and management infrastructure for inter-processor communication and I/O. By unifying the network's interconnect with a feature-rich managed architecture, it manages to provide native cluster connectivity, thus simplifying application cluster connections, supporting scalability, and sustaining reliability. With QoS mechanisms built in, InfiniBand can provide virtual lanes on each link and define service levels for individual packets.

The current network hardware implemented in the simulation models is Ethernet, which uses a hierarchical switched topology. Unlike Ethernet, InfiniBand uses a switched fabric topology. Other commonly-used network topologies are Fat-Tree (Clos), mesh, and 3D-Torus. Any of the previously mentioned topologies, after implementation, would create a very different interconnection simulation component, in comparison to the current component. InfiniBand also transmits data in large packets (maximum size of 4 Kbytes). Packets are used to form messages, which could be as large as 2 Gbytes. There are multiple types of messages, such as direct memory access (RDMA), channel send or receive, transaction-based operation, multicast transmission, and atomic operation. Due to implementation complexity reasons, PVFS over InfiniBand implementations are using Internet Protocol (IP) over InfiniBand technology [102]. This is also a very good basis for the PVFS simulation model. Many network components and client components as well as server components can be reused.

Based on the same principle as PVFS, a much improved PVFS2 is also a very nice addition to the file system simulation models. A PVFS2 improvement that has a significant impact on the simulation models is how the file system interacts with networks and storages. PVFS1 relies on the socket networking interface and local file systems for data and metadata storage. PVFS2 uses the Buffered Messaging Interface (BMI) and the Trove storage interface to provide Application Programming Interfaces (APIs) to network and storage technologies respectively. PVFS2 can support several different network types, such as TCP/IP, Myricom's GM message passing system, and InfiniBand (both Mellanox VAPI and OpenIB APIs) via BMI. Supporting multiple networking technologies efficiently is a very important feature of PVFS2. As a result, implementing the BMI model is a key to successful implementation of the PVFS2 simulation model.

Similar to network technologies, many different storage technologies are also available. PVFS2 uses the Trove storage interface to efficiently support multiple storage back-end technologies. In addition to storing file data, metadata has also received much attention in

PVFS2. Instead of using a flat file on the local file system to store metadata as PVFS does, PVFS2 is using Berkeley DB database technologies for the metadata storage. In PVFS, there is only one metadata server. This creates a single point of failure, as well as a performance bottleneck. PVFS2 can distribute metadata to multiple I/O servers (which might or might not also serve data). This allows metadata for different files to be placed on different servers and reduces the congestion to the metadata servers.

BIBLIOGRAPHY

[1] I. Gorton, P. Greenfield, A. Szalay, and R. Williams, "Data-Intensive Computing in the 21st Century," *Computer*, vol. 41, pp. 30-32, 2008.

[2] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *International Journal of High Performance Computing Applications*, vol. 15, pp. 200-222, 2001.

[3] A. Hutflesz, H.-W. Sis, and P. Wildmayer, "Twin Grid Files: Space Optimizing Access Schemes," in *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, 1988, pp. 183-190.

[4] E. Morenoff and J. B. McLean, "Application of Level Changing to a Multilevel Storage Organization," *Communications of the ACM*, vol. 10, pp. 149-154, 1967.

[5] B. Randell and C. J. Kuehner, "Dynamic Storage Allocation Systems," *Communications of the ACM,* vol. 11, pp. 297-306, 1968.

[6] G. A. Gibson and R. V. Meter, "Network Attached Storage Architecture," *Communications of the ACM,* vol. 43, pp. 37-45, 2000.

[7] K. Jensen, *Coloured Petri nets (2nd ed.): basic concepts, analysis methods and practical use: volume 1*. London, UK: Springer-Verlag, 1996.

[8] L. M. Kristensen, S. Christensen, and K. Jensen, "The practitioner's guide to coloured Petri nets," *International Journal on Software Tools for Technology Transfer,* vol. 2, pp. 98-132, 1998.

[9] A. V. Ratzer, *et al.*, "CPN tools for editing, simulating, and analysing coloured Petri nets," in *Proceedings of the 24th international conference on Applications and theory of Petri nets,* Eindhoven, The Netherlands, 2003, pp. 450-462.

[10] M. K. Johnson. (2001). *Whitepaper: Red Hat's New Journaling File System: ext3*. Available: http://www.redhat.com/support/wpapers/redhat/ext3/

[11] J. S. Bucy and G. R. Ganger, *The DiskSim simulation environment version 3.0 reference manual*. Pittsburgh, Pa.: School of Computer Science, Carnegie Mellon University, 2003.

