University of Stuttgart

Faculty of Computer Science

Masterarbeit Nr. 3196

Analysis and Optimization of Storage IO in Distributed and Massive Parallel High Performance Systems

Salem El Sayed Mohamed

Course of Study:	Infotech
Examiner:	Prof. Dr. Sven Simon
Supervisor:	Dipl. Inf. Simeon Wahl M.Sc. Wenbin Li Supervisor in IBM Dipl.-Ing. Heiko Schick
Commenced:	May 16, 2011
Completed:	November 15, 2011
CR-Classification:	C.2.4, C.4, D.4.2, D.4.3, H.3.4

Institut für Parallele und
Verteilte Systeme
Abteilung Parallele Systeme
Universitätsstraße 38
D-70569 Stuttgart

Acknowlegement

Thanks to Prof. Dr. Sven Simon for allowing me to work on such an interesting topic. Thanks to Dipl. Inf. Simeon Wahl and M.Sc. Wenbin Li for their supervision and advice.
But most of all thanks to Dipl.-Ing. Heiko Schick and everyone in the Blue Gene Active Storage team for allowing me to be part of them for a complete year. My hope is that you will benefit from my work as much as I benefited from your advice and guidance.

Salem El Sayed M.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

____

(Salem El Sayed Mohamed)

Abstract

Although Moore's law ensures the increase in computational power, IO performance appears to be left behind. This minimizes the benefits gained from increased computational power. Processors have to idle for a long time waiting for IO. Another factor that slows the IO communication is the increased parallelism required in today's computations. Most modern processing units are built from multiple weak cores. Since IO has a low parallelism the weak cores will decrease system performance. Furthermore to avoid added delay of external storage, future High Performance Computing (HPC) systems will employ Active Storage Fabrics (ASF). These embed storage directly into large HPC systems. Single HPC node IO performance will therefore require optimization. This can only be achieved with a full understanding of the IO stack operations. The analysis of the IO stack under the new conditions of multi-core and massive parallelism leads to some important conclusions. The IO stack is generally built for single devices and is heavily optimized for HDD.

Two main optimization approaches are taken. The first is optimizing the IO stack to accommodate parallelism. Conclusions on IO analysis shows that a design based on several parallel operating storage devices is the best approach for parallelism in the IO stack. A parallel IO device with unified storage space is introduced. The unified storage space allows for optimal function division among resources for both read and write. The design also avoids large parallel file systems overhead by using limited changes to a conventional file system. Furthermore the interface of the IO stack is not changed by the design. This is a rather important restriction to avoid application rewrite. The implementation of such a design is shown to result in an increase in performance.

The second approach is Optimizing the IO stack for Solid State Drives (SSD). The optimization for the new storage technology demanded further analysis. These show that the IO stack requires revision on many levels for optimal accommodation of SSD. File system preallocation of free blocks is used as an example. Preallocation is important for data contingency on HDD. However due to fast random access of SSD preallocation represents an overhead. By careful analysis to the block allocation algorithms, preallocation is removed.

As an additional optimization approach IO compression is suggested for future work. It can utilize idle cores during an IO transaction to perform on the fly IO data compression.

Contents

List of Figures

List of Tables

Nomenclature

ASF	Active Storage Fabrics
BIO	Block IO
CPU	Central Processing Unit
Ext	Extended File System
FIO	Flexible IO
FTL	Flash Translation Layer
GPFS	Global Parallel File System
HDD	Hard Disk Drive
HPC	High Performance Computing
IO	Input Output
IOPS	IO Operations Per Second
IP	Intellectual Property
MLC	Multi-Layer Cell
MPI	Message Passing Interface
NSD	Network Shared Disk
NVM	Non-Volatile Memory
POSIX	Portable Operating System Interface for Unix
PVFS	Parallel VFS
SLC	Single Layer Cell
SSD	Solid State Disk
VFS	Virtual File System

1 Introduction

As electronic devices advance new challenges rise. The advance in the area of High Performance computing or HPC has generally taken place in the realm of processing power. Indeed personal computers have also been advancing quite rapidly in the area of processing. This has been insured by Moore's law, which suggests that transistor capacity per chip doubles every 18 month. Moore's law has been in effect for quite a long time. Not only has the processing power increased but a single processor now contains more complexity. As the modern Central Processing Units or CPUs hit their frequency limit expansion took place in a different dimension. Modern processors contain multiple cores each representing a full blown CPU. More over threading and hyper threading meant that cores can operate on different threads and switch between them with less latency.

In recent years the amount of data available grew dramatically. The capacity offered by Hard Disk Drives or HDD has risen. This meant that processing units now had available capacity of data to process on massive scales. However as processing power and capacity grew, the performance of Input Output or IO did not keep pace. In turn the gap between processor and storage widened. Old processor units operated with a single cache level. In comparison modern processors contain up to four levels of cache. These attempt to hide the delay exerted by HDD data requests. Still the processor exhibits latency penalties whenever a cache miss occurs.

The IO gap increased as the processing power changed shape. Modern processors employ multiple cores. Meanwhile HDDs although growing in speed and capacity, still managed all accesses using a single queue. This is a preset limitation due to classical disk based design. The single head available for read and write operations resulted in an inherit bottleneck. Furthermore the need for a movable needle and a rotating disk leads to mechanical limitations in speed. Moving the head exerts delays called seek time. As each core runs different threads data is anticipated to be scattered all over the storage space. This in turn leads to additional seek time as the HDD attempts fulfilling the requests made by different cores.

The IO gap can especially be observed on HPC systems. Not only do they employ more complex IO settings, but also contain far higher computational power than traditional personal computers. HPC systems are mainly build from dense integration of computational nodes. In addition to that, data amounts for scientific applications have been rapidly increasing. Performance of HPC processors can cover these massive amounts of data. However due to the IO gap feeding processor with all this data requires the processor to wait. These CPU idle IO periods lead to performance degradation [1]. Nevertheless due to its complexity the IO issue has been previously ignored. As the IO become the systems bottleneck, this was no longer possible [2].

As the scale and size of modern HPC increases, the use of separate storage and computing units is no longer possible. Some suggestions for the integration of storage and computing units have already been made. Such integration would result in so called Active Storage Fabrics or ASF [3]. These will require fast single node storage IO. In turn requiring optimization of single node IO on HPC systems. The optimization at that level is mainly governed by Kernel IO stack issues. To perform such optimizations a detailed analysis of HPC single node storage IO is needed. This is a new scale for IO optimization in HPC systems.

The task of finding more optimized approaches for accessing data is important. Most elements and algorithms used in modern storage access are still based on old technologies. The use of new storage technologies meant that these algorithms are obsolete. The reason for continues use of these IO algorithms is the inherit complexity of the system. A modern operating system has to support a wide range of different storage devices. In addition to that some new technologies do not yet have any interface standardization. Optimization of IO holds the promise of better performance using new storage technologies. Yet the optimization is prevented by complexity and scale of challenge.

A common HPC proverb suggests that a super computer reduces a CPU bound problem into an IO bound problem [4]. The same logic can be applied to IO optimization. The perfect IO machine would convert any IO bound problem into a CPU bound problem. The bottleneck thus keeps moving between these two computational boundaries. As the processor has been rapidly growing in speed and computing power, it is time to do the same for IO.

The outline of the thesis is as follows. Chapter 2 introduces the problem in more detail and explains the optimization approaches studied. Chapter 3 contains a detailed analysis of the IO stack of the traditional Linux Kernel. The chapter also explains basics of Solid State Disks or SSDs. Chapter 4 deals with optimization of parallelism in the IO stack. Chapter 5 explains issues and optimizations done for using storage class memories and focuses on the emerging technology of SSDs. Finally conclusions made from the study are given along with suggestions for future work.

2 IO in High Performance Computing

There are several challenges for storage IO on HPC. The scale of IO is massive compared to personal computers. Some scientific applications require large data sets. Although the processing power is increasing rapidly the IO can no longer match the required performance. Thus leading to a gap that is rapidly increasing [5]. The gap between IO performance and computational power of the HPC increased interest in the area. The target is to achieve rapid data transfer between storage and computing systems [6]. The main difficulty with achieving high IO performance on HPC is the complexity of the system. The HPC IO has been developed in several layers. These layers are typically very complicated. The complexity is a direct result of the scale on which each layer has to operate.

A typical personal computer contains a simple stack of layers through which the IO request has to move to be fulfilled. The request starts at the application. The operating system then channels the request towards the file system. Given that the file system exists on a single device the file system directs the request to the device driver. The device accesses the storage to fulfill the request. In comparison the HPC system has more massively scaled layers. Furthermore the IO request initiated by a node is usually not executed on the same node. Computation in HPC is densely packed and storage is located on separate systems. Therefore there are additional layers that exist on an HPC system.

The complexity of the HPC IO stack can be seen in Figure 2.1 [7]. Each of these layers has to operate on large scales. The IO libraries available on a typical high performance system have the complexity of dealing with massively parallel applications. The libraries therefore have to offer both sequential and parallel access [7]. Applications on HPC systems require access in different methods and using different patterns than that of typical user applications. The IO interface for HPC applications has to also adapt for parallelism. Some HPC systems still provide the application with the typical IO interfaces such as POSIX short for Portable Operating System Interface for Unix . Other more common HPC IO interfaces also exist. One such interface is the MPI-IO or Message Passing Interface IO . These scale better with the HPC applications. They also provide operations that are required by parallel applications such as synchronization and data coherency. The massive scale and parallelism of HPC applications indeed aggravates the HPC IO libraries and interface complexity.

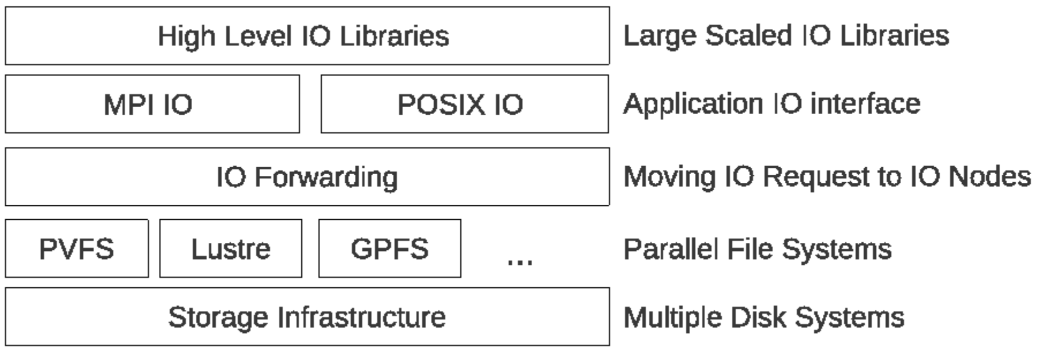


Figure 2.1: Storage IO stack for HPC

Typical HPC nodes are built to provide large scaled application with as much computational power as possible. These therefore contain complex multi-core processors. To avoid wasting any computing power the nodes run minimalistic operating systems [7]. The IO however requires additional complex algorithms. Therefore to avoid running these algorithms on the computing nodes dedicated nodes are used for IO. These nodes are typically referred to as IO nodes on the IBM Blue Gene system [7]. The IO nodes can run more complex IO algorithm. However the distance has been further increased between computation and storage.

Typical disk access can no longer keep up with the rapid increase in processing power. Figure 2.2 shows how drastic the gap between disk and CPU has increased over time [1]. The disk can no longer sufficiently feed the processing unit with data. The figure shows that CPU cycles needed over the years for accessing disk data has been increasing. Mean while capacity of disk has been rapidly increasing and cost per gigabyte storage has been dropping. HPC systems overcome disk limitations using several disk operating in parallel. The storage space is increased by the addition of a new disk. Also by using the disks in parallel the bandwidth and IO Operations per Second or IOPS are increased.

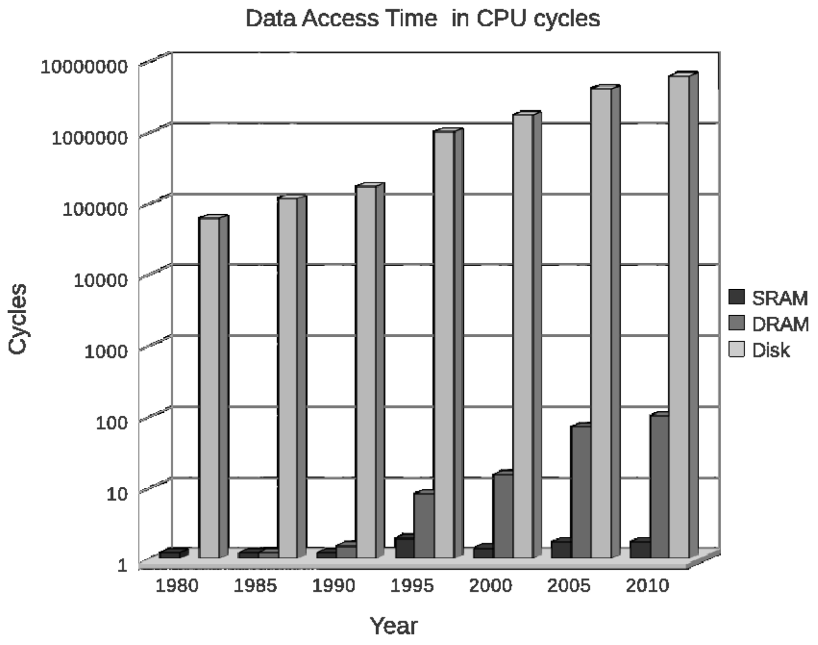


Figure 2.2: Data access time in CPU cycles for SRAM, DRAM and disk over the years

Parallel disk operation could virtually increase performance to infinity. Each disk added to the system promises the increase of performance. However utilizing all disks together increases overhead. In addition to that the disks parallelism is dependent on application data sets. Conventional personal computer file systems operate on a single device. Therefore HPC with distributed storage employs so called parallel file systems. These operate across multiple devices. Parallel file systems have to attempt switching as many devices together as possible. As a result they use different types of data distribution. Some parallel file systems stripe data into small chunks to be divided across available disks. Others replicate the same files on different disks to read or write different sections of files in parallel.

The HPC IO setting has served the applications well for the past years. Nevertheless the dramatic increase in data, storage capacity and the increase of IO gap means new more flexible settings have to be researched. In addition to that the implication of utilizing new storage technologies must be found. The IO performance of the peta-scale HPC systems must improve.

Thus making way for even higher IO demands of the next coming exa-scale HPC era.

2.1 Active Storage Fabrics

One of the main difficulties introduced in HPC systems IO is the distance between compute and storage units. Compute nodes have to forward IO requests to separate IO nodes. These access a network to request the data from the storage systems. Due to compute nodes limited RAM capacity the nodes cannot request large data amounts for computation. The compute nodes therefore spend longer times idle awaiting IO.

The ASF brings storage closer to compute nodes. The idea is to embed storage units into the densely packed computation systems. A Blue Gene rack would therefore contain in addition to compute and IO nodes storage units. This means a shorter distance between compute and storage untis [3]. These ASF systems would be employed for data intensive applications. The method for implementing ASF using Blue Gene is shown in Figure 2.3 [3].

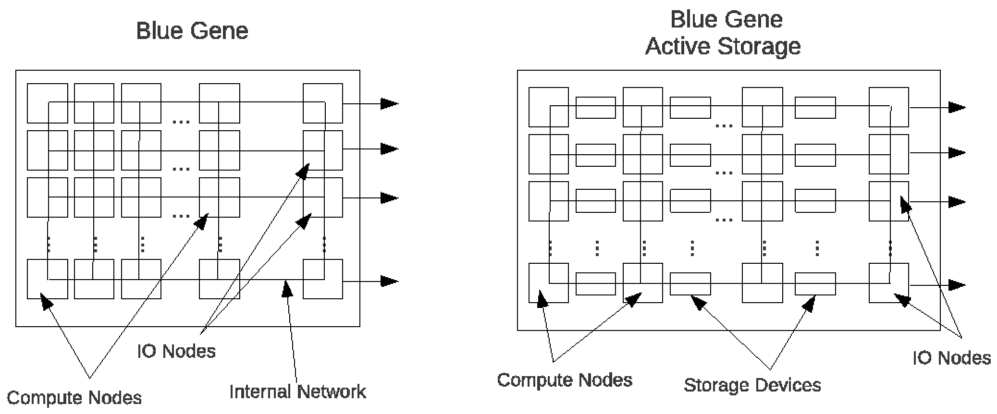


Figure 2.3: Traditional Blue Gene setting and Blue Gene Active Storage

The ASF promises better performance of IO bound applications. The challenge is to embed the storage into the HPC system. Placing HDD into a densely packed compute system is not possible. HDD are based on mechanical components and are rather large in size. Thus new storage technologies should be used instead. Storage class memories are an ideal candidate for ASF. One of the storage class memories is SSD. These offer a better IO performance and a more dense packing. Another challenging aspect of ASF is that ordinary storage devices are not build for parallelism on this scale. Blue Gene employs multi-core processing units. Although the new storage technologies offer a more parallel access model, traditional IO stacks do not. Therefore to achieve high IO on an ASF system single node IO to the embedded storage has to be optimized.

The traditional IO stack of common operating systems does not yet support massive parallelism. Furthermore the development of the stack has been mainly done in the HDD era. Due to the basic technology differences between SSD and HDD there is room for improvement. Optimization on a single node level would increase performance as a whole for the system.

2.2 Optimization Approaches

The operating system running on Blue Gene is a form of Linux. Thus the Linux IO stack has to be optimized and adapted for ASF. The optimization has to employ ASF basics. Furthermore it has to be based on the massive scale of parallelism present in the Blue Gene. As a sign of improvement the IO stack should be able to communicate with the multitude of available cores on the single processing unit. Additionally it should be able to better preform in relation to the new storage technologies.

An important concept to optimization is that it can only take place if the system is fully understood. Every detail in the Linux Kernel relative to the IO issue should be analyzed. The effects of different settings should be researched. This is not an easy task. The Linux IO stack is complicated and includes many operations. Storage device basics should also be studied. The device properties can be used to further optimize the IO stack.

This study has targeted two main optimization approaches. The first is IO parallelism and the second is optimization for storage class memories. IO compression is suggested as a third possible optimization approach. However the first two optimizations show more promise for IO performance. In comparison IO compression is and optimization that could be used for other properties such as allocation of idle CPU time.

2.2.1 IO Parallelism

Processors internal parallelism is rapidly increasing. However the IO stack requires adaptation of more parallel data access forms. The difficulty therefore becomes adapting IO for parallelism. High level parallelism of IO has long existed in the HPC systems. HPC has been employing multiple disks and parallel file systems for many years. Since this form of parallelism has not yet found its way into personal computers, the traditional IO stack remained for the most part unchanged.

The IO parallelism required by the ASF is implemented on a single node level. The challenge is to adapt solutions made on the large scale for a single node. Technologies already offer some of the large scale parallelism for single nodes. Parallelism using multiple cores is one form offered. Another is the implementation of SSD using several controllers. Therefore an analogy could be made between large and small scale IO. Optimizations done on large scale multi processors systems could be adapted for multi-core on a single processor. On the other hand parallel file systems can run on a single node and use different controllers on a single SSD as different devices.

Large scale parallelism offers a good start for finding optimization approaches that address single node IO parallelism. However adapting these optimizations faces single node limitation. A single node does not have unlimited processing power. On that level most resources are shared. The target therefore becomes scaling back high level parallelism to fit single node operations. This is indeed a difficult challenge. Large scale parallelism has dedicated hardware and can further add more resources if necessary. In comparison resources are limited on a single node. In fact single nodes IO parallelism will share resources with application. Thus the overhead induced by the IO on single nodes has to be as small as possible.

2.2.2 Storage Class Memory

Storage technology has remained dependent on HDD for many years. The main difficulty with improving HDD performance is mechanical limitations. This is an inherit property in all disk based storage systems. The limitation does not exist in emerging storage class memories. Solid state storage units such as flash can be written and read electronically. Thus these technologies offer better performance in comparison to traditional disk based systems. In addition to that SSDs consume less power and can be better packed into the dens HPC systems.

The IO stack has been mainly developed in the HDD era. This means that optimization is done in relation to HDD access patterns. The SSD however is based on a completely different technology. SSDs adhere to a different structure and organization. Furthermore inherit reliability issues of flash based memory requires SSDs to be handled in a different method than HDD. Some attempts have been previously made for SSD access optimizations. However the IO stack still induces overhead running HDD access algorithms that do not benefit SSDs.

The IO stack controls most of the underlying device performance. This means that an optimization of the IO stack for SSD would promise a better integration. For that purpose both SSD and the IO stack has to be analyzed. It is crucial that adapting the IO stack for SSD is done in relation to the system in which the SSDs will be integrated. The ASF requires SSDs to operate in correlation with multiple weak cores. This further limits the performance and optimizations possible. The challenge of integrating SSDs into the ASF design is a balance between several factors. Among these factors is the reliability of SSD, system properties, performance and others. The difficulty of the balance is increased due to limited allowed changes that can be implemented on the system in general. In essence the optimization should not change interfaces. The target is therefore to avoid application rewrites.

2.2.3 IO Compression

The combination of processing power of modern CPUs with the slow IO performance leads to CPU idle times. During those periods the CPU has almost no task, but still consumes power. IO compression indicates the ability of utilizing free CPU time to improve IO performance. If compression can be performed transparently the system could achieve more bandwidth with no implications to the applications.

IO compression is a difficult subject that requires answers to many questions. The method of compression should be identified first. There are many different algorithms for compressing data. The compression algorithm is dependent on a multitude of factors. These include compression rate, complexity of compression algorithm, computing power required and application data form. The selection of algorithm should also be done based on speed of compression versus IO performance gained. An additional difficulty is performing seamless compression. Since the applications should not be aware of the compression process, data should not be lost.

The next issue is the IO layer in which compression is performed. The IO stack is formed of both hardware and software components. The software itself is divided into several layers. The decision of compression layer is strongly related to purpose of compression. A compression done on the device for example would not benefit the IO speed without system involvement. An additional problem is read versus write compression in relation to CPU utilization. An idle CPU during write used for compression might not be idle for decompression.

IO compression could be built on top of the previous two optimization approaches. Therefore it was preferred to focus on parallelism and storage class memories. The topic is still interesting for future work. This is particularly true in combination with conclusions made for the previously mentioned optimizations.

2.3 Outline

The rest of the thesis is outlined as follows. Chapter 3 is a detailed analysis of the IO stack of the Linux Kernel. A special focus has been given to operation on SSD. Therefore the final Section in Chapter 3 explains SSD basics.

Chapter 4 deals with optimizing the stack for parallelism. After introducing the topic Chapter 4 shows the related work done for this optimization approach. The next section in chapter 4 explains the test environment. After that different IO tests that analyze the IO stack are provided with conclusions drawn from the results. The final two sections in Chapter 4 introduce and test a new implementation for parallelism based on the analysis and their conclusions.

Chapter 5 deals with optimization done to better accommodate SSDs. The first section in Chapter 5 shows previous work done on optimizing for SSD. The second section introduces several IO tests analyzing SSD and related factors. It also provides conclusions that can be drawn from them. The test environment used has already been explained in Chapter 4. The last Section in Chapter 5 contains a detailed implementation of an example for improving the IO stack for SSD.

3 IO Stack

The representation of the IO in the form of a stack is a common visual interpretation to the flow of data from a device to the application or visa versa. It also facilitates the functional division among different Kernel components. This in itself makes traversing such stack simpler and easy to follow. Nonetheless, the requirement for clear function definition sometimes becomes difficult. To achieve this it is important to keep tasks of the same type in the same level. However grouping tasks together is made difficult by the complexity of these tasks. Some features requires changes made in several stack layers, which complicates IO development even more.

It is expected that most operating systems will have to move through almost the same steps to accommodate the same underlying IO hardware. Therefore it is safe to assume that conclusions made about one operating system could be in some form migrated to another. The Linux Kernel being open source presents a viable example as to how the IO stack could function.

The chapter is organized as follows. Section 3.1 discusses method and problems of analyzing the Linux Kernel. Section 3.2 introduces the layers of the IO stack. Section 3.3 explains the first layer, the Virtual File System or VFS . Section 3.4 discusses the task of individual file systems by explaining one of Linux native file systems the Ext2. Section 3.5 explains the task of the block layer and refers to some of the device drivers requirements. Finally Section 3.6 explains the underlying basics and reliability issues of SSDs.

3.1 Linux Kernel Complexity

The stack might looks quite organized, with clear definitions for each layers task. However things get quite complicated when taking a look at the inner workings of the stack itself. The Kernel is made of millions of lines of code and thousands of functions. Since the code has evolved over different phases, it is an enormous task to understand most of what has been coded into the Kernel basics. The code is made of hundreds of different functionalities, which makes the task of finding that which is relevant to the IO stack difficult. Some division exists within the source code. Despite that most of the functions depends on each other and require function calls that lie outside of the layer itself grouped into one folder. Most of Linux fans take the magnitude of the Linux kernel as a sign of success of open source projects. Nevertheless the size and complexity of some trivial tasks implementation shows some difficulties in that model of code development. Most Linux code developers cannot hold the overview of such a complex Kernel. This leads most towards focusing on the development of new functions which would later be built into the Kernel release. Mean while little is done to optimize the already existing code.

One of the issues that makes the Linux Kernel attractive for most modern systems is compatibility. The Linux Kernel support numerous processor architectures and promises easy use of multiple file systems. Not to mention direct support of a multitude of devices. All this is promised with almost no change to the applications. In case of file systems the communication is well hidden under the VFS. File system operations is hidden to the extend that no change, not even a recompile is needed for applications using different file systems. Although this compatibility makes life easier for application developers, it over complicates tracing the Kernel code. One example of such complexity is the need of function pointers. Since each file system implementation has to overwrite basic function calls, the VFS has to use function pointers within certain constructs, which these file systems can then overwrite. This means that the normal function call graph tools cannot follow this function calls, leaving the Kernel programmer to fend for himself. Additionally this technique of hiding true functions under a heavy layer of abstraction means that the function call graph are long and hard to trace. Therefore finding where the true task is carried out sometimes becomes itself a burden, even more so for trying to optimize this task. The difficulty increases when editing or adding new functions. A wide complicated net of functions exists within the Kernel. Therefore editing Kernel functions means retracing the function call to make sure that this net has not been broken somewhere along the function call.

Even though most of what has been mentioned previously appears to only show difficulties in tracing and optimizing, the Linux Kernel is still preferable to use for analysis and optimization. As an open source project, the Linux code is well documented. Several books exist that help with tracing. Nonetheless, caution is needed when referring to Linux Kernel books. Most only contain a high level view of how the Kernel is implemented and seldom mention any code. Therefore mapping book explanations onto existing Kernel code is difficult. For example in [8] Linux is handled with a higher level view. Few attempts have been found for books that contain code references. The Linux Kernel explanation found in [9] refers to some code details. Although code was well documented and a fine attempt to show as many call graphs as possible was made, still a lot of important information is missing. This is not a failed attempt on behalf of the writer, but can only be the result of the Kernels magnitude and complexity. Open source projects provide more documentation, yet it also results more versions. Numerous Kernel versions exists and each changes just little enough for some patches to be useless. Therefore working with multitude of machines running different Kernels, as was the case of this study, dealing with these changes becomes a burden. Therefore caution has to be kept when trying to use books such as to follow function calls. The Linux Kernel changes fast, adding functions as well as changing existing function names. All make the use of books possible for basic understanding, but never sufficient for full realization of how actual Kernel code functions.

Aside from being open source and well documented, the Linux Kernel contains a multitude of smart tricks. These have been developed over long years by the maintainers. The abstraction layers such as VFS and block layer might be difficult to follow, but make quite a good job when it comes to adding new features. Well support is given therefore for rapidly changing architectures and changing applications. This feature seams to make the Linux Kernel an attractive operating system for even the most specialized of systems such as high performance computing and embedded systems. Another typical brilliant implementation for which the Linux Kernel can be appreciated is the fact of it being object oriented. Although written in a none object oriented language, the Kernel is built on the concept itself. Structures or *structs* are used as a method to maintain many different combinations of systems. This sometimes facilitates tracing and optimizing the Kernel. Developers find the attributes neatly gathered together in a group of structures. Even more fascinating is the method of linking one structure to the next to make different attributes accessible over almost the entire Kernel. This sometimes, however, makes the optimization and development of new parts of the code more difficult. Specific guidelines have to be followed during code development. Not to mention how much effort has to be spend trying not to destroy the entire complex web of pointers that are contained in the Linux Kernel. Although being complex and difficult to trace and optimize, as is expected from an operating system, the Linux Kernel still is the most suitable for this task. Aside from previous mentioned reasons, the Linux Kernel runs on Blue Gene in different forms.

To facilitate tracing the Linux Kernel a documenting tool has been used to trace function calls. The documenting tool called Doxygen produces both call and caller graphs. A call graph shows the relationship of functions. The graph is organized by showing all functions called by the function undergoing analysis. The call graph starts on the left with the function under examination and moves to the right with the called functions. On the other hand the caller graph is the opposite. A caller graph shows all functions that call the function under examination. The caller graph starts on the right and moves to the left ending with the function under examination. Arrows always start at the caller and end at the callee. As previously mentioned function pointers are ignored by the documenting tool. Therefore function calls done with function pointers must be found explicitly in the Kernel code. Furthermore, due to the complexity of the Linux Kernel running the documenting tool on a normal laptop can take several days. It is preferred to run the tool in a virtual machine to be able to store the machine state in case of a system crash. To further facilitate the documenting task, Kernel components not related to the IO stack should be excluded.

3.2 Linux Kernel IO Stack

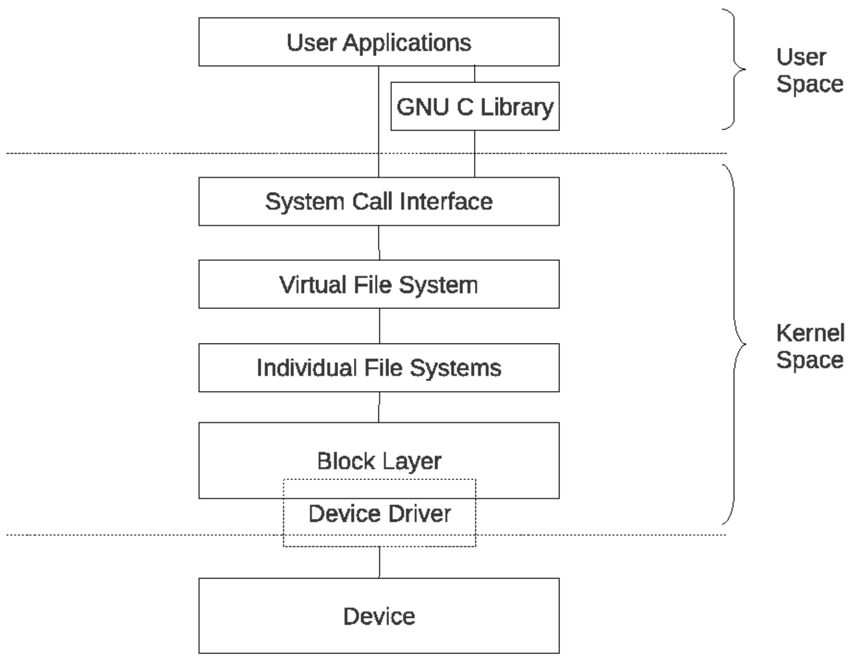


Figure 3.1: Kernel IO Stack

As seen in Figure 3.1 the Linux Kernel IO stack is formed of several layers. The user application can make a system call directly using the corresponding system call interface function. Another option is to use intermediate methods such as the GNU C library or any other library that implements the POSIX standards. Although depicted here as a separate layer the system call interface is a mere collection of functions placed at different positions within the Kernel code. As an example the function call for mounting a file system is placed in *fs/namespace.c*. Therefore the mounting option exists within the VFS and not in a separate collection. This is more convenient for developers as there is no need to pass the function parameters defined in the VFS to some other module.

The main part of the Linux Kernel IO stack is formed of three main components:

- Virtual File System (VFS)

- Individual File Systems
- Block Device Layer

The VFS has the main task of abstracting all underlying layer functions. Thereby the VFS unifies access to all different types of file systems. This is not a trivial task. Although needed for implementing unification over all file systems, it also needs to allow all these to perform there different tasks unhindered [9, Ch. 8]. Once the VFS determines which file system the request is intended for it moves to the next level selecting the corresponding file system interface or functions.

The individual file systems on the other hand have the task of managing data on the lower level. Files, folders and metadata are all well defined in this layer. But since file system implementation differs, some of these have to be reimplemented in the VFS. The individual file systems organize the data directly on the storage block device and therefore also store that data on the device itself. In comparison all VFS metadata is kept in memory and is never written to the underlying layers. Therefore the individual file systems offer a group of interface functions that overwrite some of the VFS generic functions. These functions are then called using function pointers from the VFS layer to operate on the file system. Some functions have the task of mapping the changes that happen in VFS metadata onto the individual file system metadata, which later could be written to the block device. On mounting the file system these functions are also used to fill in the VFS needed structures.

Most Linux books disregard the block layer in the IO stack representation and go directly into the device driver. Such disregard to the block layer can be found in [9] and in [8] to the extend that the first doesn't even contain a chapter under that name. Although explaining the device driver might be adequate to explain basic IO functions it does not represent the actual implementation. The block layer has the same task towards the different device drivers as does the VFS towards the different file systems. Therefore the device driver overwrites some of the generic functions written in the block layer. On the other hand the device driver runs mostly in the kernel space. Nonetheless some devices move implementations between kernel space running on available CPUs and the device hardware itself. This is demonstrated in Figure 3.1 by the overlap of the device driver with the space usually left for the device itself.

Although here the functionality of each layer is clearly defined, the details will show how difficult it is to follow the call graphs throughout all of these layers. In the next sections these layers will be explained in more details. The explanations are following the outline in found in [9], but is mostly done through observation and code reading. As previously mentioned it is difficult to find a single source for Linux code explanations. Not to mention how difficult it is to directly read a C code and try to decode all pointers and function names.

3.3 Virtual File System

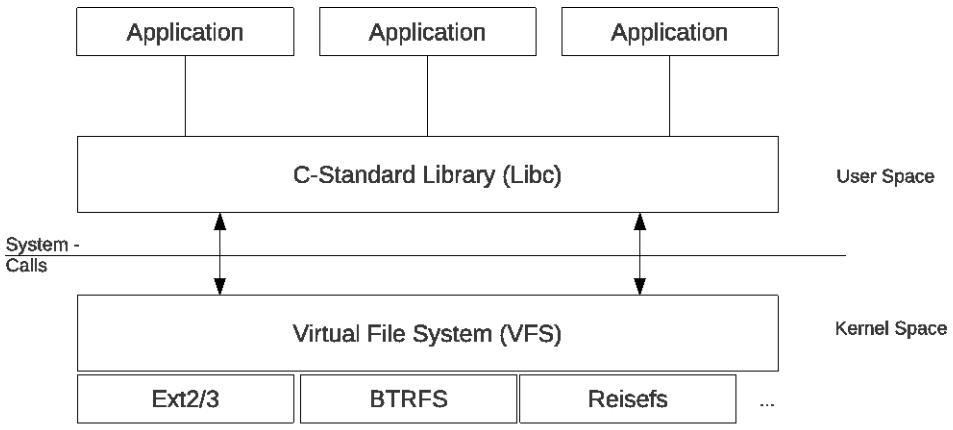


Figure 3.2: File system abstraction using VFS

As seen in Figure 3.2 the VFS has the task of abstracting the file system access for the above applications [9, Ch. 8]. This in turn enables the Kernel to support a large number of different file systems. This is done by providing uniform functions for above applications to manipulate and access underlying data. Although file systems are stored on a block device they have no control over direct manipulation of data on those device. This task is left to the block layer and is therefore hidden from the file systems and in turn from the VFS. Using this approach, theoretically changes can be made on one layer without effecting the other. Practically however on the interface between all layers is a heavy exchange of functions and parameters that need to be kept in mind while optimizing any give layer.

To achieve the function implementation on basic level the VFS needs to provide abstract function pointers, not only the functions but also the file system itself. The VFS therefore needs to mirror some of the file system components. The components of the VFS are shown in Figure 3.3. It therefore needs to show a uniform view of files and data that the applications need for execution of these functions. The VFS then extracts the needed data to implement the view of this unified files system. This fits almost every view of files, excluding files that have specific functionalities such as device files. These types of files cannot be supported on any other file system except Linux native file systems. This can be attributed to the fact that these files requires the underlying file systems to store additional data needed by the VFS to manipulate them. This in turn is not possible on a none Linux native file system.

It is important to know that the VFS is not a file system in itself. The VFS requires underlying file systems that carry out the functions that cannot be performed by the generic implementation of the VFS. Even attempting such a unified file system would defeat the purpose of simplicity for which the VFS has been written [9, Ch. 8]. Therefore the VFS acts only as an abstraction layer for the file system modules forwarding all real file system requests to the corresponding file system. The VFS uses a simple technique of presenting the information required for each file system in the form of *objects* or *structs*. These *structs* in turn contain function pointers defined by each individual file system to refer to the functions needed by VFS to carry out application requests.

Given that the Extended file system family are Linux's native file system, the VFS has to be specifically optimized for accessing Extended file systems [9, Ch. 8]. This can be seen from

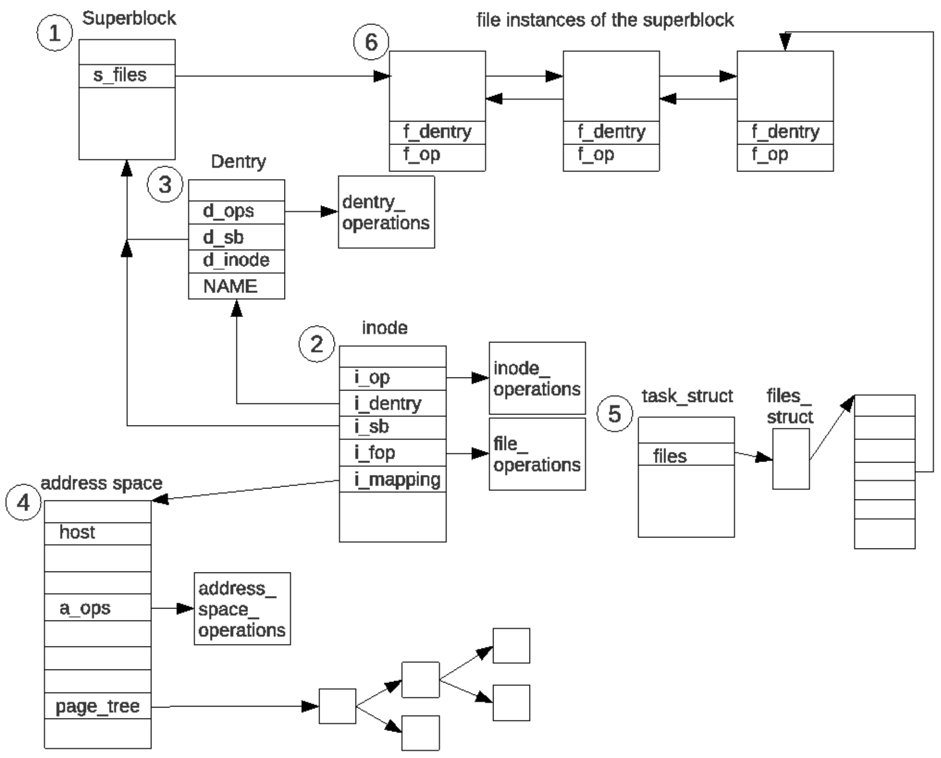


Figure 3.3: VFS components shown with pointers

the method by which the VFS organizes its *structs*. It also can be observed by the fact that the Ext2 file system uses mostly the VFS generic functions provided. This has to be kept in mind while editing the Extended file systems. Most Extended *structs* have the same name and almost the same configuration as those in VFS. Yet changing the Extended *struct* names will not be effecting the VFS *structs*.

3.3.1 VFS Components

Each process running in user space requires a list of all files opened for it. As seen in Figure 3.3-(5), the *task_struct* stores a file list for all opened files by this process [9, Ch. 8]. The *task_struct* is used by the Linux task scheduler to track running tasks and the relevant information. Therefore *task_struct* is not a direct component of the VFS, but needs to be mentioned here since it is a tool by which opened file users can be tracked. This becomes relevant for the process of opening and closing files. On opening a file the process is therefore handed a file descriptor. This integer is only valid for this process and is hence useless for any other process running at the same time. It also means that the Kernel can use the same descriptor for two different files opened by two different processes [9, Ch. 8]. The file descriptor is passed to the file on using the open system call. The process then uses this descriptor as a parameter for all functions requiring this file. The file descriptor in this case can also be used by processes as a parameter to system calls. Once the file is no longer in use the process should use the close system call. This in turn gives the file descriptor back and allows another file to get it.

Each file is however uniquely identifiable using an *inode*. An *inode* contains the relevant metadata and data segments or at least pointers to the data segments. The Kernel issues a unique integer to identify each file. The curious thing about *inodes* is that they don't contain the file name. Therefore the link is done through the unique Kernel identifier. To speed up *inode* access from the Kernel point of view, the Kernel keeps a global variable *inode_hashtable*. There are additional lists keeping track of the *inodes* within the Kernel. These mostly serve the benefit of defining the state of the *inodes* currently used by the processes [9, Ch. 8].

While file descriptors are used by processes to uniquely identify a file within the same process, the Kernel uses *struct file* for file identification. The file *struct* is shown in Figure 3.3-(6). These as seen are linked to the process file list and to the *superblock*.

It is worth mentioning here that everything under the Linux system is a file. Therefore directories are also files with their data segments containing entries representing the list of files contained within this directory. The entries refer to the *inode* number of the file or directory that exists inside this directory [9, Ch. 8]. Using this concept the file lookup becomes clear. A step by step operation traverses the directory *inode* for the next directory in the list. Once found the next directory *inode* is traversed. This continues until the intended file or directory is found. To speed up this operation for future lookups caches are used. This can be made obvious by researching twice through the same list on a Linux system. The second search is more faster and finds the target in less time. As a side note on accessing and searching for *inodes* it is worth mentioning that the optimization of *inode* access will not increase read or write time. These optimizations will only speedup the lookup operation. Nonetheless applications which deals with large numbers of files might need such optimizations. It might be of interest to compare this lookup technique with those implemented by other systems such as GPFS and other parallel or large file systems.

The *superblock*, shown in Figure 3.3-(1), contains required metadata for the mount point. This includes among others a pointer to the block device on which this mount point is stored. The *superblock* also contains a pointer to *struct file_system_type*. This is a structure that is unique for every file system that can be mounted and is registered by the VFS. The important part of *struct file_system_type* definition is shown below.

```
struct file_system_type {
    const char *name;
    int fs_flags;
    int (*get_sb) (struct file_system_type *, int,
               const char *, void *, struct vfsmount *);
    struct dentry *(*mount) (struct file_system_type *, int,
               const char *, void *);
    void (*kill_sb) (struct super_block *);
    struct module *owner;
    struct file_system_type * next;
    struct list_head fs_supers;
    ...
};
```

As seen the *file_system_type* contains function pointers that are needed by the VFS to mount or unmount a given file system. Any given file system needs to set this structure using its own function pointers. As an example the *file_system_type* of the Ext2 is shown below.

```
static struct file_system_type ext2_fs_type = {
    .owner      = THIS_MODULE,
    .name       = "ext2",
    .mount      = ext2_mount,
    .kill_sb    = kill_block_super,
    .fs_flags   = FS_REQUIRES_DEV,
};
```

These functions are then defined by the Ext2 such as *ext2_mount*. Another option is to use the predefined VFS generic functions such as *kill_block_super*.

Below are the main variables needed to explain the *struct super_block*.

```
struct super_block {
    struct list_head    s_list;     /* Keep this first */
    dev_t           s_dev;      /* search index; _not_ kdev_t */
    unsigned char       s_dirt;
    unsigned char       s_blocksize_bits;
    unsigned long       s_blocksize;
    struct file_system_type *s_type;
    const struct super_operations   *s_op;
    struct mutex        s_lock;
    int         s_count;
    struct list_head    s_inodes;   /* all inodes */
    struct list_head __percpu *s_files;
    struct list_head    s_files;
    struct list_head    s_dentry_lru;   /* unused dentry lru */
    int         s_nr_dentry_unused; /* # of dentry on lru */

    struct block_device *s_bdev;
    struct backing_dev_info *s_bdi;
    fmode_t         s_mode;

    const struct dentry_operations *s_d_op;
/* default d_op for dentries */
    ...
};
```

The *s_files* list is shown in the Figure 3.3 (6). Basic information for the mount point such as the blocksize or block device are also kept in the *superblock*. Additionally as seen the *superblock* keeps a list of pointers to *inodes*, files and *dentries*.

As mentioned previously, the *structs* used by the VFS are not the same as those used by the underlying file system. This is true even for Ext file systems which have almost the same representation. The fact that *inodes* during runtime require additional variables than those stored on the block device, makes the reason clear. Although the VFS uses this concept to support almost any file system, it is still reasonable to expect a delay from file systems that don't exhibit the same construction as the Linux native file systems. For these particular file systems the functions needed to gather the information to construct the runtime *structs* would be more complex than their Linux native equivalent.

Using runtime *structs* show that most of the operations taking place on *superblock* and *inodes* are done in the memory. For that reason the *superblock* has to keep track of all *dirty inodes* on a single list. This is another art of caching. *Dirty inodes* are not immediatly written back to the file system. On calling a synchronization function or unmounting the file system the Kernel can refer to the *superblock* for which *inodes* need to be written back. Additionally to keep track of accessed *inodes* each *inode* contains an access counter for counting number of processes currently accessing the file of this *inode*. On reaching zero the *inode* is placed on a least recently used list. This means that the *inode* could be removed from the memory as it might no longer be needed [9, Ch. 8].

Since file system are to a greater part kept on slow block storage media, the VFS uses a directory entry cache or for short *dentry*. The VFS therefore defines a *struct dentry* as shown in Figure 3.3-(3). These *dentry* objects are kept in the cache for all previously done lookup operations [9, Ch. 8]. This speeds up future accesses to the same files. Most operations used for maintaining the *dentry* cache is kept in *fs/dcache.c*. Cache *dentries* form a network and link to each other. During the lookup process for each traversed *inode* a *dentry* is created and linked to the previously traversed *dentry* which lies above it in the folder structure. The *dentry* therefore plays a crucial role in cache organization [9, Ch.8].

As a final note on caching, the *struct file* contains a readahead field. On marking this field the files data is readahead and stored in the cache. The consistency of cached files is guaranteed by *f_version* which is a variable in the *struct file*. Another pointer is stored in the *struct file* for the current position in the file. This pointer is used for readahead and sequential file reading.

