Hierarchical Performance Measurement and Modeling of the Linux File System

Hai Nguyen
University of Arkansas
Fayetteville, Arkansas, USA
1-501-342-2932
hqn01@uark.edu

Amy Apon
University of Arkansas
Fayetteville, Arkansas, USA
1-479-575-6794
aapon@uark.edu

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 when 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, affecting several million dollars of investment. This paper presents a detailed simulation-based performance model of the Linux ext3 file system. The model is developed using Colored Petri Nets. A performance study using the model 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.

Categories and Subject Descriptors
C.4 [**Computer Systems Organization**]: Performance of Systems – *Modeling techniques.*

D.2.2 [**Software Engineering**]: Design Tools and Techniques – *Petri nets.*

D.4.3 [**Operating Systems**]: File System Managements.

General Terms
Design, Experimentation, Measurement, Performance.

Keywords
Colored Petri Net, file system modeling, file system simulation, Petri Net, Linux file system, L2 cache model.

1. INTRODUCTION AND MOTIVATION
Today's scientific data intensive research applications place very high demands on storage systems in both performance and capacity [3]. 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. A consequence is that personnel costs for storage management for, say, tuning performance, now dominate capital costs over the equipment's useful lifetime. With the introduction of cloud computing and the concept of resource on demand, the management of storage performance and capacity faces an even bigger challenge. This paper describes a proactive solution to performance research and capacity planning for data intensive computing environments.

Among several storage architectures, three most common ones, Direct-Attached Storage (DAS), Network-Attached Storage (NAS), and Storage Area Networks (SANs), prove to be able to provide a shared, adaptable, and high-performance storage system for data intensive applications. The performance of each 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 a network storage architecture.

The ext3 file system is chosen to be the candidate for this study. Ext3 is a standard file system on every Linux distribution. It was released and officially supported by Red Hat since 2001 [7]. 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.

In this research a simulation model based on the well known Petri Nets formalism is used to simulate and evaluate complex data services in a repeatable and controlled environment. This formalism offers a flexible framework that is well adapted for the analysis of I/O flow performance. CPNTools [15] is utilized for simulation and analysis. Section 2 discusses related work in the storage simulation research area. Section 3 examines the structure and code flow of the Linux file system in general and the ext3 file system in particular, and is central to the implementation of the performance model. Section 4 presents an I/O performance study of the ext3 file system. Multiple design decisions and assumptions are described by the performance study. Section 5 discusses the implementation of the simulation model using Colored Petri Nets. Section 6 presents the model performance validation against the performance of real file systems. Section 7 concludes the paper.

ICPE'11, March 14–16, 2011, Karlsruhe, Germany.

2. RELATED WORK

J.S. Bucy and G.R. Ganger describe in [1] a disk simulation system called DiskSim that has been made publicly available to the research community. 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.

Previous research in performance modeling of storage and file systems includes work using timing-accurate storage emulation [5], which is a technique in which the simulator appears to the system to be a real storage component with service times similar to the component it is simulating. In [5], the authors describe a prototype called the Memulator. 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 MEMS-based storage device and a modern Linux system equipped with a disk whose firmware had been modified.

J.L. Griffin 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 [4]. He demonstrates the incorporation of intrusion detection capabilities into processing-enhanced disk drives.

Maghraoui et al. in [2] presents a method of modeling a Flash device and build a Flash simulator. The authors capitalize on the throughput behavior of the Flash disk with none 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 SSDs without having to simulate every minor detail and internal organization of a Flash device.

Wang and Kaeli in [16] present ParIOSim, a validated execution-driven simulator for network storage systems. Their simulator provides a flexible simulation environment for performing storage optimizations and 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.

3. EXAMINATION OF THE LINUX FILE SYSTEM

In the area of file systems, Linux offers a very flexible environment that supports a large number of file systems, including journaling, clustering, and cryptographic file systems. As an open source operating system, Linux provides a system analyst the ability to read the code and to determine exactly what is happening for a particular system request, helping the process of designing a simulation model significantly.

3.1 Overview of the Linux file system

The architecture for file systems in Linux, shown in Figure 1, is designed as an abstract layer that supports a large variety of file systems over a large variety of storage devices. When using the abstract layer function calls, the application is completely unaware of the true file system types or the storage medium. In this clean and well designed layer system, an upper component hides the details of the lower components and presents a more unified interface and simpler information to the layer above it. This design helps both Linux and the simulation model be more flexible and provides for the support of multiple types of storage devices. This feature is used to develop a realistic simulation without having to model lower level details of the hardware layer or particular storage devices. In other words, from the perspective of the file system, a direct attached disk drive is treated the same as a SAN storage array. This architecture also leads to very interesting performance characteristics of Linux file systems.

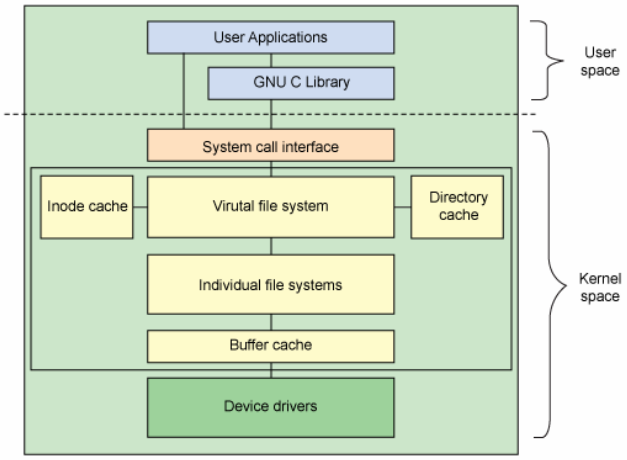

Figure 1: Architectural view of Linux file system [8]