[12] J. L. Griffin, J. Schindler, S. W. Schlosser, J. C. Bucy, and G. R. Ganger, "Timing-accurate Storage Emulation," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Monterey, CA, 2002, p. 6.

[13] J. L. Griffin and G. R. Ganger, "Timing-accurate storage emulation : evaluating hypothetical storage components in real computer systems," Thesis (Ph D), Carnegie Mellon University, Carnegie Mellon University, 2004., Pittsburgh, PA, 2004.

[14] K. E. Maghraoui, G. Kandiraju, J. Jann, and P. Pattnaik, "Modeling and simulating flash based solid-state disks for operating systems," in *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, San Jose, California, USA, 2010, pp. 15-26.

[15] Y. Wang and D. Kaeli, "Execution-Driven Simulation of Network Storage Systems," in *Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, 2004, pp. 604-611.

[16] D. S. Batory, "Modeling the Storage Architectures of Commercial Database Systems," *ACM Transactions on Database Systems (TODS),* vol. 10, pp. 463-528, 1985.

[17] H. Gomaa, "A Simulation Based Model Of A Virtual Storage System," in *Proceedings of the 12th annual symposium on Simulation*, 1979, pp. 273-303.

[18] J. E.G. Coffman and M. I. Reiman, "Diffusion approximations for storage processes in computer systems," in *Proceedings of the 1983 ACM SIGMETRICS conference on Measurement and modeling of computer systems,* 1983, pp. 93-117.

[19] P. Jacobson and E. Lazowska, "Analyzing queueing networks with simultaneous resource possession," *Communications of the ACM*, vol. 25, pp. 142-151, 1982.

[20] A. Kraiss and G. Weikum, "Integrated document caching and pre-fetching in storage hierarchies based on Markov-chain predictions," *The VLDB Journal — The International Journal on Very Large Data Bases*, vol. 7, pp. 141-162, 1998.

[21] D. Menasce, O. Pentakalos, and Y. Yesha, "An analytic model of hierarchical mass storage systems with network-attached storage devices," in *Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 1996, pp. 180-189.

[22] L. Zhaobin and L. Haitao, "Modeling and Performance Evaluation of Hybrid Storage I/O in Data Grid," in *Network and Parallel Computing Workshops, 2007. NPC Workshops. IFIP International Conference on*, 2007, pp. 624-629.

[23] X. Molero, F. Silla, V. Santonja, and J. Duato, "Modeling and simulation of storage area networks," in *Modeling, Analysis and Simulation of Computer and*

Telecommunication Systems, 2000. Proceedings. 8th International Symposium on, 2000, pp. 307-314.

[24] R. Routray, S. Gopisetty, P. Galgali, A. Modi, and S. Nadgowda, "iSAN: Storage Area Network Management Modeling Simulation," in *Networking, Architecture, and Storage, 2007. NAS 2007. International Conference on*, 2007, pp. 199-208.

[25] J. Staley, S. Muknahallipatna, and H. Johnson, "Fibre Channel based Storage Area Network Modeling using OPNET for Large Fabric Simulations: Preliminary Work," in *Local Computer Networks, 2007. LCN 2007. 32nd IEEE Conference on*, 2007, pp. 234-236.

[26] N. Aizikowitz, A. Glikson, A. Landau, B. Mendelson, and T. Sandbank, "Component-based performance modeling of a storage area network," in *Proceedings of the 37th conference on Winter simulation*, Orlando, Florida, 2005, pp. 2417-2426.

[27] H. Hung-Chang and K. Chung-Ta, "Modeling and evaluating peer-to-peer storage architectures," in *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, 2002, pp. 24-29.

[28] P. DeRosa, K. Shen, C. Stewart, and J. Pearson, "Realism and simplicity: disk simulation for instructional OS performance evaluation," in *Proceedings of the 37th SIGCSE technical symposium on Computer science education*, Houston, Texas, USA, 2006, pp. 308-312.

[29] A. Ali and R. d. Souza, "Modeling and simulation of hard disk dive final assembly using a HDD template," in *Proceedings of the 39th conference on Winter simulation: 40 years! The best is yet to come*, Washington D.C., 2007, pp. 1641-1650.

[30] D. Lugones, *et al.*, "High-speed network modeling for full system simulation," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, 2009, pp. 24-33.

[31] N. Agarwal, L.-S. Peh, and N. Jha, "Garnet: A Detailed Interconnection Network Model inside a Full-system Simulation Framework," Princeton University CE-P08-001, 2008.

[32] E. Argollo, *et al.*, "COTSon: infrastructure for full system simulation," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 52-61, 2009.