3.3.2 VFS Operations

As previously mentioned, the VFS depends on the file systems overwriting the function pointers provided by the VFS. The VFS also gives the option of using generic functions. As seen in Figure 3.3 almost every *struct* contains a pointer to a list of operations. This even includes the *superblock*, which is not shown in the figure, but can be seen in the *struct super_block* definition. The *inode operations* are defined in the *struct inode_operations* for which the implementation is present in *include/linux/fs.h*. The list of possible functions is long. It is mostly contained in changing the variables in the *inode* itself. Therefore there are no read or write functions. Just those which manipulate the file metadata. As an example of *inode operations* the function *fallocate* which is used for preallocation, for which the support exits starting at Ext4 [9, Ch. 8]. This example shows how complex it is to gather all operations necessary for all file systems under one interface.

On the other hand, files rely on file system setting the *file operations* in the VFS layer. These operations mainly deal with the editing and manipulation of the data section of the file. The complete list of pointers for the functions is stored in *struct file_operations* which is defined in *include/linux/fs.h*. The following is a short list of some of the more important file operations with a short explanation. The complete list can be found in [9, Ch. 8].

read/write used for reading and writing file data.

aio_read/aio_write used for asynchronous read and write. In realty all read and writes are asynchronous. Synchronous reads or writes are done using aio_read or aio_write and waiting for the result of the read or write operation to be done.

fsync/fdatasync has the task of synchronization between cached data and data kept on the storage medium.

fasync signals processes that there has been a change in a file. This is important in case two different processes are accessing the same file.

readv/writev is used to read or write a vector for fast scatter-gather operations.

lock is used to lock a file for a process. The application have to be careful using such harsh synchronization methods. Locking a file that is needed by multiple processes might mean a massive reduction in performance.

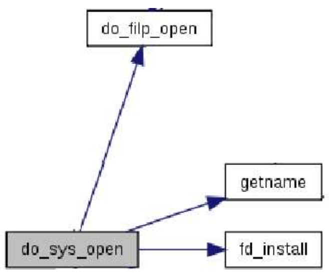


Figure 3.4: do_sys_open call graph

Figure 3.4 shows part of the function *do_sys_open* call graph, which is used by the system call open. The following describes part of this operation.

open is the system call invoked by the application. In [9, Ch. 8] the name of the system call is mentioned as being *sys_open*, this was not the case with the Kernel under observation. The system call almost immediately moves to calling *do_sys_open*.

do_sys_open takes care of the actual file opening. It finds an empty file descriptor within the process and then calls the function *do_filp_open*.

do_filp_open finds the *inode* related to this file using the traversing lookup. It then calls many other functions to take care of initializing the readahead *structs* and updates the *superblock*

fs_install is finally called by by *do_sys_open* to update the *task* of the calling process before returning to the user.

It is important for the process to invoke the close system call once it is done reading or writing the file. This allows another file to get the descriptor. Additionally each process is allowed by the Kernel a maximum number of opened files defined by a global variable. The Kernel can increase the number should the need for it arise [9, Ch. 8]. Yet this requires additional overhead that should be avoided if possible.

3.3.3 VFS File Read

To read a file another system call called *read* has to be invoked. The system call then checks the existence of the file using *fget_light*. Then under the condition of file existence the current position in the file is checked and the function *vfs_read* is invoked. Figure 3.5 shows the call graph of the function *vfs_read*. Since the read operation is file system dependent, the functions first checks for the existence of a read function by the underlying file system. This is done

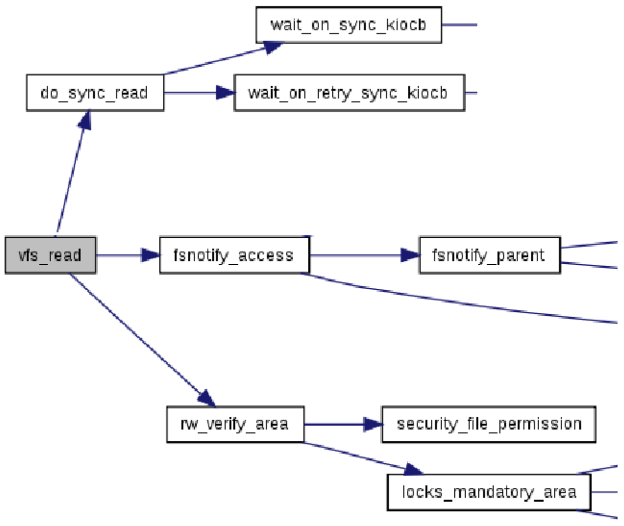


Figure 3.5: vfs_read call graph

by checking the value *file->f_op->read*. If the value is *NULL* the generic read VFS function *do_sync_read* Figure3.5 is invoked. Throughout this entire call chain a buffer pointer is handed from one function to the next. The buffer is finally filled with the needed data once the final read function is done. The issue of caching makes this simplified version far more complicated. The complicated caching function call graph will only be mentioned if a need for it exists.

The generic read routine *do_sync_read* invokes an asynchronous read function. Since this function is again file system dependent, *do_sync_read* invokes the corresponding function from the file operations. Therefore it uses *file->f_op->aio_read*. It has to be noticed here that the function itself is an asynchronous read function. To turn this into a synchronous read *do_sync_read* waits until *aio_read* is done. This particular function call method shows how complicated it is to trace such implementations. The function call graph given in Figure 3.5 does not contain any reference to *aio_read*. This is due to the function call being done via function pointers. Documentation tools such as doxygen cannot follow those function calls and therefore ignores them. The best method to trace this is to move directly into the code. In this case one has to move constantly between different parts of the Kernel, including moving to underlying file systems or even further into the device layers.

File systems can use the generic VFS function implementation for *aio_read*. This function can be found in *mm/filemap.c* under the name *generic_file_aio_read*. The function checks whether direct IO will be used by checking the value of the variable *O_DIRECT* which the application can set. For direct IO just a few lines of code are needed and most of the work is again delegated to the function pointer *a_op->direct_IO* which is set by the file system in *struct address_space* operations.

Another option is to try to access the file using cached data. This is called mapping read. In this case *generic_file_aio_read* finds *O_DIRECT* unset. This access is itself very complicated and involves a lot of function calls. The Kernel developers themselves admit to the difficulty if this implementation. Although the explanation found in [9, Ch. 8] tries its best to show how mapping read is done, it fails to fully explain the details. As an example in [9, Ch. 8] there is an explanation for the dependency of the mapping read function on a function called *do_generic_mapping_read*. This function could not be found in the Kernel under observation. To decrease complexity this study will focus on the direct IO. This is reasonable when trying to benchmark a system IO. Because of the fluctuation that the caching might bring into the benchmark results, a direct IO call is preferred.

The write operation is not that different and takes almost the same path. The main difference is that the process needs to fill the buffer with the data that will be written to the file. Most of the functions found here for reading are implemented with the same name for writing. Using that system *generic_file_aio_read* simply becomes *generic_file_aio_write*.

Important to know about operation pointers set by the file system implementation is that the none defined or *NULL* pointers allows the VFS to use its own generic functions. The file system developers have to be therefore aware during file system implementation about all possible VFS operations. That way they could either prevent or implement the operation calls that might be run by the VFS.

Most of the remaining topics concerning the VFS are implementations of the mounting or unmounting and registration of new file systems. Mounting is done using the system call *mount*. It is interesting to know that the VFS uses the file system implementation of a mount function. For example the Ext2 file system uses its own implementation *ext2_mount*. The VFS also needs the file system to provide a *fill_super* to the mount function. This functions has the task to gather the necessary information from its file system to fill the VFS *struct super_block*. Again file system developers have to make sure that the *struct super_block* was filled correctly or the VFS might not be able to continue using the file system. The unmount has another system call called *unmount*. Still the file system has to provide a *kill_sb* function which has the task of killing the *superblock*. The Ext2 in this case uses the VFS generic function *kill_block_super*. There are additional mounting and unmounting complex options. These include different mounting modes such as shared mounting, slave mounting, unbindable mounting and private mounting [9, Ch. 8]. These are options that could be kept in mind during block layer optimization.

It is essential to remember that the VFS explanation provided here is a simplified view. Details are provided on a need to know basis. Any details overlooked in this summery will be mentioned if necessary within the optimization and implementation sections.

3.4 File System

The Linux Kernel supports a multitude of diverse file systems. In [9, Ch. 9] the number is mentioned to be more than 40. It seams that in this case the Linux Kernel is unique in the amount of supported file systems. It has to be kept in mind though that most of the supported file systems cannot be used as boot file systems for the Linux Kernel. This can be simply explained by the different types of files that the Linux Kernel requires from a file system to store. In this case only the native Linux file systems can be used. These file systems are able to store the additional information required by the different types of files related to Kernel operation. As an example, the device file used by the Kernel to store device information can only be stored on Linux native file systems.

The main task of the file system is organizing the data on the storage device. It builds the link between the raw bit data that is stored on the hardware and the files. Even though the VFS construction might look the same as that of the file system the two have very different functions. For instance the VFS cannot access the files that lies in the file system underneath without using the file systems functions. While the VFS has to store data used during runtime, file systems have to store data that is needed for other purposes such as recovery. Therefore, although file systems such as Linux native file systems might have the same *struct* names, they don't contain the same variables. For the file system developers editing or optimizing the Linux native file systems, special care has to be taken whether dealt with *structs* belong to the VFS or to the file system. This is made in part more easier by the Kernel developers. Names in the file system begin with the file systems name. As an example, the *superblock* in Ext2 is called *ext2_super_block*.

File systems also have the task of preventing fragmentation of the underlying device. This is done because most of the file systems have been designed for the use on disk based devices. Since disks have a high seek time, which is the time to move the disk head to the required position, fragmentation means higher read latency. The overhead taken by computing the prevention of fragmentation is therefore justified.

Another issue that file systems have to take care of is consistency. Since devices can be removed at any moment, the file system has to ensure that the data in memory is consistent with data on device. If power drops during the write of an important part of the metadata the entire file system could be lost. For example writing data without updating the *inode* means that the data is forever lost. Even worse is a *superblock* write back that is could not be completed. This might lead to a full file system destruction. Journaling is one of the techniques that file systems use to provide consistency. The file system has to write every action that has been taken onto the device. That way the file system can undo any action that has not been successfully completed after the power is restored.

It is impossible to explain details on all file systems that the Linux Kernel supports. Therefore in this work only one file system was chosen. The Ext2 was the best candidate. Not only is the Ext2 simpler than other file systems, but also is well documented by different books. Additionally the Ext2 file system is the basis for Ext3 and Ext4 which are the following versions of the Ext2. Therefore most functions can be expected to be the same. Also the Ext2 does not implement any journaling [9, Ch. 9] which is an added benefit for SSD due to their limited write/erase cycles.

3.4.1 Ext2

Since Ext2 is intended for block devices it can only deal with data in the shape of blocks. The data for the Ext2 can only be kept in blocks and no smaller units of data can be saved without using up an entire block of data. This might lead to losing some space. Files that do not occupy an integer multiple of block, waste the remainder of a block that was not fully filled. An example can be seen in Figure 3.6. The file system therefore has to decide on a block size that is not going to waste too much space. Increasing the block size means wasting a lot of space when a file does not fill the block. On the other hand, if the block size is too small the amount of metadata including pointers and indirections means that a lot of administration has to be done. In addition to that small blocksizes lead to larger metadata that consumes a large part of the provided storage space. Therefore a optimum blocksize has to be found. Ext2 only supports a limited number of blocksizes [9, Ch. 9]. Selecting an appropriate blocksize is also related to the file sizes stored. If files are relatively large it is preferable to have a larger blocksize and vice versa. The blocksize is stored in the *ext2_super_block* defined in *include/linux/ext2_fs.h* using the variable *s_log_block_size*. The variable stores the log to the base 2 of a multiple of 1024. *s_log_block_size* can therefore only store the values 0, 1, and 2 giving the blocksizes $2^0 \times 1024 = 1024$, $2^1 \times 1024 = 2048$ and $2^2 \times 1024 = 4096$ respectively [9, Ch. 9]. Even though Ext2 communicates with the device on a block base, the method of the device storage has to be investigated. Some devices operate better with certain blocksizes. One example is alignment in case of SSDs. These perform better when the block can be stored on a single flash memory page.

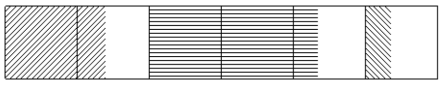


Figure 3.6: Storage is lost when using files that are not integer multiples of a block

3.4.2 Ext2 Storage Layout

Figure 3.7 shows the device layout in Ext2. The device is divided into block groups. Each block group stores a redundant *superblock*. In some later revisions the file system no longer stores one *superblock* per block group, but stores it on either odd or even block groups only [9, Ch. 9]. Having the *superblock* closer to the block group means less seek time for disk device. Redundancy also means the *superblock* cannot be easily lost. Here it appears that a further optimization of storage space can be made in respect to devices with low or no seek time such as SSDs. This could be performed by removing the *superblock* from all block group and using other means of backup. An additional advantage of low seek time devices is the possibility of sizing the block group with no fear of increasing the seek time. For devices with high seek time however increasing the block group means increased seek time from *inodes* to data blocks. As will be seen later from the reading process this can significantly increase latency.

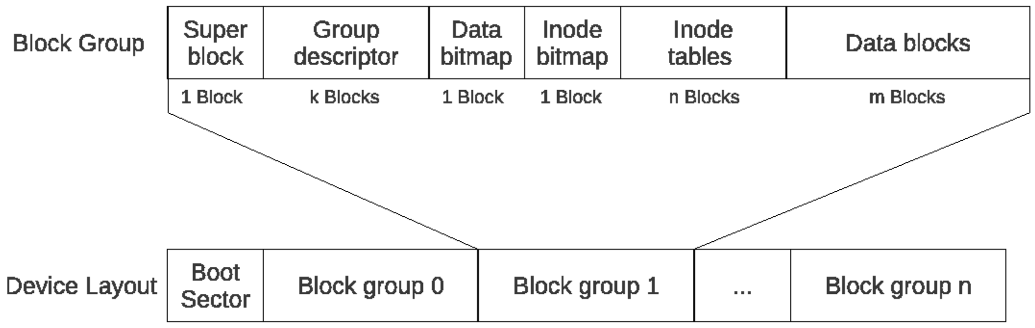


Figure 3.7: Block groups and device Layout in Ext2

The following is a short description of all elements in a block group [9, Ch. 9]:

Superblock is the main file system metadata. Contains all relevant information on the file system, including which block groups are empty, blocksize and current file system status. It also holds information on consistency with help of status variables. The Kernel only uses the *superblock* of the first block group. The rest are kept for backup or fast access.

Group descriptor Reflects the status of the block groups of the entire file system.

Data and inode bitmap contains one bit for each data block or *inode*. The bit indicates if the data block or *inode* is in use or free. A bit with value one means that the corresponding data block or *inode* is occupied.

Inode table contains all *inodes* of this block group.

Data blocks contains the actual data contents that are stored in the files.

The data structures used to define both the *superblock* and the *inode* in Ext2 are almost the same as those used in the VFS. The main difference is the name of the structures. The *superblock* is called *ext2_super_block* and the *inode* is called *ext2_inode* with both definitions found in *include/linux/ext2_fs.h*. Another important difference is the variables data type. Since the same file system can be used on different processor architectures, Ext2 has to define the variables in a bit format [9, Ch. 9]. Therefore all internal *struct* variables are defined using the *__leXX* data type. The data type stands for little endian with *XX* indicating the bit length. The Kernel then uses the architecture specific functions to convert these variables into the types needed by the processor architecture. This might mean a slight slow down on architectures using big endian when reading these variables.

It has to be kept in mind that during a read or a write the file system actual *superblock* is not used. As mentioned before in the VFS description, the *superblock* information are kept in the memory to speed up storage access. The file system has to therefore provide consistency and a frequent write back of changes made to the *superblock*. Also for mounting or unmounting operations done by the VFS to the file system it has to offer functions to fill or remove the data structures kept in memory. Since these functions are lengthy and only serve an administration factor in the file system they will not be further explained here. The required parts for implementation will be explained when needed.

Since the *inode* can only hold a limited number of pointers to data blocks, another method has to be found to increase the maximum size of a file. The *inode* employs indirection as a method of referencing more blocks [9, Ch. 9]. As Figure 3.8 shows, indirection means using a pointer to point to a block that contains pointers to data blocks. This means increasing the size of the file to $(blocksize/pointersize) \times blocksize$. This is also known as single indirection. To increase the file size even further double indirection can be used. This means using a pointer which points to blocks containing themselves pointers to blocks of pointers. These final pointers then point to data blocks. Using this method $(blocksize/pointersize)^2$ data blocks can be referenced. Eventually triple indirection has the same effect using three pointer levels. Although indirection appears to be quite simple to implement it will later be seen that it implies an overhead during the read or write process.

The number of direct data blocks is fixed in Ext2 to 12. There is only one single, one double and one triple indirection pointers. Therefore the position of each pointer is known in advance and Ext2 defines these into constants as seen below.

```
#define EXT2_NDIR_BLOCKS        12
#define EXT2_IND_BLOCK          EXT2_NDIR_BLOCKS
#define EXT2_DIND_BLOCK         (EXT2_IND_BLOCK + 1)
#define EXT2_TIND_BLOCK         (EXT2_DIND_BLOCK + 1)
#define EXT2_N_BLOCKS           (EXT2_TIND_BLOCK + 1)
```

EXT2_NDIR_BLOCKS defines the number of directly addressable blocks or direct pointers. These have an offset of *zero* to 11. The offset is relative to the beginning of data pointers within the *inode*. *EXT2_IND_BLOCK* is the single indirection pointer offset and has a value of 12. *EXT2_DIND_BLOCK* is the double indirection pointer offset and has a value of 13. Finally *Ext2_TIND_BLOCK* is the triple indirection pointer offset and has the value 14. *EXT2_N_BLOCKS* has a value of 15 and is the total number of pointers stored in a single *inode*.

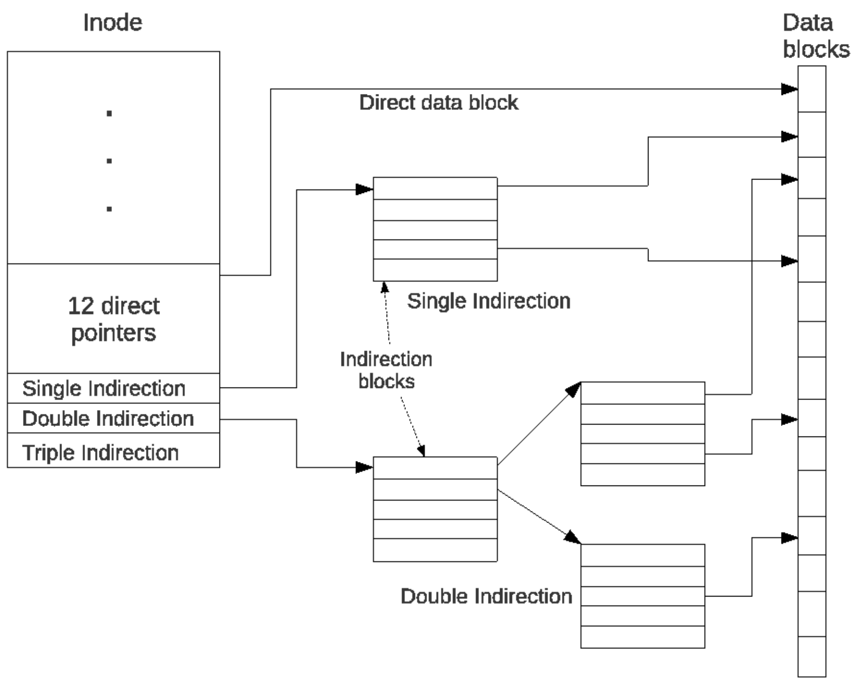


Figure 3.8: Indirection in Ext2

3.4.3 Ext2 Operations

As was mentioned in the VFS, the file system has to overwrite the functions given in the operation structures. These functions will later be referred to for carrying out the corresponding functions by VFS.

There are three operations that are defined by the Ext2:

file_operations are used to manipulate files, which include read and write functions. Their Ext2 definition is found in *ext2/file.c*. Part of the definition is shown below. As can be seen from the comment above the definition of the *file_operations*, most of the function pointers are defined as *NULL* to use the generic VFS defined functions.

```
/*
 * We have mostly NULL's here: the current defaults are ok for
 * the ext2 filesystem.
 */
const struct file_operations ext2_file_operations = {
    .llseek     = generic_file_llseek,
    .read       = do_sync_read,
    .write      = do_sync_write,
    .aio_read   = generic_file_aio_read,
    .aio_write  = generic_file_aio_write,
    .fsync      = ext2_fsync,
    ...
};
```

There is another *struct file_operations* defined in *ext2/dir.c* for directories called *ext2_dir_operations*. This is referred to when dealing with directories and is defined partially as follow.

```
const struct file_operations ext2_dir_operations = {
    .llseek     = generic_file_llseek,
    .read       = generic_read_dir,
    .readdir    = ext2_readdir,
    .fsync      = ext2_fsync,
    ...
};
```

inode_operations are mostly concerned with changing the *inode* variables. There are several definitions of *inode_operations* such as *ext2_file_inode_operations* defined in *ext2/file.c* or *ext2_dir_inode_operations* defined in *ext2/dir.c*.

address_space_operations are used for general address space manipulations. The *address_space_operations* form a connection between the file system and the block layer [9, Ch. 9]. The definition used in the Ext2 is shown below.

```
 const struct address_space_operations ext2_aops = {
    .readpage       = ext2_readpage,
    .readpages      = ext2_readpages,
    .writepage      = ext2_writepage,
    .sync_page      = block_sync_page,
    .write_begin    = ext2_write_begin,
    .write_end      = ext2_write_end,
    .bmap           = ext2_bmap,
    .direct_IO      = ext2_direct_IO,
    .writepages     = ext2_writepages,
    .migratepage        = buffer_migrate_page,
    .is_partially_uptodate  = block_is_partially_uptodate,
    .error_remove_page  = generic_error_remove_page,
};
```

3.4.4 Ext2 Read

To read data from an Ext2 file system the application has to use the corresponding system call as discussed in the VFS. The VFS then selects the function pointer from the implementations of operation given by the Ext2. Using the VFS *superblock* the device on which the file system lies is given to the functions that need this information as a parameter.

The main function that is used for reading or writing blocks to or from the Ext2 file system is *ext2_get_block* [9, Ch. 9]. As seen from Figure 3.9, which provides the caller graph for *ext2_get_block*, almost all other read or write operations depend on *ext2_get_block*. As seen from the function definition below the function requires a pointer to the *inode* of the file to read or write from, the sector to be read or written, a buffer to place the data into and an integer. The integer is called *create* and has a value of zero for read and anything else indicates a write. The *inode* handed over to the function is the runtime *inode* and not the one stored on the Ext2 file system. This can be seen from using *struct inode* and not *struct ext2_inode*.

```
int ext2_get_block(struct inode *inode, sector_t iblock,
    struct buffer_head *bh_result, int create) {
...
    int ret = ext2_get_blocks(inode, iblock, max_blocks, bh_result, create);
...
}
```

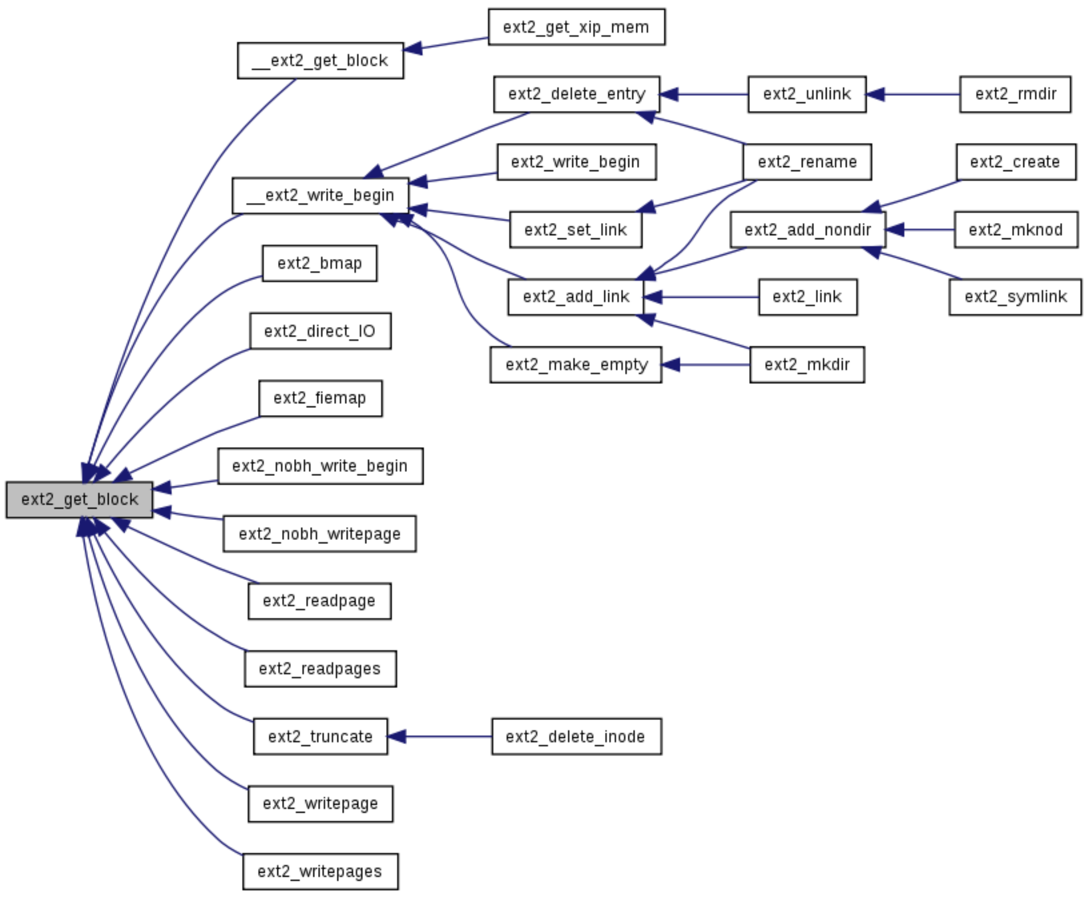


Figure 3.9: *ext2_get_block* caller graph

The function *ext2_get_block* has to be in the form of *get_block_t* provided by the VFS. That means it has to take the same variables as parameters and return the same data type [9, Ch. 9]. This is quite important as this function will be given as a parameter to a lot of other higher level functions in the VFS layer. As an example, the function *__blockdev_direct_IO* has a parameter *get_block_t* which *ext2_direct_IO* has to provide.

ext2_get_block is a front function that calls the function *ext2_get_blocks*, which is the function that carries out the block request. As seen in Figure 3.10 the function *ext2_get_block* does not do much. It simply calls *ext2_get_blocks*.

Figure 3.11 shows part of the complex call graph of *ext2_get_blocks*. The functions seen in the call graph are the ones needed for the Ext2 read operation.

The first function *ext2_get_blocks* calls is *ext2_block_to_path*. The function is used to find the path to the block in the indirection. It therefore returns an array of offsets for each indirection

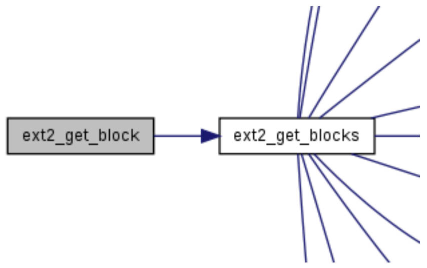


Figure 3.10: *ext2_get_block* call graph

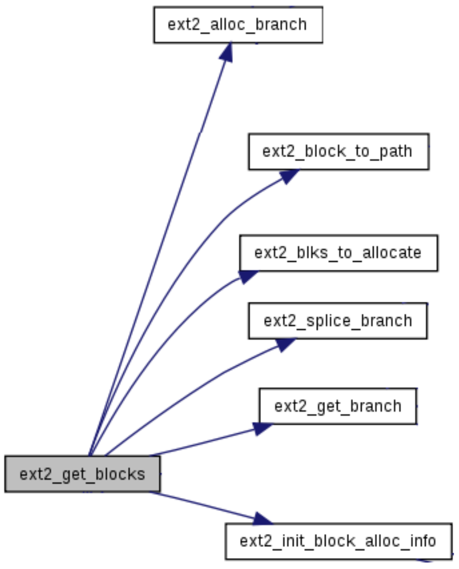


Figure 3.11: *ext2_get_blocks* call graph for read

level. Since *ext2_block_to_path* only needs to perform integer operations on the block number to find the offsets, no actual IO is done in this step. The block number is simply compared to the possible numbers of blocks in each level to find the offset. The number of indirection levels needed to address this block is then calculated by *ext2_get_blocks*.

Now that the offsets for each indirection level is known *ext2_get_blocks* can proceed to find the address of the data blocks to be read. This is done by calling *ext2_get_branch* which has the task of finding the physical address of the blocks involved in the indirection. It therefore returns a chain of triplets containing a key, a pointer and a buffer. These are a unique identification for each indirection and final data block on the physical level. If *ext2_get_branch* reaches the end of the indirection levels and finds a valid pointer it returns *Null*. This means that the *ext2_get_blocks* was called for a read request.

The important thing to remember here is that *ext2_get_block* does not read or write any data. The buffer is the returned data containing the physical address of the data blocks that are to be read or written. These physical addresses can then be used by the VFS layer to forward the request to the underlying block layer. The only function that a file system has is data organization and it therefore can only point the VFS as to where data should be read from or written to. This means that more complex read operations such as *blockdev_direct_IO* can use *get_block_t*, which the VFS matches to *ext2_get_block* on an Ext2 file system, to find physical block address.

3.4.5 Ext2 Write

On the other hand, requesting more blocks for a write operation from the Ext2 file system is a much more complicated process. In [9, Ch. 9] there are around four main tasks mentioned that the file system has to go through in order to supply more blocks.

- Number of indirection levels have to be found to address the requested block
- Free blocks are found and reserved
- Adding the blocks to block list of the file
- Finally block preallocation is used to enhance performance of next write operations

Performing these tasks for finding, allocating and preallocating new blocks is highly complicated. These however will only be explained in case of need for optimization. This has the added advantage of being able to compare original and optimized implementations side by side.

As was the case at the end of the VFS description, the Ext2 file system is complicated and contains many other features. Only the most important and needed functions and features has been explained here. The rest will be explained if required during optimization.

3.5 Block Layer

Most books consider this part to be simply an interface for driver modules. Others mention it briefly and do not go into details. As was shown in Figure 3.1 the driver lies in between the block device layer and the device space. As will be seen from the explanation of the block layer, the device driver has the ability to overwrite some functions provided. The device designers have therefore the option of implementing some of the device operations in the device driver. Another possibility is implementing the same device operations in hardware and directly placing it on the device. Hardware design means longer design time and higher device costs. On the other hand device driver implementation means higher CPU utilization. Therefore the choice of implementation is not just dependent on device design and device complexity, but also dependent on target architectures. As an example, Blue Gene contains multiple weak cores leading to a higher latency with device driver implementations.

Figure 3.12 shows the communications from user space to block layer [9, Ch. 6]. In this figure the user space is at the bottom. The VFS has the task of communicating with the user space. File operations defined by the file system are then used by the VFS to talk to the block device.

The Kernel has another type of devices called character devices. The main difference is that character devices can only communicate in a stream of data and cannot be randomly accessed. A keyboard for example is a character device. Thus keyboards cannot be accessed randomly. The interest is in storage IO. Therefore character device operations will not be explained.

As block device need to address data in the form of a block their is a requirement to define different chunks of data. Block devices have a limit for the size of the smallest addressable block of data. This is referred to as *sector*. Usually the size is 512 bytes. On the other hand the file system has to define its smallest chunk of addressable data which is usually defined as a block. Therefore the Kernel has to store a block size variable on *structs* representing the file system. Since the block device cannot offer smaller chunks of data, the block size has to be a multiple of sectors. Finally a page size is the data size which the memory management unit (MMU) uses. This is architecture specific [8, Ch. 14]. For most x86 architectures the page size is 4KB. On

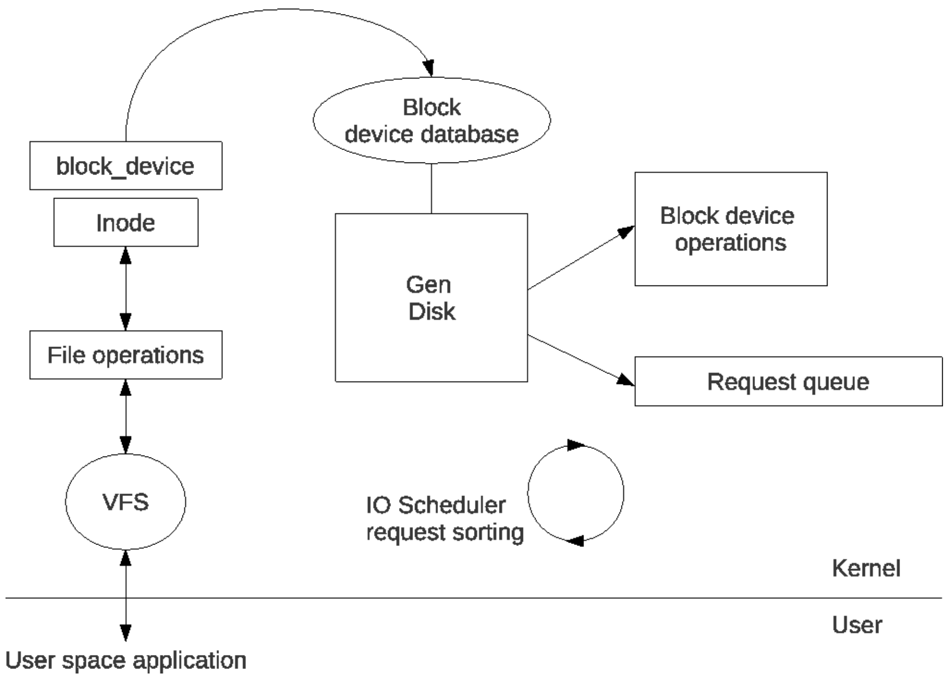


Figure 3.12: Block layer overview

the other hand some POWER architectures offer the possibility of choosing between 4KB and 64KB page size.

3.5.1 Block Device Representation

Under the Linux Kernel motto ”Everything is a file”, devices are represented using a file called device file [9, Ch. 6]. The Kernel also uses additional identification numbers for each device. This includes major and minor numbers. Important here is to know that every device can be uniquely identified by the Kernel. Also usually these unique identification numbers are stored in the structures that are relevant to the device. The Kernel also keeps track of every added device using a hash table this is called a block device database and is shown in Figure 3.12 [9, Ch. 6]. To add to this list of devices the Kernel uses *add_disk* which is defined in *block/genhd.c*.

Since devices are represented by files, normal read or write operations can be performed on such files. This also means that devices can be written to or read from without the need for file systems or mounting. This will be of further benefit when it comes to benchmarking overhead caused by file system in an IO operation. Considering that block devices can contain partitions, the Kernel creates a device file for each partition. As mentioned in the file system, the device files cannot be stored on none Linux native file systems. This is a direct result of the device files needing additional *structs* and data stored for their operations. In addition to normal read and write operations Linux has to provide special functions to manipulate devices. The Kernel therefore provides an IOCTL or Input Output Control interface [9, Ch. 6]. The IOCTL provide an interface for configuring and editing devices. The VFS defines IOCTL functions which are used by calling the corresponding system calls.

The representation of devices in the form of a file means that the VFS needs pointers to the file operations. These are usually provided by the file system. Yet device files are special files on which the file system, even Linux native file systems, file operations cannot operate. Therefore the block layer has to define special file operations for the block device. Below is part of the file operations specific for the device files. These file operations are defined in *fs/block_dev.c*.

```
const struct file_operations def_blk_fops = {
    .open       = blkdev_open,
    .release    = blkdev_close,
    .read       = do_sync_read,
    .write      = do_sync_write,
    .aio_read   = generic_file_aio_read,
    .aio_write  = blkdev_aio_write,
    .mmap       = generic_file_mmap,
    .fsync      = blkdev_fsync,
    .unlocked_ioctl = block_ioctl,
    ...
};
```

The function *init_special_inode* then defines the file operations for the *inode* of the device file as shown below. The function simply selects the appropriate file operations from the file mode.

```
void init_special_inode(struct inode *inode, umode_t mode, dev_t rdev)
{
    inode->i_mode = mode;
    ...
    } else if (S_ISBLK(mode)) {
        inode->i_fop = &def_blk_fops;
        inode->i_rdev = rdev;
    ...
}
EXPORT_SYMBOL(init_special_inode);
```

Additionally the block layer has to define address space operations for the device file which can also be found in *fs/block_dev.c*. This and the previous file operations definition show how difficult it is to differentiate between the layers in the Kernel. Although it is considered that these definitions are part of the block layer, their implementation is found in the file system folder of the Linux Kernel.

The device layer also defines another group of operations specific for devices called *block_device_operations*. These can be found in *include/linux/blkdev.h*. Part of the *struct* definition can be found below.

```
struct block_device_operations {
    int (*open) (struct block_device *, fmode_t);
    int (*release) (struct gendisk *, fmode_t);
    int (*locked_ioctl) (struct block_device *,
      fmode_t, unsigned, unsigned long);
    int (*ioctl) (struct block_device *,
      fmode_t, unsigned, unsigned long);
    int (*compat_ioctl) (struct block_device *,
      fmode_t, unsigned, unsigned long);
    int (*direct_access) (struct block_device *, sector_t,
                        void **, unsigned long *);
    int (*media_changed) (struct gendisk *);
    struct module *owner;
    ...
};
```

Each device driver can then define its own function pointers to overwrite these functions. As an example the *struct block_devic_operations* for the loop device is shown below. Any operation that is not defined or has a *Null* value means either generic functions will be used or this option is not defined by the block device.

```
static const struct block_device_operations lo_fops = {
    .owner =    THIS_MODULE,
    .open =     lo_open,
    .release =  lo_release,
    .ioctl =    lo_ioctl,
#ifdef CONFIG_COMPAT
    .compat_ioctl = lo_compat_ioctl,
#endif
};
```

3.5.2 Block Layer Components

As seen in Figure 3.12 the *gendisk* has an important role in the communication with the device. Part of *struct gendisk* is shown below. The *struct gendisk* keeps track of the entire device which makes it an additional abstraction level on the generic level. Hence the name generic disk [9, Ch. 6]. Additionally as seen below the *struct block_device_operations* of the block device has a pointer within the *struct gendisk*. These are then linked to the ones defined by the device driver. It is important to note here that these operations should not be invoked directly. Instead the file operations or *file_operations* should be used. In turn the file operations will invoke the necessary *block_device_operations* [9, Ch. 6].

```
struct gendisk {
    int major;          /* major number of driver */
    int first_minor;
    int minors;

    struct disk_part_tbl *part_tbl;
    struct hd_struct part0;

    const struct block_device_operations *fops;
    struct request_queue *queue;

    atomic_t sync_io;       /* RAID */
    struct work_struct async_notify;
    ...
};
```

The most important variable present in the *struct gendisk* is the *struct request_queue*. Any communication with a block device is done using a single request queue. This is known as request queue management. It has to be noted here that, while the Kernel has control of the caches, manipulating the queues is done by the block layer [9, Ch. 6]. Part of the definition of *struct request_queue* is shown below.

```
struct request_queue
{
    struct list_head    queue_head;
    struct request      *last_merge;
    struct elevator_queue   *elevator;

    request_fn_proc     *request_fn;
    make_request_fn     *make_request_fn;
    prep_rq_fn          *prep_rq_fn;
    unprep_rq_fn        *unprep_rq_fn;
    unplug_fn           *unplug_fn;
    merge_bvec_fn       *merge_bvec_fn;
    prepare_flush_fn    *prepare_flush_fn;
    rq_timed_out_fn     *rq_timed_out_fn;
    dma_drain_needed_fn *dma_drain_needed;

    sector_t        end_sector;
    struct request      *boundary_rq;

    struct timer_list   unplug_timer;
    int         unplug_thresh;  /* After this many requests */
    unsigned long       unplug_delay;   /* After this many jiffies */
    struct work_struct  unplug_work;
    ...
};
```

The *request_queue* contains pointers to functions which can be set by the device driver. These include *make_request_fn* which is used to add a request to the queue. Any communication with the device is done using the *request_queue* functions. Using a request queue is important for performance. The overall performance would drop if there was no queue in case of disk communication. To prevent long seek times the Kernel employs an IO scheduler, whose task is to reorder and merge requests. The main task of the IO scheduler is an overall better performance. This might result in some tasks being delayed more than others [8, Ch. 14]. The IO schedulers are also known as *elevators*. To give the IO scheduler enough time to reorder or merge requests, the Kernel plugs the request queue after inserting a request. Plugging the queue means stopping the device driver from carrying out the requests. Once sufficient time has passed or enough requests have been made the queue is unplugged. The functions and variables needed for this operation can be found in *struct request_queue* shown above.

Another important data structure shown below is *struct request*. The task of the function *make_request_fn* is to fill a request and add it to the queue. Older Kernel versions used to admit requests in the form of buffers. This meant that a request could only contain a single block on the device. To improve performance the Kernel now admits requests in the form of a *struct bio* [8, Ch. 14]. BIO stands for Block IO.

```
struct request {
    struct list_head queuelist;

    struct request_queue *q;

    int cpu;

    unsigned int __data_len;    /* total data len */
    sector_t __sector;      /* sector cursor */

    struct bio *bio;
    struct bio *biotail;

    struct gendisk *rq_disk;
    ...
};
```

A *bio* shown in Figure 3.13 contains multiple vectors each containing pointers to different physical pages [8, Ch. 14].

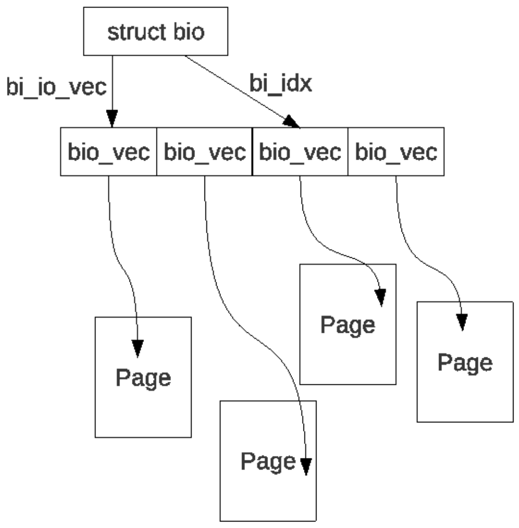


Figure 3.13: Representing requests in the form of a BIO

The final structure that the Kernel needs to represent a block device is *block_device*. It is mostly used by the VFS layer. The *struct block_device* represents partitions as well as complete block devices. The relation between *struct block_device* and *struct gendisk* is shown in Figure 3.14. In the figure there is a single *struct block_device* representing the complete block device. Each partition on the device is also represented by a separate *struct block_device*. The connection between *struct gendisk* and the different partitions is done using a list called *part*. This contains *hd_struct* which points to the *block_device* of each partition [9, Ch. 6]. Additionally each *block_device* contains a pointer to the *struct gendisk* that represents the block device it is stored on. This can be seen in the *struct block_device* definition shown below.

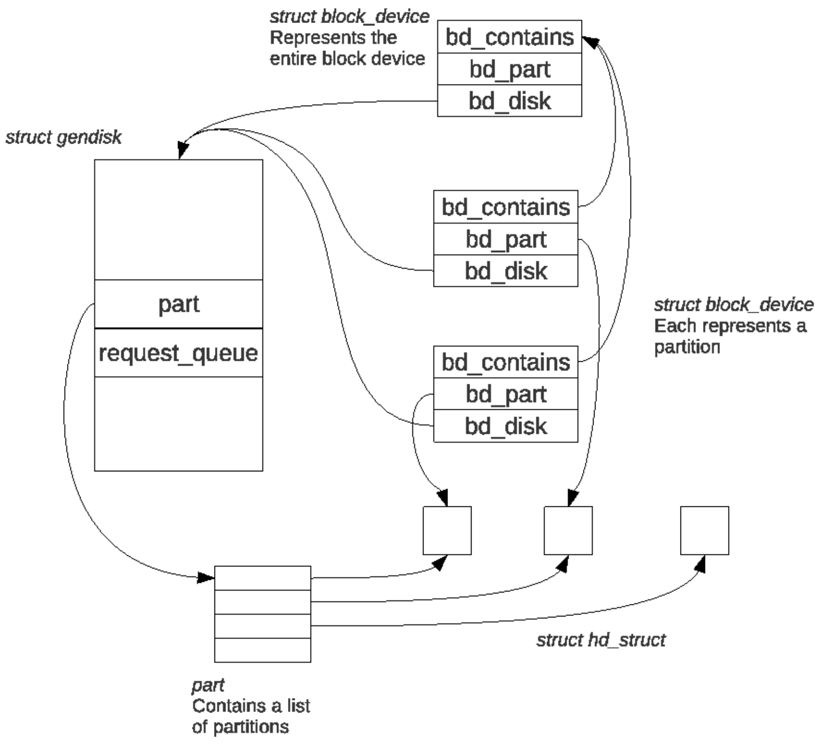


Figure 3.14: Connecting block layer data structures

```
struct block_device {
    dev_t           bd_dev;  /* not a kdev_t - it’s a search key */
    struct super_block *    bd_super;
    int         bd_openers;
    struct mutex        bd_mutex;   /* open/close mutex */
    struct list_head    bd_inodes;

    struct block_device *   bd_contains;
    unsigned        bd_block_size;
    struct hd_struct *  bd_part;
    ...
};
```

3.5.3 Submitting Requests

The final point that has to be made about the block layer is how the VFS can submit a request. The file system controls the method by which the access is going to take place. To make a simple example, the direct IO call graph could be followed. As seen in Figure 3.15 the main function called by the direct IO access of a file system is *__blockdev_direct_IO*. The numbers in Figure 3.15 show the order of relevant function execution. From the figure *__blockdev_direct_IO* calls *direct_io_worker* (Figure 3.15 (1)) which in turn calls *do_direct_IO* (Figure 3.15 (2)). This function has to take care of where the blocks are to be read from or written to. Therefore *do_direct_IO* has to call *get_more_blocks* (Figure 3.15 (3)). The VFS decides on the appropriate file system *get_more_blocks* implementation. The file system only provides the blocks physical addresses will be read or written. The next steps of the call graph have the target of creating a *struct bio* that will be finally admitted to the lower block device levels using *submit_bio* (Figure 3.15 (4)).