3.2 The flow of ext3 I/O operations

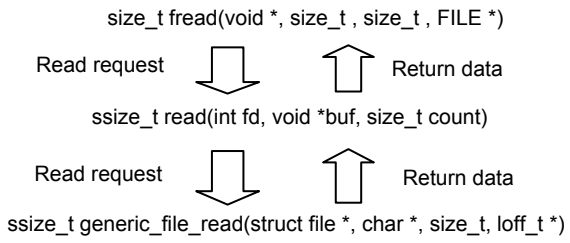

Figure 2: fread I/O flow

There are many ways to read and write data from and to secondary storage. Typically, application developers use the C library functions fread and fwrite to perform I/O operations. These two functions utilize the read system call and write system call to implement buffered I/O operations. The fread function is illustrated in Figure 2. These functions allow an application to perform I/O operations in blocks of data with configurable block size, which improves I/O performance tremendously.

The two system calls read and write are implemented so that file system developers can use their own code if they want to. However,

ext2, ext3 and even ext4 file system [13] use the Linux default read/write functions. These are defined as generic_file_read and generic_file_write. It is clear that in order to model ext3 I/O performance, it is vital to be able to simulate generic_file_read and generic_file write functions.

The detailed implementation of generic_file_read and generic_file_write can be found in the filemap.c file located under the mm subdirectory of the Linux kernel source code.

4. PERFORMANCE MEASUREMENT 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.

4.1 Experimental setup

Performance measurement experiments were executed on production computers (Dell PowerEdge 1750) at Acxiom Corporation with the hardware configurations shown in Table 1. The test computers are setup to have a single drive with no RAID, RAID 0 with 2 single drives, 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 affected performance study. The primary I/O testing suite used in the following experiments is iozone.

Table 1. Test hardware 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 Clarrion CX700
HBA card	Qlogic 2340

4.2 I/O Performance study with different file size and block size

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 1Gbytes).

The results for the sequential I/O read measurement experiments are presented in Figure 3. For space reasons, only a section of the experiment results are displayed.

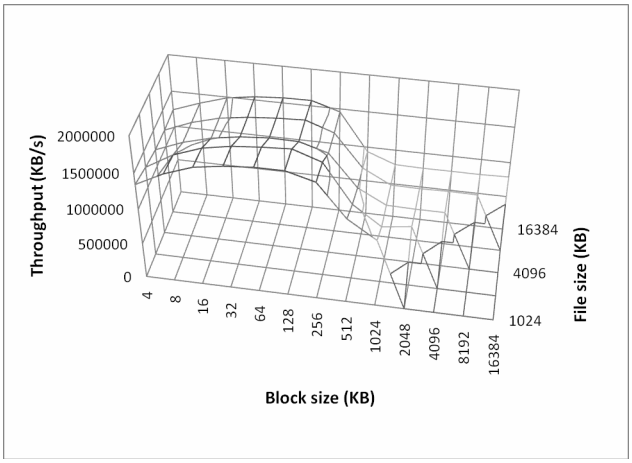

Figure 3: Sequential I/O read performance

Two observations can be made from the measurements in Figure 3. 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 performance drop are described in section 4.3.

Similar performance behavior can also be observed for I/O write operations. The write performance using a similar set of files and the block sizes varies the same way as with read performance. The results of I/O write measurements are not shown for space reasons but are very similar to the read results.

Read and write measurement 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 hold 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 [12].

Figure 4 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.

Sequential write shares the same characteristic. However, under Linux, a threshold (dirty ratio) is usually in place to synchronously flushing data to disk. This threshold is configurable via 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 5 the dirty threshold is set to the default value put forth by Red Hat (~512MB for the test machine). This shows the dirty ratio threshold affects file write performance.

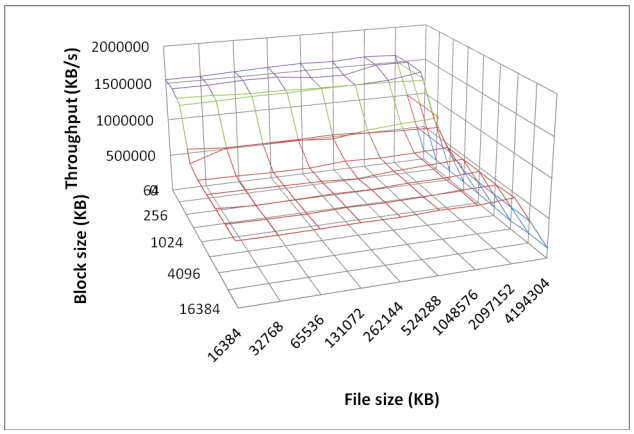

Figure 4: Sequential read file size and performance

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.

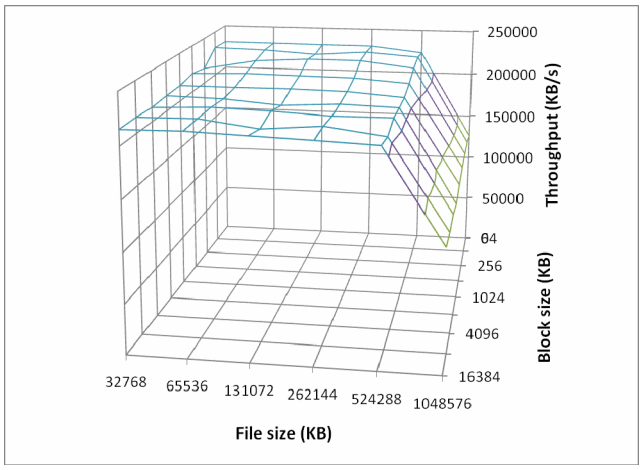

Figure 5: Sequential write file size and performance

Generally an application is designed to avoid processing files that are bigger than its physical memory capacity all at once 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.