[33] J. Lee, *et al.*, "Modeling communication networks with hybrid systems," *IEEE/ACM Trans. Netw.*, vol. 15, pp. 630-643, 2007.

[34] S. Bohacek, *et al.*, "A hybrid systems modeling framework for fast and accurate simulation of data communication networks," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, pp. 58-69, 2003.

[35] A. Kavimandan, W. Lee, M. Thottan, A. Gokhale, and R. Viswanathan, "Network simulation via hybrid system modeling: a time-stepped approach," in *Computer Communications and Networks, 2005. ICCCN 2005. Proceedings. 14th International Conference on*, 2005, pp. 531-536.

[36] J. Liu, "Packet-level integration of fluid TCP models in real-time network simulation," in *Proceedings of the 38th conference on Winter simulation*, Monterey, California, 2006, pp. 2162-2169.

[37] J. Liu, "Parallel Simulation of Hybrid Network Traffic Models," in *Proceedings of the 21st International Workshop on Principles of Advanced and Distributed Simulation*, 2007, pp. 141-151.

[38] T. Verdickt, B. Dhoedt, F. D. Turck, and P. Demeester, "Hybrid performance modeling approach for network intensive distributed software," in *Proceedings of the 6th international workshop on Software and performance*, Buenes Aires, Argentina, 2007, pp. 189-200.

[39] M. Yu and M. Zhou, "A performance modeling scheme for multistage switch networks with phase-type and bursty traffic," *IEEE/ACM Trans. Netw.*, vol. 18, pp. 1091-1104, 2010.

[40] D. F. Kassa and A. E. Krzesinski, "A queueing network model of TCP performance," in *Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, White River, South Africa, 2005, pp. 56-65.

[41] T. Katakami, T. Tabata, and H. Taniguchi, "I/O Buffer Cache Mechanism Based on the Frequency of File Usage," in *Convergence and Hybrid Information Technology, 2008. ICCIT '08. Third International Conference on*, 2008, pp. 76-82.

[42] Y. Zhou, Z. Chen, and K. Li, "Second-level buffer cache management," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 15, pp. 505-519, 2004.

[43] W. Shenggang, C. Qiang, H. Xubin, X. Changsheng, and W. Chentao, "An Adaptive Cache Management Using Dual LRU Stacks to Improve Buffer Cache Performance," in *Performance, Computing and Communications Conference, 2008. IPCCC 2008. IEEE International*, 2008, pp. 43-50.

[44] X. Ding, S. Jiang, and F. Chen, "A buffer cache management scheme exploiting both temporal and spatial localities," *Trans. Storage*, vol. 3, p. 5, 2007.

[45] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang, "DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality," in *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, San Francisco, CA, 2005, pp. 8-8.

[46] A. R. Butt, C. Gniady, and Y. C. Hu, "The Performance Impact of Kernel Pre-fetching on Buffer Cache Replacement Algorithms," *IEEE Trans. Comput.*, vol. 56, pp. 889-908, 2007.

[47] O. Ozturk, S. W. Son, M. Kandemir, and M. Karakoy, "Pre-fetch throttling and data pinning for improving performance of shared caches," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Austin, Texas, 2008, pp. 1-12.

[48] S. W. Son, *et al.*, "Profiler and compiler assisted adaptive I/O pre-fetching for shared storage caches," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, Toronto, Ontario, Canada, 2008, pp. 112-121.

[49] S. Subha, "An Algorithm for Buffer Cache Management," in *Information Technology: New Generations, 2009. ITNG '09. Sixth International Conference on*, 2009, pp. 889-893.

[50] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, Anaheim, CA, 2005, pp. 8-8.

[51] P. Vijayan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Model-based failure analysis of journaling file systems," in *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, 2005, pp. 802-811.

[52] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 29-43.

[53] M. Karlsson, C. Karamanolis, and X. Zhu, "Triage: Performance Differentiation for Storage Systems Using Adaptive Control," *ACM Transactions on Storage (TOS)*, vol. 1, pp. 457-480, 2005.

[54] I. K. Georgiev and I. I. Georgiev, "An Information-Interconnectivity-Based Retrieval Method for Network Attached Storage," in *Proceedings of the 1st conference on Computing frontiers*, 2004, pp. 268-275.

[55] M. Andrews, M. A. Bender, and L. Zhang, "New algorithms for the disk scheduling problem," in *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, 1996, p. 550.

[56] D. T. Altilar and Y. Paker, "Optimal Scheduling Algorithms for Communication Constrained Parallel Processing," in *Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, 2002, pp. 197-206.