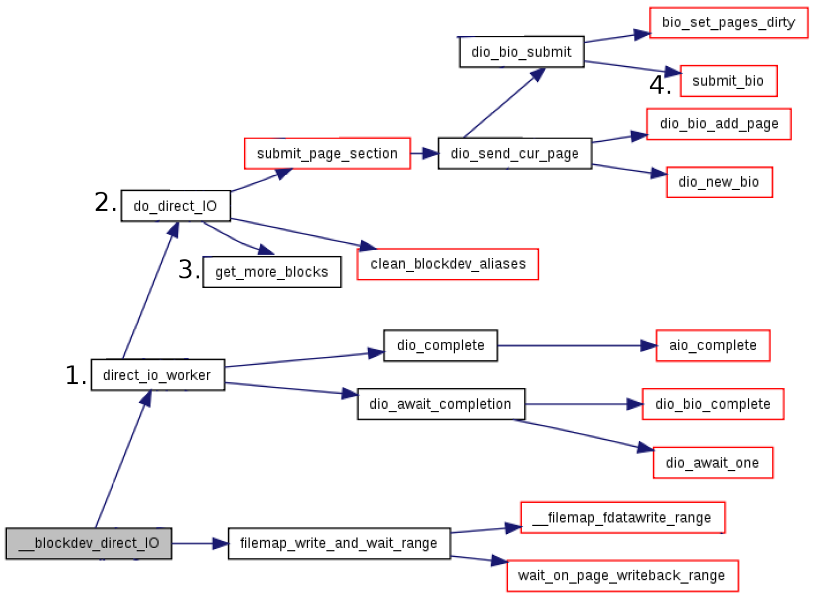


Figure 3.15: *__blockdev_direct_IO* call graph leading to *submit_bio*

The function *submit_bio* is the main function used in communicating with block device. Part of the call graph is shown in Figure 3.16. *submit_bio* calls *generic_make_request* providing it with the *bio*. As the *generic_make_request* function takes only the *bio* as a parameter, *submit_bio* has to mark the *bio* as being either a read or write request. It is important to know that the only function that calls the *generic_make_request* is *submit_bio*. This means that no request can be submitted to the block layer without the use of this function.

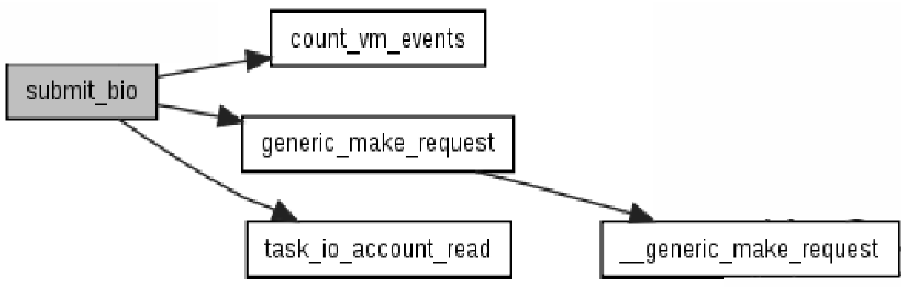


Figure 3.16: submit_bio call graph

The next step is for *generic_make_request* to call *__generic_make_request* as seen in Figure 3.16. The reason for this complicated call scheme is the amount of accounting that has to be done to admit a single request. Another problem that complicates the implementation of *__generic_make_request* is the possibility of recursive calls. Due to the limited stack available for the Kernel, the *generic_make_request* has to limit the number of recursive calls done using *__generic_make_request* [9, Ch. 6].

__generic_make_request finds the appropriate *struct request_queue* to which the request has to be made. The request queue has a pointer to the *make_request_fn* which was set by the driver. The block layer provides a generic *make_request* function. Yet some driver prefer to use their own implementation. The reason is usually the different approach that these drivers take to writing to the underlying device. For example, while schedulers and queue plugging is useful for a disk device, a *ramdisk* would not benefit from these methods. In fact the *ramdisk* implementation found in *driver/block/brd.c* provided by the Linux Kernel does not use any queuing system. The implementation of the *make_request_fn* for the *ramdisk* carries out the request as soon as it arrives by calling *memcpy*. Device drivers have to set the *make_request_fn* used by calling *blk_queue_make_request* which takes the request queue and the *make_request_fn* as parameters. The function *blk_queue_make_request* links *make_request_fn* to the request queue of this device.

In case of disk based devices the *make_request_fn* would use the elevator functions provided by the block layer to find the appropriate position to insert the request into the queue. The elevator functions represents the IO scheduler operations. This is done by calling the functions found in *struct elevator_ops*. The definition of the elevator operations is done based on which scheduler is used. It is important to give the scheduler sufficient time to merge and reorder the requests. Therefore the *make_request_fn* has to implement the queue plugging. This will prevent the queue requests from being carried out until the scheduler had enough time to perform reordering and merging of requests. After the timeout, which by default is 3 milliseconds, request execution is allowed.

The final step in a request is the execution. This is done by calling *request_fn*. This function however is device dependent and therefore irrelevant to the discussion here. Additionally most devices taken into consideration in this study do not contain a *request_fn*. This is because these drivers try to bypass the queue and schedulers provided by the Linux Kernel. One example was mentioned before is the *ramdisk*. The *ramdisk* simply carries out the request as soon as it is made. Therefore there is no need for *request_fn*.

As has been the case with the former Linux Kernel IO stack layers, the file system and VFS, the block layer is complicated and too lengthy to describe in full. Only components and functions that are at the heart of the block layer were mentioned. In case needed, additional parts and components will be described in the implementation.

3.6 Storage Class Memory and Hard Disks

Few years ago the choice for storage medium was limited to either fast but very expensive RAM or cheap but slow HDDs. Given that RAM is a volatile memory that would require long life batteries to support the storage unit, the choice was practically limited to HDD. The main problem with HDD is the mechanical component. The concept of rotating disks can only be realized with addition of motors. This meant that there is a physical limitation to the speed by which such devices can operate. An additional limitation for HDD is the use of a single request queue. HDDs cannot support more request queues since only a single head can be packaged into the device. This meant that the storage can only handle requests in serial and never in parallel. Considering that the amount of storage on single HDD was constantly increasing the time to read all the data present on the storage also increased. Although IO links also increased their speed it quickly became obvious that storage will be the new system bottleneck. Servers and large storage systems tried to increase speed by running several HDDs in parallel. Yet as the gap between CPU and IO widened more and more HDDs had to be added. The system grew in complexity and the overhead for managing the large number of HDDs became visible.

Due to the inherit limitations of HDDs developers were looking forward to a new technology. The flash technology had already been invented in the late 80s [10]. Flash presented properties that HDD could not provide. The ability to be programed, erased and read electronically represented a huge advantage. Additionally being none-volatile meant that no external power is needed to keep the data. Despite all the advantages flash was not adopted as a storage device until the beginning of the 21th century. Some reliability issues which will be later explained prevented the immediate adoption. To increase the reliability many different inventions and algorithms had to be created. Also the amount of stored data was limited by the transistor size [10]. Therefore the flash devices had to wait till the transistor size allowed for a viable storage to be built using this technology.

Figure 3.17 shows how data is stored in a flash device [11]. A floating gate is suspended between the gate and the substrate. The target is to store data by trapping electrons in the floating gate using tunneling. The gate is considered erased and containing the value 1 when there are no electrons present on the gate. On the other hand if the floating gate contains trapped electrons it signals the value 0. This means that all units contain the value 1 until a write operation changes it to 0. The write is done by applying high voltages to the gate which allows electrons to tunnel from the substrate to the floating gate. An erase takes place by applying opposite high voltage to allow the trapped electrons to tunnel back to the substrate [11]. Reading the value is done by applying the normal voltage level on the gate and detect the formation of a channel in the substrate. In case of having trapped charge on the floating gate the channel would either be not formed or weakly formed. The circuit can therefore conclude that a 0 was written to the circuit. If there is no charge trapped on the floating gate, the transistor will function as a normal transistor and a 1 can be detected.

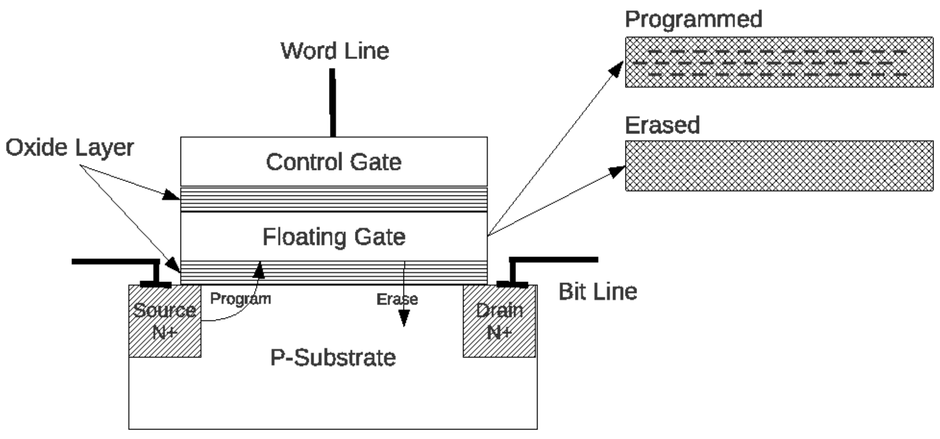


Figure 3.17: Floating gate in flash memory

There are two different methods in which the flash transistors can be grouped together. The method of the cell interface suggest how the data is read from the flash memory. The first method is a NOR construction. The advantage of a NOR flash memory is that it can be read and written one byte at a time. The write however has to assume a previous erase [12]. However the complexity of the NOR circuit means that the storage density of such a circuit is limited. That is why NOR can be applied in other more smaller units of storage. NOR is mostly used in embedded systems [12]. Since the NOR flash is not suitable as a massive storage unit it will not be further investigated.

The second possible cell interface is the NAND which is shown in Figure 3.18 [11]. The main difference to NOR is that the NAND cells can only be accessed in a page or block construction. Also NAND promised higher storage density which made the NAND more suitable for storage than NOR. As seen in Figure 3.18 the cells take the form of an array. Each *wordline* contains 4K pages. Each page contains additional spare bytes. The *bitline* runs across 64 pages. The figure represents an 8Gb SLC 50nm flash [11].

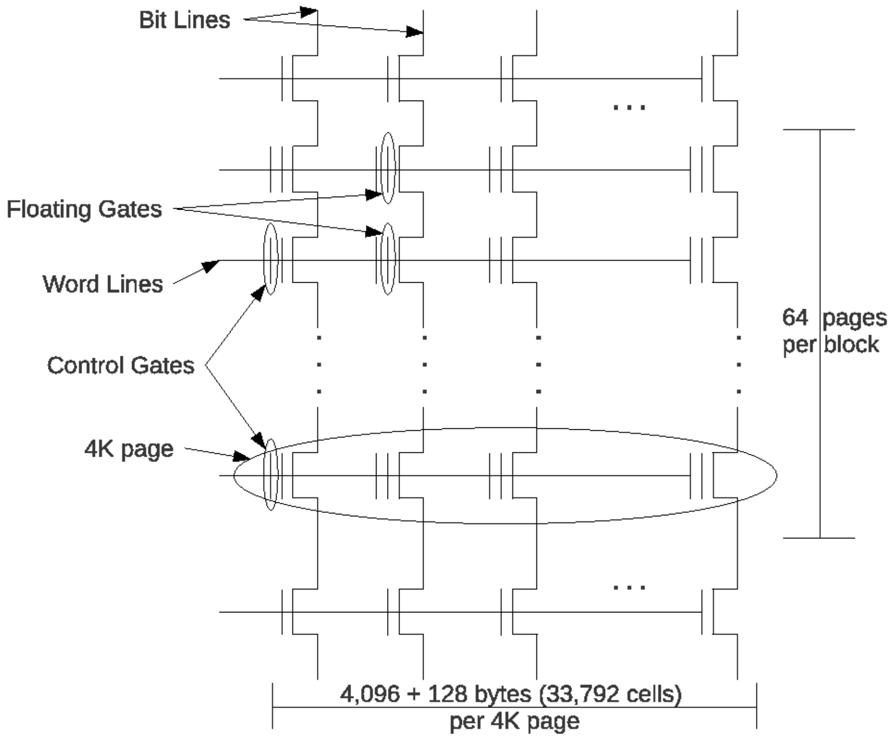


Figure 3.18: NAND flash cell architecture

3.6.1 Flash Memory Operations

The flash memory allows only three basic operations. These are read, erase and program. The NAND cells layout limit the minimum number of cells to be operated on. The read operation can be done in random and the size is not limited. This shows the most important advantage of the flash, which is the high speed random access. In comparison the erase can only be done on a complete block. The target is to drain all floating gates. This means that all bits are erased to a value of 1. The reason for being only able to write one block at a time is the shared *wordlines* and *bitlines*. The difficulty with erasing a complete block lies with the high voltages required. These stress the block leading it to fail after a limited number of erases. The final operation is cell programming which is used to write a 0 on a flash cell. The NAND layout allows for programming only a single page at a time. Furthermore programming can only occur on a previously erased block. This is because the floating gate charge cannot be drained by the programming operation. Another important aspect of writing is that it can only be done on a whole page. Additionally a block must be programmed sequentially one page at a time but not at once. Further details on the read, erase and program operations can be found in [10], [11] and [12].

There are two main types of flash memory the Single Layer Cell or SLC and the Multi-Layer Cell or MLC. In a SLC each flash transistor can only contain a single bit. In comparison a MLC flash transistor can contain more than one bit. This is achieved by using different voltage levels to signal different values. Both SLC and MLC use the same voltage range. Yet the SLC can only detect two values with a wide voltage threshold. On the other hand MLC uses smaller thresholds to divide the voltage range into four or more values. That means that a 2x MLC can store 2bits instead of one. The storage density is therefore effectively doubled for MLC compared to SLC. The main problem however with using MLC is the sensitivity of detecting the different values. This means that the circuit for reading and writing is more complex. MLC also requires several write operations to achieve the correct voltage level. Additionally reading with this high voltage sensitivity means a higher error rate. Therefore MLC has orders of magnitude lower write/erase cycles than SLC. In fact many sources report 100,000 write/erase cycles for SLC and only 10,000 write/erase cycles. This main difference makes SLC more suitable for enterprise applications where heavy read and write will strain the flash devices. However MLC is more suitable for consumer products that allow for a higher error rate and a shorter product life. A more detailed comparison between SLC and MLC can also be found in [10], [11] and [12].

3.6.2 Flash Memory Reliability

The main reason for the late adoption of the flash memory is the reliability challenges observed in the technology. As has been mentioned before each block needs to be erased before it can be rewritten. The erase and the write operations require high voltages which strain the floating gate construction. The cell will therefore eventually fail due to trapped electrons in the oxide layer. Another factor that leads to failure is the break down in the oxide structure. The oxide layers are pointed to in Figure 3.17 [11]. The damage caused cannot be fixed. Due to that the cells have a limited number of erase/write cycles.

Another reliability factor of flash is the data retention limit. Due to leakage the floating gate cannot retain the charge indefinitely. Therefore the storage will eventually lose all data. Manufacturers target a data retention period of 10 years. This might be sufficient for most application. Despite that the system has to be prepared for data retention errors. The reason is that the data retention is dependent on the already performed write/erase cycles. Therefore more frequently used blocks have a shorter data retention period [11]. Not only does the system prepared for blocks failing during runtime, there are also bad blocks that are delivered in a newly produced flash chip. These are guaranteed to be no more than 2% by the manufacturer [11]. Bad blocks are the result of the complex manufacturing cycles that flash chips have to go through.

Not only does the flash units contain failing blocks due to limited write/erase cycles, but also reading and writing causes disturbance in neighboring bits. A read operation can flip unintentionally other bits. The risk however is higher with a write operation. The reason is that write uses higher voltage than read. Also several read operations on the same bit might lead to draining the trapped electrons in the floating gate. This effectively removes the written data. Write on the other hand has a higher chance of disturbing charges on other neighboring floating gates

[12]. The system has to detect these bit flips and fix the error. This is more complicated than it looks. Given that a value can only be written back to the bit after erasing the whole block, the bit flips cannot be simply rewritten. This indicates that blocks which contain a lot of write and read operations will have to be refreshed after a given time. Until the block is refreshed the system will have to endure the errors during runtime. The read and write disturbs have more effect on MLC. This is because the MLC uses lower voltage thresholds between bit values.

3.6.3 Increasing Reliability in SSDs

In order to increase reliability the flash storage card or solid state disk (SSD) has to employ a wide range of techniques. This is crucial for using NAND flash as a storage unit. The first and far most important issue that has to be addressed is the limited write/erase cycle. File system designed for HDD are allowed to write to the same address as often as necessary. In a flash that would mean destroying a group of pages or blocks more rapidly than others. This cannot be allowed, as it might lead the device as a whole to be destroyed. SSD manufacturers have therefore resolved the issue by use of wear leveling. The target of wear leveling is to use all blocks on the SSD equally. This can be practically achieved using one of two possible ways. The first method is using SSD specific file systems. These file systems will have to be responsible for dividing the load across all existing blocks [13]. This however has the drawback of preventing the use of SSD with conventional file systems. Therefore the second method is more widely used. In this method the SSD itself becomes responsible for wear leveling. On these basis blocks cannot be grounded to a specific address since the SSD is allowed to move blocks around. Conventional file systems however require a stable address space. Thus the SSD employs a Flash Translation Layer or FTL. The idea is that the file system would see a unified address space. The SSD controller on the other hand would be allowed to place blocks where ever the wear leveling algorithm requires. The controller then records the physical address in a the FTL and links it to the address seen by the file system. Although this might require additional overhead, it is commonly used due to its backwards compatibility with HDD. Additional information on wear leveling can be found in [11] and [10].

Wear leveling algorithms can choose between two different approaches. The first is called static wear leveling. On placing a new page into the SSD the controller only checks to find the least used free page. The second approach is called dynamic wear leveling. In this case the controller checks for the least used page among all free and used pages [12]. By moving data around the controller can achieve higher wear leveling. Most SSD manufacturers are not satisfied by the added product life time using wear leveling. To increase the life time of the product even further a method called over-provisioning is used. Most SSDs contain more storage than is reported. The controller therefore has added space for further wear leveling. This allows the controller to retire some blocks once they have reached a certain error level [10].

As mentioned earlier flash devices are error prone. SSDs therefore have to overcome this limitation. Most SSDs incorporate strong error correction and detection techniques. The target is to discover any error that might have occurred during the read process. The SSD has to carry extra data to support these mechanisms. However there is a limit for how much error can be detected and corrected. This limit is determined by the used algorithm [11]. On the other hand a write failure can be easily detected by the controller. The write then is repeated on the same or on a different pages. Pages that result into too many write or read failures are retired. Each page carries bits that can be used to mark the page or block as bad or defect [10]. The controller then has to avoid those blocks for any future write operations.

Each page has to contain additional data for error detection and correction mechanisms. This means that writing a certain amount of data into a SSD might result in writing more than was intended. This is called write amplification [11]. In fact write amplification is worse than first realized. A write failure might result in having to repeat the write operation and mark the failed block as a bad block. The difficulty becomes more visible when noticing that a page cannot be written without being erased first. In that aspect if a write was intended for a certain data unit within the page the whole page has to be repeated in another block. Another option is to erase the whole block and rewrite it again in the same location. Since an erase cycle has a high latency SSD controllers try to avoid them as much as possible. The SSD therefore opts for using a different page instead of updating the old one. The old page is then marked for deletion [11]. This technique is called garbage collection. Performing the deletion at later time allows for usage of idle time during which the controller is not fully utilized. However if the controller uses up all of the freshly erased blocks a costly erase cycle will have to be performed. SSDs might therefore have a different performance for different conditions.

The complexity of the controller of a typical SSD targets mainly the increase of endurance. The increase of life time of SSD is an important factor in Using this technology as a storage device. The more mechanisms are used to increase reliability the further the life span of a flash device can be extended. For example, by using proper wear leveling the RamSan-500 can sustain constant writes at full bandwidth for up to 3.25 years. By using additional units and tolerating higher block failure rate the RamSan-500 can sustain the constant write at full bandwidth for more than 15 years [10]. This shows how much the usage of different techniques can make SSD endurance comparable to that of HDD. Since SSDs are more robust than HDD the system is promised as a whole a higher reliability by overcoming SSD challenges.

From the previously mentioned reliability issues it becomes obvious that SSD must be handled in a different method than HDD. The problem however is that systems have been primarily designed to operate with HDD. One of the major issues is the single queue operation of HDDs. In comparison a SSD operates with a wide range of inherit parallelism. The reason for the needed parallelism in building such SSDs is the limited single flash chip performance. A single chip is divided as shown in Figure 3.19 into several Dies. The chip as a whole shares data pins [10]. The chip therefore gives a preset limitation for performance. However by the use of complex controllers an SSD can combine the usage of multiple NAND chips to increase performance. The inherit included parallel operation of a SSD can process more than one operation at a time. Yet applications and operating systems have not been designed for that purpose. In fact most applications use sequential reads and writes. An application is suspended until the IO operation takes place preventing it from creating new requests to be processed in parallel.

The system design for HDD has other implications. The most critical design aspect is seek time for HDD. Operating systems and file systems are designed on the basis that CPU time is cheap and seek time is far more expensive. This concept is rooted into the design of the operating system so that it performs a lot of merging and reordering on the requests submitted to the device in an attempt to lower the seek time. Aggressive prefetching is hindered to avoid moving the head to far and cost the system expensive seek time. Since SSDs have practically no seek time, the CPU time used for merging is wasted. In ordinary personal computers lending part of the CPU time to do worthless seek time reduction is not a problem. The number of cores is fairly low in those applications and single core performance is still relatively high. This however is not the case for HPC. In these systems their is a larger number of cores all awaiting IO. In addition to that these cores are fairly weak. The system cannot afford additional computation.

Although many have looked at SSDs as being a simple performance increase in IO, backward

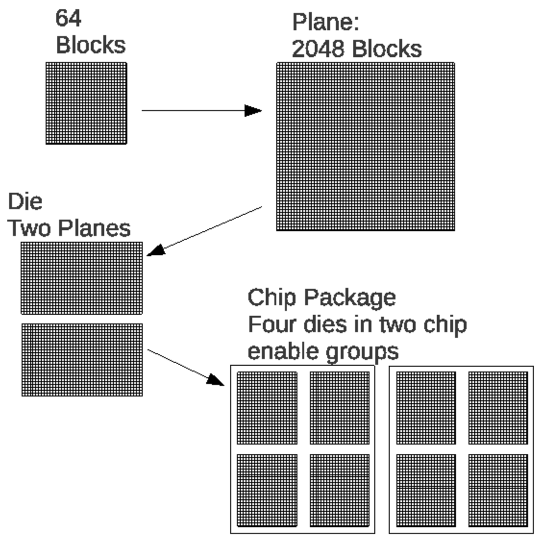


Figure 3.19: Typical SLC NAND flash chip packaging

compatibility has lead to an inefficient use. As mentioned before SSD have numerous reliability issues that need to be taken into consideration when accessing the device. Manufacturers develop their cards trying to shield users from reliability issues. Despite that, applications, file systems and operating systems can further help to increase reliability. Efficient access also can result in a major performance increase when it comes to SSDs. This however requires rethinking the entire IO stack to accommodate SSDs unique access properties.

There are several different types of flash devices available in the market. The most commonly used are MLC based flash USB devices. In fact MLC has overwhelmed the market. MLC represents 80% of the total NAND market versus SLC for 2007 [12]. However the systems targeted in this study cannot tolerate the high error rate produced by the MLC designs. Therefore the focus will be on PCIexpress adapted SLC devices. These are enterprise grade units that exhibit far less errors and failures. It remains to be said that in comparison to HDD, price per storage unit for SSD is very high. This can be noticed by observing the user market as well as the enterprise markets. A single 1TB PCIexpress SLC SSD can cost as much as 30K US dollars. It is therefore unfair to compare the prices on the basis of storage units. The comparison might be more logical on the basis of performance. Hundreds of HDD might be needed operating in parallel to achieve performance of a single SSD. The price therefore becomes more convenient and understandable for high end applications.

4 Optimizing IO Stack For Parallelism

Moore's law dictates that the number of transistors per chip have to keep increasing. This still remains true until today. In previous days most of the increase in performance offered by new chips was based on frequency scaling upwards, a trend that has stopped by the turn of the 21st century. This lead processor designers into the new era of multi-core. Current common market processing units contain at least two cores. In fact Intel has presented an 80-core chip prototype [14]. Not only does this strain applications towards applying themselves onto a many core design, but also strains the system to supply sufficient resources to cover requests from all cores. One such strained resource is IO.

The scale of the IO problem is known in the technical community, more so for the high performance computing systems. Scaling IO to meet the requirements of a massively parallel system is a difficult task. Although many scientific applications running on these systems have been using all processing units available, IO has been preventing the further scaling [15]. Under current IO design many applications lead to long processing idle time waiting on IO requests. This meant that the further optimization of applications for parallelism would not be effective in case the application is IO bounded. In fact adding more processing power would not be of benefit once any application has reached the furthest limit to its computational aspect. The previous point explains the known HPC proverb ”A super computer is a device for transforming a compute-bound problem into an IO-bound problem” [4]. To put this aspect into a practical sense, it is considered that a peta-flop system should be able to perform a single bit of IO for each instruction. This leads the system to needing at least a 100TB/s of sustained bandwidth [14]. Given the scale of the problem many researchers have been working on solving IO on the large scale of HPC.

As demonstrated in the introduction, one solution that shows promise is the implementation of active storage. Since the design of HPC usually involves a massively parallel computational unit, there is no space left for storage within the same unit. This means that storage is kept in another system. Under these circumstances an IO request has to move through a long stack involving a network request. To solve this the active storage introduces storage units directly into the computational system or vise versa.

The direct integration of storage into the computational units meant that these two have to communicate on a more basic level. In other words the multi-core design became relevant to the IO construction of the active storage design. Seen from the previous chapter, the IO stack of Linux has been mostly designed for single queue access. In fact preventing race conditions using locking mechanisms was done on a file level and have been only recently moved to a block level. Such limited single node storage IO performance might limit the active storage model performance.

Another aspect that introduced an additional challenge is the use of SSDs. Not only do SSDs add difficulty by having a different access pattern than traditional disks, but also introduce the presence of multi-controllers. To increase performance and saturate PCIexpress links, SSD designers have opted to adding multi-controllers onto a single SSD card. This meant that the operating system has to deal with not one but multiple devices. Trying to convince HPC application developers to handle multiple devices on a single node is not useful. The process requires too much effort and eventually means that the application will be too architecture specific. An effort has to be made to handle these multiple devices in either a global or local manner. Thus finding an appropriate handling method of multiple devices is an important condition for better performance in ASF. Due to the multiple layers that exist in a HPC system, it is important to decide on an appropriate handling layer for the IO problems on single nodes.

Reading through many of the current HPC research concerning IO, leads to the conclusion that there is very few papers related to single node storage IO. Nonetheless high level or global storage IO research is applicable in this case. The argument could be made that problems which were rising a few years ago on a large IO scale have now moved into the single node. Indeed the presence of multi-core and multi-controllers dictate that some of the solutions used on the global level might be useful for a single node. This argument has to be carefully dealt with. The compute power present on a single node might not be sufficient for large scale middleware. After all it is not desirable for the service computation to over utilize the single node. In other words, although single node might implement large scale solutions, such as parallel file systems, the node still needs to be able to meet application requirements.

The outline of the chapter is as follows. In Section 4.1 some of the related work is presented in the form of possible optimizations for IO access. Most of these studies as mentioned before address the HPC system on a large scale. Therefore the adaptation of these solutions on the single node will be pointed out. This will serve the further explanation of the tests and implementations concluded later in the chapter. Section 4.2 provides a detailed explanation of the test environment. Section 4.3 shows the analysis which includes tests done on the IO performance of different settings and the conclusions that can be drawn from the results. Section 4.4 explains in details a suggestion for a possible parallel IO design based on the previous analysis results. Finally Section 4.5 shows the results from testing the suggested parallel IO design.

4.1 Optimization Approaches

There are many different types of optimization approaches described in the research. In [1] three main types of optimization categories for optimizing parallelism are provided. These are optimization of runtime IO libraries, optimization of parallel file systems and caching, prefetching and data distribution strategies. An additional target for optimization is functional partitioning. The term functional partitioning is used to indicate binding resources to specific functions. Due to the presence of multi-core processors which use multi-controller storage devices, a well defined functional layout might increase performance. The following sections give insight into some of these optimization approaches and signify how these could be added to the IO stack on a single node level.

4.1.1 Runtime IO Libraries

Runtime libraries are well suited for optimization. Although POSIX is thought to be the standard for IO libraries, most tend to disregard it due to its limitation [2]. This comes as an understandable problem in case of HPC systems. Since there exists thousands of processing units in a single HPC system, collecting data on the fly and rearranging the IO in accordance could increase performance. One such method is the use of collective IO. In this technique all processes share information on their IO requests. In that way the collective IO libraries can decide on an appropriate approach to requesting the IO from lower layers [16].

One approach to improving collective IO is mentioned in [16]. Here spawning IO specific threads that carry out the collective IO services is used to improve performance. Yet the paper warns against uncontrolled thread spawning. Not all IO services can be performed in a separate thread. Therefore only part of the IO can be done in the background [16].

The reason for using runtime IO libraries vary according to applications using them. The main reason is avoiding clashes on either the disk access or on cache level. Having multiple applications accessing a group of disks at the same time might mean that seek time is not taken into consideration. It might be left to the drivers how the seek time is minimized as has been explained in the block layer. But as mentioned driver IO scheduling is done to improve overall performance. In turn this might lead to some applications being deprived of IO for sake of seek time minimization. While an IO scheduler on the block layer might not be able to communicate with the application for IO access patterns, runtime IO libraries can. Therefore they give the opportunity of avoiding application starvation during IO access due to IO scheduling.

On the other hand, the main target for using collective IO libraries is to avoid cache misses. A multi-core processor means multi-threads running on the same cache. If each process is left to regulate its own cache use, avoidable cache misses might occur. Collective IO has the opportunity of dividing the cache not equally but logically. In that sense processes that need the same data might share cache space and would not overwrite each others data. In short collective IO can predict cache accesses, combine data space and prevent overwriting of needed cache. Although caching is an important advantage for runtime IO libraries it is a complex subject that runs on multiple levels.

As seen in Figure 4.1, the optimization mainly focuses on the interface between the application on the lower levels. Some optimization will have to be done in the VFS. This is because the VFS controls the lower level layers. Therefore the VFS will have to be updated with functions that might be needed by the runtime IO libraries to perform their collective IO.

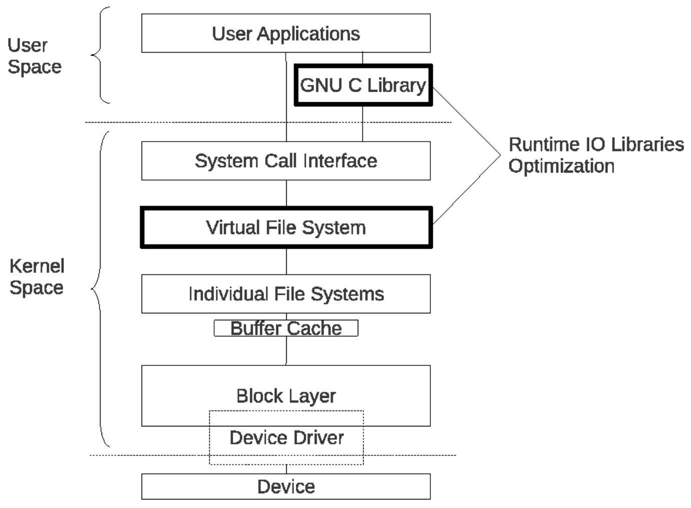


Figure 4.1: Optimization of IO stack using runtime IO libraries

Although optimization of IO libraries appears to be viable and simple to implement, it has a multitude of disadvantages. One obvious disadvantage is changing the interface by which the application communicates its IO requests. This means that all applications have to be rewritten on a basic level. Not only do these rewrites have to change the application IO interface, but also need to identify the amount of IO and the timing for these IO before handing them to the libraries to make the optimization useful. A solution could be applying these IO optimizations to an already existing IO library. In [17], it is suggested to use MPI-IO library as a promising ground for potential optimization. The reason mentioned is that the MPI-IO is a software layer between the user and the file system. This however still might involve some user intervention.

This optimization is still limiting to those applications using the optimized libraries.

Another obvious disadvantage that the optimization of runtime IO libraries have, is handling multiple devices. The IO libraries have to have an underlying multiple device handler. Most papers therefore mention the presence of an underlying parallel file system. This means that the distribution of data among these devices is not just dictated by the library. In turn this leads to a further complication of the design and the necessary optimization. Even though implementing a runtime library that can deal with multiple devices is possible, nonetheless it is not compelling to do so. Unless the application developer is prepared to deal with multiple devices on multiple systems, the library will not offer better performance. In such a case the application will have to signal IO schemes to the library. It is therefore inevitable that some applications will not perform well.

Also, while all applications using the optimized runtime IO library might improve performance, those not using the library might degrade overall performance. This mean that all applications running on the system must be using these optimized libraries. In addition to that it means that the Kernel has to be searched for IO access and rewrite them according to the rules set by the libraries. This might also lead to some complicated Kernel rewrites that must be performed to avoid IO clashes.

As mentioned before, runtime IO libraries optimize for specific considerations. The most important one is seek time. Since the optimizations considered in this work are based on SSD access on single nodes, decreasing seek time would not increase performance. In fact some of the optimizations mentioned in the papers are not very suitable for the Blue Gene setting. As an example, in [16] the possibility of spawning threads to perform IO is provided. While this might increase parallelism, the performance still remains under the mercy of the single core performance. Specially in case many calculations have to be done to merge request and decide on an appropriate request layout. Thus using such optimizations will hold the systems performance hostage to Blue Genes weak single core performance.

4.1.2 Parallel File Systems

Since HPC systems cannot function without IO, they have been always dependent on increasing parallelism. Most HPC and in fact servers scale their IO by increasing the number of disks and controllers used [18]. Since no application is prepared to deal with that amount of disks and controllers, parallel file systems had to be implemented. There are numerous types. The goal from a parallel file system is to combine the access of multiple disks into a single unit. That means that the application eventually has to deal with a single file system. Access division among devices is left to the parallel file system.

Parallel file systems allow multiple nodes access to multiple devices. The main task of HPC parallel file systems is providing global data access from all nodes present in the system [2]. Some file systems therefore employ multiple IO servers for that purpose. The data is then scattered among the devices. For that parallel file systems use different schemes. For example, IBM GPFS (Global Parallel File System) uses striping [4]. That means that each device shares part of the file. Using this method the parallelism is guaranteed through activation of several disk devices even if all applications operate on the same file.

Another example for data distribution is Hadoop file system. In this case data is not striped but replicated several times across multiple nodes [19]. The difference of design between GPFS and Hadoop file system means they have different applications. GPFS is build to allow multiple nodes access to the same data located on different devices. This implies an inherit separation between compute nodes and data servers. This is reflected in the processes that run on each side. Compute nodes requesting data mount the file system and run as clients that request data from the servers running as GPFS service nodes. The setup of such a system is shown in Figure 4.2. From the construction it is obvious that this system is adapted for high scale IO communication and is appropriate for current Blue Gene setup.

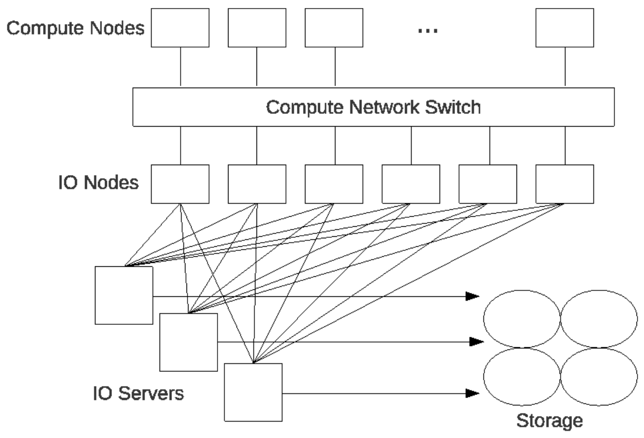


Figure 4.2: System setup for GPFS

One of parallel file systems biggest challenges is metadata. These are the data which indicates where and how the actual data is stored. Since loosing metadata means loosing the complete set of data, these have to be put in a setting that guarantees their survival. Another issue revolving around metadata is fast access. Shared data means shared metadata. That in turn leads to a possible bottleneck. GPFS offers complete parallel access to both data and metadata [6]. This has the advantage of avoiding the bottleneck and the possible metadata loss. Another factor in GPFS is that it can employ RAID for data backup and fault tolerance. Such property is a direct result of using striping and supporting multiple devices.

Hadoop on the other hand uses a different construction as seen in Figure 4.3 [19]. The setup uses a single node for metadata. This node is called name node. Not only is the name node a potential bottleneck, it also presents a failure risk. If lost the entire file system will be rendered useless. The rest of the setup contains clients which communicates with the data nodes to read or write data. Additionally the Hadoop file system can implement data replication on several data nodes. Not only does this help with data protection, but also ensures parallelism.

The major difference between Hadoop and GPFS is how the applications run on the two file systems. GPFS is implemented to run separately on different machines to service nodes that request data from storage units. In comparison Hadoop file system is meant to run on the same nodes requesting the data. Hadoop has the intention of writing the data once and reading it multiple times [19]. In other words, Hadoops intention is for the programs to move to the data and not the other way around. As a direct result Hadoop file system is indented only for applications using its own implemented map reduce interface. That is also why normal POSIX requests cannot be used to access the Hadoop file system. GPFS however, can be mounted as a normal file system and accessed using normal POSIX.

The Hadoop file system could be a suitable candidate for an active storage setting. However it is a more high level IO concept which is not interesting on a single node IO performance level. Additionally due to the difficulty in benchmarking IO as will be seen from the next sections,

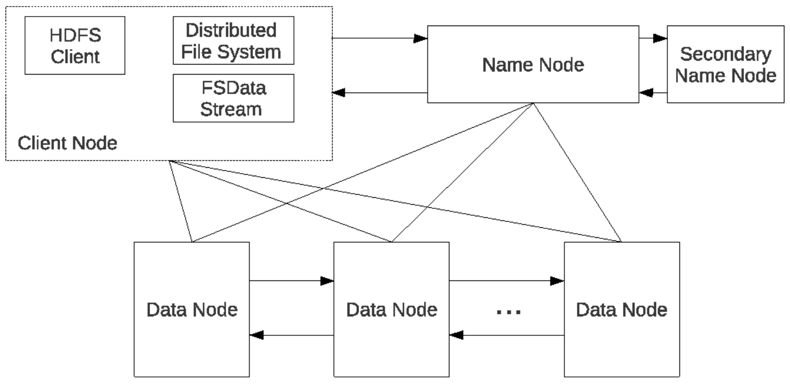


Figure 4.3: System setup for Hadoop File System

comparing Hadoop performance with GPFS or normal IO is not possible. Hadoop requires special benchmarks or needs the IO benchmarking tools to be rewritten in a map reduce construct. Single node IO performance will therefore be lost in the middle of high level IO. Additionally Hadoop file system does not support multiple devices it much rather supports multiple nodes with storage on board. Again this makes the single node performance of a Hadoop cluster irrelevant for this study.

In comparison to Hadoop, GPFS is more appropriate for single node IO performance analysis. Although GPFS offers many advantages, there still are difficulties that have to be dealt with. One important problem is the setup. Client and service nodes in a GPFS cluster are separated. This means that an active storage node must operate as both client and service node. This might strain the limited computation power. GPFS therefore introduces an overhead that could be avoided. For example multi-controllers on a single node could be detected as a single device. This can be implemented in either the VFS or on the driver level as will be demonstrated by the following sections.

There are factors that might imply that GPFS data distribution is not suitable for the active storage setup. One such factor, is a direct consequence of data striping. As data is distributed evenly among the nodes with files being striped no one node holds a complete set of data. This leads the active storage nodes request to be burdening not only its own IO but the IO of the neighboring nodes as well. This might look like it is defeating the purpose of the active storage. It would be much more convenient to have the single node only using the data stored on its own storage device. Nonetheless it still remains interesting to see how GPFS could lead to a speed up in performance of single node IO. In other words, an active storage would benefit from a two view file system. External nodes access a global file system striped across all active storage nodes. Mean while each active storage node has a local view of data and does not require a global access to fulfill its data requests. This combination would therefore benefit both targets equally.

As can be seen from Figure 4.4 the optimization from using a parallel files system focuses on optimizing individual file systems. The VFS might also need to be optimized. In fact the entire implementation of the parallel file system can be done on the VFS layer. The idea is to make the VFS aware of data distribution among the underlying mount points. In fact this has already been implemented as a separate file system called PVFS or Parallel VFS [6].

Additional changes have to be made on the block layer level to accommodate the multiple devices. Figure 4.4 does not include these changes. This is because these changes are irrelevant

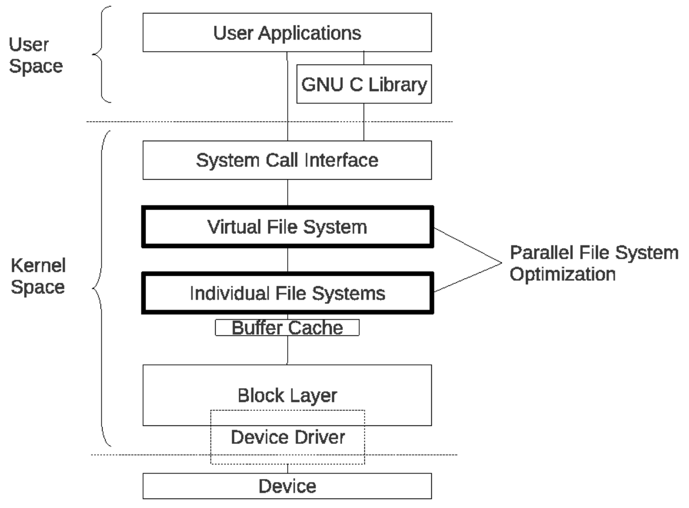


Figure 4.4: Optimization of IO stack using parallel file systems

to the parallel file system. The block layer needs only to present the multiple devices and leave the parallel file system to deal with them.

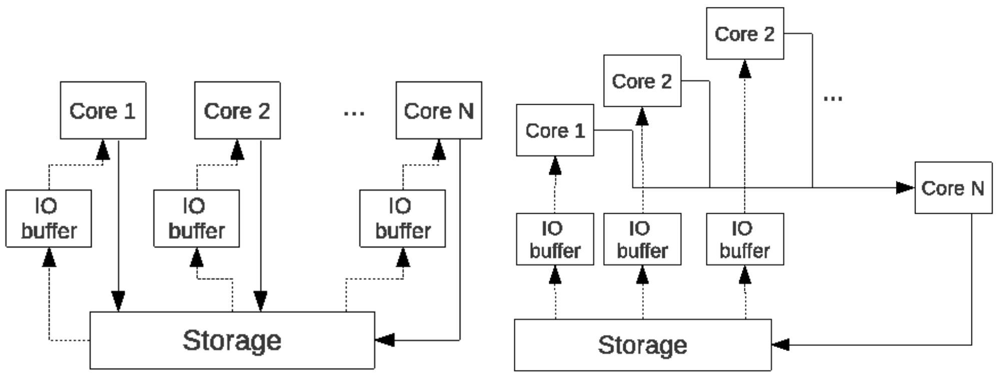
4.1.3 Functional Partitioning

As the number of cores per processing unit increases so does the expectations. The problem however is that most applications performance does not scale with the assignment of new cores [14]. Although HPC applications should have a better scaling performance, still they exhibit difficulties. In the current HPC systems an overwhelming number of multi-core processors exist. Not only does the application and the operating system scheduler have to divide these cores among themselves, but also have to manage a complex hierarchical memory system. Managing these cores adds difficulty to the application and the operating system. Considering the number of cores to be managed means that old scheduling techniques might not be efficient.

Another aspect that is also overwhelming the applications and operating systems is the amount of available data. While parallel file systems might be a well suited tool to store large data amounts, most are not designed for application fast access. Usually the intention of a file system is to speed up current access and has no overview over data placement. On an active storage system this becomes even more problematic. Giving up expensive storage embedded into the computing system without prior knowledge of data usage, might not be in the best interest of performance. The same concept could be said for caches. Additionally the existence of multiple controllers on a single card means that the system has to manage different devices. Not only does the file system or in this case local parallel file systems have to decide on data location, but also decide which device to use in addressing this data. While it might seem that choosing the currently available device or the one with least traffic is reasonable, things are more complicated. In most, if not all cases data has to be read and written using the same device. This means that the parallel file system will have to consider read traffic before executing a write. On higher levels parallel file systems can simply stripe or duplicate data to avoid parallel read conflicts on the same device. This is not a viable solution with active storage. Single nodes are expected in this case to have expensive limited storage space.

An alternative for traditional scheduling is functional partitioning. The term functional partitioning refers to binding certain resources to specific functions. Thus the target is to assign a number of cores to certain applications or certain services. In [14] a description is provided for using a functional runtime library to assign cores to part of the application. For example an application that contains significant IO operations would signal that it needs to run IO threads on a separate core. This however dictates rewriting applications which has to be possibly avoided. Another viable preposition made in [14] is giving specific cores to specific SSD services. As described in the previous section, some SSD need additional services that could be implemented on a driver level. To avoid letting these services slow down applications an additional core could be awarded to these services.

Functional partitioning of CPU affinity decreases the switching time from process to process. The technique can be even taken further. Instead of just allowing applications to request cores for services, cores could be completely allocated to specific IO services. By removing a core from the CPU scheduling list in Linux the CPU will be registered by the operating system but no process will ever be scheduled on it. That leaves the core open for specific allocation which could be done using *taskset*. As an example to that approach can be found in [20], which targets core allocation for collective IO. Instead of running a collective IO library a core is allocated to collective IO. All other cores send their IO requests to the collective IO core, which processes them and sends a total request on behalf of all cores. The concept for this can be seen in Figure 4.5. It has to be noted though that this design carries the same limitations of collective IO. The optimization still remains mainly for seek time reduction and therefore of minimal relevance to SSD.