4.3 I/O performance behavior of ext3 file system

This section describes the measured performance behavior of an ext3 file system. Figure 6 shows the I/O read performance of the ext3 file system that is measured with different hardware sub-systems. The measurements in Figure 6 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 6 also shows that when the block size reaches 64Kbytes, the performance of the file system start to drop. Figure 7 shows the I/O write performance exhibits a similar behavior but not as dramatic.

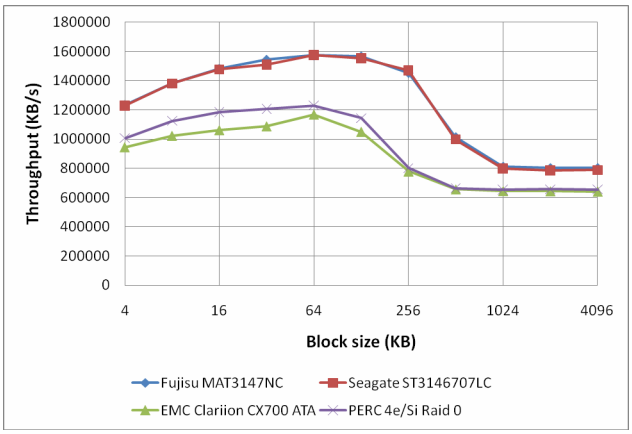

Figure 6: I/O read performance, different hardware

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 have interesting response times. Figure 8 shows the average response times of copy_to_user functions as I/O block size increases. The response times of copy_from_user functions are nearly identical to that of copy_to_user functions and are not shown for space reason. This is also the reason why the performance drop in the I/O write performance is not as dramatic as in the I/O read case. The response time of the copy_to_user function when compares to the overall I/O read time is much more significant than the same response time of copy_from_user function when compares to the overall I/O write time.

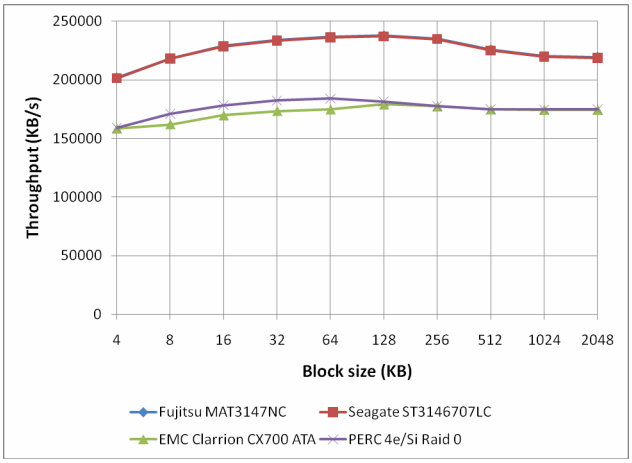

Figure 7: I/O write performance, different hardware

Examination of the kernel code for functions copy_to_user and copy_from_user shows no evidence to support this performance behavior from the functions' codes [14]. 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 [10] while running the I/O experiments. The results of L2 cache behavior of the copy_to_user function and copy_from_user function obtained during I/O benchmark are shown in Figure 9.

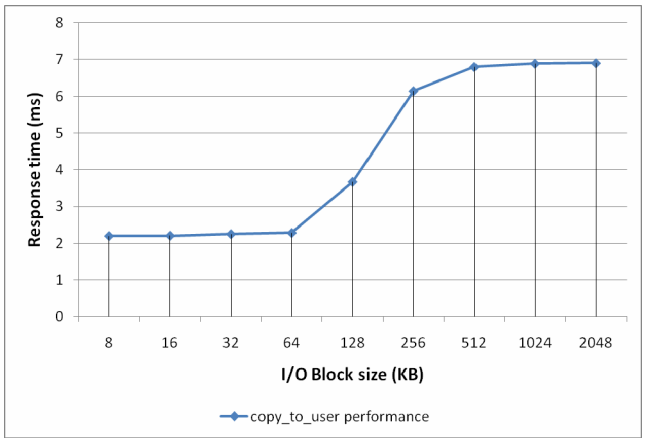

Figure 8: Average performance curve of copy_to_user and copy_from_user

The measurements in Figure 9 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.

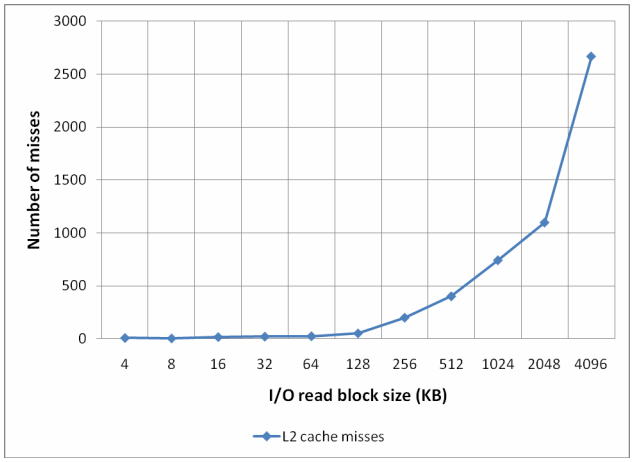

Figure 9: L2 cache behavior of copy_to_user and copy_from_user

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 8.

Table 2: L2 cache size vs. I/O block size where performance starts to drop

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 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 the size of about 1/8 of the total L2 cache size then copy_to_user and copy_from_user performance starts to drop.

5. IMPLEMENTATION OF THE SIMULATION MODEL

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. 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.