[57] B. Hillyer, R. Rastogi, and A. Silverschatz, "Scheduling and Data Replication to Improve Tape Jukebox Performance," in *15th International Conference on Data Engineering (ICDE'99)*, 1999, p. 532.

[58] S. Prabhakar, D. Agrawal, and A. E. Abbadi, "Optimal Scheduling Algorithms for Tertiary Storage," *Distributed and Parallel Databases*, vol. 14, pp. 255-282, 2003.

[59] C. Moon and H. Kang, "Heuristic Algorithms for I/0 Scheduling for Efficient Retrieval of Large Objects from Tertiary Storage," in *Proceedings of the 12th Australasian conference on Database technologies*, 2001, pp. 145-152.

[60] S. Prabhakar, D. Agrawal, A. E. Abbadi, and A. Singh, "Tertiary Storage: Current Status and Future Trends," Dept. of Computer Science, Univ. of Calilfornia, Santa Barbara1996.

[61] S. Prabhakar, D. Agrawal, A. E. Abbadi, and A. Singh, "A brief survey of tertiary storage systems and research," in *Proceedings of the 1997 ACM symposium on Applied computing*, 1997, pp. 155-157.

[62] B. Hillyer and A. Silberschatz, "Random I/O scheduling in online tertiary storage systems," in *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, 1996, pp. 195-204.

[63] T. Johnson and E. Miller, "Performance Measurements of Tertiary Storage Devices," in *Proceedings of the 24rd International Conference on Very Large Data Bases*, 1998, pp. 50-61.

[64] B. Liu, J. Li, L. Nie, and Y. Zhang, "Non-blocking Disk-Tape Join Algorithm for Data on Tertiary Storage Systems," in *Proceedings of the The Fifth International Conference on Computer and Information Technology*, 2005, pp. 58-64.

[65] S. Christodoulakis, P. Triantafillou, and F. Zioga, "Principles of Optimally Placing Data in Tertiary Storage Libraries," in *Proceedings of the 23rd International Conference on Very Large Data Bases*, 1997, pp. 236-245.

[66] A. Vakali and E. Terzi, "Multimedia Data Storage and Representation Issues on Tertiary Storage Subsystems" An Overview," *ACM SIGOPS Operating Systems Review*, vol. 35, pp. 61-77, 2001.

[67] J. No, R. Thakur, and A. Choudhary, "Integrating parallel file I/O and database support for high-performance scientific data management," in *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, 2000, p. 57.

[68] S. Berson, S. Ghandeharizadeh, R. Muntz, and X. Ju, "Staggered striping in multimedia information systems," in *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, 1994, pp. 79-90.

[69] P. Triantafillou and T. Papadakis, "Continuous Data Block Placement in and Elevation from Tertiary Storage in Hierarchical Storage Servers," *Cluster Computing,* vol. 4, pp. 157-172, 2001.

[70] J. Wilkes, R. Gelding, C. Staelin, and T. Sullivan, "The HP AutoRAID hierarchical storage system," *ACM Transactions on Computer Systems (TOCS),* vol. 24, pp. 108-136, 1996.

[71] K. Holtman, P. v. d. Stok, and I. Willers, "A cache filtering optimisation for queries to massive datasets on tertiary storage," in *Proceedings of the 2nd ACM international workshop on Data warehousing and OLAP*, 1999, pp. 94-100.

[72] E. Otoo, F. Olken, and A. Shoshani, "Disk cache replacement algorithm for storage resource managers in data grids," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing,* 2002, pp. 1-15.

[73] A. Shoshani, A. Sim, L. M. Bernardo, D. Rotem, and H. Nordberg, "Multidimensional Indexing and Query Coordination for Tertiary Storage Management," in *Proceedings of the 11th International Conference on Scientific on Scientific and Statistical Database Management*, 1999, p. 214.

[74] A. Shoshani, A. Sim, L. M. Bernardo, and H. Nordberg, "Coordinating Simultaneous Caching of File Bundles from Tertiary Storage," in *Proceedings of the 12th International Conference on Scientific and Statistical Database Management (SSDBM'00)*, 2000, p. 196.

[75] G. A. Alvarez*, et al.*, "MINERVA: An Automated Resource Provisioning Tool for Large-Scale Storage Systems," *ACM Transactions on Computer Systems (TOCS),* pp. 483-518, 2001.

[76] T. Kagimasa, K. Takahashi, and T. Mori, "Adaptive Storage Management for Very Large Virtual/Real Storage Systems," in *Proceedings of the 18th annual international symposium on Computer architecture*, 1991, pp. 372-379.

[77] H. Tang and T. Yang, "An Efficient Data Location Protocol for Self-organizing Storage Clusters," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003, p. 53.