(a) Parallel IO on multiple core (b) Diverting one core for collective IO

Figure 4.5: Dedicating core for collective IO

Despite the fact that functional partitioning on the basis of core affinity might increase performance, it could also limit it. Some modern processing units use multiple weak cores for increased parallelism. Some operations divided among applications that run on different cores might not perform well if assigned to a single core. Even if the core has no other task than the allocated operation, the limitation remains due to low core frequency and limited compute capacity. In fact the proposed design of collective IO in [20] might limit the total performance of the system. Cores might idle waiting for the so called coordinator core to send their requests to the storage.

On the other hand, memory or storage affinity might be a method for improving efficiency. For example, in [21] a method is introduced for memory affinity on hierarchical multi-core multi-processor level. This is of great benefit for modern processor designs. Although modern multi-core processors might contain cache for each core, on some memory level they eventually have to share. In fact that is the case when it comes to RAM which will be shared by all applications running on all cores. If applications do not share data they will eventually overwrite each other. RAM misses are expensive, even more so for HPC. If data is present in a global file system the HPC node will have to forward a request through complex networks. Therefore memory affinity becomes a possible candidate for limiting the number of RAM misses. In case of an active storage layout memory affinity techniques can even serve determining which application could write into the active storage. Using memory or storage affinity, performance could be enhanced on an even larger level.

Storage affinity is a technique by which storage can be divided among applications. It can also be a technique by which device allocation can be performed in case of using multiple controller storage units. This means that data has to be allocated with a global layout. Applications are bound to certain devices or certain storage space. Considering the fact that active storage systems will contain fast but expensive storage this might be a viable solution to efficiently use the available storage space. Additionally it will limit the number of writes, which is necessary for some storage technologies such as SSD. In fact global data layout aware systems have been introduced to increase parallel IO performance as seen in [1]. Here data layout was shown to be the description as to how the data will be divided among multiple file servers. For a single node system the same could be defined for dividing data among separate controllers or devices.

Active storage will contain fast and expensive storage units that could be dealt with as either slow memory or fast storage. Therefore both techniques for storage and memory affinity could be applicable. In fact storage affinity will have to be mixed with global application scheduling in an HPC. Data will have to be stored as close as possible to the currently operating application. Therefore it is of advantage to have a parallel file system that communicates or is controlled by the process scheduling unit.

Using memory or storage affinity might be a good solution to reduce data access time. Nevertheless, unless applications signal their data use before hand, it might be complicated to implement. Some data layouts that might be of benefit for one group of applications might be a disadvantage for another. Active storage units will have to support data layout changes. Therefore simple data use counters will have to continue to exist to signal data relevance. Other more general memory or storage affinity algorithms have to be defined as well. The main target from these algorithms would be to keep application developers independent from the system architecture. The ideal case is achieved if the application developer can run the same application on an active storage system and still achieve the most efficient storage use possible.

As seen in Figure 4.6 optimization has to be done on many different layers. CPU or core affinity can be implemented in the application layer. Since this is not in the interest of the IO layer it might be ignored. Another possibility is adding service cores. These overtake certain responsibilities such as file system defined functions. In fact one can implement storage affinity using core affinity. In this case a core would be awarded to finding best methods for dividing the storage on currently running applications. This could already be done in case of using parallel file systems. These already implement different threads for different clients or services. Therefore cores can be allocated for such processes to guarantee a high degree of quality of service. Additionally service cores can be applied to carry out device driver specific operations. For instance queue handling can be carried out on a single core. Another option is to allow device driver developers to take over the scheduling of a core to carry out device services. Therefore an SSD might implement garbage collection and wear leveling in the device driver and allocate a complete core for these processes.

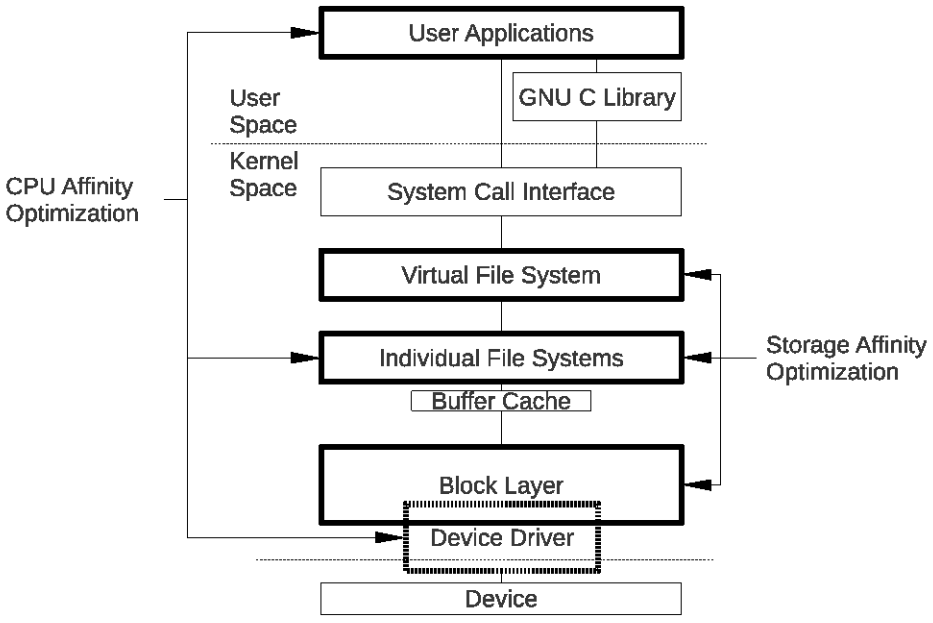


Figure 4.6: Optimization of IO stack using functional partitioning

On the other hand storage affinity can be implemented into several different IO stack layers. Figure 4.6 shows storage affinity optimization pointing to three layers VFS, individual file systems and block layer. Since all three might select the storage device used, allocating certain device controllers to certain tasks could be done in one of the three. The VFS keeps track of *superblocks* of mount points which contains the block device on which the file system is stored. Therefore the VFS could have a list of devices and point to any of these in case certain tasks call the appropriate system call. The VFS however cannot select the data allocation layout. This task is performed by the file system. Therefore a parallel file system might be a good solution to perform storage affinity. Nonetheless the VFS will have to keep the file system aware of current allocation needs. Another possible candidate for storage affinity implementation is the block layer. There already exist device drivers that control device selection. These have the added advantage of simplifying the operating system view of the system. The device driver is registered as a single block device and the file system does not even have to consider device boundary effects or allocation schemes over different devices.

Although resource binding is an appropriate approach to todays multi-core multi-controller systems, the implementation is complex. The functional partitioning cannot be done in one layer. A complete system view is needed. If the implementation takes place in the lower layers, upper layers will have to pass system information to lower layers and vice versa. The complexity increases even more if the implementation is to be architecture independent.

4.1.4 NVM Express

Hard disk interfaces have evolved over many years. The device interface became so important that even operating systems changed their block device interface to accommodate them. File systems and allocation strategies are heavily optimized for hard disks. The main concept on which these optimization have been taking place is that CPU time is cheap and seek time is expensive. Therefore file systems implement allocation strategies targeting contiguous file allocation. Random read or write is avoided as much as possible.

During the recent years SSD and other Non-Volatile Memory (NVM) started to appear on the market. Since these technologies needed to be integrated into present system setup, the interface of a disk was adapted. Manufacturers designed their card interface to function as a hard disk. The target was to avoid file system rewrite. However the CPU kept decreasing seek time which is no longer necessary for the new technology. SSDs have almost no seek time. Additionally, while hard drives are easily integrated into the system, each SSD manufacturer has to spend money on developing complex drivers. This is because of the missing unified interface.

As a solution Intel and other companies set out to define a new interface standard called the NVMexpress [22]. The standard targets the SSD design on a PCIe interface. The idea is to create a uniform communication method by which any SSD can be integrated into the system without the need for lengthy and difficult device drivers. The NVM Express is defined as a scalable host controller interface. This has been specifically designed for enterprise needs and client systems that use SSD on PCIexpress [22].

The basic idea of the NVM Express interface is to use multiple queues. Processes form their requests and add them into a submission queue. The controller would then complete these requests and return them into a completion queue. There are many differences between this interface and the old queue interface provided by Linux. There can exist as many queues as needed. The interface supports up to 64K IO queues with each queue supporting up to 64K commands [22]. This means that queues could be binded to cores or to processes. Figure 4.7 shows the possibility of binding queues to cores. Another possibility is shown in Figure 4.8. In this case several applications each having their own submission queue share one completion queue [22].

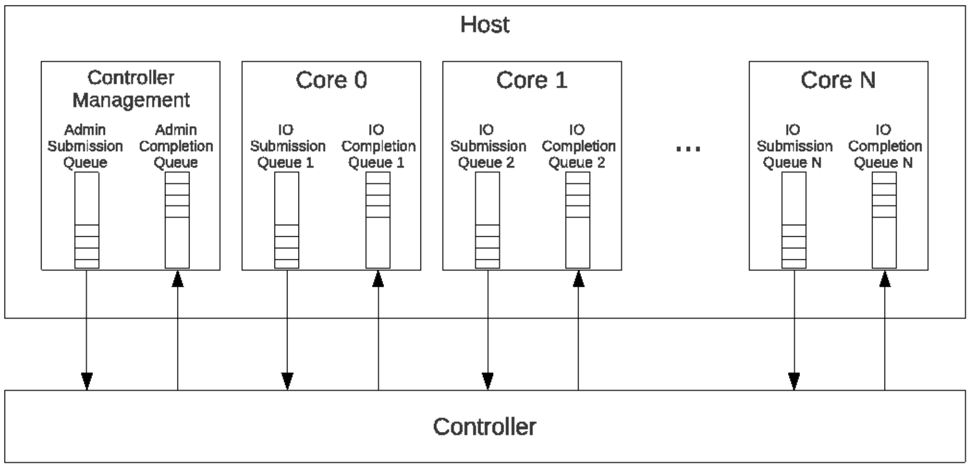


Figure 4.7: NVM Express multiple queues with core binding

The NVM as explained in the literature has a lot to promise. However there is still a lot of unknown. Not only do SSD manufacturers have to implement the standard into their cards, but also IO libraries have to be rewritten. This might also mean a rewrite of applications. The disadvantage of the NVM Express is that it has not yet been widely adopted. Once the manufacturers of SSD PCIexpress cards adopt the standard IO libraries will be changed. The final point will

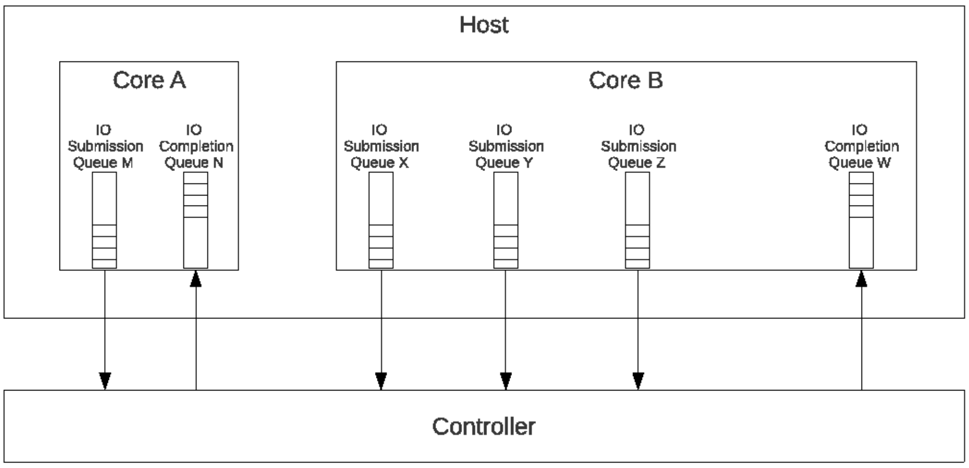


Figure 4.8: NVM Express with different queue mapping

be for Kernel developers to follow. The intention then is to be able to place any SSD card into any system and have it function as needed. Not only will that mean saving efforts of device driver development, but also means easier optimization of Kernel architecture for the sake of supporting none-volatile memory.

4.2 Test Environment

A single node in a Blue Gene/Q contains a processor with 18 cores. Since one core is used for operating services and one is kept in reserve only 16 are available for applications. Each of these cores supports four threads and runs at a modest 1.6GHz [23]. Given that the Linux Kernel counts each thread as a separate CPU, performing a *cat /proc/cpuinfo* prints the total to be 68 CPUs.

The architecture of the cores implemented into the Blue Gene/Q is the A2 core based on the POWER architecture [23]. Due the unique implementation, pre-tests had to be done to port code onto the complex Blue Gene/Q architecture. For example, on testing diverse SSD cards it was necessary to go through several testing phases prior to Blue Gene/Q tests. The card would be first tested on an *x86* processor architecture. The *x86* is the most commonly known processor instruction set. For that reason, the card is expected to function at specification rates on an *x86* processor. Several different *x86* processing units have been used in the testing process. One was a quad-core Intel i7 CPU 920 with a frequency of 2.67GHz and dual thread per core.

The next step would be to test on a similar architecture to that of the Blue Gene/Q. Seeing that the A2 core is based on the POWER architecture, testing should be done on a POWER processor. Therefore a G5, which is a POWER Macintosh, is used. The G5 contains two dual core 970MP with a frequency of 2.5GHz. Porting from the *x86* to the POWER architecture on the G5 is complicated. This is because of the differences present in the instruction set between the two processor architectures. This meant that some of the optimizations done for *x86* would not work on the POWER processor. It has to be noted that in some cases such as testing loop devices or *ramdisks* there was no need for the middle step of testing on the G5. The reason is that these devices do not contain any architectural specific implementations. Therefore a direct comparison can be done between the performance of such devices on *x86* and Blue Gene/Q or between the G5 performance and Blue Gene/Q.

The final step is testing on the target architecture the Blue Gene/Q. Due to porting the device on the G5 most of the architecture problems should have been solved in the previous step. However the Blue Gene/Q has its own unique architecture. Therefore additional changes had to be made to complete the porting to the Blue Gene/Q. Although the porting process is long and complex it gave the opportunity of comparing performance. Not only does Blue Gene/Q contain a unique processing architecture, but also is a HPC build for parallel operations. Comparing Blue Gene/Q performance with that of a G5 meant comparing a commercial computer performance with that of an HPC system. On the other hand comparing Blue Gene/Q performance with that of a *x86* meant comparing performance of different processor architecture.

Due to the diversity of architectures it is important to label performance plots. Therefore every performance plot shown will have a description of the processing unit on which the test was performed on the top.

4.2.1 Test script

The used IO tester is called Flexible IO or FIO. It is an open source benchmarking tool that has been used within many other projects. FIO spawns many threads that read or write from or to a specified file that it creates. This file is placed into the device under test. The test script itself was perfected over a multitude of tests. Since FIO contains a large number of different settings and parameters, the testing scheme is complicated. The following will be a short description of some of the FIO test parameters. There is a brief explanation of all FIO options available in the *HOWTO* file provided with the tool. However parameters effect on the output is not mentioned, as well as special use of some other parameters. The explanation given here therefore also includes test observations.

Benchmarking IO is complicated. On one side there are different things to measure. IO can be measured as the number of IO operations per second or IOPS. It can also be measured as bandwidth, which is the amount of data that can be transfered per second. There is a direct relation between IOPS and bandwidth. This relation is defined by the amount of data that is requested in a single IO operation. The correlation will be defined and tested in the next section.

Another factor that complicates the testing of IO is the large number of parameters that affect the IO performance. The parameter list used in the test script is shown below.

```
fio \
     --name=$FIO_NAME \
     --rw=$FIO_MODE \
     --size=$FILESIZE \
     --directory=$FIO_DIRECTORY \
     --bs=$FIO_BLOCKSIZE \
     --numjobs=$j \
     --runtime=$FIO_RUNTIME \
     --loops=$FIO_LOOPS \
     --direct=$FIO_DIRECT \
     --ioengine=$FIO_IOENGINE \
```

The following is a detailed explanation of used parameters.

name is the name given to the test.

rw is the mode of the test. Possible modes are read, write and read/write for sequential IO or random read, random write and random read/write for random IO.

size is the size of the file to read or write from. The file size is an important parameter. Increasing it means larger seek times in case of hard disks. It also means that the data in the file is addressed using indirection. In case of hard disks this results in even larger seek times. On the other hand for SSD it might mean that during a write more blocks have to be updated. This leads to write amplification. While choosing the file size it has to be noticed that unless the filenames are set and therefore the number of files is kept constant, each job will create a file of this size. Therefore the aggregate file size will be $numjob \times size$.

directory / filename the *directory* is the location where the files are placed for testing. The location has to be on the device under test. There is also the possibility of using the *filename*. By defining not only the filename but also the path to the file the test directory can be chosen. Another useful trick is the ability to use more than one file. This means that more than one test directory and therefore device can be tested at the same time. This is useful in case multiple devices or parallel operation should be tested. However if only the directory is set FIO will create one file per job. If several filenames are given, then all jobs will operate on all files at the same time. This means that all devices under test will be accessed from all jobs. Moreover setting *filename* can be used to test raw device performance. Raw device data is useful in finding overhead of file systems processing.

bs holds the value of the *blocksize*. As mentioned before their is a direct relation between IOPS and bandwidth. The relation is dependent on the amount of data per IO request. The blocksize is the amount of data that is contained in a single IO request. Since the IOPS and bandwidth are inversely proportional, increasing the blocksize means decreasing IOPS and increasing bandwidth. On the other hand, decreasing blocksize leads to increasing IOPS and decreasing bandwidth. This will be further investigated in the tests.

numjobs is the number of threads that the FIO job will spawn. The main target is for the performance to increase with increasing the number of jobs. There are many factors that have to be kept in mind. For example if *filename* is set then all jobs will operate on all files at the same time.

runtime indicates how long the test will run. Unless set the FIO job will run for a very long time. There has been no indication as to how long that is. Most probably the test will terminate after reading or writing the entire file size. The *runtime* has to be set at a reasonable amount. It was found that $300s$ or $5min$ are sufficient for a single test. It has to be noted here that the test does not have to run for $5min$. This is just an indication that the test will be terminated in case it took longer than $5min$.

loops is the number of times the test should be repeated. In some cases if the runtime is too short as in testing *ramdisks* the number of loops helps increase runtime. This increases the precision of testing.

direct if set to one indicates that non-buffered IO should be used. In order to benchmark the IO link correctly direct IO was almost always selected. Not only is cached and buffered IO more complicated, but also means that performance will vary over many tests. Using direct IO means testing the IO under the worst case, which means all accesses resulted in a cache or buffer miss.

ioengine defines the IO engine used. The IO engine defines how a job issues an IO request to the file. The *ioengine* has been set to *libaio* which is Linux native asynchronous IO engine. An additional advantage is that *libaio* only supports the use of non-buffered or direct IO.

The following shows an example for the output of a FIO test.

```
Fusio: (groupid=0, jobs=1): err= 0: pid=5030
  read : io=65536KB, bw=1158.6KB/s, iops=1158 , runt= 56570msec
    slat (usec): min=186 , max=23738 , avg=837.08, stdev=247.00
    clat (usec): min=3 , max=143 , avg= 7.56, stdev= 2.36
     lat (usec): min=196 , max=23761 , avg=849.06, stdev=247.29
    bw (KB/s) : min= 1028, max= 1972, per=12.52%,
        avg=1159.10, stdev=140.72
  cpu          : usr=2.99%, sys=48.76%, ctx=127005, majf=0, minf=5
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%,
        16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%,
32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%,
32=0.0%, 64=0.0%, >=64=0.0%
     issued r/w/d: total=65536/0/0, short=0/0/0
     lat (usec): 4=0.01%, 10=97.56%, 20=1.70%, 50=0.73%, 100=0.01%
     lat (usec): 250=0.01%
```

As can be seen the name of the test is used to identify the output. The next identification is the mode, read or write. Whether the test was a random or sequential read or write is not indicated. It is therefore preferred to use the test name to indicate a sequential or random test. As can be seen there are many measures taken from a single job. Since we are only interested in the IOPS and bandwidth, we can ignore most of the rest. Bandwidth of a single job is given by *bw*. IOPS are given by *iops*. FIO also can give the output in the form of numbers separated by semicolon. Although this might be the better choice for plotting the results, this output does not contain the IOPS.

The output given above is one out of eight for each job that has been spawned by FIO. This means that the IOPS and bandwidth given is not the total. To find the total IOPS and bandwidth the individual results have to be added. The semicolon separated output form provided by FIO cannot be used due to the missing IOPS results. Therefore the script has to find a method of choosing the correct value and adding to find the total IOPS or bandwidth. The method for finding the IOPS is given below. As can be seen the trick is to use string manipulation to find the value. The loop then adds all found values and sets the variable to the total IOPS that will be printed. The same is done with bandwidth.

```
IOPS=0
iops=‘grep iops= < $tmpfile | cut -d’=’ -f 4 \\
    | cut  -d’ ’ -f 1 | tr -d ’,’‘
for i in $iops; do
   ni=‘norma $i‘
   IOPS=‘echo $IOPS + $ni | bc‘
done
```

At the end of each test output the aggregated values of all jobs is given. This could be an alternative to the previous method for finding the total bandwidth. However it was observed that the total bandwidth calculated using the script was different than the aggregate bandwidth given by FIO. As the method of calculation of the aggregate bandwidth is not known the summation script was used. Another reason for using the summation script is that the IOPS aggregate value is not given in the list of total results.

The test script has gone through several revisions. The most important problem is finding the parameters against which the IOPS and bandwidth will be tested. At first the tests were done against changing both the number of jobs and the blocksize. This results in the count of different job number multiplied by the number of different blocksizes tests to be done. Due to the complexity of reading the performance change for increasing job number the test was later changed to output the results only against the jobs number. The test was usually then repeated twice, once for a blocksize of 4KB and again for a blocksize of 64KB. Tests results might be shown in either performance against job count or performance against blocksize.

Below is the output from the test script. The date and time of the test is printed. This is important in case the tests need to be recreated. As the Blue Gene/Q environment is under development and therefore constantly changes it is important to know which revisions can be used for getting the same test results. Using this technique the cause for a test result change can be identified. The rest of the upper parameters show the test parameters as given to the FIO thread. Additionally the file size is printed before each job number result. The total file size however remains the same as has been explained in the FIO parameters.

```
#DATE=2011-08-29 09:53:17
#FIO_BLOCKSIZE=4k
#FIO_DIRECTORY=/mnt/TEST
#FIO_MODE=randread
#TOTAL_FILESIZE=1024M
#jobs  BW[MB/s] IOPS
FILESIZE=1024M
1 21 5493
FILESIZE=512M
2 41 10710
FILESIZE=256M
4 82 21029
FILESIZE=128M
8 154 39541
FILESIZE=64M
16 283 72568
FILESIZE=32M
32 477 122287
FILESIZE=16M
64 746 191153
```

Finally gnuplot is used to plot the output. An example for the final result can be seen in Figure 4.9. The figure shows the IOPS as plotted against the number of jobs.

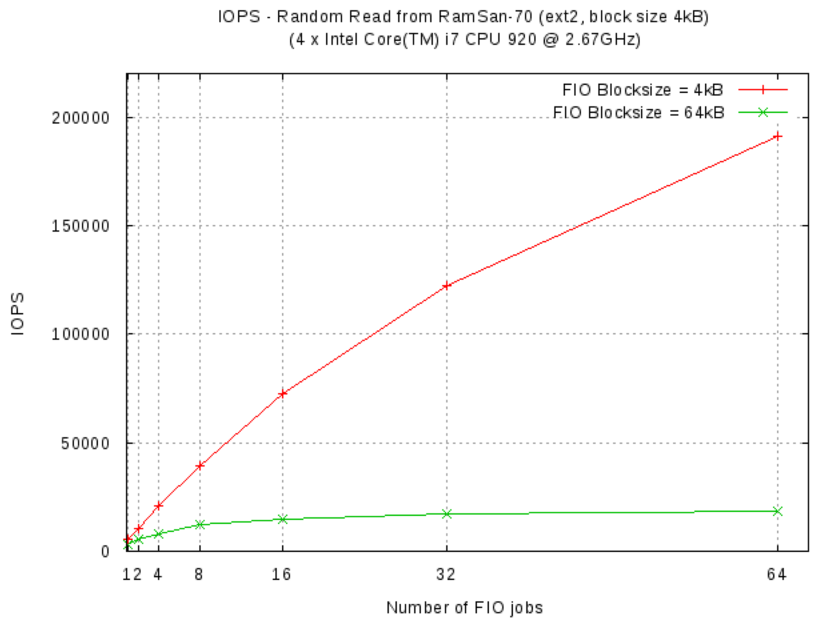


Figure 4.9: Example of a gnuplot figure for a FIO script output

4.3 Parallel IO Analysis

Previous sections have made some points which show how parallelism is increased on a global level. As the active storage will need to communicate on a single node level parallelism needs to be pushed lower. That is, solutions that were previously implemented across several computing units has to be implemented on single nodes. In fact observing the development of HPC over the years, the same has been done on different layers. The modern processor contains several cores and the modern SSD storage unit could contain multiple devices. Therefore IO parallelism is required on an even smaller scale. This is not as simple task. On a global level implementing complex algorithms and configuration is possible. In contrast trying to implement the same on the scale of a single node has many limitations. As an example, parallel file systems are massive programs. Implementing the same on a single node might need redefinition of the function of a parallel file system. Under these circumstances it appears that some of the previously mentioned concepts should be tested given the new limitations. The following sections define some of the tests done over multiple settings. The target was to find some efficient methods of increasing parallelism.

4.3.1 Testing Effect of CPU Frequency on IO

As mentioned before, modern CPU are moving towards the usage of smaller weaker cores. Besides using a different architecture, the Blue Gene/Q employs an A2 core working at 1.6GHz. The target here is to find the effects of lowering the frequency on the performance of the IO. The best candidate for testing the IO under the effect of lowering the CPU frequency is using an actual physical SSD. Hard disks require a good deal of CPU utilization in order to avoid seek time. However the CPU is not expected to dominate the IO operation. Under this assumption using the hard disk would not be useful. On the other hand if a virtual device is used such as

Read Bandwidth (64kB)	770 MB/s
Write Bandwidth (64kB)	790 MB/s
Read IOPS (512 Byte)	140,000
Write IOPS (512 Byte)	135,000
Mixed IOPS (75/25/r/w)	119,000
Access Latency (512 Byte)	26 µs

Table 4.1: Table for ioDrive performance specification

loop devices or ramdisk, there might be no separation between performance drop due to IO and drop due to module operations.

SSD performs many different services to support its operations. Manufacturers have the option of implementing these services either directly on the card in hardware or into the driver. For example, garbage collection could be implemented as a hardware feature into the SSD controller. On the other hand it could be programmed into the driver module. Due to that different SSDs might not be equally dependent on the CPU frequency. Nonetheless it is still important to know how the CPU frequency would effect such combination.

The SSD card used for testing frequency effect is an ioDrive produced by Fusion-io one of the most known SSD manufacturers. The ioDrive contains a single controller and is therefore detected as a single device. This helps decrease the overhead of combining performance of multiple devices. The ioDrive used is a SLC with a capacity of 320GB. The performance expectation are given by the table [4.1] [24]. Although the card has a driver that supports 64K pages, it can only be used for the *x86* architecture. This meant that the Blue Gene/Q used Linux had to be patched to use 4K pages instead of 64K pages. In turn this might already result in some of the performance for IOPS and even bandwidth being lost.

It is expected that the results will not be the same as those given by the specifications. The reason is that the ioDrive driver has been optimized for *x86* architecture and not for POWER. The ioDrive driver is closed source. Therefore there is no room for observing how the internals of the driver work. There is also no possibility of optimization of the driver. Additionally the G5 is the most appropriate device to test frequency change. This is due to it containing a POWER processor which is similar to that of the Blue Gene/Q. Additionally using simple tools it is possible to decrease the frequency of the CPU. The test results shown will be that of the random read. Although random write was also performed, concerning frequency change, the same conclusions can be drawn. Thus the random read tests are sufficient for this purpose.

Figure 4.10 shows the ioDrive bandwidth versus blocksize. As can be seen the bandwidth gradually increase with increasing the blocksize. This is due to the fact that the amount of data read per request is becoming larger. This continues until the link is saturated. Additionally the bandwidth increases with increasing the number of jobs. This as well continues until the parallel IO limit is reached. As seen from the figure the bandwidth end performance is close to 800MB/s which is given by the ioDrive specification. However this limit is reached with a blocksize of 128KB. The blocksize shown in the specification for the read bandwidth is 64KB. Only for a job count of 32 and 64 can the maximum bandwidth be reached at the specified 64KB blocksize. This means that we need double the blocksize to achieve the same performance as the one given by the ioDrive specifications. It should be noted that increasing the blocksize is dependent on the access pattern of the applications. Not all application will benefit from a larger blocksize. In fact increasing the blocksize specially in case of writing will decrease SSD endurance. It should also be noticed that this bandwidth can only be reached by using 4 jobs or more.

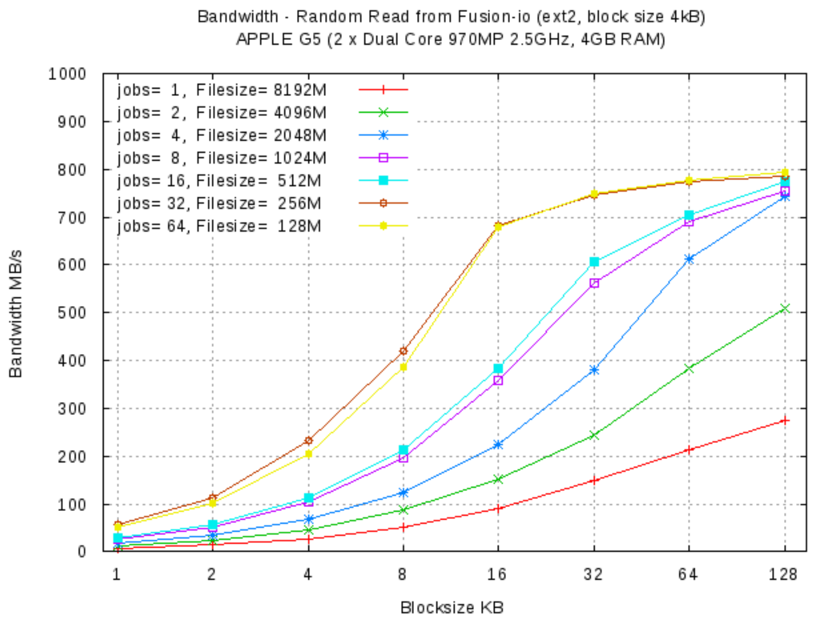


Figure 4.10: Bandwidth vs blocksize for ioDrive on an Apple G5

On the other hand, Figure 4.11 shows the IOPS versus blocksize of the ioDrive. In this case the IOPS decrease with increasing the blocksize. As has been mentioned before there is an inverse proportional relation between IOPS and bandwidth which is governed by the blocksize. This indicates that either the system can run many IO operations with small blocksize or few IO operations with large blocksize. The figure shows how the performance starts to gradually decrease at the blocksize of 8KB. The maximum value however for the IOPS does not meet the ioDrive specification. This indicates that the architecture effects the IOPS and not the bandwidth. It is therefore expected that decreasing the frequency would effect the maximum IOPS. Additionally the IOPS increase by increasing the number of jobs. In fact there is an expected symmetry between the bandwidth and the IOPS in the vertical direction. That is by increasing the number of jobs both IOPS and bandwidth will increase. However once the number of jobs either saturate the bandwidth or the IOPS, adding more jobs will no longer increase the performance. On the contrary adding jobs beyond a certain point might mean a decrease in performance. In comparison to hard disks however the performance drop does not happen due to increase in seek time. It appears due to increase in CPU utilization. This correlates with difference between hard disks and SSD systems. While systems containing hard disks consider that seek time is expensive and CPU time is cheap, modern systems containing SSD have no seek time and have expensive CPU time.

Given that the target of the test is to find the effect of frequency decrease on IO performance the CPU clock has to be decreased. The tool used to decrease the CPU frequency is called *cpuspeed*. Using the tool the CPU frequency is scaled to half and is monitored during the FIO test using *watch grep clock /proc/cpuinfo*. This makes sure that the CPU scaling tool does not undo the frequency decrease. Thus the CPU frequency of the G5 was kept at 1.25GHz while running the same FIO test.

Figure 4.12 shows the bandwidth versus blocksize for the ioDrive on a G5 with frequency

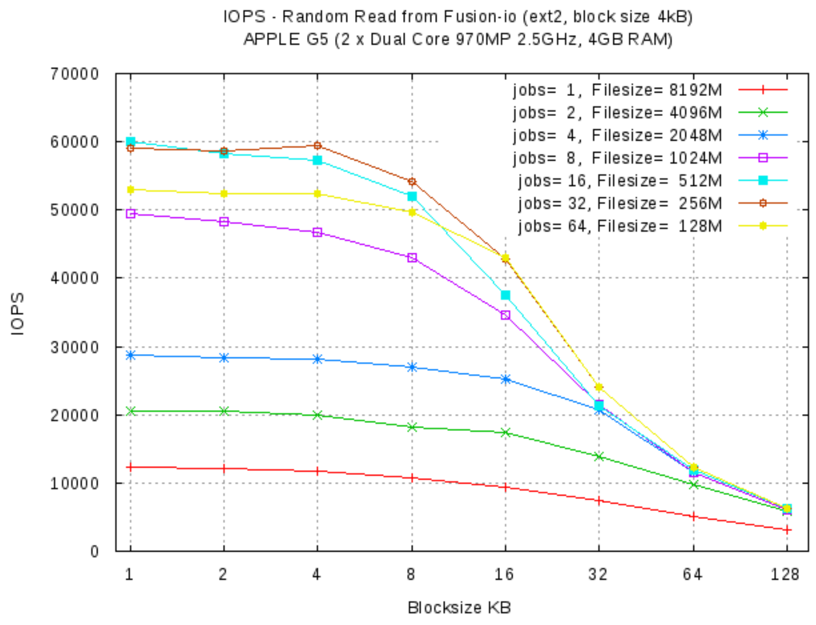


Figure 4.11: IOPS vs blocksize for ioDrive on an Apple G5

scaled to 1.25GHz. Although the frequency is decreased it can be seen that the maximum bandwidth can still be reached. This means that the maximum bandwidth is not effected by the frequency. Nonetheless the frequency change can still be observed on the bandwidth curves. As can be seen from the figures, the saturation is first reached at a blocksize of 64KB. This means that the decrease of frequency leads to moving the saturation of the IO link in the direction of larger blocksizes. This indicates that processors with lower frequency will require larger blocks to achieve maximum bandwidth. Another important observation is the vertical behavior of increasing number of jobs. Using lower CPU frequency meant that there is no longer any difference between having 8 or more jobs at lower blocksizes. Having full frequency meant that, for example, at a blocksize of 16KB there was a large difference in bandwidth between using 16 and 32 jobs. One can conclude from this that having a low frequency CPU using larger blocksizes has a better effect on bandwidth than using more jobs.

In contrast to the bandwidth the maximum of IOPS is affected strongly by the decrease of CPU frequency. This can be seen in Figure 4.13. Not only does the frequency change decrease the number of IO requests that an application can make, but also decreases the performance of the driver. Therefore if the card is too heavily dependent on worker threads the IOPS will decrease even more. The figure also shows how drastically the decrease of frequency is on parallelism. The single job performance at a blocksize of 1KB drops from 12254 to 7411 IOPS. On the other hand the 32 job performance drops from 59093 to 29294 IOPS. This means that almost half of the performance has been lost. In fact the maximum of the full frequency cannot be reached. Observing the figure in the vertical direction it can be concluded that the loss of performance increase with increasing the number of jobs. Finally the limit is reached and increasing the number of jobs no longer increases the performance. This results in no performance change between having 8 jobs or more. By close observation and comparison with the bandwidth it is possible to detect a correlation between the drop in IOPS performance and the change in the bandwidth. At full CPU frequency the blocksize 16KB allowed high bandwidth due to a large

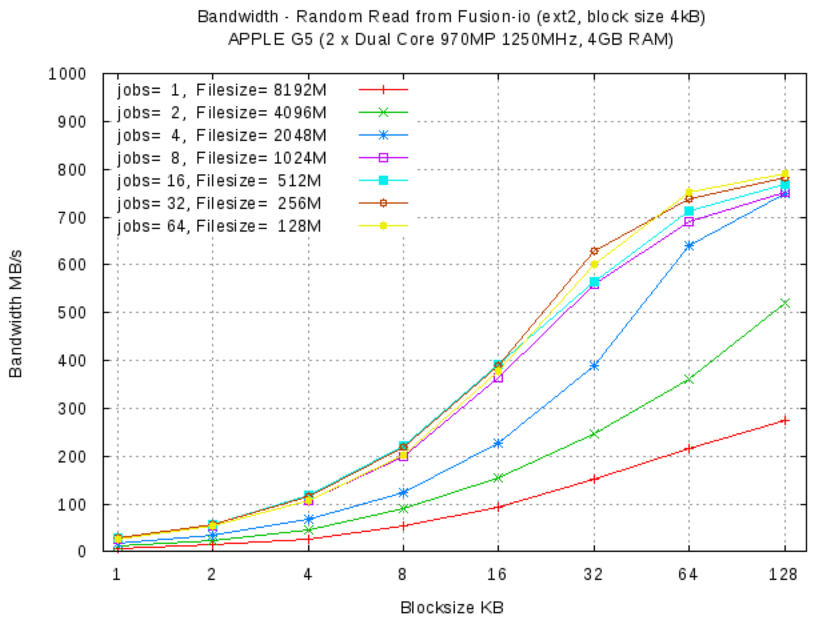


Figure 4.12: Bandwidth vs blocksize for ioDrive on an Apple G5 with clock set to 1.25GHz

number of IOPS. At half CPU frequency this is no longer the case. Therefore IOPSs decrease at low blocksizes pushes the bandwidth saturation to higher blocksizes.

As mentioned before the decrease in IOPS on decreasing CPU frequency is due to the performance drop in both request making and driver performance. Since on modern CPU increasing the frequency is not possible by design, other methods need to be found to increase performance. In case needing high bandwidth the blocksize has to be increased, as well as increasing the number of reading or writing threads. On the other hand achieving higher IOPS is not as simple as achieving high bandwidth. First the driver has to be optimized for lower frequency by pushing most services into hardware. SSD cards are should preferably perform garbage collection and other services on card and not in the driver. Another possibility is using multi-controller devices. This means that many drivers and file systems and IO stack operations will run in parallel. Increasing number of controllers should continue to increase the IOPS. In case the selection of appropriate device for read or write is kept to the application, the performance should linearly increase. This should continue until CPU utilization is at maximum. This is true if the application is simply mapped to one device. On the other hand, if the selection is done by the system, the overhead of selection will also take part of the CPU utilization. For example, a parallel file system needs to spawn worker threads that divide the file access on the available devices. One last factor remaining is the link speed. Once the PCIexpress bus is saturated the increase of devices used will not increase IOPS.

4.3.2 Testing Functional Partitioning

Mentioned in the previous section is the possibility of using functional partitioning to enhance the performance. This is especially true in case of having service threads that can be mapped on different cores. Additionally if the core have multiple threads, functional partitioning might be an optimization step. On such architecture avoiding placing two processes on the same core is

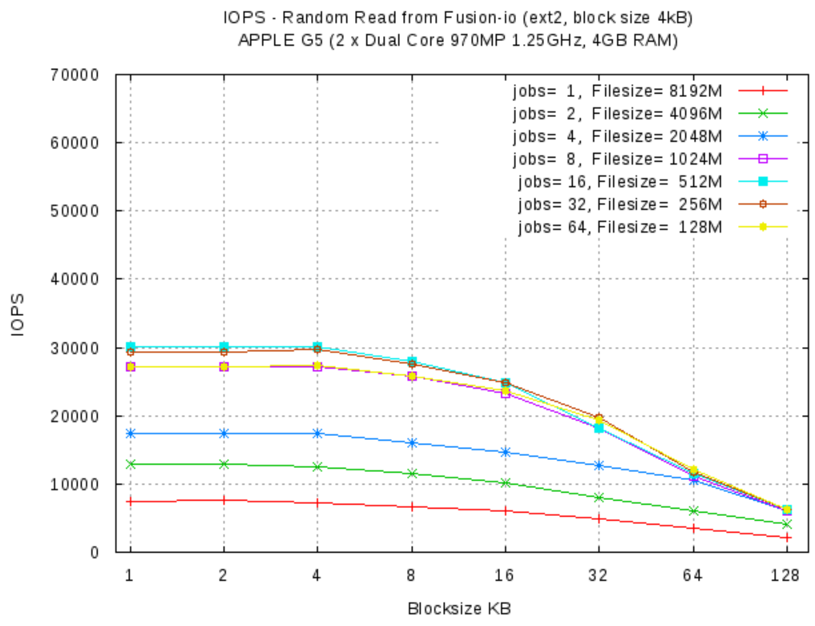


Figure 4.13: IOPS vs blocksize for ioDrive on an Apple G5 with clock set to 1.25GHz

helpful. For example, the Blue Gene/Q has 17 cores each contains 4 threads. The Linux Kernel however, has 68 possible CPUs to schedule the tasks to. This is because the Kernel considers every thread a CPU. Thus the Kernel might consider that giving a process exclusively CPU 1 means that the process is free to run at all times. Considering that CPU or thread 0, 1, 2 and 3 all share the same core, the Kernel is mistaken. Therefore a utilization of a CPU given by the Kernel is already at its maximum allowed utilization at only 25%.

Since the Fusion-io driver is closed source there is no method to know how the services are implemented. However by observing the CPU utilization during an IO test, tasks which take up most of the CPU time can be found. *top* which is a tool provided by Linux to show the current running processes can be used for that purpose. The tool has additional features such as showing CPU utilization and which CPU each processes is running on. By observing the *top* command output during FIO tests on the Fusionio card ioDrive, it was found that there are three tasks spawn by the driver.

A full observation of the CPU utilization during a FIO test is needed. Shown in Figure 4.14 is the CPU utilization in percentage during a FIO test. The test was done on a Blue Gene/Q, but limited the number of CPUs to 20. This helped observe the behavior over a limited number of CPUs. As can be seen the Utilization of the first few CPUs is very high. This indicates that some threads are needed by the driver to carry out the requests submitted by the FIO jobs. It is also worth knowing that CPU 0 is used for interrupt requests. That means that every interrupt is automatically routed to core 0 thread 0.

Considering that the first four CPUs share the same core, a CPU utilization of more than 50% for CPU 2 means that the core is taken for half the time by a single thread. That is why a better CPU scheduling might lead to improving performance.

There are three main components that can be edited to achieve a better scheduling. These are the Kernel scheduler, the FIO tool and the test script. By manipulating the Kernel scheduler the processes can be placed into the appropriate CPU. This however requires changing the Kernel

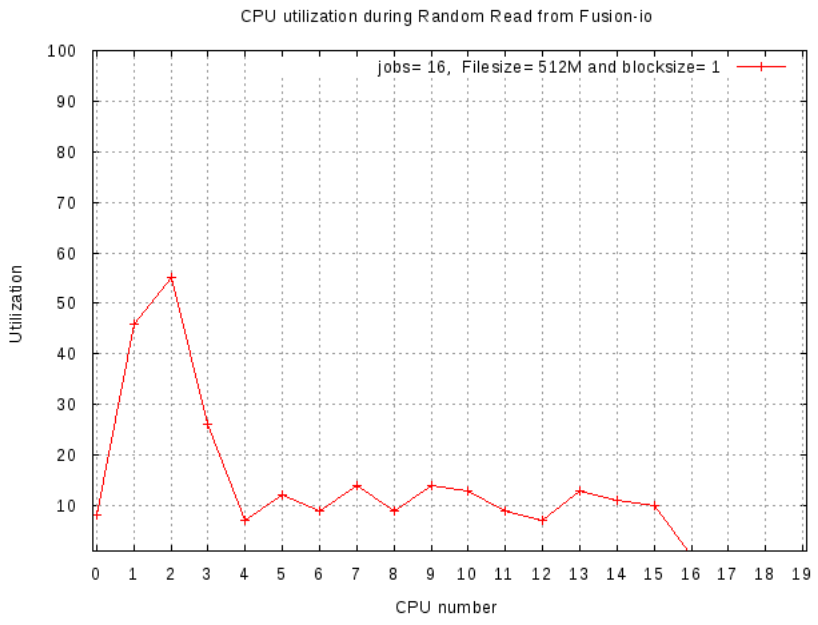


Figure 4.14: CPU utilization during FIO test reading from ioDrive on Blue Gene/Q

inner workings. Not only is this complicated, but also might disturb other side processes needed. This might unintentionally result in performance changes that would not be considered. The second option is to change the FIO tool. Although this is less complex it still involves the changing of a complex tool. Also the risk of unintentionally changing performance calculation through side effects is still present. In fact it is seldom a good idea to manipulate the tool used for benchmarking. The final option is changing the test script. Linux provides the option of setting core affinity for processes using the *taskset* command. The test script spawns the FIO job then iterates waiting for all jobs to spawn. Once the Process ID or PID can be found they are passed to *taskset*. The appropriate core is chosen and the thread is mapped. Changing the test script meant that there is a risk of part of the CPU utilization going to the script. This can be avoided by breaking the iteration once all processes are found. Another disadvantage for using the *taskset* in the test script is that the tasks could have operated for a few cycles on another CPU than the one intended by the script. This might result in error contained results. The margin of error can be decreased by increasing the time of FIO test operation. This will increase the average utilization of the intended CPU towards the ideal case.

The main target of the division of CPU among driver threads and the different FIO jobs is the minimization of collision. Therefore giving each task a CPU is not sufficient. As mentioned before each four adjacent threads share one core. Thus the target became giving each related IO job its own core. Since the first core receives the interrupts nothing will be scheduled on core 0. To the first three cores the three threads spawn by the ioDrive driver will be binded. This means that *taskset* will set the CPU affinity of these three threads to CPU 4, 8 and 12. Since there is no need to repeat the test for all different numbers of jobs it is sufficient to perform the test once. It was found that 8 jobs reach the highest performance possible for both IOPS and bandwidth on the Blue Gene/Q in combination with the ioDrive. Therefore performing the test once for 8 jobs is adequate. These 8 FIO jobs were binded using *taskset* to CPU 16, 20, 24, 28, 32, 36, 40 and 44.