5.1 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 I/O read followed by computing followed by a large I/O write. Many times, the execution phase 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. In this paper we will focus on the default mode – ordered mode.

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, we enforce an fsync() 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 section 4.3, 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, we use a simple queuing model for our I/O scheduler and disk sub-system model.

5.2 Modeling using Colored Petri Nets

K. Jensen proposed an extended version of classical Petri Net called Colored Petri Net [6]. 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. These concepts prove to be incredibly powerful since tokens can now carry information that is simple or complex. This feature is used extensively in this paper to carry time stamps with tokens flowing within the simulation model.

A Colored Petri Net is a graphical oriented language for design, specification, simulation and verification of systems. This language is particular well-suited to illustrate and simulate systems in which communication, synchronization between components and resource sharing are primary concerns [9]. This makes it a very good tool for modeling file systems.

5.3 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. 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 disk.

The implementation of the fread function in the 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 kernel system call read to carry out the actual read from disk. The result of fread is an array of data gathered from the read system call and this array is returned back to the application.

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 10. The application and fread Petri Net are very simple and are not shown.

The Petri net shown in Figure 10 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 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 10, is implemented using a mathematical formula presented in Section 5.5.

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

The Petri Net model of the Linux buffer cache component is presented in Figure 11. 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 disk component are implemented using a simple queuing model and are not shown because of space reason.

5.4 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 fwrite function. The difference between file read and file write is what is being transferred. 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 data, however, may or may not be written to disk right away. If the application specified the write operation is synchronous, the data is written to disk before fwrite returns to the application. If the application uses the