[78] C. Wu and R. Burns, "Tunable Randomization for Load Management in Shared-Disk Clusters," *ACM Transactions on Storage (TOS)*, pp. 108-131, 2005.

[79] H. Tang, *et al.*, "A Self-Organizing Storage Cluster for Parallel Data-Intensive Applications," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, 2004, p. 52.

[80] S. S. Vazhkudai, *et al.*, "FreeLoader: Scavenging Desktop Storage Resources for Scientific Data," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, 2005, p. 56.

[81] A. R. Butt, T. A. Johnson, Y. Zheng, and Y. C. Hu, "Kosha: A Peer-to-Peer Enhancement for the Network File System," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, 2004, p. 51.

[82] D. Colarelli and D. Grunwald, "Massive Arrays of Idle Disks For Storage Archives," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, 2002, pp. 1-11.

[83] K. Hiraki, *et al.*, "Data Reservoir: Utilization of Multi-Gigabit Backbone Network for Data-Intensive Research," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, 2002, pp. 1-9.

[84] E. K. Lee and C. A. Thekkath, "Petal: Distributed Virtual Disks," in *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, 1996, pp. 84-92.

[85] W. H. Min, *et al.*, "Dynamic Storage Resource Management Framework for the Grid," in *Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'05)*, 2005, pp. 286-293.

[86] Y. Feng, Y.-y. Zhang, and R.-y. Jia, "EPYFQ: A Novel Scheduling Algorithm for Performance Virtualization in Shared Storage Environment," in *Proceedings of the 5th international workshop on Software and performance*, 2005, pp. 263-264.

[87] A. Gulati and P. Varman, "Lexicographic QoS Scheduling for Parallel I/O," in *Proceedings of the 17th annual ACM symposium on Parallelism in algorithms and architectures*, 2005, pp. 29-38.

[88] L. Huang, G. Peng, and T. Chiueh, "MultiDimensional Storage Virtualization," in *Proceedings of the joint international conference on Measurement and modeling of computer systems*, 2004, pp. 14-24.

[89] C. R. Lumb, A. Merchant, and G. A. Alvarez, "Facade: virtual storage devices with performance guarantees," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 2003, pp. 131-144.

[90] D. A. Ford and J. Myllymaki, "A Log-Structured Organization for Tertiary Storage," in *Proceedings of the Twelfth International Conference on Data Engineering*, 1996, pp. 20-27.

[91] M. Zhao, J. Zhang, and R. J. Figueiredo, "Distributed File System Virtualization Techniques Supporting On-Demand Virtual Machine Environments for Grid Computing," *Cluster Computing*, vol. 9, pp. 45-56, 2006.

[92] W. D. Norcott and D. Capps. (2011). *IOzone Filesystem Benchmark*. Available: www.iozone.org

[93] N. Murray and N. Horman. (2004). *Understanding virtual memory*. Available: http://www.redhat.com/magazine/001nov04/features/vm/

[94] J. Pommnitz. (2010). *Kernel level exception handing in linux 2.1.8*. Available: http://www.mjmwired.net/kernel/Documentation/exception.txt

[95] J. Levon. (2009). *Oprofile - a system profiler for linux*. Available: http://oprofile.sourceforge.net/

[96] B. Lu, *et al.*, "A case study on grid performance modeling," in *The 18th IASTED International Conference on Parallel And Distributed Computing And Systems (PDCS 2006)*, Dallas, Texas, USA, 2006.

[97] A. Traeger, E. Zadok, N. Joukov, and C. P. Wright, "A nine year study of file system and storage benchmarking," *Trans. Storage*, vol. 4, pp. 1-56, 2008.

[98] N. Joukov, T. Wong, and E. Zadok, "Accurate and efficient replaying of file system traces," in *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, San Francisco, CA, 2005, pp. 25-25.

[99] J. Wu, P. Wyckoff, and P. Dhabaleswar, "PVFS over InfiniBand: design and performance evaluation," in *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, 2003, pp. 125-132.

[100] W. Jiseheng, P. Wyckoff, and D. Panda, "Supporting efficient noncontiguous access in PVFS over Infiniband," in *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, 2003, pp. 344-351.

[101] W. Jiesheng, P. Wyckoff, D. Panda, and R. Ross, "Unifier: unifying cache management and communication buffer management for PVFS over InfiniBand,"

in *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on*, 2004, pp. 523-530.

[102] R. E. Grant, P. Balaji, and A. Afsahi, "A study of hardware assisted IP over InfiniBand and its impact on enterprise data center performance," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, 2010, pp. 144-153.