Setting the CPU affinity of the processes as previously explained resulted in the CPU utilization shown in Figure 4.15. The utilization in this case has been taken over the complete FIO test. The average was then taken over the time. To ensure the accuracy of the process it was carried out using complex scripts. The scripts target finding the CPU and add up the utilization over thousands of *top* output. Additionally the script has to be capable of excluding irrelevant results such as the occurrence of a zero utilization before the job has been spawned. Another important aspect of the script is finding the average. Therefore each time a useful utilization value has been found a counter is increased by which later the total CPU utilization will be divided. Since the output of the *top* command is refreshed after a constant interval of time, the number of useful CPU utilization values is proportional to time. On these bases the CPU utilization given in the figure are acceptably accurate values. It is important to note here that the utilization values mentioned here are system utilization. These are therefore time spend by the CPU in creating IO requests or driver operations. The CPUs used for FIO job spend a high percentage of time in CPU IO utilization according to the *top* command output. This utilization percentage is the amount of time spend by the CPU waiting for IO. This reached values of more than 70%.

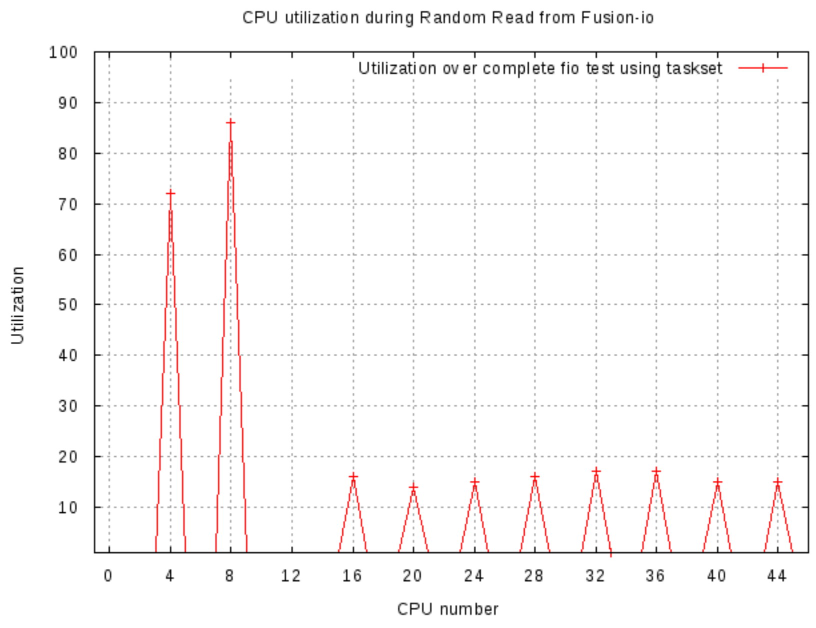


Figure 4.15: CPU utilization during FIO test and setting CPU affinity

Once the test is concluded the IOPS can be compared with the ones for a normal FIO test without setting CPU affinity. Figure 4.16 shows the percentage of improvement of using functional partitioning. The percentage is taken from the maximum achieved IOPS without setting CPU affinity. This is the IOPS at 1KB blocksize. The figure compares the performance of an 8 job test with and without using *taskset*.

Figure 4.16 shows an improvement for the usage of functional partitioning. This howeve decreases as expected by increasing the blocksize. The performance improvement comes from the fact that processes are utilizing a complete core. This means that the Linux Scheduler is not allowed to move the process around. This in turn leads to saving valuable switching time lost during process migration. Additionally the core is no longer shared among multiple processes running on different threads. This means that the process is allowed to use more than 25% CPU

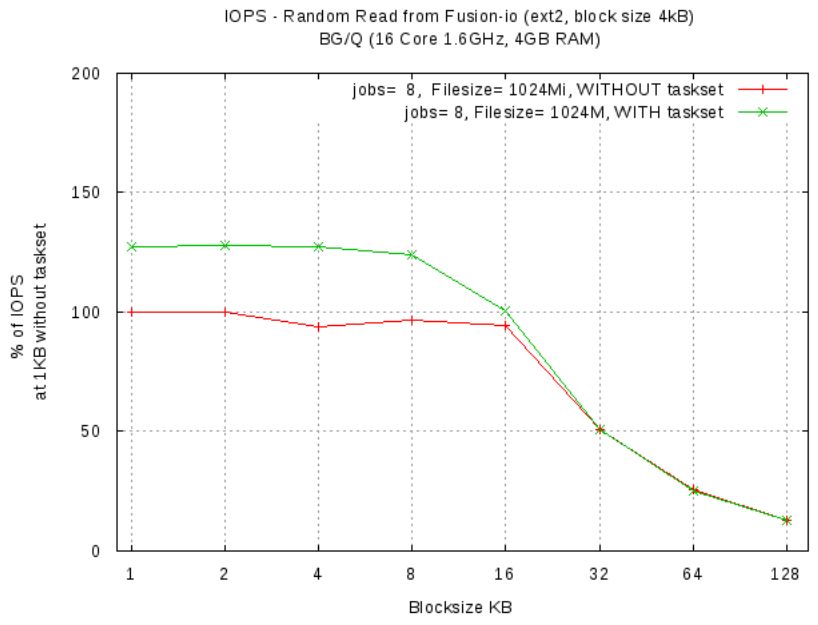


Figure 4.16: Percentage of IOPS improvement by setting CPU affinity

utilization without depriving other threads.

Although there is an achieved performance increase from using functional partitioning it is limited. As can be seen by Figure 4.15 the service processes used by the ioDrive already utilizes more than 85% of a single core. Considering that the FIO jobs all are using below 20% of CPU utilization, the bottleneck is assumed to be the driver threads. Therefore the only method for increasing the IOPS further is the use of multiple controllers. Another option is dividing the driver into several threads to perform the same task. This however might be complicated and is not possible without a deep understanding of the controller implementation. Nonetheless, this test proves that better performance can be achieved using functional partitioning. It also indicates that better resource management improves performance without the need for hardware upgrades.

4.3.3 Testing Parallel File Systems

The use of parallel file systems has been mentioned before as one of the most successful methods for increasing performance of IO. Modern servers depend on increasing number of hard disks to increase IO performance. To achieve that parallel file systems are used to handle the increasing number of devices. Therefore using parallel file system to combine the performance of multi-controller SSD is a well suited method for increasing parallel performance.

Given that IBM GPFS is the most commonly used file system for HPC, it is appropriate for using on the Blue Gene/Q. The main advantage in using GPFS is that normal POSIX can be used to access the file system. Therefore the GPFS created can be mounted and accessed just as any other file system. This makes the testing process easier. In fact the exact same script can be used making the results comparable to previous tests.

GPFS defines service and client nodes. The service nodes contains the storage devices which will be accessed by the client nodes. In order to achieve using the same node as a storage and processing unit, the Blue Gene/Q node is defined as both client and service node. This leads to having both threads operating on the same node. Another important issue with GPFS is that it only accepts certain types of devices. However these physical devices are missing on the Blue Gene/Q specially if it is necessary to test performance change with number of devices. This meant that GPFS had to be tricked to accept *ramdisks* as being actual physical devices. Considering that the *ramdisks* are suppose to perform better than any other available devices, scaling by increasing the number of devices will be due to the use of GPFS.

Under GPFS each device attached must be given a globally accessible NSD name. NSD stands for Network Shared Disk. It is expected that increasing the number of NSD should increase performance. The test therefore was performed using several NSD. Additionally the relation between blocksize and performance had already been established by previous tests. Thus the test can be performed using a single FIO blocksize. This was chosen to be 4KB. In addition to that the number of jobs are plotted on the X-axis. Considering that the test changes jobs and number of NSD keeping the blocksize constant makes the plot easier to read. In fact as will be seen later from the figures there is no longer a need to present both IOPS and bandwidth. The difference between the two becomes a constant factor which is the blocksize.

Figure 4.17 shows the bandwidth of different NSD counts. The bandwidth in the plot is represented as a percentage of the *ramdisk* performance. It should be noted that the *ramdisk* is mounted using an Ext2 file system.

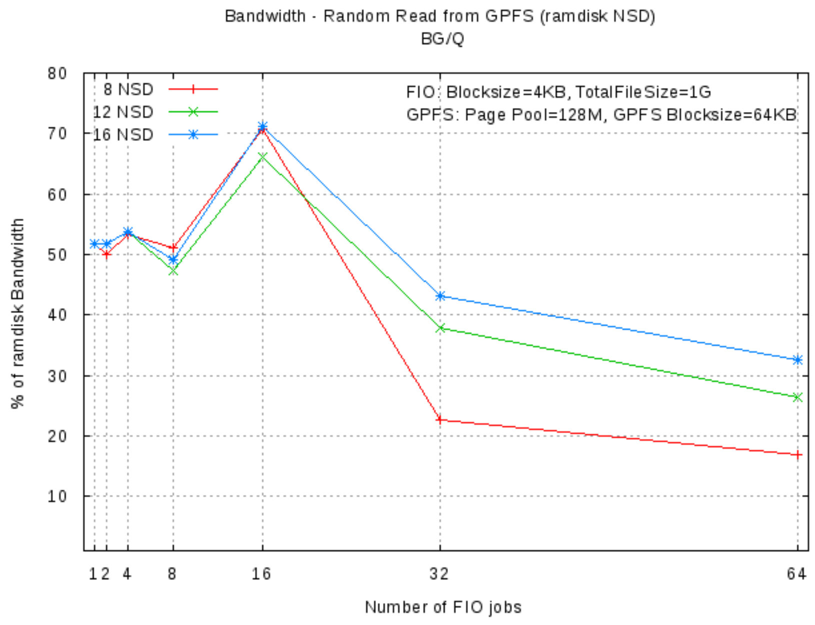


Figure 4.17: Percentage of ramdisk bandwidth vs jobs for GPFS with different NSD

There are many observation that can be drawn from Figure 4.17. First increasing the number of NSD increases performance for higher number of jobs. This was expected as more jobs are needed to saturate the bandwidth at a constant blocksize. The same can be observed for IOPS as seen in Figure 4.18. Although the figure might give the illusion that performance drops for higher job numbers, this is not true. The *ramdisk* bandwidth increases almost linearly with the increase of number of jobs. The figures therefore indicate that bandwidth using GPFS does not increase as quickly as that of a single *ramdisk* when increasing number of jobs. Therefore the percentage drops as seen in the figure.

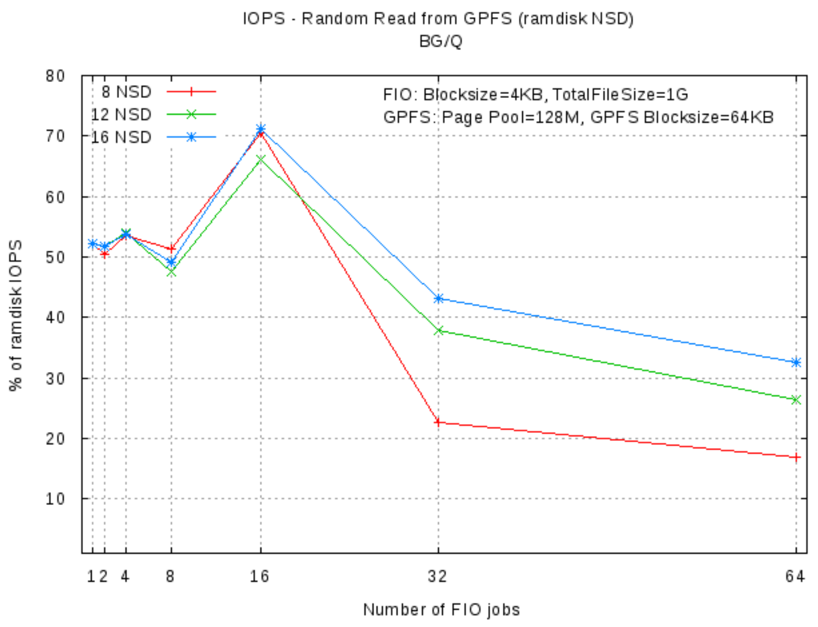


Figure 4.18: Percentage of ramdisk IOPS vs jobs for GPFS with different NSD

Considering that each of the NSD used by GPFS is a *ramdisk* there is an obvious massive drop in performance. For example, best bandwidth achieved by a GPFS with 8 NSD is 70% of single *ramdisk* performance at 16 jobs. Instead, it was expected that the performance be close to 8 times that of a single *ramdisk*. This could be explained by the number of threads that GPFS has to use to service the file system. In other words, GPFS represents a large overhead. This overhead leads to the decrease of performance. However it should be noted that this comparison is to some extend unfair. The *ramdisk* being part of the RAM is the fastest storage possible. Nonetheless the test shows the massive impact that a complex file system can have on performance. Therefore it might be more efficient to use other methods for combining multiple devices on a single node. For example, combining several devices in the driver is more appropriate for this case. It still remains to test how the overhead of GPFS will affect performance of multi-controller SSD. Although the overhead might not limit device capabilities as drastically as is the case with *ramdisk*, still there will be limitations. On increasing SSD controllers the performance will be at some point limited either by the number of processes the diver spawns or GPFS. A good parallel driver implementation should result in the GPFS overhead limitation to be reached first.

4.3.4 Testing Loop Devices Parallelism

The SSD market is one of the most expensive. A single SSD card can cost as much as 15K dollars. This is not the only reason for SSD limited available testing time. Testing hardware means that the setting cannot be changed. Therefore only the construction given by the manufacturer can be tested. This indicates the need for a more flexible testing environment that exhibits some of the SSD features. *Ramdisks* might appear as a suitable candidate. Random access cannot be dominated by seek time which makes it close to the performance of SSD. Another advantage of *ramdisks* is the ease of setting and the open source module. This makes it easy to change and manipulate the implementation of *ramdisks*. The disadvantage however is that a *ramdisk* has almost no driver like functions. The *make_request_fn* eventually leads to a simple *memcpy*, which can be seen by Figure 4.19. This means that there is almost no delay for accessing a *ramdisk* and there is no need to spawn any service threads. Therefore the CPU will never present a possible bottleneck. This results in the *ramdisk* being unsuitable for further test implementation.

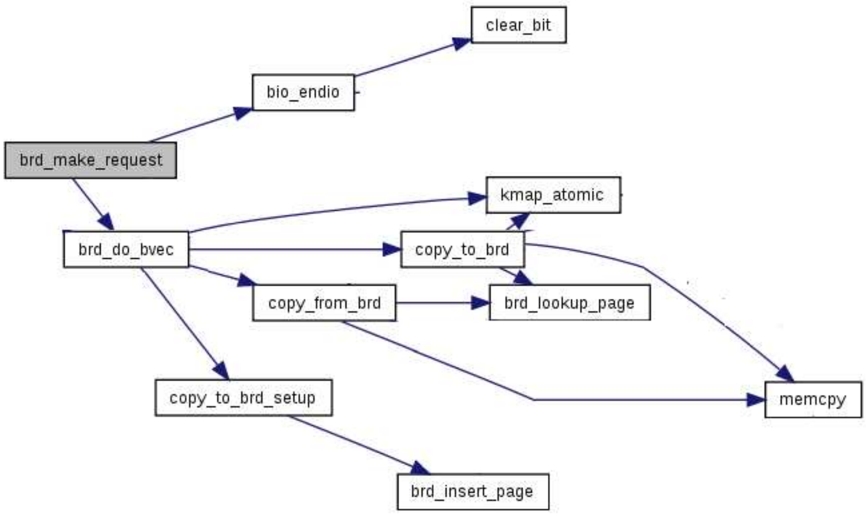


Figure 4.19: *brd_make_request* read or write result in a simple *memcpy*

Another viable candidate is the loop device. By using the loop device the access to files can be emulated as being a block device. The main advantage for using a loop device is that it spawns a thread that deals with the device access. This means that the loop device has the similar limitations to that of a regular SSD. An additional advantage is that the loop device is setup on top of a file. That file can be allocated anywhere. Therefore if the file was allocated on a *ramdisk* the seek time would be zero. This results in an almost equal relative performance to that of a SSD.

In order to estimate parallel performance of a loop device there is a need to measure how well does a single device deal with the increase of job number. As the number of jobs scale so should the performance. The result of the bandwidth ratio against a single job is shown in Figure 4.20. The ratio shows how the performance is marginally improved by using an additional job. The figure shows how the performance is doubled at low blocksizes. At higher blocksizes there is a 50% or less bandwidth increase. Additionally there is no increase in bandwidth by adding more jobs. In fact the bandwidth almost does not increase passed the performance of a two job FIO test. This indicates that the limitation factor of the loop device is the thread created to perform the read.

The exact same observation can be drawn from Figure 4.20. This shows the ratio of IOPS between single and multiple jobs random reading from a single loop device. The result is almost the exact factor distribution as that given by the bandwidth. Given that there is a direct relation between IOPS and bandwidth which is governed by the blocksize the reason becomes obvious. The figures are based on a ratio that eliminates that factor. The ratio is the division of bandwidth or IOPS values with that of a single job at the same blocksize. That way the blocksize effect is eliminated.

Figure 4.20 and Figure 4.21 give the illusion that performance decreases with increasing

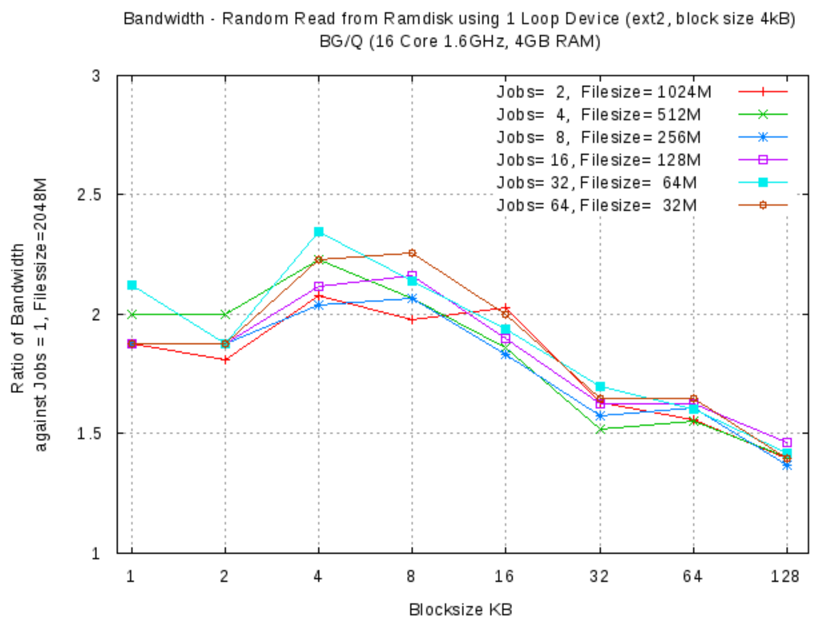


Figure 4.20: Ratio of bandwidth between single and multiple FIO jobs random reading from a single loop device

blocksize. This however is false. Increasing the blocksize still increases the bandwidth and decreases the IOPS. The figures simply point out that performance increases more using smaller blocksizes. There is a simple explanation to that. As mentioned before IOPS are affected more by parallelism than bandwidth. Given that IOPS increase by decreasing the blocksize the difference becomes more visible at smaller blocksizes. As for the middle values of blocksize both the IOPS and bandwidth are at average values. Therefore the factor of increase of performance against that of single job appears to be high.

The main observation that needs to be drawn here is that loop devices do not perform better when using more jobs. Therefore loop devices have low parallelism. As mentioned before this makes them an almost ideal candidate for testing. If it is possible to increase the performance of the loop devices without major changes to the loop device itself it might be possible to do the same for SSD technology.

The multi-controller configuration of an SSD can also be applied to loop devices. The FIO test script can use the *filename* to point to several different block devices. Using this concept and setting up multiple loop devices the parallelism can be tested. It is expected that using multiple loop devices will increase performance when increasing the number of jobs. Therefore the test was done against the increase of number of FIO jobs used. This in turn makes the resulting plot more readable. The blocksize has been kept constant at 64KB over the entire test.

Figure 4.22 shows the ratio of IOPS between using a single and multiple loop devices. The plot shows the results for three settings using 2, 4 and 8 loop devices and reading from them in parallel. As seen at higher number of jobs the performance increases when using more loop devices. The increase however is not directly proportional to the number of loop devices used. As seen the performance nearly doubles at 64 jobs when using two loop devices. On the other hand when using 8 loop devices the performance factor is only around 5.5 at 64 jobs. If the performance increase was linear the 8 loop devices would result in a performance factor increase

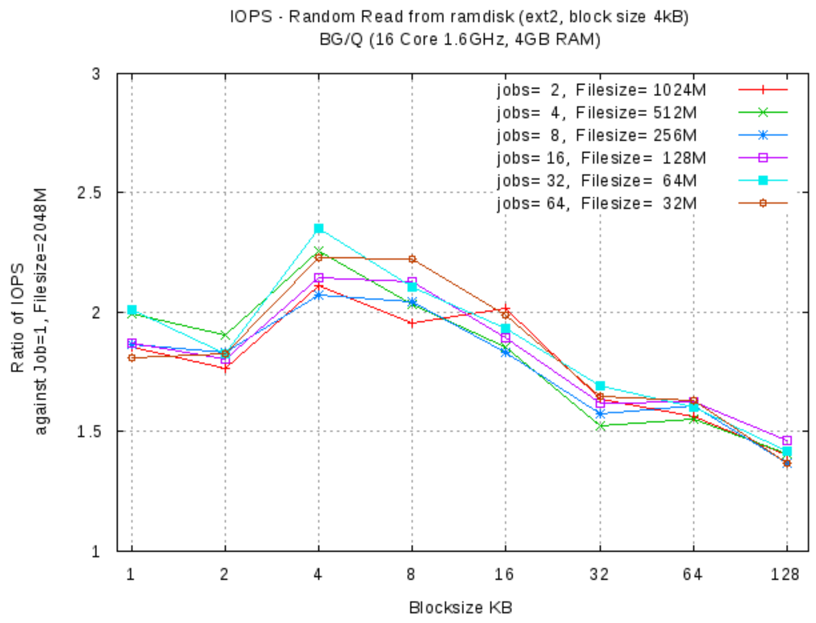


Figure 4.21: Ratio of IOPS between single and multiple FIO jobs random reading from a single loop device

of 8. This however is not possible. As the number of loop devices increase so does the overhead. Each loop device spawns a separate thread which uses up part of the CPU utilization. Another reason for the performance increase factor limitation is job device access time. As the number of devices increase so does the time needed for the jobs to access each of the devices. The access pattern has been explained before for a multiple device FIO test. As mentioned each device and therefore each file is accessed by all jobs. There is no functional partitioning in this case. However this is similar to the expected result of using multiple devices. In fact a GPFS division among the devices would have been much different. GPFS would stripe files across all devices. Therefore each file accessing all files would access all devices.

Another important observation that can be drawn from Figure 4.22 is the change of increase saturation point. As can be seen, using two loop devices increases performance until using 4 jobs. After that point using more jobs does not increase performance. In comparison using 8 loop devices keeps increasing IOPS all the way till 16 jobs. Using more jobs on 8 loop device almost does not increase the performance. This can be explained by the number of jobs that are needed to fully utilize a loop device. Shown by Figure 4.20 after using two jobs reading from a single loop device performance is not increased. Therefore using multiple devices would not increase performance passed using more than two jobs per loop device. This leads to the performance of a two loop device test to saturate at 4 jobs. In turn it leads to the IOPS of a 4 and 8 loop device test to saturate at 8 and 16 jobs respectively.

There is no reason to show the plot for bandwidth. This is because the bandwidth plot is the exact same as the one shown for IOPS. As has been explained before plotting both IOPS and bandwidth against number of FIO jobs will give the same curves. Considering that Figure 4.22 shows a ratio between performance of single and multiple loop devices, the bandwidth plot looks exactly the same. Even for IOPS and bandwidth against blocksize there would be no difference between the two plots. This is due to the ratio removing the blocksize as the factor governing

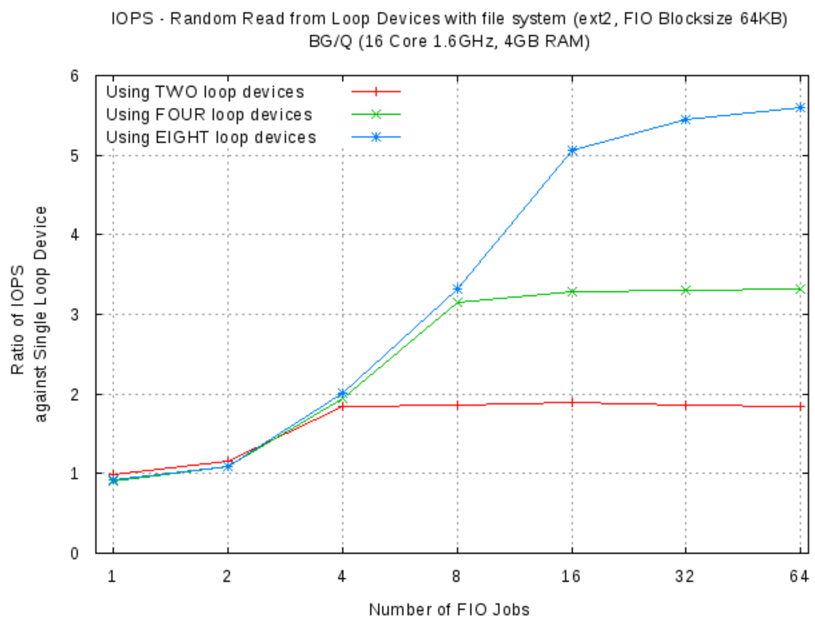


Figure 4.22: Ratio of IOPS between single and multiple loop devices

the difference between IOPS and bandwidth. That has already been shown by Figure 4.20 and Figure 4.21 which show no difference between IOPS and bandwidth when ratio is used.

4.4 Implementing A Parallel IO Design

As defined by the previous shown tests the optimization is dependent on a lot of factors. The one with the most performance increase is using multiple devices. The rest of the optimization factors can be done on top of using multiple devices. As an example, using functional partitioning for a multiple device setting can optimize performance further. There are other important observations that can be drawn as well. One such observation is the use of loop device to test improvements to SSD multiple device settings.

An important issue when dealing with multiple devices is where to combine the performance. In the tests shown in section 4.3.4 on testing loop devices the parallel performance has been achieved on the application layer. In other words the FIO test recognizes all devices and distributes them accordingly. As mentioned before FIO just makes all jobs read from all available devices. Although this seems as a possible solution it requires changing the applications. Additionally this does not allow for global distribution of performance. This can be implemented if the application has some information on the global state of the system. Since this will lead to drastic changes to the application and an over complication of implementation, this is not a possible solution.

Another layer available for combining multiple block devices is the block layer. The best position would be the driver implementation. In fact there are some cards that contain multiple controllers which are shown to the system as a single block device. The RamSan-70 is one example. The main advantage is that such implementation avoids changing the application layer. Additionally it avoids the complexity of having to deal with multiple block devices on a system level. In other words implementing a driver for multiple devices avoids the need for using complicated parallel file systems. The disadvantage however is that the block layer does not have a global view of the system. Therefore, although the driver can distribute requests over the available devices it cannot achieve any function binding. An important example for such a case is write versus read distribution. On receiving a write request the driver can distribute the request on any available device. This is specially true for SSD. These contain a flash translation layer (FTL) which is responsible for mapping addresses. Therefore changing the address to another device is allowed by the driver. In comparison when a read request arrives the driver can no longer map it to any device. The read has to be done from the block device on which the data is stored. This means that the driver is no longer free to choose the distribution. Given that data is usually read more than written, the driver is not always free to choose the distribution suitable.

The third possible option for utilizing multiple block devices is on the system layer. One possibility is to allow the operating system to bind applications to certain block devices. This might be effective if the operating system has information on data read and write done by applications. However further complicating the operating system of single compute nodes is not a good approach. On the other hand the most commonly used method on a global level is parallel file systems. Not only do they provide a global access to all devices, they also can manage data distribution. Also by adding data layout awareness into the file system data access can be optimized for different access patterns. On the other hand there are many disadvantages to using a parallel file system. As previously shown parallel file system can represent a massive overhead. This is specially true for complex global file systems. Additionally to achieve full parallelism some parallel file systems stripe over multiple devices. In turn this leads to applications having to read from all devices simultaneously. This might be counter productive.

Considering the previous points and the tests made in the previous section, there is room for improvement. There is a need for a better distribution pattern of accessing different block devices. Another factor for increasing freedom of access is being able to write then read the same data from different devices. This will allow the most optimum use of device allocation. The file system or operating system is free to distribute applications or access patterns among block devices. Therefore reaching best performance through both division of task and functional partitioning.

Figure 4.23 shows a suggestion for implementing such parallel device system. The system contains a single uniform file system that has a view of the entire storage space. Given that the file system can be mapped to the uniform storage space there is no need to use a complex parallel file system. This is the most important point of the design. The storage space is a uniform single unit space. Thus all block devices or controllers can use any available block device to access any block in the memory space. Not only does this serve the use simple file systems, it also means that any application can use any block device to access any data block. In fact the file system is free to distribute the access as it considers best. The same data can be read and written from two different block devices. This means that any suitable access pattern is possible. The file system can schedule a write on one block device and read the data later from another. Furthermore the file system can be any common file system. The edit is confined to the access of blocks. This means that the file system can operate as it would on any other single block device.

The design shown in Figure 4.23 can be implemented for testing. Considering that implementing a uniform storage on a SSD requires changes on the hardware level the design will be implemented using virtual devices. The following sections will show an implementation of the given design. It has to be noticed that the main difficulty in the design was changing as little as possible. The Kernel should not be drastically changed. Additionally the Ext2 which is the file system used for testing the implementation concept should also not be reimplemented from

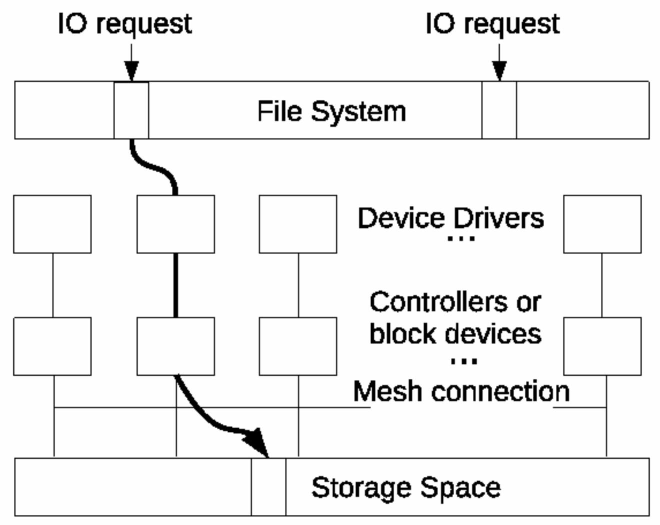


Figure 4.23: Implementation of parallel device system

scratch. The target is to achieve a better performance with as little disruption to the file system as possible. In some cases the file system was found to handle issues that have not been anticipated. That resulted into an even simpler design.

4.4.1 Block Layer Implementation

The first level of implementation is getting two or more block devices to recognize and use the same storage space. One complex solution would be to edit the *ramdisk* module available in the Linux Kernel. The idea is to get all *ramdisks* to access the same memory space. Therefore implementing the required link. As mentioned before this requires complex changes to the module. An additional disadvantage to using *ramdisks* is the fast access that these devices exhibit which was proven by previous tests. This therefore would mean that the results are not comparable. Another problem is that a *ramdisk* operates directly on RAM. Manipulating access patterns means risking accessing wrong memory locations. In turn this might result in unrecoverable errors. Thus leading to complex debugging sessions. Additionally the performance using such changes would not be comparable to that of a single *ramdisk* due to module changes.

This step has to be achieved with as little change to the device as possible. As demonstrated before the loop device is a suitable candidate for testing performance changes. Additionally using the loop device would avoid some of the difficulties exhibited by the *ramdisk*. The loop device is set up on top of a file. Therefore using it would avoid direct memory mapping. This in turn would avoid the need for complex debugging. In fact it was found that this can be achieved with no changes to the loop device module itself.

Creating a loop device is done by running the *losetup* command. The parameters required in this step are loop device to setup and the file on top of which the loop device will be set. The command does not check if the file is used by another loop device. Thus repeating *losetup* for different loop devices using the same file is possible. During a read or write operation these loop devices will therefore access the same storage space.

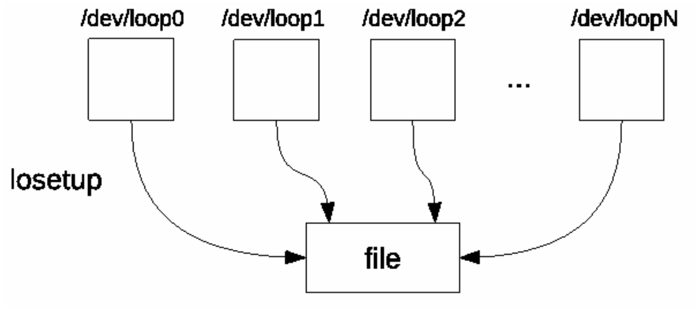


Figure 4.24: Linking multiple loop devices to a single file

4.4.2 Registering Multiple Devices

Given that the file system has to access many devices it has to know which devices it can access. How a file system registers the device it is stored on has been explained in the VFS Section 3.3 and individual file system Section 3.4. The target here is to limit changes to that of the file system. This will facilitate testing changes. Additionally by avoiding extensive changes to the Kernel other file system operations will not be affected. However since the Kernel needs to register the possible block devices the VFS has to know which ones to associate with the file system. The VFS is responsible for runtime variables that are set to cater for the file system operations. Although it is possible to store the names of the devices and there number on the file system, this will only be a temporary or a testing solution. The proper technique is for the VFS to detect how many devices the file system contains. The next step is for the VFS to associate those devices to the file system.

Given that the test is performed using loop devices it is possible to have a firm set count of loop devices needed. That count could be kept in the file system. Another option is to add an integer to the VFS. Although it is a must to add the block devices associated with a file system to the VFS, it is preferred to not add the count into the VFS directly. The number of block devices used in accessing the file system is a factor that would be changed constantly. Given that the VFS changes result into a full recompile of the entire kernel, adding the count into the VFS is counter productive. To achieve a dynamic environment the best solution is to form a list of block devices. The advantage of lists is the changeable length, which is not the case with arrays.

The Kernel already presents a list implementation. This makes forming a block device list much simpler. However a new *struct* needs to be defined. The code for this *struct* is shown below. This contains three variables. The *struct block_device *s_bdev* is the block device variable. This will be later used as an identifier for block access. It has to be noticed that the *block_device* does not contain the regular block device name such as */dev/loop0*. Later in the process of mounting the file system this variable will have to be found. The next variable is the list variable defined by the Kernel. The source code for the implementation of lists in the Linux Kernel can be found in *include/linux/list.h*. Using predefined lists in Linux is quite simple. All needed functions are already implemented. The ones used will be explained when required.

```
struct list_loop_bd {
    struct block_device *s_bdev;
    struct list_head list_s_bdev;
    atomic_t access_count;
};
```

The third and last variable in *struct list_loop_bd* is *atomic_t access_count*. This is an integer that is used for counting how many request currently access the device. The important aspect of this variable is using the *atomic_t* provided by Linux. Instead of using a normal integer the atomic integer is used to avoid possible race conditions. This is necessary given that multiple processes can operate on the device at the same time.

As proven earlier the VFS has to register the list of block devices. The next step is to find where to register the list within the VFS. As shown in Figure 3.3 there are many different runtime variables that could contain this list. The best approach is to add the list into the *superblock*. This is because the *superblock* exists once for each file system. In fact the *inode* does not contain any pointer to the block device on which the file system exists. Therefore during file operations the *inode* points to the *superblock* and then points to the block device variable in the *superblock*.

Figure 4.25 shows part of Figure 3.3 with the location of the block device list. As seen the list has been added to *struct super_block*. This is shown by the circle in Figure 4.25 (1). The list variable is defined as shown in the code segment below. Another important observation is that the variable *s_bdev* has not been removed from the *superblock*. This has two main reasons. The first is as mentioned before changing as little as possible in the Kernel implementation. The second and most important reason is that the *struct super_block* is used by all file systems. If the variable *s_bdev* had been deleted these file systems would have required patching. Therefore keeping the variable demonstrates how simple it is to add functionality without disturbing normal Kernel processes.

```
struct block_device *s_bdev;
/*
 * list of loop block devices controlled by the super_block
 */
struct list_loop_bd *loop_s_bdev;
```

The next step is to fill the list with block devices.

4.4.3 Mounting Ext2 with Multiple Devices

Mounting a file system is a complex operation that involves creating many complex components. These will later be used to access the file system, ensure coherence and proper operation. Once again the target is to confine the changes to the Ext2 module. Even then changes should be kept at a minimum. Given that Ext2 is compiled into the Kernel by default, the Kernel configuration should be changed. Another possible option is to compile the Ext2 module with a different name. This approach is useful in case the original Ext2 is needed by the Kernel for booting. In fact it was found that some Linux Kernel versions mount the first initial *ramdisk* as an Ext2 file system. These Kernels cannot boot if the Ext2 file system is not compiled into the Kernel. The target then is to change the Ext2 enough for it to be registered by the VFS as a different file system. This can be achieved by replacing every account of the Ext2 name by another. That way one guarantees that the file system is unrecognizable by the VFS as the old Ext2. The problem here is that even Ext2 interface files need to be changed. This includes not only the files found in *fs/ext2* but also the Ext2 files found in *include/linux*.

The difficulty with recompiling the Ext2 as a different file system becomes obvious once the file system is used. Changing the Ext2 module means that the file system is registered by the VFS as a different file system type. However the tools needed to use the newly defined file system are missing. The main problem is formating a device using the new file system. The

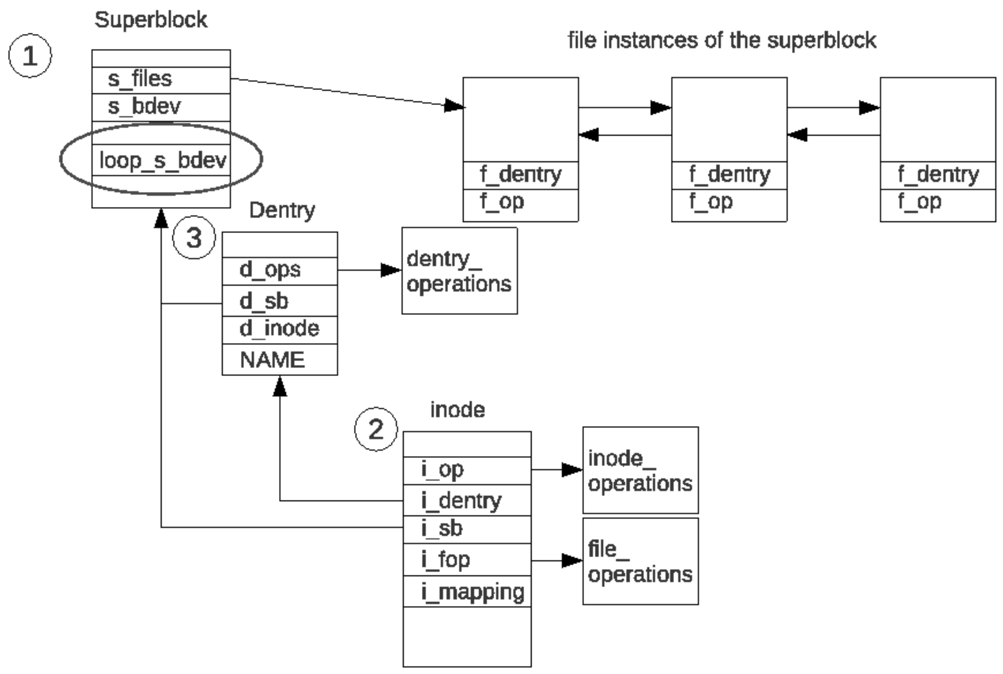


Figure 4.25: Adding list of block devices to VFS components

tools available for formating an Ext2 file system are externals added tools. These need to be rewritten to format using the new file system. Attempting to use the new file system to access a device formatted using an Ext2 tool will fail. The VFS checks the file system type before mounting. If the type does not match, the mount process is aborted. This indicates that the tools for formatting a device using the new file system requires rewriting Ext2 existing tools. This however is a complicated process.

The case for compiling the Ext2 as a new file system is unavoidable in case the Kernel does not accept loading the Ext2 as a module. However the devices used for testing do not need such drastic measures. Thus the Kernel configuration file can be updated to allow the Ext2 use as a module. The Ext2 can then be compiled and loaded after the machine has successfully booted. Compiling a Kernel for the Blue Gene/Q without the need for the Ext2 module is possible. Also booting a node using that Kernel does not show any problems. However the Ext2 mount has to be forced using the *type* option for the mount command. This is to avoid the Kernel using any other file system type to mount the Ext2.

The mounting process is carried out by VFS. This might indicate the difficulty of setting multiple block devices without changing the VFS. Nevertheless the VFS mounting process is dependent on individual file system implementation of certain helper functions. Given that each file system has a different architecture, the VFS has to employ the helper functions to fill the runtime variables. Therefore the VFS requests the file system to define these functions into the *file_system_type*. Shown below is the *file_system_type* definition of the Ext2. The code shows two important variables *get_sb* and *kill_sb* which are set to *ext2_get_sb* and *kill_block_super* respectively.

```
static struct file_system_type ext2_fs_type = {
    .owner      = THIS_MODULE,
    .name       = "ext2",
    .get_sb     = ext2_get_sb,
    .kill_sb    = kill_block_super,
    .fs_flags   = FS_REQUIRES_DEV,
};
```

On mounting a new file system the VFS refers to these pointers to access the functions needed. *get_sb* is used to get the *superblock* from the file system. As mentioned before VFS is optimized for using Linux native file systems. Therefore *ext2_get_sb* only needs to call *get_sb_bdev*. Still the VFS requires the file system to fill out the *superblock*. This however cannot be done using generic VFS functions. Thus *ext2_get_sb* has to supply a function pointer to *get_sb_bdev*. The function pointer has to point to a *fill_super* function. In case of of the Ext2 the function is called *ext2_fill_super*. As the name suggests the function has the task of filling the VFS *superblock* with the required data from the file system. Therefore the best position to initialize the list of loop devices is in *ext2_fill_super*.

It is necessary to mention here that the mounting process might be different for other Kernel versions. On the Kernel used on the *x86* system the Ext2 *file_system_type* did not contain a *get_sb* variable. Instead it contained a variable called *mount* which was set to *ext2_mount*. The function *ext2_mount* then calls a generic function from the VFS called *mount_bdev*. This function however still requires a *fill_super* function pointer. In turn the Ext2 uses *ext2_fill_super*. That means that the changes made on the *x86* Kernel are compatible with the changes made on Blue Gene/Q. The code for the initialization of the block device list is shown below.

```
    /*My variables*/
    fmode_t mode = FMODE_READ | FMODE_EXCL | FMODE_WRITE;
    int nr_loop_dev;
    int total_loop_dev=6;
    struct block_device *bdev;
    struct list_loop_bd *list_bdev;
    char loop_devices[20];

    /* Adding the additional loop devices */
    sb->loop_s_bdev = kzalloc(sizeof(struct list_loop_bd), GFP_KERNEL);
    INIT_LIST_HEAD(&(sb->loop_s_bdev->list_s_bdev));

    /* Create a new list_loop_bd */
    for (nr_loop_dev=0; nr_loop_dev < total_loop_dev ;
            nr_loop_dev++){
        sprintf(loop_devices, "/dev/loop%d", nr_loop_dev);
        bdev = open_bdev_exclusive(loop_devices, mode, sb->s_type);
        //bdev = blkdev_get_by_path(loop_devices, mode,
        //          sb->s_type);
        list_bdev = kzalloc(sizeof(*list_bdev), GFP_KERNEL);
        list_bdev->s_bdev = bdev;
        list_bdev->s_bdev->bd_super=sb;
        atomic_set( &list_bdev->access_count, 0);
        list_add(&(list_bdev->list_s_bdev),
       &(sb->loop_s_bdev->list_s_bdev));
    }
```

The code given shows the adjustment made to *fill_super*. These target the initialization of the block device list with the loop devices. The code starts with defining support variables. The most important variable is *total_loop_dev* which represents the number of loop devices that will be used. The code then initializes the list using *INIT_LIST_HEAD*. Filling the list is done in a loop. Each iteration a loop device is added to the list. The loop has to find the correct loop device name. Given that loop devices are named in ascending order, the loop adds a count to the string */dev/loop*. The resulting string however does not identify the block device on a Kernel level. The identification as explained in the block layer Section 3.5 is done using a unique *struct block_device*. To reach from the name of the block device a special function has to be used. The function is called *open_bdev_exclusive*. It opens the block device by returning the *struct block_device* associated with the given name.

Two lines are commented into the *fill_super* code given above. These show that *struct block_device* is found using a different function called *blkdev_get_by_path*. There is no *open_bdev_exclusive* on the Kernel used on the x86 system. Therefore a replacement was found. As pointed out earlier the Kernel changes rapidly. Not only does the implementation of functions change but the interface as well. Patching the Kernel is a simple method for moving changes from one machine to the other. Despite that the practical use of patches is difficult. Changes can seldom be added to another Kernel without checking functionality.

The *fill_super* has to set the *list_bdev* that will be added to the list. It does that by setting *s_bdev* to the found variable using *open_bdev_exclusive*. It also needs to set *sb* of the block device to the *super_block* being filled. This is required as the VFS needs to know which *superblock* to use when accessing a block device. Using the atomic function *atomic_set* the access counter of the device is set to zero. Finally the complete list element can be added to the list. The process shown here will be done once for mounting the file system. It therefore does not need to be optimized further. An advantage to that is the possibility of adding *printk* that would print messages to the Kernel log. These can be checked to confirm that the correct mounting process has been performed.

4.4.4 Unmounting Ext2 with Multiple Devices

Multiple devices and variables have been created to mount the Ext2 file system. Given that these are not considered by the original Ext2 implementation, the normal unmount operation does not remove these devices or variables. There is a need for a clean unmount process. The file system needs to be tested using several different configurations. If the file system cannot be properly unmounted the whole system will need to be restarted to apply these changes. It is therefore important to edit the unmount process to undo the changes done during mounting.

The Ext2 *file_system_type* defines a variable for removing the *superblock*. The VFS will call *kill_sb* when unmounting the file system. This pointer is set by Ext2 to *kill_block_super*. The problem here is that this function is a VFS generic function which is used by many other file systems. Changing *kill_block_super* therefore is not possible.