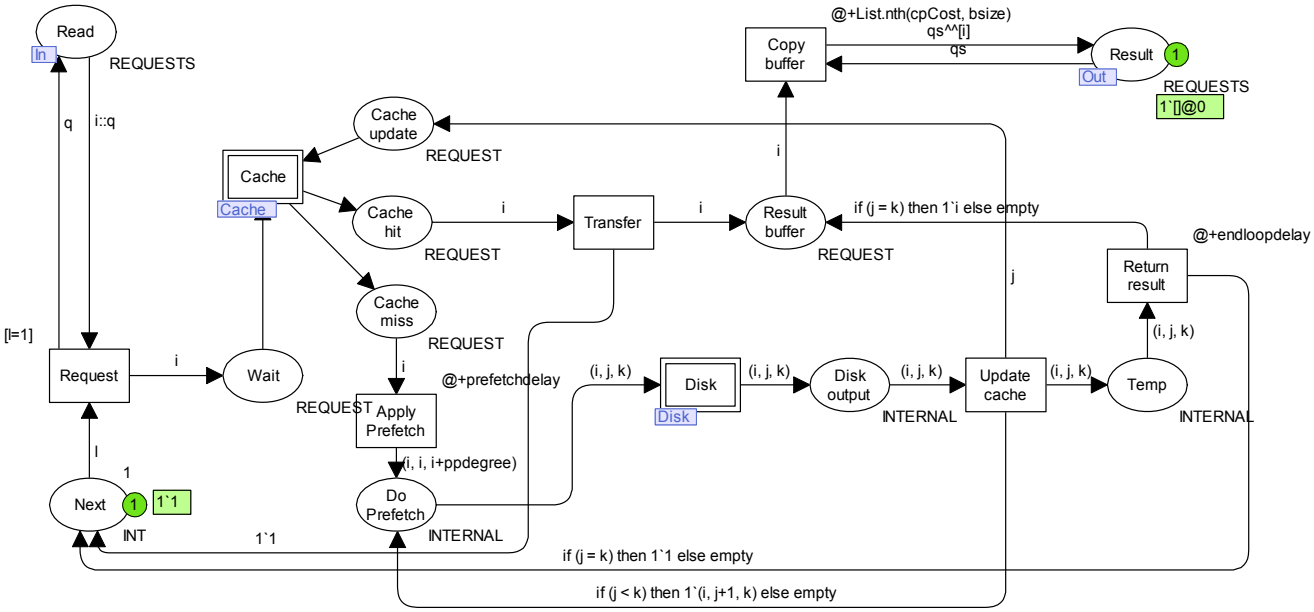

Figure 10: generic_file_read Petri Net model

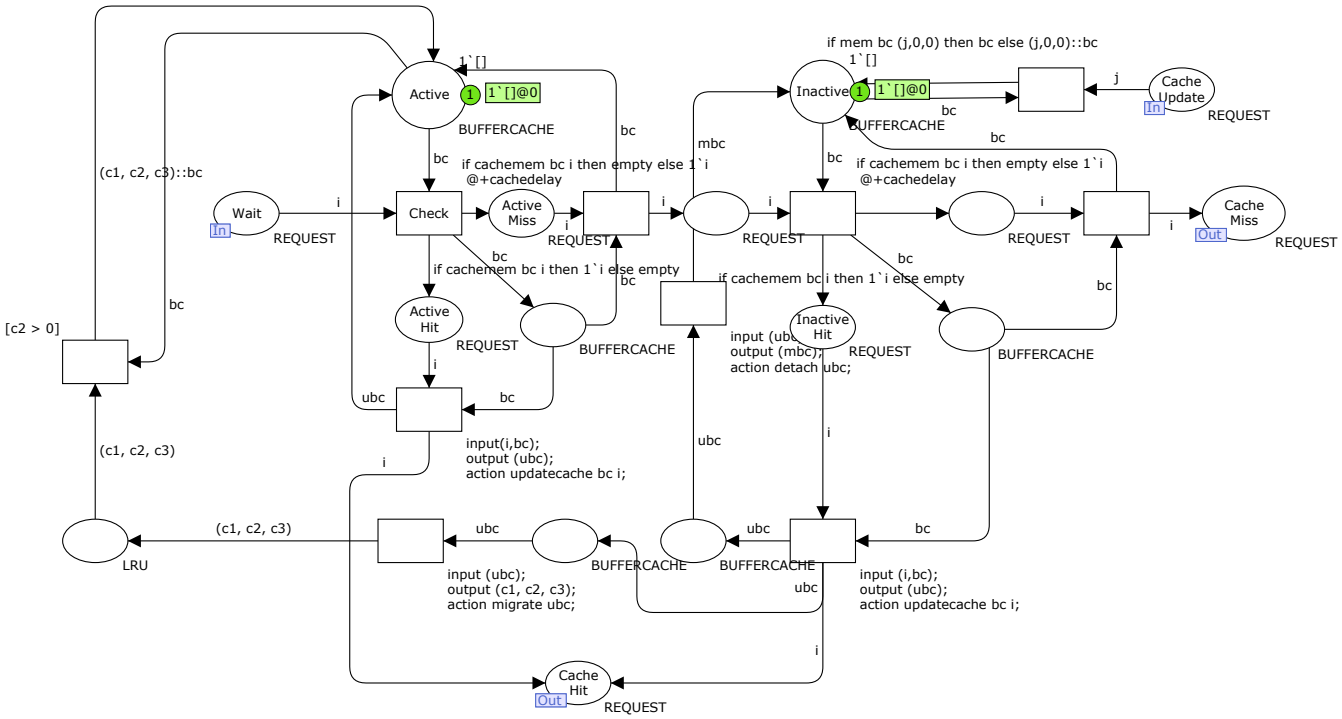

Figure 11: Buffer cache Petri Net model

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 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 fsync() function.

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 of the write system call is presented in Figure 12. The "Write begin" process prepares the system for the data from the user space such as allocate memory and journal tracking. 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 5.5. The "Commit write" process, implemented in Petri Net by several smaller processes such as "update journal", "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.

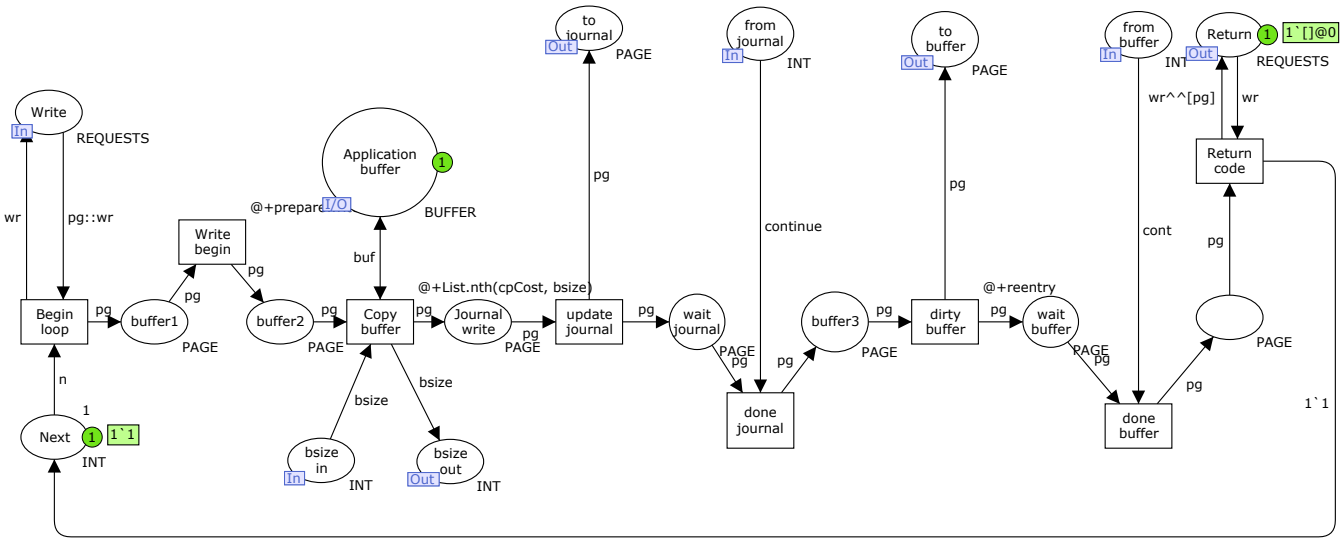

Figure 12: generic_file_write Petri Net model

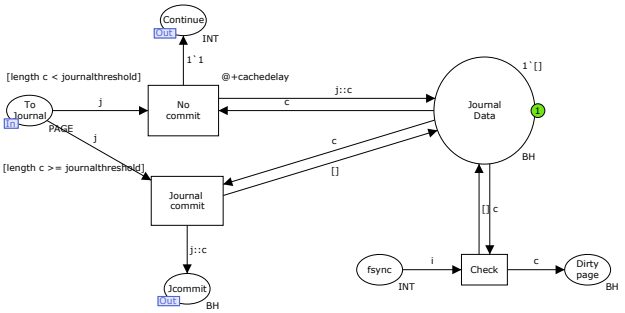

Figure 13: Journal Petri Net model

The Petri Net implementation of the journal is presented in Figure 13. Data is flushed to disk using a different mechanism. The data flush mechanism 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 fsync() function. The Petri net implementation of the data flush is presented in Figure 14. The write system call implementation uses a disk component very similar to the disk component in the read system call.