This is where the logic of the VFS implementation can be used. The Ext2 can change the function called for removing the *superblock*. This can be achieved by setting *kill_sb* to a different function. In this case the function is called *kill_block_super_loop*. The code of the function is shown below.

```
void kill_block_super_loop(struct super_block *sb)
```

```
{
    struct list_head *p;
    struct list_loop_bd *tmp;
    struct block_device *bdev = NULL ;
    fmode_t mode = sb->s_mode;
    list_for_each(p, &(sb->loop_s_bdev->list_s_bdev)){
        tmp = list_entry(p, struct list_loop_bd, list_s_bdev);
        bdev = tmp->s_bdev;
        bdev->bd_super = NULL;
        sync_blockdev(bdev);
        WARN_ON_ONCE(!(mode & FMODE_EXCL));
        blkdev_put(bdev, mode | FMODE_EXCL);
    }
    kill_block_super(sb);
}
```

Removing a block device requires calling several functions to perform a cleanup. *kill_block_super_loop* loops over all devices in the list using *list_for_each*. This is a Kernel defined function. For each device the *bd_super* has to be set to *NULL* to set the device free for another mount. Next *sync_blockdev* is called. Performing a synchronization makes sure that all data cached will be written back to the device. In turn preventing data loss. The final step is to put the block device. Therefore *blkdev_put* is called to free the block device.

As mentioned before the original Ext2 mount process called the function *kill_block_super* during the unmount process. Since there is no need to repeat the functions implementation the function is just called. Again this shows that the design makes room for normal Ext2 operations to function without disturbance. An Ext2 file system that does not require the multiple device design can disregard the mounting and unmounting of the loop devices. Eventually the file system will be unmounted as a normal Ext2 using the *kill_block_super*.

4.4.5 Accessing Ext2 Multiple Devices

The complex call graphs for reading or writing a file has been explained in the VFS Section 3.3. The data access explained has stopped at calling the *direct_IO* function implemented by the file system. The only interest in following this long function chain is to locate the point at which the device used is found. The Ext2 file system sets the *direct_IO* to its own implementation *ext2_direct_IO*. This function has to pass the device on which the file system is present to *blockdev_direct_IO*. This function has been explained in the block layer Section 3.5.

The function *ext2_direct_IO* is a good position for selecting the device. All functions below that are implemented into the block layer. On the other hand any function above *ext2_direct_IO* does not contain any reference to the block device. Shown below is the code for *ext2_direct_IO* function. As can be seen there is a block device pointer passed to *blockdev_direct_IO*. This block device comes from the file system. The *inode->i_sb->s_bdev* is the pointer to the block device element in the *superblock*.

```
static ssize_t
ext2_direct_IO(int rw, struct kiocb *iocb, const struct iovec *iov,
            loff_t offset, unsigned long nr_segs)
{
    struct file *file = iocb->ki_filp;
    struct inode *inode = file->f_mapping->host;

    return blockdev_direct_IO(rw, iocb, inode, inode->i_sb->s_bdev, iov,
                offset, nr_segs, ext2_get_block, NULL);
}
```

According to the implementation of *ext2_direct_IO* the block device is selected and handed over to the next level. This however is misleading. This device is only used for finding some mapping information on the block device. Although the use of different loop devices to call *blockdev_direct_IO* might result in correct output, it will not increase the performance results. This is because the previously mentioned loop devices all have the same mapping information. Choosing the block device from which the data is finally read is called somewhere else. To discover whether or not the additional devices are being accessed the *top* command output has to be observed. Every loop device on setup spawns a new thread. These threads have to be active during the read or write operation. Therefore if only one loop device is active then the file is being read or written to by a single loop device.

Finding the actual device which is used for the read or write operation is crucial for the implementation to be able to use all loop devices. The best method is to start at the lower level. The function used for adding a request to the block device is *make_request*. Since this function is depended on the device driver, the function calling it must be the one deciding on the appropriate device. This function is *__generic_make_request* which gets its block device from the *bio* that it takes as a parameter. Therefore it is important to find the point at which *bio->bi_bdev* is set. Given that this variable has to be set by either the block layer or the file system the search is difficult and long. The best method to facilitate the search is to take a look at the call graph that leads to *__generic_make_request*. This time however it is important to move backwards in the call graph to find the source and not the result of the call.

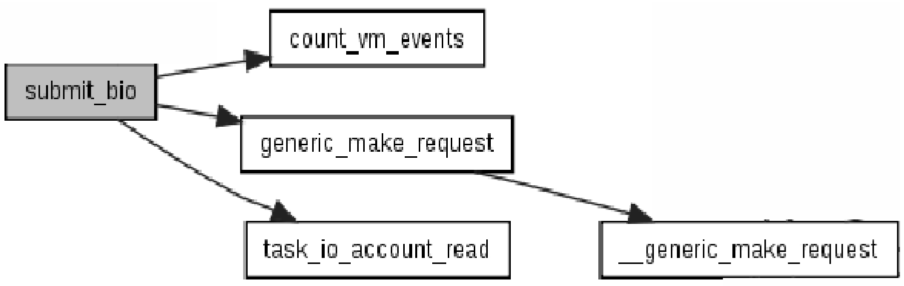


Figure 4.26: submit_bio is the source of caller for __generic_make_request

Figure 4.26 shows that the source call for *__generic_make_request* is *submit_bio*. This indicates that the *bio* has to come from the direct IO. The reason is that the direct IO operation has to put together the appropriate *bio* before using *submit_bio* to submit it to the block layer. The problem here is that the design demands not changing the direct IO layer too much to avoid disturbing other functions.

The construction of the direct IO call level can be found in *fs/direct-io.c*. The main function used by the Ext2 to carry out a direct IO is *__blockdev_direct_IO*. This function has to carry out a lot of administrative organizing to admit the final *bio* to block layer. It is important at this point to mention the difference between a *bio* and a *dio*. The *struct bio* is used by the VFS to admit any request to the block layer. The *bio* has been extensively explained in Section 3.5 on the block layer. On the other hand *struct dio* is used by the direct IO. It helps organize the data that is related to a direct IO access. Despite using the *dio* for organization in the direct IO call, the block layer can only accept a *bio*. Therefore the *dio* has to create new *bio* to be admitted to the block layer. Given the interest in knowing where the *bio* sets its block device variable it is necessary to locate the point of creating a new *bio*. The function that does the setting is called *dio_bio_alloc*. It gets the block device as a parameter.

Once again the call source of *dio_bio_alloc* has to be found to see where the block device is set. This function is called by *dio_new_bio* which gets its block device from the buffer head associated with the *dio*. A hint towards where the buffer is set is given in a comment added to the *struct dio* declaration. The comment explains that *struct buffer_head map_bh* is the last result of function *get_block*. As explained in Section 3.4 each file system provides a *get_block* function. For the Ext2 file system it is called *ext2_get_blocks*. This function sets the buffer head at the end once the blocks have been allocated. For that purpose a function called *map_bh* is used. This function finds the appropriate block device from the file system. This indicates that the *get_block* function is the one that sets block device on which the access is done.

The challenge now is to find an appropriate point for the implementation to change the block device used. This can be divided into two steps. The first step is deciding on which block device to use. The second step is to set the used block device to the chosen one.

As explained before the function *ext2_direct_IO* is the first function that uses the block device. As a result it is an appropriate point in the call chain to decide on the block device. The choice can be done using different methods. One such method is finding the least used block device and bind the request to that block device. For that purpose the atomic access counter placed into the block device list can be used. Such an implementation is shown in the code below. Given is only the part of the *ext2_direct_IO* that selects and then uses the block device in the next function call.

```
/* Dividing req using access count*/
...
struct list_head *p;
struct list_loop_bd *loop_bdev;
    struct list_loop_bd *tmp;
    int min = atomic_read(&(loop_bdev->access_count));
    loop_bdev = list_first_entry(
                     &(inode->i_sb->loop_s_bdev->list_s_bdev),
struct list_loop_bd, list_s_bdev);

    /*Finding the block device with zero or smallest access count*/
    list_for_each(p, &(inode->i_sb->loop_s_bdev->list_s_bdev))
    {
        tmp = list_entry(p, struct list_loop_bd, list_s_bdev);
        if ( atomic_read(&(tmp->access_count)) == 0 ){
                loop_bdev = tmp;
                break;
        }
        if ( min < atomic_read(&(tmp->access_count)) ) {
                loop_bdev = tmp;
                min = atomic_read(&(tmp->access_count));
        }
    }
    atomic_inc(&(loop_bdev->access_count));
    ret = blockdev_direct_IO(rw, iocb, inode, loop_bdev->s_bdev,
              iov, offset, nr_segs, ext2_get_block, NULL);
    atomic_dec(&(loop_bdev->access_count));
...
```

As seen in the code above, the function iterates through the list of block devices. In each iteration the access count is compared with the minimum. If the access count is smaller than the minimum a new minimum is set. In case a zero access count loop device is found the iteration is broken. Otherwise the iteration continues until the minimum is found. Each time the minimum changes the associated list element is kept in *loop_bdev*. Once the iteration is done the list element block device can be accessed.

The least access count method promises a division of load across loop devices. As a result a high performance might be expected. However the presence of a loop within a call that is performed hundreds of times might be a performance downgrade. An additional throttle for the performance of the access count method is the number of atomic operations that have to be carried out. Eventually there is also no guarantee that this method will divide the performance equally. The first loop device on the list is promised the first job. If these jobs can be performed in less time than that needed to add a new job the first device will continue to be used. Although there is no problem with over loading one loop device over the others, the case differs for actual physical controllers. There, dividing the performance more across the hardware is an important wear leveling technique. An additional factor is that the division of jobs is done in a list iteration. Given that finding the minimum takes time, it is not guaranteed that the least used block device will be chosen. For instance the first device might have been done with a job becoming the least used and is not chosen. This happens if the first device access count has been checked before the job finished.

A different more simpler method for choosing the device to access is Round-robin. In this method the requests are given to the next device on the list. Therefore dividing the requests equally over the list of devices. The implementation is shown below. The challenge is to find the loop device next for use without knowing the number of existing loop devices. To avoid having to pass the number of loop device from the point of setting to the point of use another method is implemented. The function uses a static defined global variable called *direct_IO_count*. The function moves into a list iteration in which the list entry is counted. Once the global variable is equal to the count the iteration is broken and the block device is taken. If the list end is reached the global counter is set back to zero.

```
/*Dividing req using remainder of count*/
...
    /*My Variables*/
    struct list_loop_bd *loop_bdev=NULL;
    struct list_head *p;
    int count = 0;
    static atomic_t direct_IO_count;
    atomic_inc(&direct_IO_count);
    loop_bdev = list_first_entry(&(inode->i_sb->loop_s_bdev->list_s_bdev),
            struct list_loop_bd, list_s_bdev);

    /*Finding the block device next for use*/
    list_for_each(p, &(inode->i_sb->loop_s_bdev->list_s_bdev))
    {
        count++;
        if ( atomic_read(&direct_IO_count) == count)
                break;
        if ( list_is_last(p , &(inode->i_sb->loop_s_bdev->list_s_bdev)))
                atomic_set(&direct_IO_count, 0);
    }
    loop_bdev = list_entry(p, struct list_loop_bd, list_s_bdev);

    atomic_inc(&(loop_bdev->access_count));
    ret = blockdev_direct_IO(rw, iocb, inode, loop_bdev->s_bdev,
            iov, offset, nr_segs, ext2_get_block, NULL);
    atomic_dec(&(loop_bdev->access_count));
...
```

The second challenge is getting the access to be directed to the block device chosen. This is complicated as shown by the previous explanation of the position of block device access. Although the final set of the block device is done in the *ext2_get_blocks* the choice cannot be moved. The reason is that within a single direct IO access several requests are made to the block layer. Each request then calls the *ext2_get_blocks* in case more blocks are needed to be added to the read or write operation. Considering that each of these requests are grouped into a unified access, it is reasonable to carry out all on the same device. The difficulty is keeping the appropriate device in the list all over the entire access. The problem becomes clear when looking at call graph of *__blockdev_direct_IO*. This is shown in Figure 3.15 which can be found in Section 3.5.

The call graph of *__blockdev_direct_IO* explains the difficulty with moving the chosen device towards the access point. The call starts in the Ext2 layer and moves into the direct IO implemented for the block device and then moves again into the Ext2 to find the block device to access. The problem is to keep the chosen device the same over all of these calls. An additional challenge is to avoid changing the block device access scheme for other file systems.

Figure 4.27 shows the problem of gapping the direct IO call towards returning back into the *ext2_get_blocks*. There are multiple gaps that the device choice have to bridge in order to reach the final access selection point. The challenge is to cross those gaps without changing the parameter list of the functions.

The first gap is between the *ext2_direct_IO* and the *__blockdev_direct_IO*. Given that the block device choice has already been handed from the first to the second the gap has already been bridged. As mentioned before the *dio* is used to keep data across all function calls within the direct IO call. Therefore every function on the gap interacts with this structure. The functions *direct_io_worker*, *do_direct_IO* and *get_more_blocks* all receive the *dio* as a parameter. This makes passing the block device far easier. By embedding the block device into the *dio* all direct

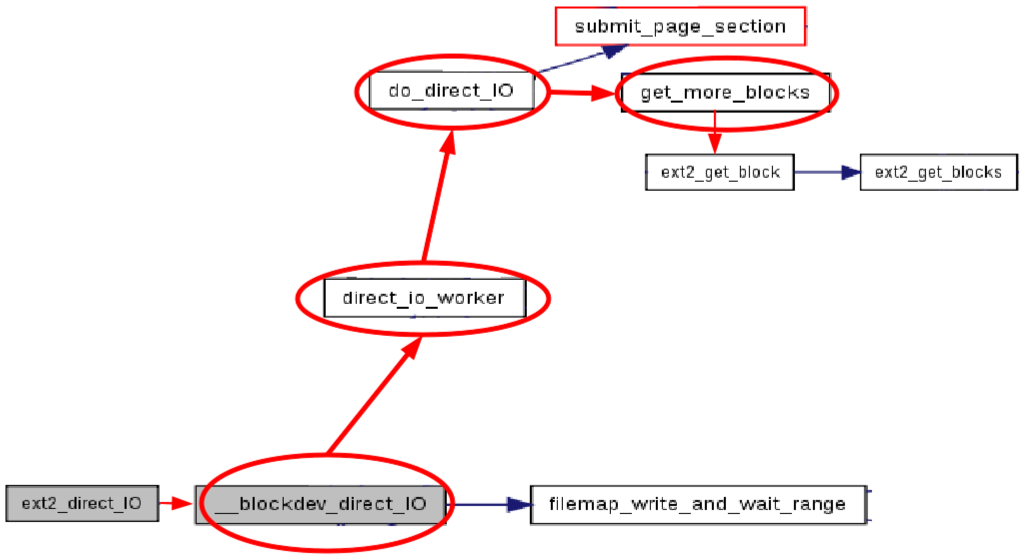


Figure 4.27: *ext2_direct_IO* call graph with block device choice chain

IO functions can use the same block device across the same access.

First the following line is added to *struct dio* definition.

```
struct block_device *list_bdev; /*Loop device for the dio operation*/
```

Next the device is set in *__blockdev_direct_IO* before calling *direct_io_worker*. This is shown by the code below. There is the possibility to allow Ext2 direct IO by a different Ext2 mount point. The mount point that does not use multiple loop devices will have to avoid changing the *dio->list_bdev* variable. To achieve that an appropriate if condition can be put before setting the variable. The if condition can check on a variable that is set in the block device variable. Another option is to try to use the *container_of* function to find the list element from the block device variable. If this element exists the mount point is a multiple loop device mount. Given that the implementation needs to be tested without any other parallel direct IO on different mount points, there is no need to further complicate the implementation. However if used the *container_of* function has to be handled with care. The function can return strange pointers. In some cases it was even found to return the same pointer it was given. Although the function itself does not throw error outputs, using these pointers will. The file system counter to normal applications cannot recover from false pointers. These will result in a Kernel panic.

```
...
dio->list_bdev = bdev;
retval = direct_io_worker(rw, iocb, inode, iov, offset,
                 nr_segs, blkbits, get_block, end_io,
                 submit_io, dio);
...
```

Now the *dio* contains the correct block device. Once the call for the *ext2_get_blocks* is reached, the device has to be moved into the buffer head. This is done in *get_more_blocks* which is shown in the code below. To avoid destroying other *ext2_get_blocks* calls from other devices that have not been mounted using multiple Ext2 design an if condition has to be used. Given that other devices will not set the *dio* device variable, the if condition has to simply check for the value of the variable if set or not.

```
if(dio->list_bdev)
       map_bh->b_bdev = dio->list_bdev; /*Set the buffer head
                                          device to the loop device */
```

Within the *ext2_get_blocks* call the buffer head has to be set to the block device passed before leaving the function. As the buffer head might be reseted during operation it is important to save it at the start of the function using a block device variable. Once the *ext2_get_blocks* has found the blocks that will be read the buffer head is set to the device. In order to make sure that this call has been done through the direct IO layer from a multiple device Ext2 the device variable set at the beginning is checked. If the variable contains a value the device is changed. This implementation is shown in the code below.

```
...
struct block_device *bdev = bh_result->b_bdev;
...
map_bh(bh_result, inode->i_sb, le32_to_cpu(chain[depth-1].key));
if(bdev)
      bh_result->b_bdev = bdev;
...
```

As seen from the above the implementation is very complicated and involves a lot of function changes. The challenge is not just to find the location at which the different variables are set and used, but also how to change them. The problem with optimizing the Linux Kernel is not disturbing normal functionality. For instance, if the previous functions have been changed permanently, access through other methods than direct IO would not have been possible. This would have meant a great deal of rewrite to other functions and probably to the libraries resulting in application changes. As explained before this has to be avoided at all cost.

A final point to be made on the implementation is coherence. Usually parallel file systems have to guarantee locking on different block devices. This adds a layer of complexity to the file system in order to prevent reading old data from a block device while the data has been updated on another. The design avoids this by having a unified storage space. The file system therefore has to only assume coherence and race conditions on the level of a normal file system. The Ext2 file system originally prevents reading a block that is currently written by another process. This will remain the case for the implementation irrelevant of whether the block is being read using a different or the same block device.

4.5 Testing Parallel IO Implementation

The test configuration on a Blue Gene/Q has already been explained in details in Section 4.2. Additionally in Section 4.3.4 the parallelism of the loop devices has been tested. The previous tests show that using multiple loop devices increases the performance. However the block devices were separated. That meant that the access took place on each individual block device using different file system and therefore different files. This however is not the case with the previously explained implementation. For the test script all devices are seen under a single file system. On that basis there is no need to change the test script when testing different numbers of loop devices. In comparison each different number of loop devices test will require a recompile of the Ext2 file system. In each recompilation the number of loop devices used for a read or write operation has to be changed in *fs/ext2/super.c*. The exact position and variable can be found from Section 4.4.3.

The Ext2 with multiple loop device automatically adds the given number of loop devices to the list. The method has been previously explained in Section 4.4.3. Despite that the loop devices have to be set up to point to the same file. This has been explained in Section 4.4.1. On changing the number of loop devices used by the Ext2 file system the number of loop devices setup has to be changed as well. A possible error will occur only if the number of loop devices setup is lower than that of the loop devices used. In this case the file system will try to access a loop device that does not exist. In case the number of loop devices setup is greater the excess loop devices will be ignored by the file system. Additionally the default of the Linux Kernel is 8 loop devices. To perform tests with more the Kernel code has to be changed. Another option is to boot the Kernel with the option *max_loop* set to the needed number of loop devices. The later option is preferred since it does not involve patching and recompiling the Linux Kernel.

As explained in the implementation and how the access is done in Section 4.4.5 there are different methods for selecting the block device. For the tests shown below only the Round-robin access has been considered. The reason for that is the difficulty of dissecting the use of access count. During the early phases of development the access count method showed an overload of access on the first device. This however was due to the use of complex *printk* commands. These consume long periods of time to be fulfilled. This in turn leads to distorting the division of requests among the loop devices. To count the division another scheme of counting access could be found. The challenge is to guarantee that the access counting will not influence the results. Since this is very complicated and meant explaining different complex results the test was done using only Round-robin. The reason for the simplicity of Round-robin is the equal division of access. Nonetheless this equality has to be proven. To achieve that the *top* command is used to observe activity of loop device threads. During a read or a write all loop device threads used in the Ext2 file system list have to be active. Using this technique all test results have been proven to operate on all loop devices in the list of the file system.

Figure 4.28 shows the result for testing the implementation with different numbers of loop devices. The test has been repeated for 2, 4, 6, 8 and 16 devices. The results shown in the figure are for IOPS. As previously explained due to a constant blocksize and plotting the ratio there will be no difference between bandwidth and IOPS plots. For that reason there is no need to show the bandwidth.

Comparing the results shown in Figure 4.28 by those achieved in Section 4.3.4 is important to have a global view of performance increase. The advantages of having a unified storage over all block devices has been already mentioned. Nonetheless it was expected that the Ext2 implementation will add overhead thereby slowing down the parallelism of block devices. The factor however that was not taken into account was the existence of multiple Ext2 mount points. That means that there are a lot more active threads catering to the different mount points. Therefore, as seen in Figure 4.28, counter to the expectation there is an increase in performance.

The performance results show that by increasing the number of devices used by the Ext2 file system performance increases. This however only happens when increasing the number of jobs performing a read or write. It has been explained before that the loop device requires at least two jobs to saturate the IOPS. This would mean that by adding one loop devices an additional two jobs are required to saturate the file system. The saturation has been confirmed by the

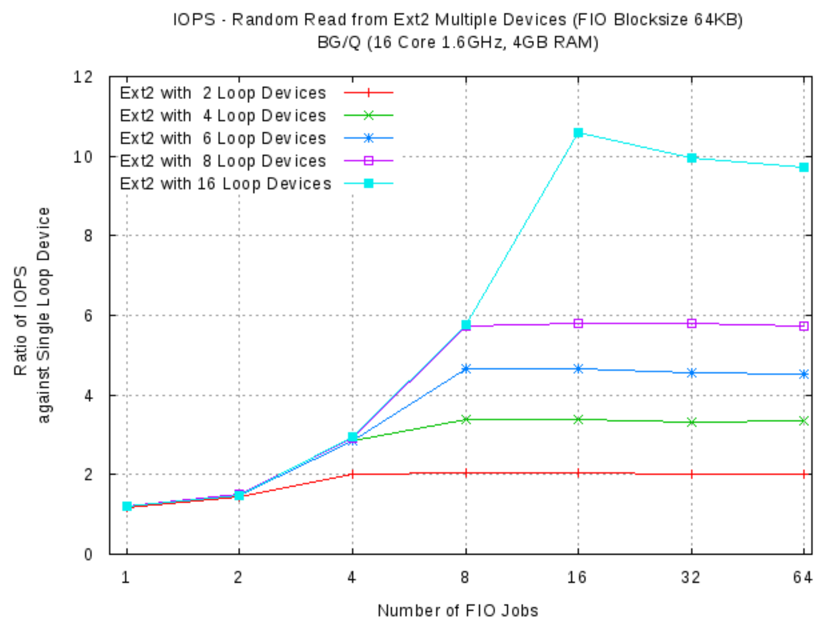


Figure 4.28: Ratio of IOPS of Ext2 with parallel device implementation against single loop device

results shown in Section 4.3.4 and in Figure 4.22. There it is shown that a 2, 4 and 8 loop device setting saturates around 4, 8 and 16 jobs respectively. However for the Ext2 parallel IO implementation the output shows different results. The 2 loop device setting still saturates at 4 jobs. In comparison to previous results the 4, 6 and 8 loop device setting all saturate at 8 jobs. This means that there is an improvement for the performance of 8 loop devices at 8 jobs. This indicates that the throttle for 8 jobs on an 8 loop devices was the excess of mount points and not the loop devices.

The 16 loop device setting behaves contrary to the expectation. Instead of having a saturation point at 32 jobs it is moved to only 16 jobs. Even the saturation behavior is different than anticipated. For other numbers of loop devices the saturation or maximum value was either constant after it was reached or slightly increased. In the case of the 16 loop device setting the performance starts to decrease for more than 32 jobs. This indicates that that the bottleneck is no longer the number of block devices or jobs. The bottleneck is the CPU. As explained earlier Blue Gene/Q contains 16 cores each with 4 threads. Given that each loop device requires one thread it is logical that the performance increase has to stop at 16 jobs. Under the assumption of perfect scheduling the 16 loop device threads will be scheduled on 16 different cores. Even if the 16 jobs are also equally divided among the cores, they will still run on the same core as a loop device. That indicates that the jobs will have to take part of the computing power given to the loop device.

As the number of jobs increase, the portion of compute power that the loop device threads have to give up increases. Thus decreasing allowed maximum performance. The target here is to show that the ceiling of the addition of block devices to the design is fully dependent on the system. Each block device requires service and driver threads, these in turn require compute power. As shown in this test there is an opportunity of adding 16 block devices. This however should not be implemented. FIO jobs only read or write data. Actual applications will require additional compute power for application operations. This has to be kept in mind while designing the system. There is no need for a perfect IO machine on which no application can run. Important to remember is that block devices come with a cost. Not only a cost of hardware and design, but also a cost to the system for support.

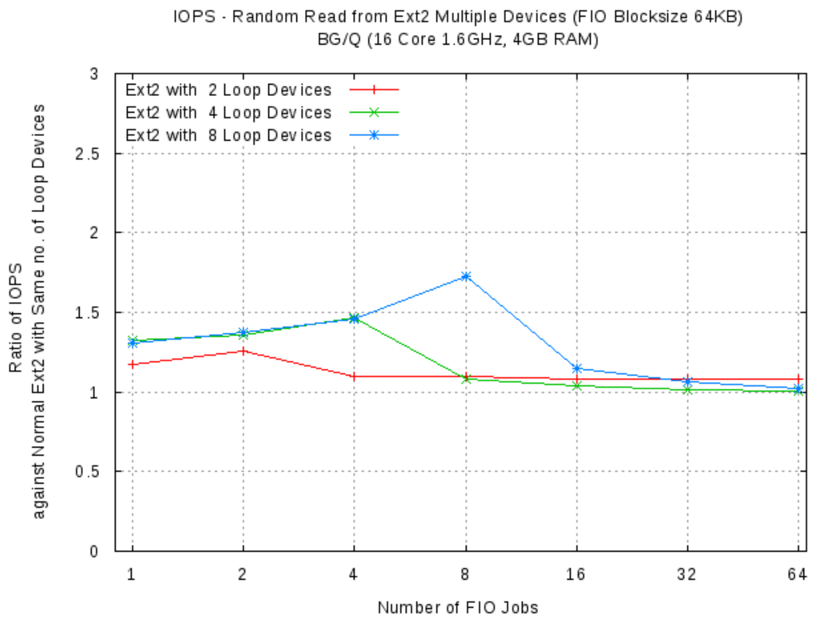


Figure 4.29: Ratio between IOPS with Ext2 parallel implementation and original Ext2

Figure 4.29 shows a ratio between IOPS for the Ext2 implementation and the IOPS for the original Ext2. The ratio is taken between equal numbers of loop devices. As can be seen the performance achieved is slightly higher than that from using original Ext2 on multiple devices. This as explained before is due to having multiple mounting points. The maximum however achieved at the higher number of jobs is almost the same. As can be seen the 8 loop device setting has a large difference at 8 jobs. The reason was as previously explained that the throttle exists due to the multiple mount points and not due the number of loop devices. Once again the figure shows only the IOPS. The bandwidth figure is exactly the same and will therefore not be shown.

Figure 4.29 shows a performance improvement over using original Ext2. Despite that the results should be evaluated with care. As explained in Section 4.2 on the test setting the FIO script uses the *filename* variable to address several devices. In turn this leads to all jobs reading and writing to all devices, but only to a single file on each device. In comparison the parallel IO Ext2 implementation uses one file for each job. Therefore the difference might be the file number and not the implementation. However in worst case the implementation still performs as well as the original Ext2. In light of the advantages for using multiple block devices without changing the application, this is still an excellent result. Additionally for the implementation any block device used can access the same files. Therefore read and writes can be equally divided on all block devices. In fact the read test results shown in this section were done on data that was written to */dev/loop0* and read from the entire list of loop devices.

5 Optimizing Kernel for Storage Class Memory

As the processing power continues to increase with multi-core the IO gap continues to increase. Although there is substantial development taking place to enhance IO links, there still remains a gap between CPU and storage. Figure 5.1 shows that there is a 5 order magnitude gap in cycles between accessing disks and accessing memory [25]. This is rather a great time for an application to wait for an IO operation to be concluded. Flash memory represents an ideal candidate to form a more faster storage unit. As shown in the figure the gap is only 3 magnitudes of cycles between SSD and memory.

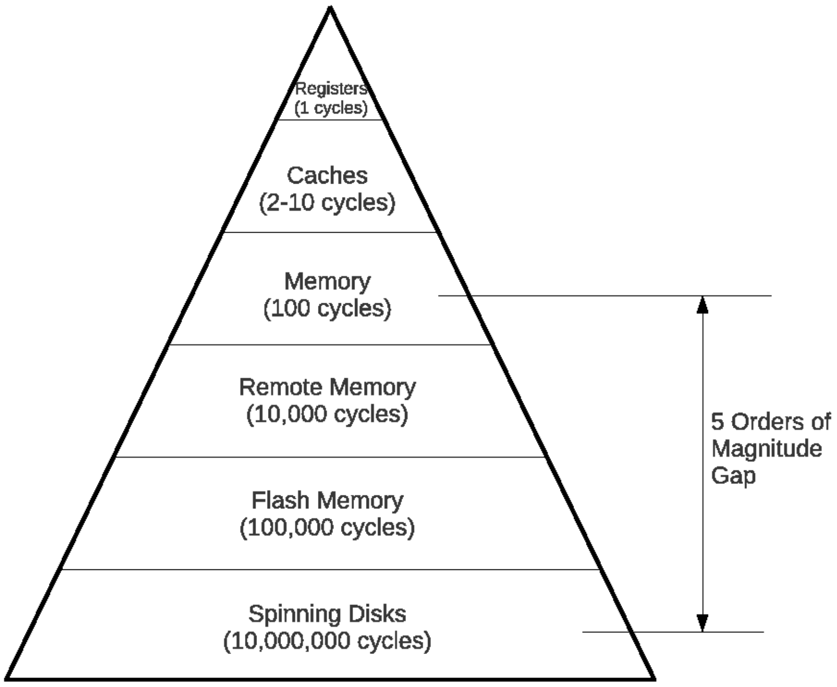


Figure 5.1: There is a 5 order magnitude difference between accessing memory and accessing disks

Many developers have so far treated SSD as being a simple substitution for HDD. In fact in the realm of the user space not much has been done to improve the storage access towards SSD. Fearing backward compatibility SSD manufacturers use the same interfaces for SSD as those used for HDD. That means building complicated layers of abstraction. For example, HDDs are accessed on the basis of sectors set usually to 512KB. In comparison SSD require access on a larger page base which is usually set to 4KB [26]. The reason for the large SSD page size is that writes can only be done one page at a time. The difference will effect the throughput of a write operation. Writes in this case need to be buffered and merged due to only sectors sizes being sent to the SSD. On the other hand reads also need to be buffered to allow a constant SSD read of a page size. The buffer is then allowed to send only a single sector size to the upper layers. This backward compatibility meant that some of the added performance is lost when using SSD.

Problems become even more drastic when referring to reliability issues of the SSD. Due to the limited write/erase cycles that flash cells can endure SSD controllers have to employ wear leveling techniques. The target is divide use among all flash cells. Since conventional file systems can only use a single physical address space, SSD controllers have to implement complex FTL. The advantage is that the file system remains unaware of block movement on the physical SSD layer. However the write becomes more complicated and require drastic overheads.

Another factor affecting SSD compatibility is the design aspects targeting HDD. The main problem is the fundamental characteristic differences between the two technologies. While HDD delays suffer the most from seek time, SSDs practically have no seek time. Yet the system design including file systems, operating systems and middle-ware have been specifically optimized to decrease seek time. Some optimizations have even went as far as removing aggressive prefetching to prevent additional seek time. All these design issues rooted into the current design is limiting the full potential of SSD.

The Chapter is outlined as follows. Section 5.1 explains related work and conclusions that can be drawn from it. This section is divided into different optimization approaches. Section 5.2 shows analysis and tests done for factors affecting SSD performance. The section introduces the conclusions that can be drawn from the test results. Section 5.3 shows an example of improving the IO stack for SSD. The section explains the details of removing preallocation which is not useful for SSD access patterns.

5.1 Optimization Approach

The SSD technology is further adopted into low and high end products due to the underlying advantages. The system built on top therefore will have to adapt to allow further improvement. However as will be shown by the optimization approaches this is not a simple task. The HDDs has become such a viable part of the design that the entire IO stack has been specifically optimized for its benefit. Due to the drastic differences between the HDD and the SSD technology this makes the migration complicated. In fact many concepts used under HDD will have to be abandoned when dealing with SSDs.

SSDs have only been recently adopted as a viable storage unit due to overcoming the reliability challenges inherited by the technology. Therefore most papers and studies found did either target optimization for a reliability defect or presented performance increase due to using SSD. An additional problem faced is no clear categorization of optimization. The techniques are strongly related to each other. Nonetheless the following subsections will show some of the found studies on targeting optimization for or to SSDs. Additionally some studies for difficulties and challenges on SSDs will be shown. The main target is to observe how the technical community is trying to adopt SSD into existing systems. However since most papers as mentioned lack a clear cut optimization approach observations and conclusions will have to be drawn on the basis of the available data.

5.1.1 SSD Optimized File Systems

The complexity of a file system was discussed in Section 3.4. As has been seen the file system is a complicated middle-ware. As has been discussed before in Section 3.6 there are two approaches to accommodate wear leveling. The first is using a specific file system. This will have to integrate the wear leveling algorithm allowing use division over flash cells. The main advantage of this approach is removing the need for a costly FTL. The complexity however has not disappeared and has to be implemented into the file system. Therefore support for new functions has to be implemented such as out of place write [13].

Although this might appear as a new concept the Linux Kernel has contained the support for such types of file systems for many years. File systems such as JFFS and JFFS2 have been part of the Linux Kernel for a long time. JFFS stands for Journaling Flash File System [13]. Most file systems for flashed are based on a log construct. Log based file systems are an additional option to perform wear leveling. The basic idea is to represent the complete file system as being a log. Only the head of the log can be updated and the log continues to move in circular form. The difficulty of this type is that updated blocks cannot be changed in place. This means that new blocks have to be used. The file system has to keep track of block versions to know which to use and which not to. The disadvantage is that these file system support information has to be kept in the RAM and is created at runtime. Therefore at each mount the whole file system has to be checked to recreate the support information [13].

Flash based file systems have a need for supporting out of place writes. The necessity originates due to the using wear leveling techniques. This means that block updates are usually not done in the same place. The block is rather copied and updated in a new write process. In fact SSD controllers usually perform the same operation to increase write speed. Erasing a block before writing is a time inefficient operation. However as the number of write operations that have been performed increases the space of free storage will decrease quickly. A log file system will therefore approach its circular end. Since all the blocks have been written the log file system will not be able to continue writing. Considering that the log file system has information on the block versions it can delete outdated blocks. This process is called garbage collection [13]. SSD controllers use the same method.

Figure 5.2 shows the optimization layers of the IO stack by using flash file systems. It is important to note that the layer names have been intentionally changed. The individual file system has been substituted by flash file systems. The block layer and device driver still are necessary to communicate with the device. However the device has been changed to NAND flash memory. This indicates that the controller design of the SSD is made far simpler due to the missing FTL. In fact the controller can be reduced even further by allowing the file system to track bad blocks and performing some error detection and correction algorithms. Although this appears to be useful for decreasing controller and therefore hardware complexity it rather complicates the file system.

The desire is to divide the complexity onto different layers in a more optimized fashion. Given that the target environment contains multiple weak cores using more complicated file systems is not desired. Therefore a different more suitable middle ground has to be found for implementing a flash file system. One such implementation is the usage of Nameless writes. The idea is for the write operation to only signal the controller with the data to be written. The controller decides on a suitable location and returns a confirmation to the file system which includes the address of the data. Hence the name nameless writes [27]. The file system will therefore have to distinguish between new writes and updates. The need for the division is that the controller must be able to identify blocks that are out of date. This is solved by allowing the file system to send a free command on a certain block to the controller [27]. In fact many SSDs already support a *trim* command that allows signaling the controller of unused blocks. The main disadvantage for the method indicated for updating is the two steps needed. The first step is an normal nameless write. The second is a free or trim command.

A nameless write results in a delayed address knowledge for the file system. The file system has to therefore coupe with the temporary set back in not knowing the address. A read request therefore cannot be submitted until the device is fully done with the write. Only after the controller sends back the address of the written data can the request be submitted. This however does not constitute a large delay. The read request would have been delayed under all conditions

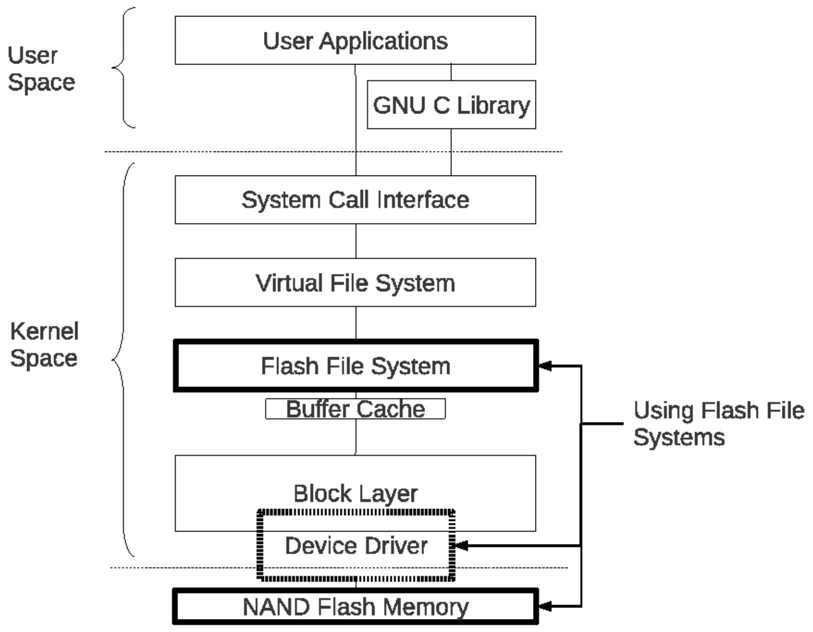


Figure 5.2: Optimization of IO stack using flash file systems

until the write operation is concluded. Another factor that complicates this method is the need for a reclaim operation requested by the controller to the file system. The controller requires carrying out garbage collection operations or additional dynamic wear leveling. This means that the controller might need to move a block that has been addressed by the file system. The controller therefore has to request a reclaim first on the address [27]. The obvious disadvantage for this process is the added IO operations needed to be performed. Not only is the controller delayed in completing garbage collection operations, but also the file system has to keep track of physical position of data versions. That means that a reclaim operation might send the file system into search mode to find the physical address to be reclaimed in its used addresses. If found, the file system then has to do an additional write to move the data to another physical address before allowing the controller to reclaim.

Although using nameless writes has disadvantages it also contains many advantages. Having a file system that is aware of the underlying physical NAND construction results in layout awareness during read and write operations [27]. For example the file system can write data in a better block form matching the need of the controller. Frequently read data can be written several times to allow for less read disturbance across certain blocks. Additionally a write operation that is known to be rewritten in a few cycles can be kept in RAM to avoid wasting a write/erase cycle. A further and rather important advantage for nameless writes is the free division of SSD services between file system and controller. A system that has weak cores can offload to the controller. On the other hand systems that have better CPU performance can utilize it. By allowing simpler controllers the time to market is significantly reduced.

The more widely used option for wear leveling in SSD is the use of FTL. This option is preferred due to the ability to use any file system on top of the SSD. The backward compatibility however comes at a high price to the controller. All NAND flash services has to be done in the controller. This rather complicates the design. Since the above IO stack layers are unaware of the existence of the SSD, requests are not optimized. Some conventional file systems such as

BTRFS have been trying to implement such an optimization for FTL designed SSDs. However the SSD does its best to hide its identity from the above system. Therefore such file systems can only engage the optimizations if the user specifically indicates the underlying storage device to be a SSD.

Shown in Figure 5.3 is the optimization of the IO stack using FTL. As can be seen, in comparison to Figure 5.2 an additional layer has been added above the NAND flash memory. This layer is the FTL required to map the file system on the lower storage. The main advantage of such low level optimization is specifically targeting SSD functions. Manufacturers can increase performance using different hardware and software layouts. Wear leveling, garbage collection and other SSD services can be done as efficiently as possible. The most difficult part for designing an SSD specific file system is the absence of detailed information on the SSD hardware. The reason is that most SSD manufacturers keep there hardware details and even driver software a secret. Therefore most file systems that are designed for SSDs will not be able to make full optimizations. The file system designer will have to make assumptions that might not fit all controller types. On these basis having the controller carrying out its own services means a higher overall optimization for the SSD access. This however might not be true for the system in general. File systems will still have difficulties in optimizing access to SSD and therefore the system loses efficiency.

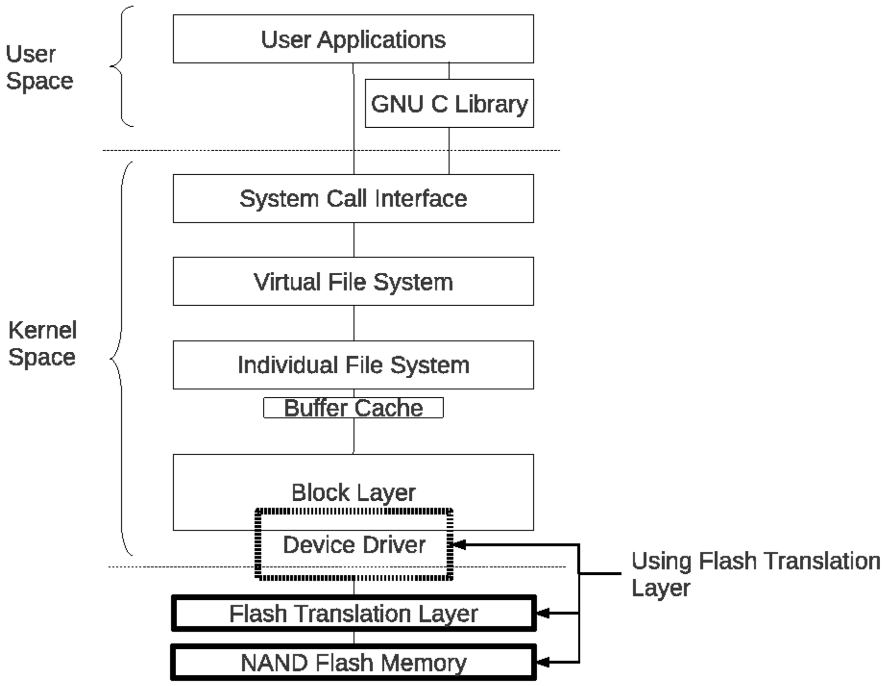


Figure 5.3: Optimization of IO stack using Flash Translation Layer (FTL)

An additional factor that rather limits the use of the conventional IO stack with the FTL is the missing parallelism. As mentioned in Section 3.6, SSDs have inherited parallelism build into them. Since HDD do not share this feature the IO stack is sequential by nature. In fact the application is the one that has to signal parallel access. Parallel access can be achieved if the application uses asynchronous IO or AIO. For that purpose the Linux Kernel offers *libaio* [28].

Abstraction is mainly done to achieve compatibility. For example, the FTL targets abstraction of the underlying SSD to allow compatibility with file systems. This however indicates loading the controller with operations to hide the SSD functions from the system. In addition to that the CPU is loaded by operations that are not needed such as seek time reduction. File system stride to allocate contingent blocks as close as possible to each other. In fact the main reason for block allocation complexity is the use of preallocation. The technique preallocates a group of blocks to the file that are close to the previously allocated data blocks. The target is for the next blocks of data to be placed in the preallocated free blocks to allow for less seek time during read and write operations. Not only is this an added overhead and a further complication to the already complex write operation, but also serves no purpose. The file system operates on a logical addresses that are irrelevant to FTL. While to the file system block 0 is next to block 1, on the physical layer these two blocks can be no where near each other. The controller will place the two blocks as it sees fit and will signal the FTL with the actual physical addresses. Therefore there is absolutely no requirement to waste CPU time on performing such complicated preallocation algorithms.

Another property of file systems is targeting the decrease of fragmentation. In HDD allowing data to become fragmented means that applications will have to move across larger disk spaces. This means that a file write operation and a later read would result in larger seek times. In comparison SSD do not exhibit any form of seek time that would justify the use of CPU for a decrease of fragmentation. Since most file systems use a block based allocation there is no fear of loss of free space due to fragmentation.

Seek time has to be avoided at all cost on HDD. This means that aggressive prefetching cannot be allowed [28]. Only prefetching on adjacent or sequential data is possible. The main reason for that is hindering the head from moving too far from an actual read or write operation. In comparison an SSD would be able to perform real aggressive prefetching during an idle period. That could result in very high performance improvement on the application layer.

Linux offers the ability of accessing the block devices without the use of a file systems. This has the added benefit of allowing applications with their own data allocation and addressing schemes to directly bypass the file system layer. The reason for mentioning this outstanding feature is the need for measuring the overhead of file systems. Some papers have already observed the overhead of some file systems on the performance of the SSD. In [25] the overhead of the XFS has been measured and found to be high in case of random writes. Given that many factors affect such a test it will be more convenient to repeat it on the available environment.

5.1.2 Access Patterns and Scheduling

One of the difficulties is the essence of accessing difference between the HDD and SSD. While the first exhibits large seek times the second does not. This however is only the tip of an iceberg. The complication and the difference increases by taking into account reliability issues of SSD. Additionally the unique write/erase cycles of SSD means that there is a completely different access pattern dependency. The problem becomes even more difficult when realizing that observations have been HDD oriented. Some performance differences between patterns are attributed to wrong properties due to miss understanding SSD layouts.

Predictability is an important property when testing storage devices. Due to the nature of storage devices predicting timing of request fulfillment is not an easy task. HDD seek time means that a write or read request can exhibit a wide range of delay. Therefore the latency becomes dependent on how fragmented the devices is and how well the IO stack layers above are able to avoid straining the HDD. As mentioned earlier the IO stack was designed with HDDs in mind. The target became to avoid reasons for low predictability using different techniques.

Seek time as the largest predictability issue for HDD became the focus point. Access patterns are kept as sequential as possible. File systems employ allocation and preallocation schemes that would allow file data to be as contingent as possible.

On designing access patterns for SSD different issues arise. Using the same preconceptions for SSD as those used for HDD is a mistake. On the other hand considering SSD to be access pattern independent is also false. The unique SSD design makes it unpredictable, yet the reasons are different than HDD [28]. While SSDs have no seek time they still contain other factors that make different access patterns behave differently. One of these unpredictability factors is the services that are required to support the SSD [28].

The most noticeable difference is that SSDs do not exhibit equal read and write times. While read can be done at random and any place in the SSD, write is limited to free space. An additional problem with write is that pages can only be written sequentially within SSD blocks. This means that page number 0 can only be written after page number 1 within the same block [10]. The need to erase a block before writing to the pages also makes the write process slower. This therefore would indicate a drastic mismatch in write versus read latency for SSD [29] [26]. The problem is however that this is not always true. Since allocation is hidden from above layers using the FTL, the controller tries to hide the side effects. As an example, controllers avoid as much as possible using a full write/erase cycle. As explained in Section 3.6 out of place writing is used to write all incoming data in free blocks. This however increases unpredictability. Once the SSD is full garbage collection will have to be switched on to free space. In fact some SSD exhibit sudden write delays after long write periods. These are due to the controller trying to free space for the incoming write requests [28].

Another issue that makes a write request slower than read is the effect of merging. Data can be read almost at different granularity. Since SSDs are forced to use the same interfaces as those of HDD, SDD needs to simulate sector sizes. The sector is usually 512KB in size. In fact most conventional file systems operate only with sectors sizes. In comparison SSDs operate at a page size of 4KB and blocks of 64 pages. Simulating a sector size for read operations is simple. Even if the controller needs to read additional data it can ignore or buffer the rest of the page and send the sector chunk required to the above layers. This however is different for write operations. These can only happen in page sizes. Since the controller is not willing to waste the rest of the page the sector written will need to be merged with others. The merge operation adds to the latency of the write requests [26].

Access patterns are usually divided into two main categories random and sequential. As explained before HDD are faster in case of sequential access due to lower seek time. SSDs however are different. The main advantage that almost all sources state for SSDs is the quick random access allowed under a fully electronic technology. In some cases this becomes misleading. As discussed in the previous paragraphs, read is different than write. A random read is large orders of magnitude faster on a SSD than on a HDD. Write are different. They highly dependent on different factors. A random write could be delayed longer than a sequential write due to the difficulty with merging sectors into pages. The problem however is that many papers tend to attribute specific latencies with random access to fragmentation. In [30] different latencies were found for different access patterns done on different SSD types. The main observation despite that is the controller effect that must have lead to such pattern differences. While fragmentation might have a large influence on HDD latency this is not the case with SSD. Once again file systems might lead to useless overhead. Avoiding fragmentation on file system level is not useful in on a SSD employing a FTL. While the file systems believes it has proclaimed contingent address spaces the controller reorganizes this into a different pattern.

It is difficult to conclude how much latency is influenced with access pattern change. The reason is the use of controllers that hide the actual access pattern that is taking place on the physical level. As mentioned before a sequential write is preferred for the control to be able to perform sector merging. Additionally the pages within the same block have to be written sequentially. Random write however might lead to a high garbage collection overhead. This is only true if the random write involves updating already existing data. The latency difference however between random and sequential writes is dependent on algorithms used to implement the controller. In fact a sequential write might be turned into a random one by the controller to write in freshly erased blocks. The opposite is also possible. Random access write might result in a fully sequential write due to use of FTL.

The issue of random versus sequential access becomes even more interesting when observing SSD layout. As mentioned in Section 3.6, SSD cards are made of several flash memory chips. This means that SSD has an inherit internal parallelism. In fact this property makes the random access more efficient. The reason is that by accessing different chips the controller can perform several read or write operations in parallel. Random access however does not dictate accessing data on different chips. This leads to the parallel access increase to be highly mapping dependent [31]. Another issue that is rather interesting is mixing the two operations read and write. As flash memory chips cannot perform both operations on some levels at the same time there is an expected performance degradation. This has been shown to be the case [31].

The life span of a SSD is to a large extend dependent on the workload layout [31] [32]. As a direct result the controller becomes responsible for acting on changing access pattern to avoid early failure. In fact all wear leveling techniques are specifically designed to approach a more random pattern. This is because random patterns allow the chip and block utilization to be further divided among all available memory units. For that reason pattern dependency cannot be simply observed from higher stack levels. In fact the file system has almost no effect on the pattern. Although preallocation, for example, tries to increase data contingency, on a SSD this would not result in physical data contingency. Under these conditions the file system has the illusion that future reads are sequential, while physically the reads are randomly divided across the SSD. Even though the write in a sequential manner into a flash block might decrease latency file system algorithms cannot help. Reserving the next page in a flash block to future writes would hinder the writing of any subsequent page in the block. Since the controller will probably not allow that, the page will be claimed for other write operations.

Many applications exhibit random access and might even be dominated by it [25]. Given that HDD do not perform well under these conditions, the IO stack tries to approach a more sequential access pattern. The difficulty however is that the IO stack has no control over applications IO patterns. As a consequence the IO stack uses other techniques to allow a more uniform access pattern. The most important technique to that purpose is scheduling. By waiting long enough there is a good probability that two IO requests will occur that are closer in the address space. The IO stack therefore holds a queue as explained in Section 3.5 for all submitted requests. The target is to rearrange and merge requests to approach a more sequential access.

As explained before SSDs do not exhibit any seek time and are mostly access pattern independent due to FTL use. Since this is the most important reason for implementing a scheduler, most conventional HDD designed schedulers will result in avoidable overhead [25]. In fact most SSD driver designs avoid using queues and schedulers. This is mostly done by overwriting the *make_request* offered by the block layer. The Linux Kernel offers several schedulers. The simplest to understand is called noop scheduler. The noop does no scheduling at all. The request is simply placed in the queue as soon as it arrives. On an HDD the noop will exhibit the worst performance. In comparison best performance is achieved using a noop scheduler on a SSD [26]. This is a strong indication that conventional schedulers are an overhead that adds latency to the SSD access. SSD manufacturers prefer overwriting the *make_request* function instead of switching to the noop scheduler. This is because there is no method to signal to the IO stack which scheduler to use. The choice is left to the Kernel and the user.

Although there is no direct performance improvement from using HDD schedulers, this by no means denies performance increase in general to using schedulers on SSD. The target is to observe SSD properties and implement schedulers exclusively for SSDs. This is the target of the scheduler design in [26]. The idea presented is that merging different sector write requests lead to additional overhead. Therefore the scheduler introduced in [26] suggests servicing all sectors in the same block as soon as they arrive. Thus reducing merging overhead. Using this scheduling scheme the controller would receive a group of contingent sectors that can be merged into a page. There is no seek reduction done in this scheduler.

The explanation of SSD access patterns showed that in some cases it is preferred to have sequential writes. This is true for SSD cards that exhibit large write to read latency mismatch. The reason is trying to get the writes done as a bulk might allow the controller to do more optimized writing. Also it allows the controller to simplify merging sectors into pages which has been pointed out before. In addition to that controllers might avoid erasing a block for a single page. However for chunks of data a block erase is worth it. Not to mention that the erase delay will be divided over the total number of write requests written into that block. Under these conditions using write deferring might be useful. The term write defer means that a write will not be serviced until a certain point in time. This is allowed since most applications do not wait for write confirmation to continue. This has been shown to provide performance improvement [29]. The problem with deferring is its need for merging different IO operations to form more contingent data units. Merging is a CPU heavy operation [28]. This comes as no surprise. CPU has to compare every incoming request with previously made ones to find possible merges. Once that is done the actual merge operation can begin. It is therefore logical that by increasing the write defer window results in larger CPU overhead [29].

Another interesting write improvement is write alignment. The target is to align all writes to block boundaries. For SSDs this becomes an advantage as the controller can avoid unnecessary block erase. This shows significant promise in improving the write performance. In [29] it is reported to increase the performance by as much as 50%.

As a byproduct to schedulers the queue became a useful tool for implementing fair access. This however is not realized by HDD schedulers. It is important to know that schedulers target mostly overall performance. To that end some schedulers allow starvation of some requests. This might be in the interest of overall system performance in the presence of seek time. SSDs schedulers however can be designed in a more efficient method. Different queues can be assigned to different users or different files and data [28]. In fact a reasonable assumption to improve performance is to assign different queues to different storage space. The decision should not be one sided. Allowing the controller to initiate different queues might give the possibility for further internal SSD parallelism. In all cases queue plugging has to be avoided as it results in overhead that is not useful for the SSD [28]. The issue on a new form of queuing interface for SSDs has been extensively explained in Section 4.1.4. The interface explained is a new standard called NVM Express.

Figure 5.4 shows the optimization layers for implementing IO scheduling. The VFS is responsible for the implementation of the schedulers. Therefore most optimization will take place in the VFS layer. Furthermore the choice for appropriate scheduler is currently reserved to the user. To change that aspect the VFS can be allowed to choose the appropriate scheduler. This would be useful if the VFS can somehow detect that the underlying storage device is a SSD.

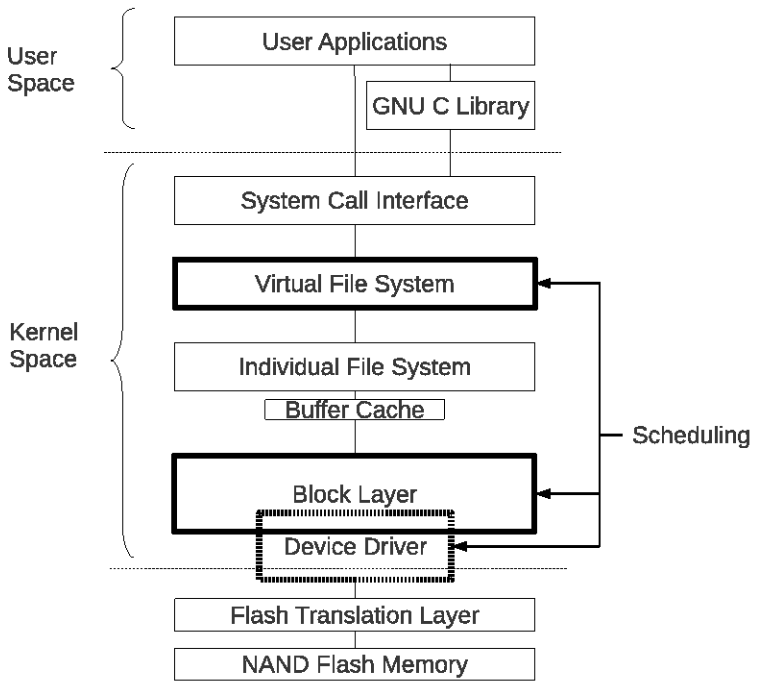


Figure 5.4: Optimization of IO stack using IO scheduling

Although the VFS chooses the scheduler, it does not decide on how to apply it. This choice is still done in the block layer. In fact the VFS merely supplies the block layer with function pointers to the scheduler. Since the block layer allows the driver to overwrite the *make_request* function the driver has the opportunity to discard all scheduling and queuing. The optimization done from a scheduling scheme therefore becomes a cooperation between three layers the VFS, block layer and driver. It remains to be noted that the device in the IO stack has been replaced by the FTL and NAND flash memory. This is done to indicate that the access pattern decided on in the scheduling scheme might be distorted by the controller. Hence the optimization would be further improved if the controller is involved.

5.1.3 Internal SSD Optimization

SSDs differ on many layers from HDD. The main difference is in the amount of service that an SSD requires. Storage access in a SSD is no longer only dependent on the device itself. SSDs access time suffers from the reliability constraints. An obvious example to limited SSD throughput due to reliability issues is write amplification [33]. A 1MB write operation on an HDD would usually result in almost exactly 1MB physical write. SSDs require error detection and correction bits. These add to the write amount, not to mention the added latency of producing these bits. The detection of errors also limits the throughput of the read operation, more so for MLC than for SLC.

Another example of how the throughput is limited by reliability of SSD is the requirement of out of place write. This is a direct result of SSDs need to employ wear leveling to increase the life span of the device. Out of place writes cannot function unless a garbage collection algorithm is utilized [33]. The problem with garbage collection algorithm is the unpredictability that is present in the amount of overhead induced by running it. A write operation on a freshly erased SSD has a low latency. As more write requests are serviced the placing of the new writes becomes difficult. Although it is preferred to run the garbage collection algorithm during idle times, at a certain storage space consumption level it will have to be switched on. The problem is that the writes will have to be delayed till some blocks are erased. Over provisioning and other techniques can be employed to minimize the effect. Despite these techniques the delay of writing to a SSD will degrade over time. The amount of degradation is dependent on the SSD design.

The most important aspect of optimization is rethinking internal SSD functions. As shown before reliability factors dominate performance. These can therefore be handled on a system basis or an internal SSD basis. While the previous sections point to the problems related with systems adoption of SSD, the most important optimization can be done from within the SSD itself. Using special algorithms and further optimizing the IO stack to operate at high efficiency with SSDs is not a simple task. The biggest obstacle is analyzing the SSD itself. SSDs contain many different algorithms most of which is based on manufacturers Intellectual Property or IP. This in turn means that most information on the actual implementation of the SSD is not available.

Analyzing and comparing SSDs is not just complicated due to the use of IP units, but also difficult due to the interface used. SSDs attempt to simulate HDD interfaces to be used with legacy designs. This however means that many of the underlying implementation is hidden [34]. In fact as mentioned before the interface hides even the aspect of the device being a flash based storage unit. This extremely complicates the analyses and in turn makes designing special stack operations for SSD almost impossible. However some conclusions can be drawn from either flash memory basics or from tests done on SSD.

The most important observation for SSD is the need for driver support. Due to the common interface used to support most available HDD these can function without the need of installing drivers. SSDs on the other hand require a driver to operate correctly. As a consequence designing a high end SSD is done both in hardware and software. In fact many of the operations that have been attributed to the controller could be easily moved to the device driver. The distortion of the border between hardware and software SSD design is a massive limitation to actual SSD analysis. While one SSD is dependent on CPU frequency due to driver operations another is not. The SSD designs usually hide on which level any of the service operations are done. As a direct result performance of the same SSD might change when porting it to a different system.

Although the analysis is made difficult by the absence of a clear hardware versus software SSD design, there are advantages to this design concept. There is large room for improving SSD specifically for a given system. While some systems may benefit from a larger software driver others will not. The use of multiple weak cores means that driver is limited by the performance of a single core. Therefore the system will benefit from a more hardware implemented controller. In fact this is the expectation that has been proven by the frequency tests done in Section 4.3.1. The benefit is increased by applying a controller than has more specific hardware optimizations done for the system implementation. One such optimization is the use of less error detection and correction code in case the system already builds on higher level error handling codes. This in turn will increase read and specially write operation due to reduction in the write amplification. Additional more specific enhancements would result in even more efficient storage utilization.

Despite hardware being more efficient and faster in handling issues on the storage level, the software implementation is still used. The main reason is that hardware design is complicated and time consuming. Yet the use of larger more complex software drivers should not be viewed as a final limitation. The issue of parallel operation is a great aspect of how far the performance can be enhanced. To achieve better performance multiple thread drivers are the best approach [28]. It should be noted however that the limit of such thread design is the CPU ability to carry out parallel processing which depends on core count. Yet another limitation to the use of SSD software services is the extend by which these operations can be executed in parallel. While a process such as garbage collection can be done in parallel, error detection and correction cannot. In addition to that software is limited due runtime information loss. This means that a fully software operated SSD controller is at risk of losing more in case of a power loss.

Finding the border between hardware and software implementation in relation to performance is difficult. This is quite obvious when dealing with the out of place writes. As explained in Section 5.1.1 there are two options for achieving a viable SSD wear leveling. The first is using a flash file system that would support the out of place write. This implementation therefore pushes the wear leveling issue to an even higher stack level than the driver. The most important advantage from operating at these software levels is more stack information. File system can rather enhance placement of data and can support even longer write deference. This can also be made true for a driver that can further gather more information from the above layers. The other option for supporting out of place writes is using a FTL. The property of a controller that implements wear leveling techniques completely in hardware is more appealing to systems with low CPU performance. On the other hand complicating either the hardware or software of a controller implementation due to a large FTL is inefficient. The design of the FTL is highly dependent on the size of mapping. To allow smaller more efficient write operations the FTL will grow in size wasting both storage and processing time [28]. On those grounds as explained previously in Section 5.1.1 in [27] a middle solution was presented. Nameless writes give the opportunity of implementing hardware wear leveling without the need of a FTL. Nonetheless there are other constraints to that approach that have been mentioned in Section 5.1.1.

SSDs internal design offers an added dimension of optimization. The inherit parallelism of SSD give a high interest to the controller performance and viable enhancements [31]. The parallelism appears due to the need of using multiple flash chips in the design of SSDs. The controller therefore has the ability of further enhancing efficiency by dividing requests among these different chips. Until now only the SSD controller stands to benefit from this feature. However by specifically optimizing the upper IO stack layers the optimization can be increased. For example, single read and write queues need to be reformed into more complex parallel operations. To achieve this the internals that supply the SSDs parallelism have to be understood [31]. Yet another more important aspect of internal SSD parallelism is the difficulty of the level of operation. Challenges need to be handled on a much more global level now need to be performed in the firmware [31]. On one hand the lower level design of parallelism has a larger set of limitations, on the other it has the opportunity to benefit from previous experience. While firmware could not support complex stripping algorithms from parallel file systems, it could use similar simplified techniques.

One of the benefits for parallelism on the low level is hiding delays induced by write operations in flash memory [35]. While HDD can service a single write or read operation at a time, SSD can use multiple chips to induce several reads or writes. By observing single chip behavior the performance can be increased even further. As previously reported in Section 5.1.2 performing mixed reads and writes degrades performance. This can be explained by chip behavior which cannot perform two different operations in the same enable group. Therefore by allowing the controller to detect future read operations writes can be performed on different flash chips than the ones used for the read. That can only be achieved if the upper layers somehow inform the controller by the upcoming access pattern.

One of the major difficulties in implementing parallelism on this low level in SSD is the requirement of a parallel FTL [36]. This is not a simple requirement. The FTL should not act as a performance bottleneck. The overall SSD layout is complex and requires addressing many different spaces using multiple buses. The FTL has to be able to deal with all of these issues and more. Thus making the FTL a viable candidate for optimization. For instance engaging several different controllers into a unified FTL will eventually lead to a loss of performance. On the other hand using several FTL is not a simple task. A FTL needs to be constantly aware of controller wear leveling and changes done on the lower level. The FTL therefore could divide the mapping of the underlying storage space among smaller more efficient tables. Another important aspect that complicates parallelism of FTL is the added dimension of parallel services that request changes. The controller might be accessing and updating the FTL for dynamic wear leveling. Mean while the parallel operating garbage collector might be trying to inform the FTL of moving blocks to other locations to perform an erase. In addition to that loss of FTL data is fatal. While bit and byte errors are recoverable by error correction and detection algorithms, loosing FTL mappings is not. If the file system attempts an access using a logical address to which the physical address has been lost by the FTL the data cannot be recovered. In fact such data loss cannot be recovered even though the data exists on the storage units. Furthermore the location at which the data is written is not recovered by the garbage collector. The FTL in such a case will have to handle the incoming request from the controller to relocate the data for an address that the FTL doesn't recognize. All this factors into the complexity of a parallel FTL.

Figure 5.5 shows the layers on which the optimization is done. Device driver, controller and FTL changes have been previously explained. The problem however is that limitations are inherited into the controller due to the block layer. This means that applying changes to the device driver might require removing some of these limitations. In fact the block layer is the first line of layers that could inform upper layer of the device type. In turn this can be used to trigger a more SSD efficient upper layer operation. The block layer also holds the ability to take on some of the functions required by the lower layers. There has been no literature found on the opportunity of enhancing the wear leveling or the parallel access on the block layer. In fact the block layer might be useful in functioning with specified optimizations for SSD without changing the upper layers. Nonetheless the power of the block layer remains limited and therefore might not offer real optimization chances.

Although the optimization of the SSD design only points to lower layers an optimization on the complete IO stack might be the result. As discussed before designing the SSD controller might be useful if done with upper layer functionalities in mind. The SSD controller has been attempting to shield upper layers from physical address change using FTL. The file system therefore engages costly allocation algorithm in the hope of decreasing already absent seek times for SSDs. One suggestion is to find a method to enhance internal SSD access towards the access patterns of the HDD. Not only will this increase general internal SSD performance but it will make the allocation algorithms useful. A case can therefore be made for keeping the same scheduling algorithms but enhance the SSD controller towards this scheme. This however is counter to the norm, which dictates that software be optimized for hardware and not the opposite. Nonetheless the method might show room for improving performance of SSD on legacy HDD interfaces.

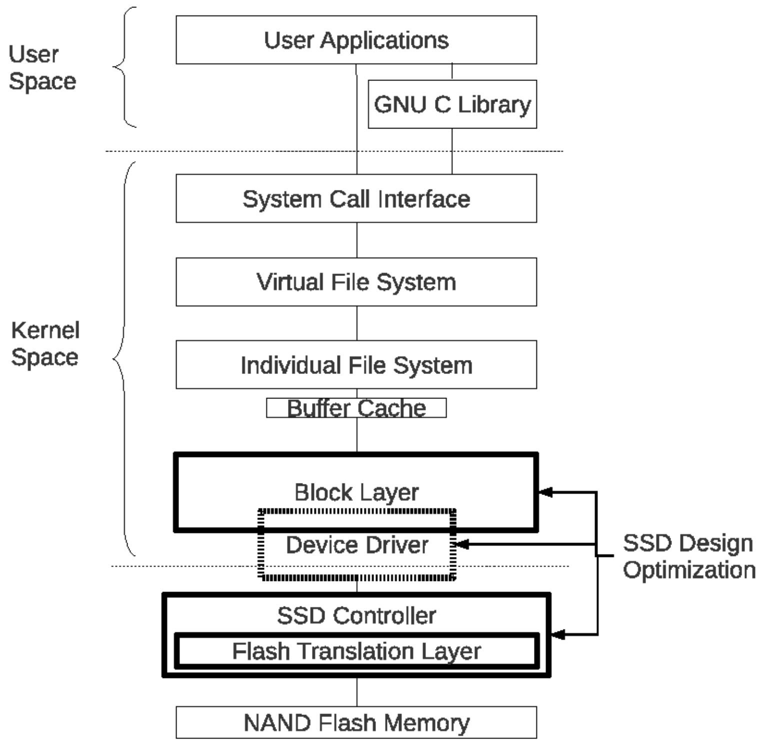


Figure 5.5: Optimization of IO stack by changing SSD design

5.2 SSD IO Analysis

In the previous sections some inferences have been made as to how SSD access can be improved. To find which implementations show more improvement it is crucial to perform several IO tests under different conditions. The challenge however is the difficulty with testing IO on SSD. As previously mentioned SSDs hide most of the implementation under HDD interfaces. While being able to test a typical SSD it is quite complicated to attribute performance change to certain properties.

Due to the unique architecture of the Blue Gene/Q machine porting SSD drivers is not made easy. Almost all SSD manufacturers develop their drivers on *x86* machines. Since Blue Gene/Q is not compatible with the *x86* architecture the porting is made difficult. Furthermore the SSD are mostly optimized for the *x86* architecture. Therefore a high drop in performance could be expected when porting the driver. Some SSDs perform some of their services in the driver itself employing different algorithms. The driver becomes a complex unit that requires detailed information on hardware to optimize for a given system. In addition to that these drivers hold IP. which the manufacturer prefers not to share with others. Thus the drivers of such cards are closed source. Not only does this prevent optimization on the Blue Gene/Q, but also hinders detailed analyses of the cards internals.

The difficulty of porting drivers onto the unique Blue Gene/Q architecture was seen on many levels. For example, the *x86* architecture mostly runs with a default page size of 4KB. The Blue Genes/Q Linux Kernel defaults to the 64KB page size. Since the cards drivers are designed on the *x86* some are incompatible with the page size of 64KB. This prevents testing the actual difference in performance between a card on the Blue Gene/Q and other architectures. Other difficulties arise also from the implementation of the Blue Gene/Q on the basis of the POWER architecture. Yet another difference is the massive parallelism required on the multi-core Blue Gene/Q to utilize the computing power available. This is not anticipated by the SSD manufactures and therefore might not be utilized. Under these circumstances testing has to be made by accounting for the differences. To conclude different aspects on SSD performance a set of tests have to be done. Despite the difficulty of testing SSD, this study is not interested in specific SSD performance. The interest is in finding general properties of SSD that could lead to a more optimized IO access on SSD. Therefore tests might not be performed only on SSDs but also on general upper layers. In addition to that comparing absolute performance numbers is not necessary. More information can be found in comparing job scaling performance and ratios between different test conditions.

The next sections attempt to perform some tests and elaborates on some conclusions that can be drawn from these tests. It should be noted that the test environment used here is the same as has been explained in Section 4.2. In addition to that the same test script has been used in an attempt to unify the test. This in turn allows comparison of test results across different settings.

5.2.1 Apple G5 and Blue Gene/Q Tests

As explained earlier there is an inherit difficulty with testing SSDs. One difficulty is testing the difference between hardware and software implementation of SSDs. The main problem is finding two cards that can be compared on these basis. Most manufacturers do not supply specific informations on the SSD provided. Therefore attempting to find the layer of service implementation difficult and in some cases even impossible. Under those conditions finding two cards to compare based on the software to hardware implementation is not possible.

Given that the target is to find how the new architecture of the Blue Gene/Q might effect performance, tests have to be performed on a wide set of machines. This can benefit the finding differences between hardware and software SSD service implementation. As mentioned before in Section 4.2 the Blue Gene/Q contains 18 cores that run at 1.6GHz. In comparison the mentioned G5 only contains two dual cores that run at 2.5GHz. In plain numbers it is apparent that the Blue Gene/Q should outperform the G5. However the frequency of the G5 is higher than that of a single core in Blue Gene/Q.

The test comparison between IO on Blue Gene/Q and IO on G5 becomes interesting due to the architecture differences. A well designed SSD card performing most of its operations in hardware should increase performance on the Blue Gene/Q due to the increased computational power. The expectation therefore is that performance should increase on the Blue Gene/Q specially when it comes to job scaling. In other words, by increasing the job count a higher performance increase is expected on the Blue Gene/Q than on the G5. For that purpose the ioDrive introduced in Section 4.3.1 is used.

Figure 5.6 shows the performance of the ioDirve on the G5. The figure only shows the ratio between single job performance and different tests with increasing number of jobs. The target is to find the scaling behavior of the ioDrive on the G5. As can be seen in the figure the IOPS scale well with increasing the number of FIO jobs operating on the driver. The increase in IOPS decreases at higher number of jobs. This however could be attributed to the limited compute power available on the G5 to accommodate the number of jobs running. In fact the scaling of FIO at 64 jobs at lower blocksize shows this effect. The IOPS of the 64 job FIO test should have remained at the maximum level. This would have meant that the bottleneck is either IO link or the driver. Since the IOPS dropped at higher job numbers it means that performance has its bottleneck in the computation. The G5 no longer can support all these FIO jobs without them sharing CPU utilization.

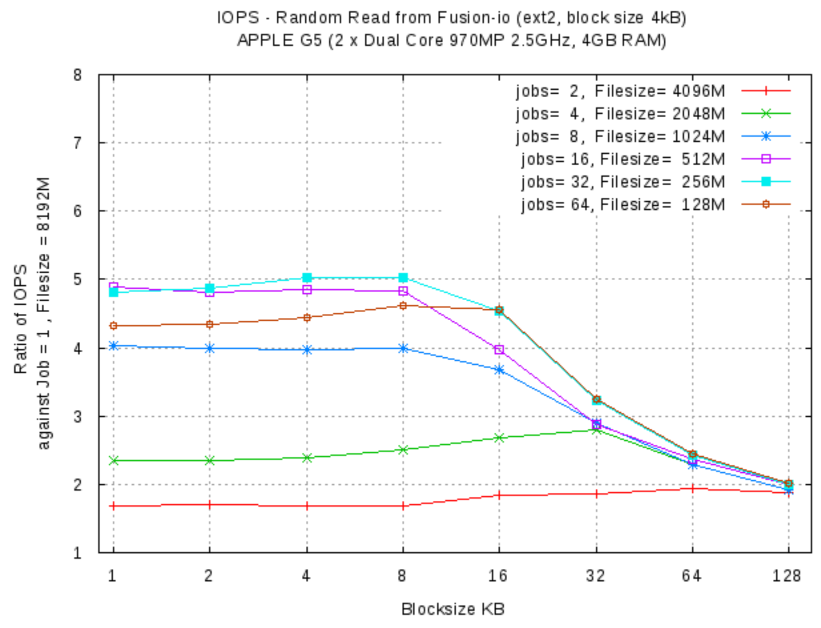


Figure 5.6: Ratio of IOPS against single job for ioDrive on Apple G5

It should be noted here that the G5 contains processors that also employ the POWER architecture. This makes the results compatible with the Blue Gene/Q. The optimizations pointed to in the introduction of this section still are not available. The performance therefore is expected to be worse than *x86* architecture. This however means that the difference between the G5 and the Blue Gene/Q is limited to the number of cores and frequency. There are still more architectural differences that might effect the IO performance. This comes from the fact that the G5 is a more general purpose personal computer. However the Blue Gene/Q differences to G5 are still less than that of differences to the *x86* architecture. In addition to that by limiting comparison to performance differences and not using absolute IOPS or bandwidth the comparison becomes more relevant. The target therefore is to find the difference in performance scaling on multiple jobs.

Since the Blue Gene/Q contains a far superior computation power on a single node than that of the G5 the expectations point to better performance. However as seen in Figure 5.7 this is not the case. It is obvious that the performance increase has almost the same maximum or even lower than that of the G5. Contrary to the expectations the performance saturates at 8 jobs. The problem here is obviously not the number of jobs used by the FIO which have plenty of computing power to utilize. This conclusion can be made as the saturation at 8 jobs did not appear on the G5 which has less computational power. In addition to that all tests including 8 jobs or more have almost the same distribution along the different blocksizes. If the bottleneck is the computational power of the Blue Gene/Q then adding more jobs would have decreased performance. However as can be seen this is not the case. An 8 job FIO tests performance almost exactly as a 64 job test. Thus it can be concluded that the bottleneck is not the overwhelming number of FIO threads started in each test.

Given that the performance on the Blue Gene/Q saturates after 8 jobs there has to exist a bot-

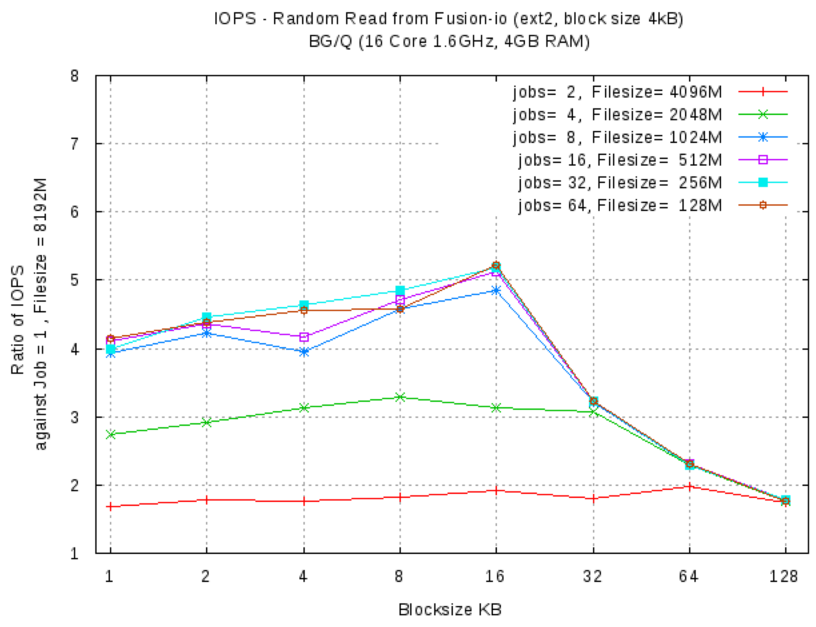


Figure 5.7: Ratio of Bandwidth against single job for ioDrive on Blue Gene/Q

tleneck. The reason for the saturation of the Blue Gene/Q IOPS performance could be defined by the card limitations. There are two observations that could lead to a clue on what this bottleneck might be. The first is the single device present on the card. The ioDrive specifications point to it using a single controller. In addition to that the ioDrive presents a single block device to the Kernel. This however is not always a proof single device operations. But as mentioned the specifications refer to a single device. For the device to be a bottleneck the performance on the Blue Gene/Q has to be exactly the same as that on the G5. In other words the bottleneck should not just appear on the Blue Gene/Q but also on the G5. Since this is not the case it could be concluded that the controller is not fully utilized on he Blue Gene/Q. Therefore the controller is not the bottleneck.

The second clue to the Blue Gene/Q bottleneck is the CPU utilization. Figure 5.8 shows the CPU utilization of during a FIO test on the Blue Gene/Q. This figure has been previously shown in Section 4.3.2. The utilization shown in the figure dictates that the bottleneck is the driver performance. The thread that has been binded to CPU 4 and 8 are ioDriver threads. This explains the reason for having a worse performance on the Blue Gene/Q than that on the G5. Although Blue Gene/Q possesses higher computational power, the single thread performance is still limited by the core frequency. The driver appears to set the limit for the possible performance of the ioDrive.

The main design issue of a SSD is choosing the level of operation for the services. It should be noticed that the tests shown here cannot speculate on whether the ioDrive performs such services in the driver or in the controller. Despite that it appears obvious that using software limits performance. To mend this limitation two possibilities exist. The first option is to implement a more parallel driver. If the driver can therefore utilize more CPUs the performance can be further enhanced. The challenge with driver parallelism is how far the SSD services can be performed in parallel. Some services do not allow parallel performance. Indeed there are several situations in which these parallel services will have to synchronize to for example update the

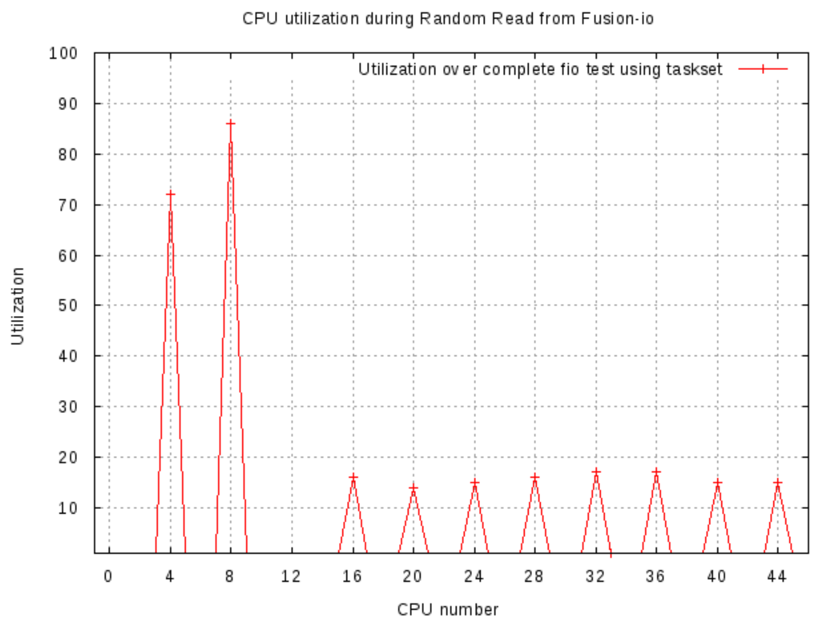


Figure 5.8: CPU utilization during FIO test and setting CPU affinity

FTL. Furthermore creating these threads indicates that part of the computing power will be utilized by SSD services. This means that applications will be deprived from computational power that was reserved for them. In fact Section 4.3.2 has shown that complete cores have to be reserved to the driver to allow for better performance. Therefore a software SSD service design is not advisable. In addition to that the performance of the driver threads will still be bound by the frequency of the modern cores which is being lowered due to power consumption issues.

Another option for solving the bottleneck of a single driver is implementing hardware serviced SSDs. The driver in this case should only operate as an abstraction to the SSD and for SSD optimized access. All other SSD services should be implemented in hardware. This design promises almost system independent performance. The bottleneck would then no longer be the single core frequency. It should be noted however that the controller performance could also be the bottleneck. Thus the device design is a balance between system single core performance versus hardware controller performance. That is proven by the Blue Gene/Q. Although single core performance is limited, any SSD controller could not provide such massive parallelism as present on the Blue Gene/Q. Therefore single block device performance might be limited by the controller speed. However on the Blue Gene/Q a multi-threaded driver has the opportunity of utilizing more cores. It should be noted that most SSD designers provide parallel access using multiple controllers. These then operate in parallel to provide higher performance. There are other disadvantages to this design which have been discussed in details in the previous Chapter 4.

It remains to be said that Figure 5.6 and Figure 5.7 represents only the IOPS. However given that these figures are showing IOPS ratio the bandwidth performance would show the exact same curves. For that reason the bandwidth figures are omitted. The IOPS still do decrease with increasing the blocksize. On the other hand the bandwidth increases with increasing the blocksize. The inverse relation between two performance counters shows that increasing one does in fact decrease the other. This should be kept in mind on choosing an appropriate SSD.

The bandwidth given in the specifications should be associated with the blocksize for which the bandwidth has been achieved. It should however be noted that most SSDs prefer a lower blocksize to increase life span. By allowing the applications to request larger blocksizes means that the SSD has to write or read larger storage data. As mentioned before there is a limit to number of write/erase cycles allowed on a flash memory. On the other hand too small blocksizes means that the controller has to do time consuming merges. The reason as mentioned before is that there is a lower limit to size of written data to an SSD. The lower limit is set at the SSD page size.

5.2.2 Read vs. Write

As elaborated in Section 5.1.2 on access patterns there is a mismatch between read and write operations on SSDs. Therefore it is important to find how this mismatch effects performance. Up to this point only read results have been presented. This was due to the fact that the conclusions that were drawn were mainly dependent on scaling. However it is important to address how write performance differ from read. In this case more conclusions can be drawn from absolute figure of performance. In addition to that it is important to show both bandwidth and IOPS performance change across blocksize changes.

The main problem with performing these tests on the Blue Gene/Q is risking having degraded performance due to limited single core frequency. In comparison the G5 avoids this limitation. The tests therefore should be done on the G5 to have better comparison. In order to provide viable results an actual SSD has to be used. The issue of write to read difference should show how these two operations should be handled.

Figure 5.9 shows the IOPS of random read from the ioDrive. As can be seen, IOPS decrease by increasing the blocksize. The maximum performance is limited to 60K IOPS. In comparison Figure 5.10 shows the IOPS of random write on the ioDrive. The maximum as shown has saturated at around 50K IOPS. In fact most jobs show lower IOPS at about each point. Some show slight increase to the write versus read. However this increase could be due to test fluctuation. The main conclusion that could be drawn is that IOPS are decreased by performing a write operation.

The two figures for the IOPS reflect the expectations. Write requires more operations than read. Therefore write has to have a lower performance than that for a read operation. However it was expected that the performance difference would be much larger. As explained a single write operation might even require an erase cycle. Placement and wear leveling make the write operation complicated. The write operation used here however did not reach those limitations. The main difficulty with testing write on SSDs is making sure that the storage is fully utilized. Only then can the effect of the service and reliability avoidance process be tested. For example the test should be repeated until the garbage collection has to be engaged. Since there is no intention here to destroy the available SSD the test cannot run the SSD to the limits. Despite that the shown results are important. SSDs do not always operate at the maximum limit. In fact the target from these designs is an active storage. These do not fully utilize the cards performance. In addition to that the tests shows the basic difference between write and read operations on a card. The IOPS difference between a write and a read can be fully attributed to the difference of flash memory write to read mismatch. In addition to that random write has been used. This means that some updates has to be done to previously written locations. The controller therefore has the choice of either trying to write in place or performing out of place writes. The first choice delays the write until performing an erase. The later however requires switching to the

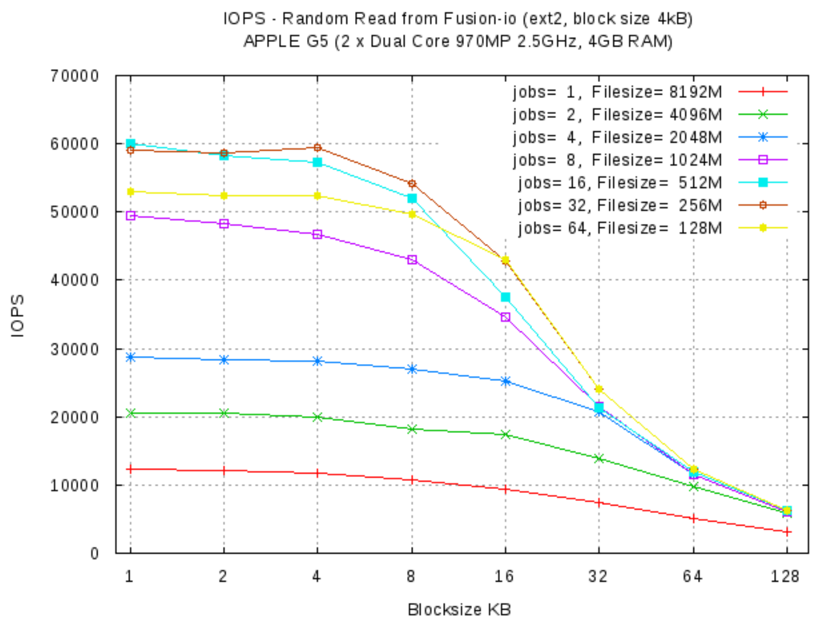


Figure 5.9: IOPS of Random Read from an ioDrive on Apple G5

garbage collection later. The test however is not long enough to have the results influenced by side operations such as garbage collection. Therefore an out of place write would result in almost a pure read.

There is a difference between the distribution of IOPS and bandwidth across blocksize. As a result the bandwidth curves could lead to different conclusions than that of the IOPS. Figure 5.11 shows the bandwidth of random read from an ioDrive. As can be seen the bandwidth increases with increasing the blocksize. The maximum is reached just short of the 800 MB/s. In comparison Figure 5.12 shows the bandwidth of random write on an ioDrive. The maximum has massively dropped to just over 300 MB/s. As mentioned earlier it is expected that write is a higher delay. This delay highly effects larger blocksizes. As expected performance drops. However as discussed earlier the effects that change read to write performance is purely due to flash memory. A write operation requires the use of high voltages. Furthermore the write requires creating or updating the FTL table entry. This in turn leads to further delay. However the delay would be more effected if the controller decides on an in place write. In such a case an entire block would be erased.

The effect of an out of place write should not drastically degrade performance of writes. By combining the results of both performance differences on the bandwidth and IOPS several conclusions can be drawn. IOPS dominate the performance at smaller blocksizes. The decrease at smaller blocksizes detected on the IOPS curves is modest. Therefore it can be concluded that performance drop on smaller blocksizes is mainly due to write to read performance difference. On the other hand bandwidth dominates the performance at larger blocksizes. The decrease at lareger blocksizes detected on the bandwidth curves is massive. Therefore it can be concluded that performance drop on larger blocksizes is due to more than just difference of read to write performance on flash. The controller therefore appears to be selecting out of place writes at low blocksizes or for small sized write requests. Yet on larger blocksizes or for large sized write requests the controller updates in place. This means that the write has to be delayed for the erase

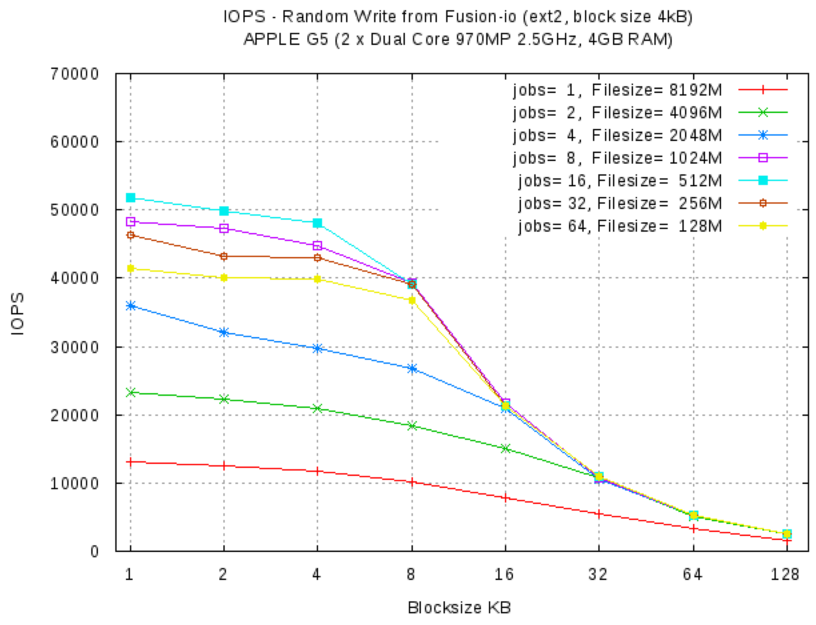


Figure 5.10: IOPS of Random Write on an ioDrive on Apple G5

to take place first.

As mentioned before, most SSDs prefer smaller blocksizes. The main reason is that the controller can more easily select blocks to place them into. In addition to that several write request could be fitted into the same flash block. Thus all these write requests would share the erase delay that might be required. Furthermore the life span of the SSD is increased. The controller becomes able to perform more optimized wear leveling. Additionally the file system would not have to write data that is unnecessary to fill the data request. The write to read mismatch therefore becomes a difficult issue that requires solving. It should be noted that conclusions made here concerning controller implementation is based on the results shown. It still remains difficult to make decisive determinations on how the controller is built. The performance is governed by complex algorithms hidden beneath layers of interfaces. The complexity therefore prevents finding actual bottlenecks. In fact the difference between read and write at small and large blocksizes could be due to garbage collection. The controller could be switching garbage collection algorithms for large blocksizes to free these as quickly as possible for future writes. This would contradict the conclusions made on in place versus out of place writes. However the reason why garbage collection effect in this case is dismissed is the uniform curves at higher blocksizes for bandwidth. If the write was delayed due to garbage collection or any other reliability service there should have been more fluctuations in the performance. It is expected that reliability services do not operate at full capacity during write. Therefore their influence on curves cannot be leading to such uniform results as those shown in Figure 5.12.