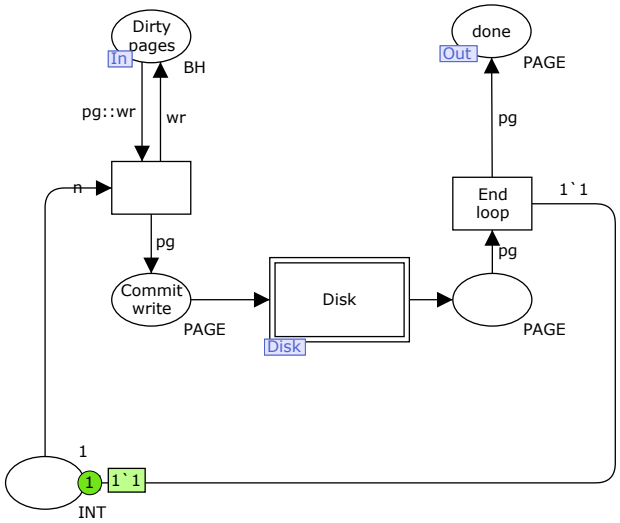

Figure 14: Data flushing Petri Net model

5.5 L2 cache effect model

The measurements in Section 4.3 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 TL2. 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 $S_{threshold}$, the average response time is

$$t = T_{L2}$$

If the I/O block size is greater than $S_{threshold}$, 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 does not fit within available L2 cache. When x becomes very large, t approaches T_{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 15 shows a comparison of this model for the L2 cache effect as compared to the measured data from Figure 8. Figure 15 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 L2 cache on copy_to_user performance is captured well by the model.

6. MODEL PERFORMANCE VALIDATION

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 mechanizing 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, function exit and provides function duration.

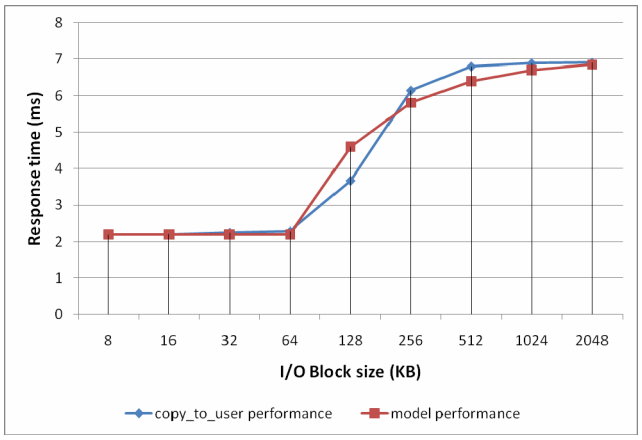

Figure 15: L2 cache model versus measured data (Figure 8)

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 function's constants ($S_{threshold}$, S_{page}, T_{L2}, T_{memory}) are measured from the test system. The Petri Net model (Figure 10, top right corner, and Figure 12, 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.

6.1 Synthetic sequential workload

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

The result of I/O read performance model is presented in Figure 16. The errors bars are set at 10%.

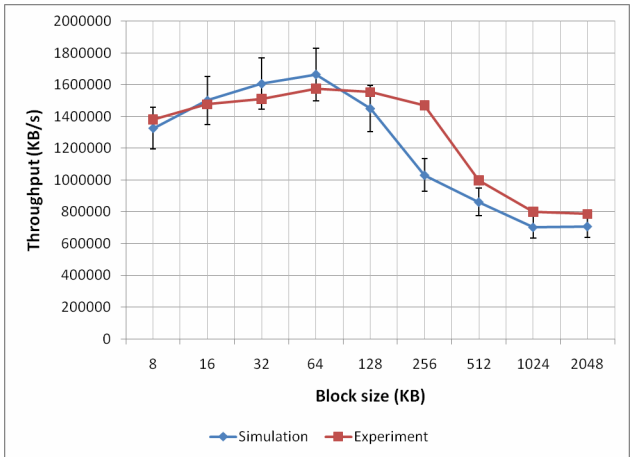

Figure 16: Sequential I/O read performance validation

Most data points fall within 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 16 shows that the Petri Net model result captures the trend of file system performance very well, and behaves very similarly to the real file system. The error for block size 256Kbytes is somewhat larger than that of other block sizes. This result is interesting, since the L2 cache model is actually closest for an I/O block size of 256Kbytes. This result emphasizes that the performance of the file system is affected by many factors, not just the size of the L2 cache, and is an area of further investigation.

I/O write performance experiments were performed in the exact same manner. Measured data from the actual kernel I/O path were inserted into the Petri Net model. The write simulations were 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 17. The error bars are also set at 10%.

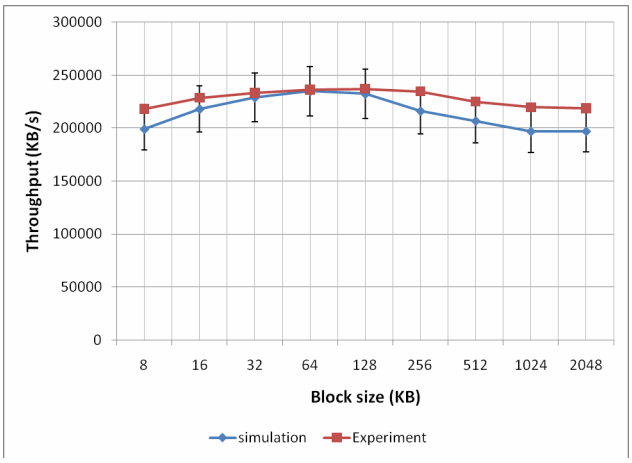

Figure 17: Sequential I/O write performance validation

The write result is even better than the I/O read performance result. Figure 17 shows that all data points falling 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 [11].

6.2 Synthetic random workload

Simulations were 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 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 was also used to feed the iozone benchmark to produce performance results on the test system. The I/O access pattern of the workload is presented in Figure 18. The Y axis represents the location of the I/O request. The X axis represents the order the I/O requests occur. Figure 18 presents the randomness of the workload very clearly.

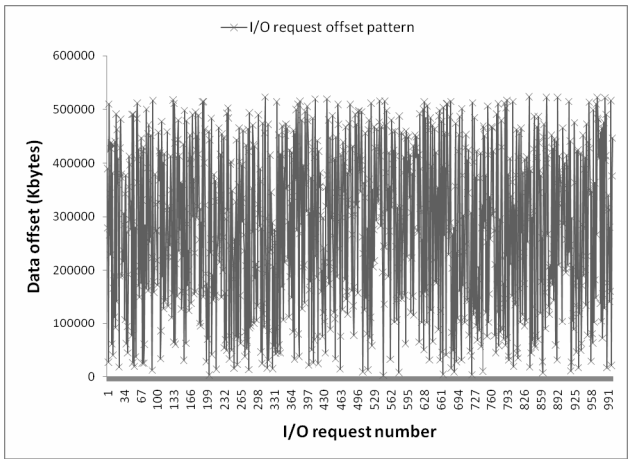

Figure 18: synthetic random I/O pattern

The random I/O read performance results are presented in Table 3.

Table 3: Random I/O read performance

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 random requests is also used in I/O write experiment. The performance results of random I/O write are presented in Table 4. 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 4: Random I/O write performance

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%

6.3 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 condition. 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 19 presents the I/O read requests pattern of the first captured trace.

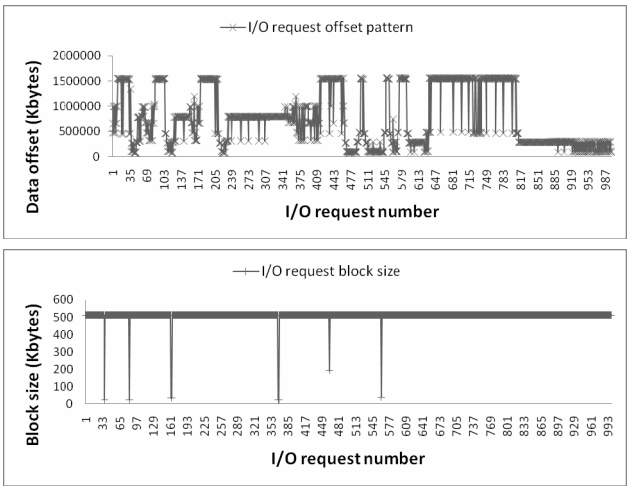

Figure 19: First captured trace I/O read pattern

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

Figure 20 shows the I/O write request pattern of the first captured trace.

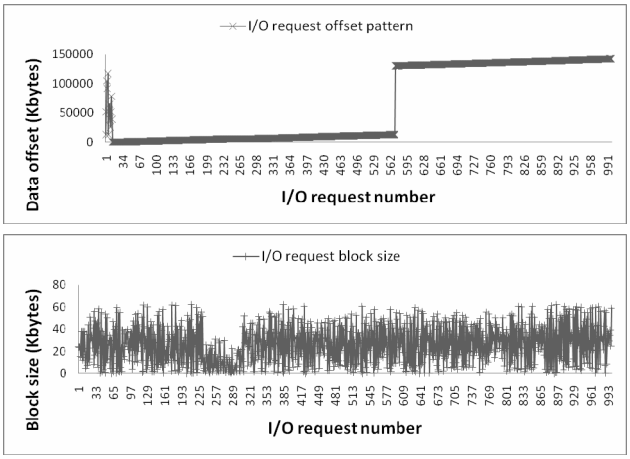

Figure 20: First captured trace I/O write pattern

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

Figure 21 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.

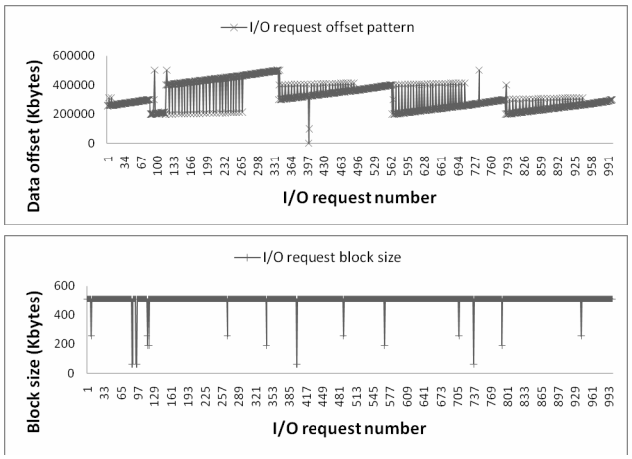

Figure 21: Second captured trace I/O read pattern

Figure 22 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 I/O write are also varied a lot in the duration of the trace.

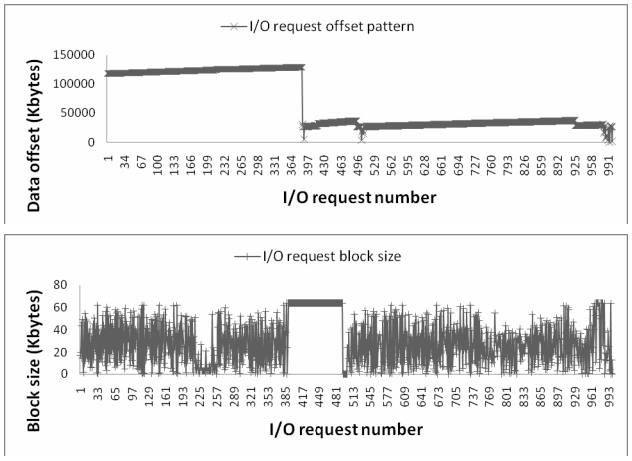

Figure 22: Second captured trace I/O write pattern

These two I/O read traces are fed into the model and iozone benchmark to produce the I/O performance comparison. Similar to the previous performance studies, simulations were run several times and produced the average result. The I/O performances are higher than previous experiments due to caching effect. Table 5 presents the performance results of the I/O read performance result.

Table 5: I/O read performance using traces

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 are also fed into the model and iozone benchmark to produce the I/O performance. Table 6 shows the performance results of the I/O write performance result.

Table 6: I/O write performance using traces

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%

6.4 The impact of the dirty ratio kernel parameter

The kernel parameter – dirty threshold – discussed in section 4.2 influences 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 dirty threshold setting on the system (~512MB). Figure 23 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.

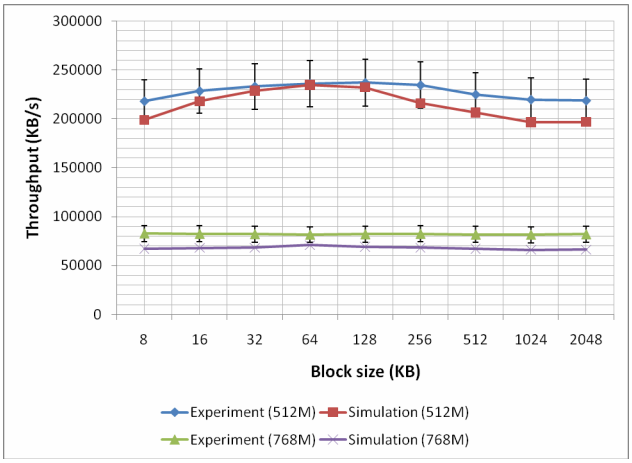

Figure 23: Dirty ratio kernel parameter effect

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 underestimate the performance of the actual system for both file sizes.

7. CONCLUSION

This paper 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 is performed. A model for the L2 cache behavior captures the behavior of the L2 cache and is used directly in the full model.

In previous work presented in Section 2, storage hardware performance models were examine and developed. However, end-to-end file system performances were generally overlooked. In this paper, both file read and file write including buffering effect and caching effect are modeled and the results are very close to the performance of the real file system. 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 are less than 35% of the real file

system. For I/O traces captured from live systems, the simulation performances are less than 20% in most cases. Additional performance factor such as dirty ratio is also modeled and validated.

A future paper will present performance study and model validation for the write back mode and journal mode of the ext3 journaling system. Also, 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. The model will be extended to include models of the network communications.

8. ACKNOWLEDGMENTS

This research is based upon work supported by the National Science Foundation under Grant No. 0421099.

The authors would like to thank Larry Dowdy for his feedback and sharing his invaluable knowledge and insights into file system behavior and performance modeling.

9. REFERENCES

[1] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger. The disksim simulation environment version 4.0 reference manual. Technical Report CMU-PDL-08-101, Carnegie Mellon University - Parallel Data Laboratory, May 2008.

[2] K. El Maghraoui, G. Kandiraju, J. Jann, and P. Pattnaik. Modeling and simulating flash based solid-state disks for operating systems. In *WOSP/SIPEW '10: Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, pages 15–26, New York, NY, USA, 2010. ACM.

[3] I. Gorton, P. Greenfield, A. Szalay, and R. Williams. Data-intensive computing in the 21st century. *Computer*, 41:30–32, 2008.

[4] J. L. Griffin. *Timing-Accurate Storage Emulation: Evaluating Hypothetical Storage Components In Real Computer Systems*. Phd dissertation, Carnegie Mellon University, Department of Electrical and Computer Engineering, 2004.

[5] J. L. Griffin, J. Schindler, S. W. Schlosser, J. C. Bucy, and G. R. Ganger. Timing-accurate storage emulation. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 6, Berkeley, CA, USA, 2002. USENIX Association.

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

[7] M. K. Johnson. Red hat's new journaling file system: ext3. Red Hat Support White Paper, 2001. http://www.redhat.com/support/wpapers/redhat/ext3/.

[8] T. M. Jones. Anatomy of the linux file system. IBM developerWorks Linux Technical Library, 2007. http://www.ibm.com/developerworks/linux/library/llinux-filesystem/.

[9] L. M. Kristensen, Søren Christensen, and K. Jensen. The practitioner's guide to coloured petri nets. *International Journal on Software Tools for Technology Transfer*, 2:98–132, 1998.

[10] J. Levon. Oprofile - a system profiler for linux. Sourceforge, 2010. http://oprofile.sourceforge.net/.

[11] B. Lu, A. Apon, D. Hoffman, L. Dowdy, D. Brewer, and F. Robinson. 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.

[12] N. Murray and N. Horman. Understanding virtual memory. Red Hat Magazine, 2004. http://www.redhat.com/magazine/001nov04/features/vm/.

[13] L. K. Organization. Linux kernel source code. The Linux Kernel Archive, 2010. http://www.kernel.org/.

[14] J. Pommnitz. Kernel level exception handing in linux 2.1.8. Linux Kernel Documentation, 2010. http://www.mjmwired.net/kernel/Documentation/exception.txt.

[15] A. V. Ratzer, L. Wells, H. M. Lassen, M. Laursen, J. F. Qvortrup, M. S. Stissing, M. Westergaard, S. Christensen, and K. Jensen. Cpn tools for editing, simulating, and analysing coloured petri nets. In *ICATPN'03: Proceedings of the 24th international conference on Applications and theory of Petri nets*, pages 450–462, Berlin, Heidelberg, 2003. Springer-Verlag.

[16] Y. Wang and D. Kaeli. Execution-driven simulation of network storage systems. In *Proceedings of the 12th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04)*, pages 604–611, Los Alamitos, CA, USA, 2004. IEEE Computer Society.