The disadvantage of trying to change performance using blocksize is that it is a application factor. It is therefore a specification of the system rather than a changeable factor. Some settings necessitates the use of IOPS while others build on bandwidth. From the previous results it is obvious that specifically for writes both is not achievable using these settings. Write requires better bandwidth performance. This could for example be achieved by further stripping data across more internal chips. Figure 5.13 shows the performance decrease along blocksize for

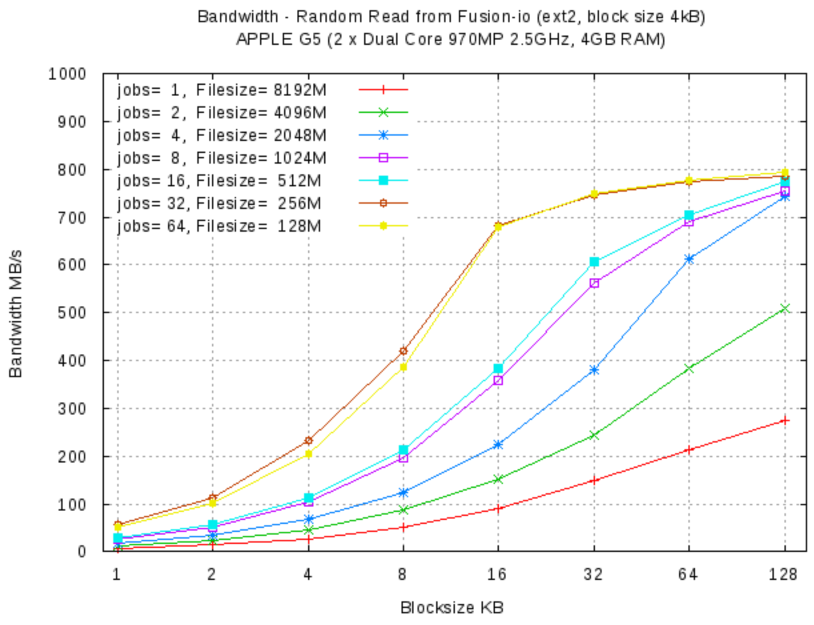


Figure 5.11: Bandwidth of Random Read from an ioDrive on Apple G5

writes as a percentage of reads. The performance drop shown could therefore be explained by the failure of the controller to divide larger sized blocks across different flash chips. Thus leading to performance drops at larger blocksizes which become limited by single chip bandwidth. One reason for the controller to keep larger blocks as a single group in a flash is to decrease the size of the FTL. The block written as a unit could be addressed using a single physical address, leading to only a single entry in the FTL.

As parallelism by utilization of several flash chips would increase bandwidth it would be advisable for the controller to attempt breaking larger blocksizes. Thus giving itself the ability to write these smaller chunks on different flash chips. The utilization however of the write and read of such breaking is highly dependent on access patterns. If multiple applications access their files in sequential order at the same time, the controller should sacrifice write bandwidth and allocate the block into a single chip. This would allow the controller to read from many chips in parallel to serve all applications. On the other hand if the applications are performing more parallel file read and writing the controller should preferably stripe the block across multiple flash chips. This would allow later the application to access the file using parallel operation and increase performance. It is shown by this that application information on access patterns is crucial for design of optimum performance. Although this might not be possible for general purpose computers, HPC systems are unique and mostly built for scientific research. This in turn leads to understandable and application uniform access patterns that could be researched.

Another issue that has not been anticipated or accounted for is caching. Figure 5.13 shows slight increase of write operations versus read at smaller blocksizes for 1, 2 and 4 jobs. Although these could be explained as normal test fluctuations, they can be attributed to caching. At low number of jobs at small blocksizes the controller could be avoiding a write delay by placing the block into the cache. The write would then be handled later. This would therefore prevent future write operations into the cached data from having to erase a block. It should be kept in mind however that FIO test cannot detect whether the delay is for writing to the physical flash storage

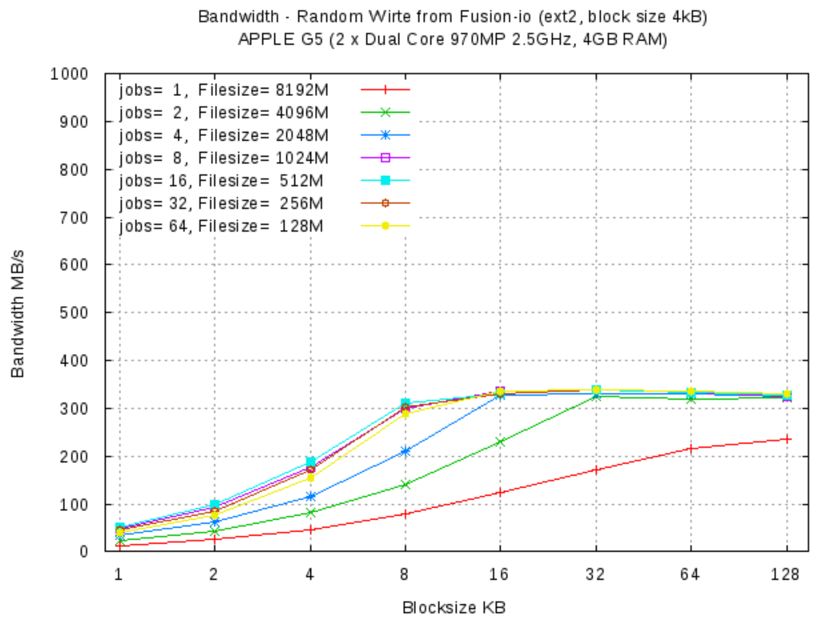


Figure 5.12: Bandwidth of Random Write on an ioDrive on Apple G5

or into the temporary SSD cache. Given that both are reported by the controller to the upper layers as successful writes FIO adds these cache writes to the performance. This is an additional proof to the complexity of trying to test SSDs.

5.2.3 File System Overhead

A file system has to add as little overhead as possible. However there is almost no file system that does not add any overhead. The file systems task during a read or write operation is to find requested blocks or allocate new blocks. During a read operation the application requests blocks of data from the file system. These blocks are marked by the block number. The file system has then to find the physical addresses of these blocks. On write the file system in turn provides physical address for blocks in which the data can be written.

Linux gives the ability of raw device access. The application can be allowed to directly access the block device without the need of a file system. This is done by providing the application with the block device file. Since the file system determines allocation of physical addresses, in raw device access the application has to do the allocation. This however gives the opportunity of finding overhead in case of a simple file system such as the Ext2.

Section 4.3.3 on testing parallel file systems has previously shown the overhead for parallel file systems. The Ext2 is expected to have a lower overhead. This is mainly due to the simplicity of Ext2, in comparison to the complex parallel file systems. In addition to that parallel file systems depend on using multiple devices spread across several servers. The Ext2 uses single devices mounted on a single point. However to achieve relatively good performance using Ext2 on loop devices several devices have to be used in parallel. Therefore the test has to compare how performance changes with changing the number of loop devices as well.

Figure 5.14 shows the overhead produced by the usage of the file system. As can be seen the performance difference for a single loop device is relatively low. This is because a read operation

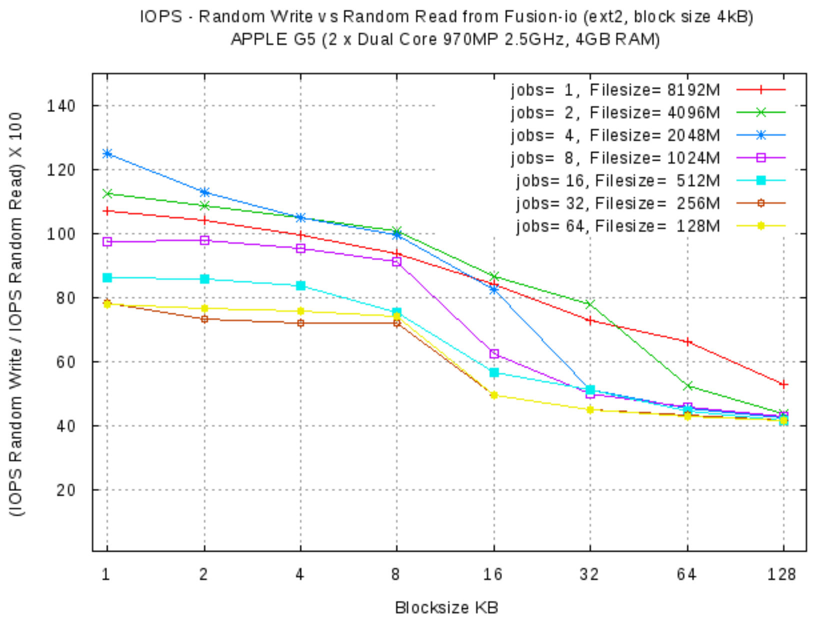


Figure 5.13: Percentage of Bandwidth of Random Write versus Random Read on an ioDrive on Apple G5

in Ext2 does minimum operations. In fact all that is required from the file system is to return the physical addresses corresponding to the block number. Nonetheless it requires separate block access. The file system has to access the device in case of having indirection. This is because the pointers or physical addresses are saved on the device. The detailed explanation of the read operation in an Ext2 file system is explained in Section 3.4.1.

The overhead in Figure 5.14 was expected to increase with increasing the number of loop devices. This is because the number of loop devices require themselves processing power. In addition to that each mount point requires separate operations. However the result show that performance difference for some points does not increase much with increasing number of loop devices.

Raw block device access removes overhead of file system. However the physical address selection is left to the application. Since the FIO benchmark does not show access patterns and how division is divided among devices, it becomes difficult to compare results. In addition to that loop devices are more bound by the processing frequency. This therefore indicates that while the overhead exists, it cannot be attributed fully to the file system.

On the other hand write operations are more complex and require further calculations from the file system. Based on that the write operation overhead should increase. This cannot be proven using loop device. The loop device is bound to the processing power available for a single thread as has been shown in Section 4.3.4. Therefore the difference between write and read operations does not exist on loop devices. In addition to that *ramdisks* cannot be used to find write overhead due to their speed and direct access to RAM.

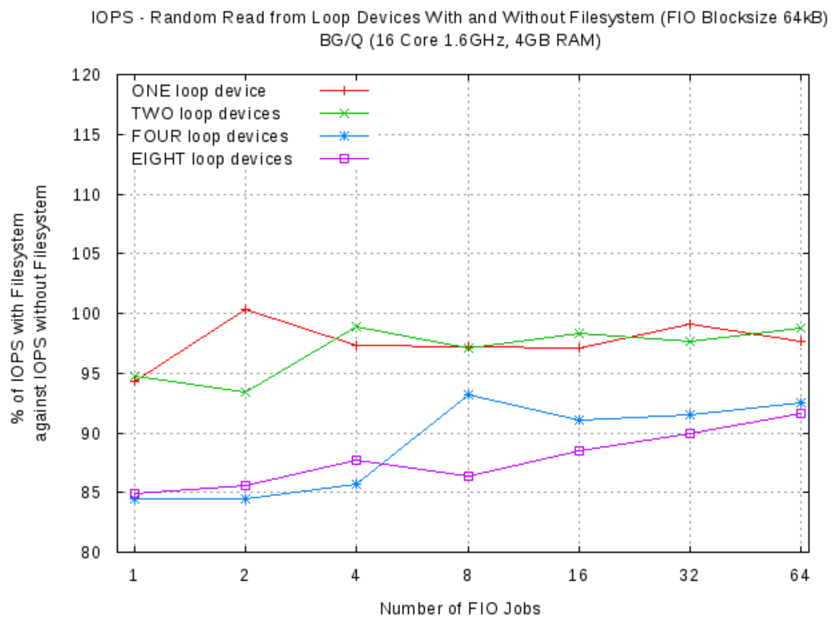


Figure 5.14: Percentage of IOPS for Random Read with file system against without

5.3 Removing Ext2 Preallocation

The difficulty with finding an appropriate implementation for improving SSD is supporting the different types and implementations. Therefore it is preferred to find an optimization that fits the basic flash memory access. File systems exerts efforts in attempting to decrease seek time. Since flash devices do not exhibit seek time, these algorithms become a wasteful overhead. One such algorithm is preallocation.

File systems operate on the basis of blocks. Section 3.4.1 has explained that the storage space of an Ext2 file system is divided into block groups. Within these block groups the data blocks are stored. The file uses its *inode* to store the pointer to the data blocks associated with it. For that purpose the *inode* employs direct and indirect pointers. On the basis of such file layout a single file can be fragmented across the entire file systems storage space. Such layout on HDDs which suffer from seek time would result in massive read and write delays. Things are even worse when accounting for indirection blocks which contain data block pointers. These indirection blocks have to be read to find the physical addresses of the data blocks. Any distance between the indirection blocks and the corresponding data blocks has to be covered by the HDD head. This in turn will result in additional seek time. Therefore the file system attempts to place indirection and data blocks of the same file as contingent as possible. Those should also be placed as close as possible to the files *inode*. The result is far better sequential read time on HDD. Thus the file system allows a relative high CPU utilization to avoid file fragmentation.

The difficulty in avoiding file fragmentation is that files are usually not written all at once. Data blocks of a file seldom come as a single unit. Furthermore the file system usually writes and reads several files at the same time. This means that data blocks which are close to the file are at risk of being allocated to a different file. The file system therefore employs reservation windows to anticipate file allocations [9]-Chapter9. This type of reservation is called preallocation. The file system uses several complex allocation schemes to keep all writes of a file as close together as possible. In addition to that the file system allocates new unused data blocks to the file that could be used for future writes. This however is not an easy task. Since the file system has no record of future writes these preallocated blocks could be a waste of space. Therefore the file system has to be able to reclaim such blocks without a free operation from the application. Furthermore the file system has to be able to balance between neighboring files and their preallocated blocks. Another issue is the free space available on the storage device. The file system has to be able to reclaim these preallocated blocks for actual data.

As will be shown later, preallocation complicates a write operation. The file system however has the notion that such CPU utilization is cheaper than expensive seek time. In comparison SSDs exhibit no seek time. This makes them ideal candidates for random access. Conventional file systems such as the Ext2 have no method of identifying the underlying storage type. Therefore the contingent allocation algorithms are used no matter the device type. Despite that Ext2 supports switching off preallocation using a file or application setting. Nonetheless the contingent data allocation cannot be switched off. In addition to that several avoidable function calls are still performed regardless of preallocation setting. CPU time in a multi-core environment with weak single core performance is expensive. Therefore such preallocation overhead should be avoided.

Although sequential write operations on some SSD show better performance, this would not be addressed with the usage of preallocation. Preallocated blocks are not written back to the device. That is no actual IO appears to be happing for preallocation. The file system just keeps a record of preallocated blocks. Even pointers should not be written back to the device. This is done to avoid permanently allocating preallocated blocks to the file in case of a system failure. The target of preallocation is the avoidance of future seek time for write and specially for read operations. On the other hand SSD employ a FTL that hides the actual physical address space from the file system. Thus the file system is allocating logical addresses that will not map to the lower physical storage. Therefore preallocation has no benefit whatsoever for SSD operation.

The following explanation will attempt to explain the effort of removing preallocation overhead and decreasing write complexity in the Ext2 file system. The write operation will be demonstrated and changes will be provided side by side to the original write call construction. An effort is made to keep code inclusion to the minimum. The target is to show how a rewrite for such a complex operation might could be approached. It should be mentioned that there is no detailed source on how such operations are done. The information given here are found by analyzing the Ext2 implementation. Readers interested in further information should try to have a look into the original Kernel Ext2 code. However such approach should be done with caution. The implementation is complicated and requires advanced knowledge on file systems. In some cases detailed knowledge on processor architecture is required as well.

5.3.1 Ext2 Get Blocks Function

Due to the complexity of the write operation done in the Ext2 file system the approach will be made in a hierarchical form. This form is an analogy to the call path of the *ext2_get_blocks* function. Figure 5.15 shows the call graph in a hierarchical form. The numbering of the functions is done in accordance to the execution sequence. The second number represents the hierarchical order. If there is no number it is directly called by *ext2_get_blocks*. Using this scheme 6.1 is called by 6., while 6.2 is called by 6.1 and so on. It of absolute importance to follow the complete write operation. Some function names is deceiving. For example a function for allocation with no reservation could still contain some reference to preallocation. Despite that some of the more complex issues that are irrelevant to the issue will be omitted.

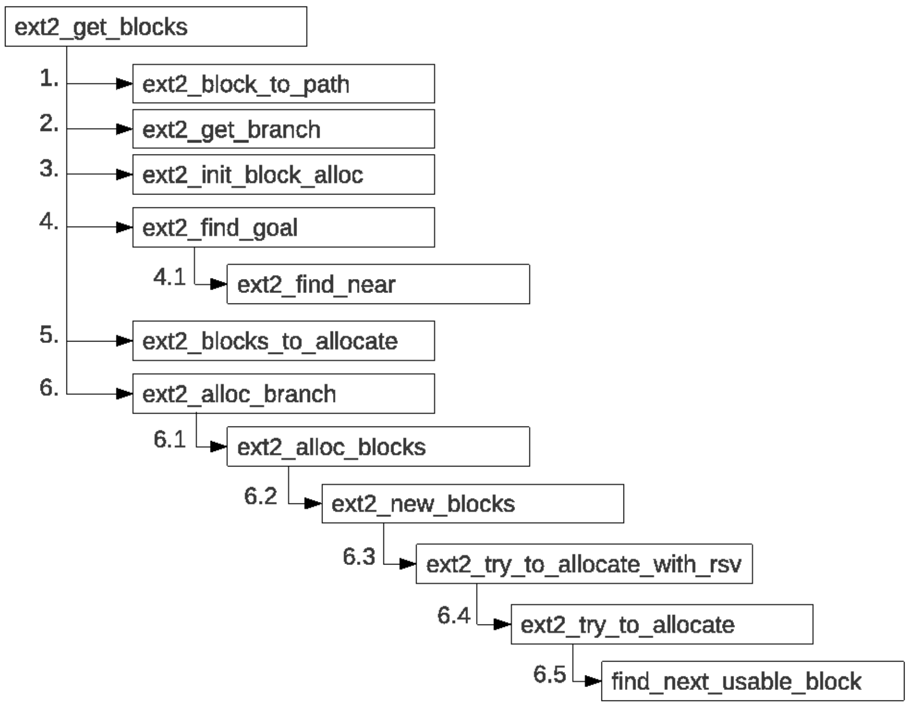


Figure 5.15: ext2_get_blocks call graph with hierarchical and execution sequence

The first function called is 1. *ext2_block_to_path*. The task of the function is to fill the offset list of the indirection blocks. The indirection has been explained in Section 3.4.1. The offsets within each indirect block can be found using block number. Therefore 1. *ext2_block_to_path* does not perform any IO. The next function called is 2. *ext2_get_branch*. It has the task of finding the actual physical addresses of both indirection and data blocks. *ext2_get_branch* produces a chain of addresses using the offset list within the indirection units. This function has to use IO to access each indirection block with the offset to find the physical address saved in the indirection block.

The *ext2_get_blocks* can be used for both read and write operation. In a read operation the function 2. returns a *NULL* pointer. Using a condition *ext2_get_blocks* finds this case and loops over the sequence of addresses. Finally it jumps to the end and sets the final resulting buffer to the physical address. This buffer will be handed to the VFS to perform the block read operation.

In case of a write the *ext2_get_blocks* task becomes providing the VFS with physical addresses to write to. For that purpose the function has to first find available data blocks for allocation. Next the data blocks physical address must be added to the *inode* as either a direct or indirect pointer. The later case might result in adding indirection blocks to provide space for the pointer.

Assuming an allocation call, the next function called by *ext2_get_block* is 3. *ext2_init_block_alloc_info*. The task of this function is to initiate block allocation info. Mainly this is done to allocate the information necessary for the reservation window. If preallocation is turned off the reservation window is set to a size of zero. For the purpose of removing preallocation function 3. can be completely removed. Any setting done within 3. that is still necessary for finding data blocks can be moved to *ext2_get_block*. One such setting from 3.1 is the block allocation info structure. This is set to the *inode* block allocation info which can be performed in *ext2_get_block*. The block allocation info is needed to find last allocated logical and physical block.

The next function called is 4. *ext2_find_goal*. As the name suggest the function attempts to find a goal block near of which the new blocks should be allocated. The goal block is usually set to the block next to the last allocated block. If the allocation info has just been initialized meaning the last allocated block is zero then 4. *ext2_find_goal* calls 4.1 *ext2_find_near*. This in turn attempts to find another block with sufficient locality. It first attempts to set the goal to a previous pointer if it exists. Next option is for it to allocate near the indirection block. Finally if that is not possible it attempts to allocate in the same block group as that of the *inode*. Important is to realize that the goal block will only be used for locality. The data does not have to be actually placed into it. Under this condition the two function calls can be avoided. The goal block can be immediately set to the some place within the same block group as the *inode*. This is could be done in *ext2_get_blocks*.

The function *ext2_get_blocks* has to find the number of blocks to allocate. This does not just hint to number of data blocks but to indirect blocks as well. This is done by calling 5. *ext2_blks_to_allocate*. The function returns the count of total blocks to be allocated.

5.3.2 Function 6. ext2_alloc_branch

Up to this point no allocation has been made. Previous function calls were made in an effort to prepare the data necessary for allocation. The actual allocation happens in 6. *ext2_alloc_branch*. As can be seen in Figure 5.15 many functions are called to achieve successful allocation. The first called function is the 6.1 *ext2_alloc_blocks*. As the name suggest the allocation is done in this function. The task therefore of 6. *ext2_alloc_branch* is to allocate the branch and prepare it to be added to the *inode*. The blocks provided by 6.1 are zeroed out and then the indirection chain is written to them. It should be noted here that 6. *ext2_alloc_branch* can only allocate a single branch. The final indirect block contains a list of pointers to the data blocks. This means that 6.1 *ext2_alloc_blocks* allocates blocks for the branch and as many data blocks as required or stops once it cannot allocate more to the same branch. Therefore on return 6.1 has to signal the number of direct or data blocks allocated.

5.3.3 Function 6.1 ext2_alloc_blocks

The function 6.1 *ext2_alloc_blocks* attempts the allocation of minimum set of blocks. The number required for allocation is the total of indirect blocks for the branch in addition to at least a single direct block. If more blocks are available allocation continues until all required direct blocks are allocated. Still 6.1 *ext2_alloc_blocks* does not do the block allocation itself. Instead it calls 6.2 *ext2_new_blocks*. As the name suggests function 6.2 finds and allocates a new set of contingent blocks. Since 6.1 *ext2_alloc_blocks* could require more blocks than currently available in a contingent group, it repeatedly calls 6.2 *ext2_new_blocks*. This continues until the required number of blocks is allocated. It is important that 6.1 *ext2_alloc_blocks* keeps track of whether the indirection blocks have been allocated or not. If the minimum is not achieved an error has to be returned.

5.3.4 Function 6.2 ext2_new_blocks

The most complex of all functions is 6.2 *ext2_new_blocks*. This is because the actual allocation takes place in this function. The allocation scheme is mentioned in the comment above the function as being relative to the goal block. The goal block is used if empty. If that is not possible, a free block is searched for within a distance of 32 blocks from the goal block. Otherwise a forward search is conducted for a free block. For that purpose the function 6.2 *ext2_new_blocks* uses the bitmaps. The function starts by checking for sufficient quota allowed by the VFS for allocation to this file. The function then tries to set reservation information. If the file system is mounted without preallocation the reservation window has the size of zero. This part can be removed as there is no need for an added check operation if the preallocation will not be used. An additional reservation part that can be removed is the check for sufficient free blocks for reservation.

Since the file system has a limit of available blocks, 6.2 *ext2_new_blocks* has to check if free blocks are available. If sufficient free blocks are available 6.2 finds by usage of the goal block a group target. This is the block group that the allocation should be done within. For actual allocation block allocation within the target group 6.2 *ext2_new_blocks* uses 6.3 *ext2_try_to_allocate_with_rsv*. It should be noted that 6.3 can only allocate within the same block group. If allocation is not possible within that group another group with sufficient free blocks for both allocation and preallocation blocks has to be found. On failing to find an appropriate block group 6.2 *ext2_new_blocks* switches off preallocation and repeats the block group finding process. Finally if no block group is found error is returned, signaling that no free space is available. This shows the amount of overhead such reservation tables can place on the file system allocation process. Therefore in the interest of removing preallocation block groups are immediately searched for one containing sufficient free blocks without reservation. The issue of preallocation is repeatedly checked. Thus several other changes have been made to this function. These however address the same issues as mentioned before.

Once 6.2 *ext2_new_blocks* gets the blocks allocated it has to check for conflicts. The allocation blocks have to be out of the range of the data block bitmap, the *inode* bitmap and the *inode* table. Now 6.2 has to adjust the number of free blocks available. Additionally 6.2 *ext2_new_blocks* already took the number of blocks required for allocation from the VFS quota. Given that not all of these blocks were allocated the remainder has to be returned to the VFS. Further administrative tasks are done at the end of 6.2. It should be noted that information that is stored on the block device is kept in buffers. As 6.2 *ext2_new_blocks* changes these buffers it marks them as dirty. Therefore the final step in 6.2 is to synchronize with the block device. This will write all dirty buffers back to the storage.

5.3.5 Function 6.3 ext2_try_to_allocate_with_rsv

As mentioned earlier, 6.2 *ext2_new_blocks* calls 6.3 *ext2_try_to_allocate_with_rsv*. This is a function that attempts allocation using reservation. However it starts by checking for for reservation window. If the file system is mounted with no reservation then 6.3 *ext2_try_to_allocate_with_rsv* calls another function 6.4 *ext2_try_to_allocate*. This shows that preallocation can produce an overhead even when switched off. Again due to lack of interest in preallocation this function call can be removed. Therefore every call for 6.3 *ext2_try_to_allocate_with_rsv* is replaced with 6.4 *ext2_try_to_allocate*. This has to be done in 6.2 *ext2_new_blocks*. Based on this setting the rest of 6.3 *ext2_try_to_allocate_with_rsv* can be ignored.

5.3.6 Function 6.4 ext2_try_to_allocate

The actual block allocation happens in 6.4 *ext2_try_to_allocate*. The function sets a range of blocks to search within for free blocks. In case of preallocation the range is set to the reservation window. Since preallocation no longer exists this check can be omitted. For no preallocation the window is set to start at the goal block and end at the group block end. 6.4 then calls 6.5 *find_next_usable_block*. As the name suggests this returns a free usable block or a negative number in case of failure. Once a usable block is found 6.4 *ext2_try_to_allocate* loops on subsequent blocks to check the bitmap. The loop ends at either finding a used block or reaching the number of requested blocks. The function 6.4 now has a group of contingent allocated blocks. The bitmap is updated to allocate the blocks. Finally the number of allocated blocks is returned with the physical address of the first block.

5.3.7 After Removing Preallocation

Although preallocation can be switched off in the Ext2 filesystem, this does not remove overhead. The previous explanation demonstrates how complicated such Kernel edits are. Figure 5.16 shows the call graph of *ext2_get_blocks* after removing preallocation. The figure cancels out functions that have been removed. It also highlights functions that have been edited.

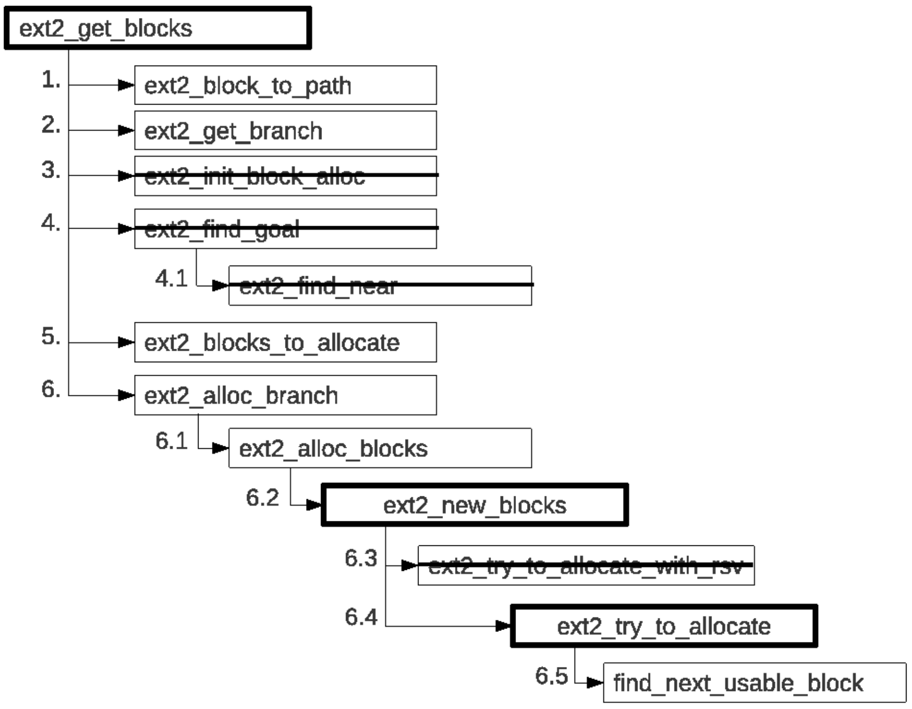


Figure 5.16: ext2_get_blocks call graph after removing preallocation

The changes done on the file system are difficult to be tested. The effect is highly dependent on the processing speed and the underlying device. On most settings the difference might not be noticed at all. However it can be shown that applying changes depending on SSDs access patterns is possible. In fact the call graph can be optimized even further. On such optimization is the repeated call of 6.2 *ext2_new_blocks* due to it being able to allocate only contingent blocks. Literally the whole call graph beneath 6.1 *ext2_alloc_blocks* can be removed. To achieve that 6.1 could use 6.5 *find_next_usable_block* to find the next free blocks. Perform the contingent block allocation explained in 6.4 *ext2_try_to_allocate*. If more blocks are needed than allocated the process could be repeated. Once done 6.1 *ext2_alloc_blocks* returns all blocks to 6.

ext2_alloc_branch to complete the allocation process. Not only will this save the cost for several function calls, but also will completely remove preallocation checks. In addition to that side administrative issues within removed functions such as variable creation will be avoided. It should be noticed however that this requires a further detailed look at functions implementation. Error handling is done by checking function return type. The task of error handling done in a removed function will have to be performed somewhere else.

The Kernel and associated file systems can adapt for SSD usage. They can be even adapted to switch between SSD and HDD. However such changes must be made with more information about complete IO stack. This does not just include block devices, but even application. The difficulty shown by removing the preallocation will also be faced in further optimizing Kernel file systems. These optimizations are worth it. The SSD might not be the only storage technology for which such optimizations are useful. In fact SSD is not the only storage technology for which optimization in general must be done. For that reason, file systems have to start acquiring information from block devices on their types. This will facilitate the developers task for setting optimizations. A file system can switch on and off certain functions for different technologies. However such switching will have to be done with better care. As seen preallocation although switched off still represents overhead that can be avoided. This appears to be due to the file system expectation that the normal case is the usage of preallocation. To optimize such switching developers will have to make more effort in more fluent switching.

6 Conclusion and Future Work

6.1 Conclusion

Computational power of HPC systems has dramatically increased over the past years. However IO has not increased enough to fulfill the transfer rate required for such powerful computing units. The IO gap is increased by the amount of data capacity available for HPC applications. Active storage fabrics is a possible solution that embeds storage into the HPC units. This promises the decrease of distance between data stored and the computing units. The challenge of active storage is to enhance single node IO performance. Most HPC nodes are not optimized for internal communication with storage devices. To achieve such optimization a detailed understanding of the IO stack is required and storage devices need to be studied. Furthermore storage IO must be analyzed under different conditions to inspect possible optimizations.

Analysis of storage IO show that the IO stack is formed of several complex layers. The complexity is increased by the abstraction required to communicate with different file systems and different devices. The IO stack is generally built for single request queues resulting in low parallelism. Since modern processors are built using multiple cores, single queue operation becomes a bottleneck. Additionally new storage technology provides more than one block device to the system on the same hardware. The IO stack has difficulty in utilizing these block devices in parallel. The emerging solid state technology such as SSD promises higher performance. However the development of IO stack has been mostly done in the HDD era. The SSDs therefore have to implement HDD interfaces to communicate with the IO stack. By hiding the SSD identity the IO stack continues to optimize access of storage for HDD. SSD is based on a different storage concept. Thus the optimized HDD access is suboptimal for SSD.

Parallelism has been generally designed on higher system levels. As technology size decreases problems faced on system levels are now faced within single units. Large scale solutions offer a starting point for optimizing parallelism on single node. The challenge of implementing the large scale solutions is the limitations existing on single nodes. System parallelism solutions can afford overheads induced by complex parallel file systems. In comparison single node computing capacity is limited and shared with applications. Furthermore analysis show that IO performance is effected by CPU frequency. In general lower CPU frequency results in poor IO performance. Operation of IO can be enhanced by the use of functional partitioning. The target is to bind IO processes to different CPUs. Also division of available block devices among processes or CPU could further increase IO efficiency. To overcome single device limitations the IO stack has to utilize several block devices. These will offer more performance if used efficiently in parallel.

The target of a parallel IO stack implementation is to achieve an efficient use of available resources. However changes to original IO stack must be reduced to a minimum. The purpose is to avoid complex Kernel changes that could result in disturbing normal IO stack operations. Other unaltered functions should continue to operate unhindered. Furthermore there is an immense difficulty in altering current existing applications. This has delayed applications from exploiting new optimized IO interfaces. A parallel IO stack should therefore offer enhanced operation without requesting changes to the applications. The parallel IO implementation should accommodate current interfaces using a transparent parallelism. However as applications and interfaces advance the implementation must be able adapt achieving same or better performance. There is an inherit advantage to functional binding in parallel systems. For that reason a parallel IO implementation should be able to divide operation among available resources. Current devices allow for free write division on block devices but restrict read division. Write operations can therefore utilize block devices in the most optimal method. In comparison read operations need to be performed on the same devices on which the data has been written. A parallel IO implementation should provide the same for read division among block devices. All these restrictions should be held without performance degradation and with a small cost. The target is to utilize parallelism for increased IO operations per time unit.

The proposed implementation addresses the limitations and specifications required from a parallel IO. By use of a unified file system, all devices can be accessed using conventional IO interfaces. Parallelism becomes transparent to applications. These can therefore operate unchanged. In addition to that by utilizing simple conventional file systems, the overhead is kept to a minimum. This could not be achieved by the use of complicated parallel file systems. Furthermore the use of a global storage space available to all devices allow efficient resources utilization. Reads can be divided among devices in the most optimal method. Data can be written and subsequently read from a different device. The unified conventional file system used carry out normal data locking algorithms. Thus there is no need for performing data locking on the device level. The implementation promises a free division of operations among resources. Allocation of devices for processes or CPU could be implemented with simple changes. The parallel IO implementation proposed achieves these specifications by as little alteration to the Kernel as possible. Other file systems can operate without the need for change. The Kernel can employ traditional access schemes without being prevented by the implementation from normal Kernel operations. Furthermore testing the implementation showed an increased performance. By increasing number of devices an increase in performance is observed. Although proposed implementation is realized using loop devices, a hardware design is possible with simple changes. The design offers the possibility of supporting current and future interfaces such as NVM Express. The support requires minimum changes to the implementation and could adapt to future application needs.

On the other hand, SSDs are superior to HDDs on many levels. They do not suffer from seek times. Thus promising higher random access performance than provided by common HDD. Previously addressed parallelism can further utilize underlying SSD properties. However the IO stack has been mainly developed and optimized for HDDs. These HDD optimizations in some cases lead to processing overhead which is wasted when performed on SSD access. Therefore their was a requirement for detailed analysis of SSD operation in correlation with IO stack HDD optimizations. Reliability issues faced by SSD means that conventional file systems cannot use their own address space. Wear leveling requires out of place writing. Therefore if conventional file systems were directly accessing the flash memory on SSD, endurance would drastically drop. Most SSDs solve this problem by the use of Flash Translation Layers or FTL. The file system is provided a different address space scheme than implemented on the physical layer. Other SSDs require complete change of file systems. This has the disadvantage of requiring application changes and is seldom compatible with existing stack layers. Another issue for the use of SSD is access pattern behavior. While HDD requires sequential access for better performance, SSD are limited by other factors. Reliability constraints of SSD employs wear leveling, garbage collection and error correction and detection codes. Also SSDs cannot write without a previous erase. There is an added complexity due to the minimum erase and write sizes allowed by a flash memory. These issues need to be addressed by the SSD controller. To achieve optimum performance the controller design should also be done in accordance with the system configuration. A system employing a processor with multiple weak cores would benefit from a hardware designed controller. A software design that is not appropriately done using several threads result in a limited performance. This design would become dependent on the weak frequency of the single core.

Due to SSD complexity these analysis are difficult. Performance is highly dependent on design. However by analyzing performance under different conditions some conclusions could be made on SSD operation. Analysis show that an added computing performance might not increase the IO parallel operation of a SSD. Software device drivers with single or small number of threads result in a binding to a single core. Thus additional compute power would not subsequently increase IO performance. Using multiple threads to implement device driver could increase performance. However the SSD controller operation should not abuse the computing power available. The SSD device drivers in an active storage configuration will share processing power with applications. Priority should remain for processing applications. In comparison a more hardware based controller implementation promises a system independence. The hardware design would not be effected by CPU frequency as much as a software implementation. Still the SSD will remain limited by best hardware performance possible. A software design however would have the full processing power available on the system to utilize. Another issue for SSD is the write versus read performance. The inherit flash memory result in a lower write performance. This affects most of all the maximum bandwidth that the device can write. Internal SSD parallelism would provide for a better write performance. Additionally by the use of more application and system related controller designs write latency can be reduced. Controllers should handle reliability issues with more efficiency. Furthermore analysis of the conventional file systems show that overhead is limited. This however is dependent on the system performance and the underlying device design. Conventional file systems utilize CPU time to avoid HDD seek time. SSDs do not benefit from seek time reduction. Thus file system overhead can be further decreased by removing seek time reduction algorithms. This is true for the whole IO stack.

As an example for overhead reduction by redesign of IO stack for SSD, preallocation is removed from the a conventional file system. Conventional file systems preallocate free blocks to a file in order to provide more contingent data allocation. To achieve preallocation in contingent form file systems employ complicated algorithms. These exert an overhead of computation to the storage IO requests. SSDs have a better random access performance than HDD. Furthermore SSDs use a different physical address space for allocation than the one shown for the file system. Thus preallocation makes no improvement to SSD performance. Although the file system might provide a method for switching off preallocation, the overhead is not much decreased by this method. The actual removing of preallocation is shown to be a complicated implementation. Despite that it is possible and could result in better performance. The process of removing preallocation required analyzing every function in the function call graph of the allocation method. Functions used for preallocation are removed. Other functions are edited to avoid preallocation and reduce complexity. This should be done with absolute care. Functions should not be altered to the extend of failing. Applications should also not be changed due to preallocation removal. Additionally the Kernel should remain unchanged if at all possible. The preallocation removal done in this study shows that these restrains can be kept. The write form is reduced which promises better performance.

6.2 Future Work

IO will continue to be in the center of attention of the HPC community. Building future massive scale computers will be dependent on how far IO can be optimized. In essence, as processing units continue to grow in parallelism, so should IO. Parallelism has to be adopted into both new storage technologies and the existing IO stack. The design mentioned in this study shows how far parallelism can achieve higher IO performance. The hardware changes required by the design are simple and easy to implement. The global storage space shared among controllers can be fabricated by the use of bus system connecting all flash memory chips with all controllers. The bus system would require an additional bus arbiter which regulates bus traffic. The controllers themselves could be implemented on Field Programmable arrays or FPGAs. Changes to the IO stack could be implemented without the need of Kernel or module recompile. The file system will have to detect on the fly how many devices exist on the same board and communicate with the same storage space. This could be achieved by a signal that the board has to send to the file system informing it with the number of devices. Another possibility is for the file system to detect the number of devices using the PCI bus registration information. Furthermore the file system could implement different access patterns. The allocation of devices could be done on a CPU or process basis. Further functional partitioning schemes could show higher IO performance.

The market already offers several SSD designs with multiple devices on a single board. These however do not contain a global storage space. Each controller can communicate to only a hardwired set of flash chips. By sacrificing the global storage space the design can adapt to these SSDs. The file system will have to map the address space during data write to a single devices. This will allow the file system to divide the write requests among the available devices. However read operations will have to be done on the same device as written to. Although this limits the design advantages, it still give the opportunity of testing the design with real physical SSD devices.

The most important aspect of parallelism is the use of an appropriate parallel file system. These file systems are however too complex for single node performance. The study done here offered the redesign of a conventional file system to adopt parallelism. Another approach could be the simplification of complex existing parallel file systems. This would require an extensive effort in understanding such file systems. In addition to that the use of parallel file systems of global scale in ASF should be studied. Parallel file systems operate on a global view of data and divided the data according to replication and performance factors. Data division however does not consider any computation issues. By using parallel file systems single nodes with storage will have to access data on the global file system including data on adjacent nodes. It would be beneficial to study the possibility of dual address space. The global file system operates and saves data on the complete ASF using one address space. The second address space would be the view of the single node to information on the storage next to it. Using this setting a single node would be allowed to do faster IO without the need of accessing global networks.

As shown by this study there is large room for improving the IO stack to adopt SSDs. To achieve better understanding of possible optimization for the IO stack more analysis has to be done on the SSDs offered by the market. Also it is important to implement and test a new SSD design in relation to current modern processors with parallelism and hardware flash memory services. Since the IO stack still requires SSDs to use HDD interfaces, optimization of SSD for HDD access patterns could result in better performance. For example an SSD could divide contingent data offered by HDD scheduling schemes on different flash chips. This would result in internal SSD parallelism.

Further optimization of IO stack for SSD is required. Since the SSD will have to coexist with HDD the IO stack will have to support optimizations for both. In the interest of user friendly systems the user should not be required to enter the type of device for one optimization to be activated. The IO stack should be able to detect the device storage type. Newly designed SSDs could set a variable signaling the storage type. The HDD could be set as the default for such a variable. Not only will this facilitate optimization for SSD but also provide developers with the ability for optimization for future storage technologies.

6.2.1 IO Compression

IO compression is a fast emerging tool for optimizing IO. It requires extensive study. There are many different types and algorithms that perform compression. These offer different compression rates and different compression speeds. The compression rate also depends on the type and form of data. This will therefore also require an extensive study of application data form. It is of absolute importance that the compression be transparent. Data should not be lost by the compression or decompression. This in turn will avoid application changes. Another important factor is compression and decompression processing requirements. As shown in this study functional partitioning increases performance. Therefore allocating cores to the compression operation could further increase benefits from IO compression. However the compression should decrease disturbance to the application. For that reason it should not take to much CPU time to compress or decompress from the running applications. Another option is for compression to utilize idle cores. An algorithm offering different compression rates at different processing requirements could be implemented. Using this compression algorithm the system could support many compute bound threads at low compression rates or few IO bound threads at high compression rates. The difficulty however with this implementation is compression versus decompression. The system will have to guarantee idle cores during decompression. During a write operation the system is free to choose the rate and speed of compression according to number of idle cores. In comparison during a read operation requiring decompression the system is bound by the rate of compression done during write.

Compression can be implemented on many different IO levels. The main two options available is software versus hardware compression. To increase IO performance software compression can be implemented on the upper layers of the IO stack. Some file systems already offer compression. Another option is to implement compression in the device driver. This would allow for compression in relation to the underlying storage technology. The use of software compression would allow for a high IOPS and bandwidth to be supported at the same time. As seen from tests done in this study bandwidth and IOPS are inversely proportional with the blocksize as the factor between them. By using compression large block sizes could be decreased. This would allow for the increase of IOPS without decreasing the bandwidth. However it should be noted that the term IOPS used in this case refer to actual IOPS. These are the IOPS achieved after compression. The compression operation itself will result in additional latency. The same issue is applicable to bandwidth.

Another option is the use of hardware compression. An SSD offering on board compression would not increase IO performance. The data has to be decompressed before returned to the system during a read operation. However on board compression offers the increase of SSD endurance. By storing less data during a write operation, number of write/erase cycles used will be decreased. The problem however becomes out of place writing. SSDs have to use out of place writes due to wear leveling techniques. Data updates are written to another location and the old data is market for garbage collection. A compression done on the sum of the data would require a rewrite to decompress, update and rewrite the whole data. Not only will this result in write delay but also might decrease SSD endurance.

IO compression requires further analysis of the IO stack. It also requires further understanding of storage technology. The analysis should be done with reference to both IO stack parallelism and SSD implementations.

References

[1] Y. Chen, X. Sun, R. Thakur, H. Song and H. Jin, “Improving parallel I/O performance with data layout awareness,” in *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pp. 302 –311, sept. 2010.

[2] A. Jackson, F. Reid, J. Hein, A. Soba and X. Saez, “High performance I/O,” *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, vol. 0, pp. 349–356, 2011.

[3] B. G. Fitch, A. Rayshubskiy, M. C. Pitman, T. J. C. Ward and R. S. Germain, “Using the active storage fabrics model to address petascale storage challenges,” in *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW ’09, (New York, NY, USA), pp. 47–54, ACM, 2009.

[4] A. Snell, *GPFS: optimizing performance with parallel data management*. Intersect360 Research, Inc., August 2010. White Paper.

[5] S. Seelam, I. H. Chung, J.Bauer and H. F. Wen, “Masking I/O latency using application level I/O caching and prefetching on Blue Gene systems,” in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1 –12, Apr. 2010.

[6] F. Schmuck and R. Haskin, “GPFS: a shared-disk file system for large computing clusters,” in *In Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pp. 231–244, 2002.

[7] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward and P. Sadayappan, “Scalable I/O forwarding framework for high-performance computing systems,” in *Cluster Computing and Workshops, 2009. CLUSTER ’09. IEEE International Conference on*, pp. 1 –10, 31 2009-Sept. 4 2009.

[8] R. Love, *Linux Kernel Development*. Addison-Wesley Professional, 2010.

[9] W. Mauerer, *Professional Linux Kernel Architecture*. Wiley Publishing, Inc., 2008.

[10] W. Hutsell, J. Bowen, and N. Ekker, *Flash Solid-State Disk Reliability*. Texas Memory Systems, Inc., Houston, Texas USA., Nov. 2008. White Paper.

[11] E. Spanjer, *Flash Management – Why and How? A detailed overview of flash management techniques*. SMART Modular Technologies, Nov. 2009. White Paper.

[12] M. Sanvido, F. R. Chu, A. Kulkarni and R. Selinger, “NAND flash memory and its role in storage architectures,” *Proceedings of the IEEE*, vol. 96, pp. 1864 –1874, Nov. 2008.

[13] C. Egger, “File systems for flash devices.” http://www-vs.informatik.uni-ulm.de/teach/ss10/rb/docs/flash_fs_ausarbeitung.pdf, Jun. 2010.

[14] M. Li, S. S. Vazhkudai, A. R. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann and G. Shipman, “Functional partitioning to optimize end-to-end performance on many-core architectures,” in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’10, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2010.

[15] S. W. Son, S. Lang, P. Carns, R. Ross, R. Thakur, B. Ozisikyilmaz, P. Kumar, W. Liao and A. Choudhary, "Enabling active storage on parallel I/O software stacks," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1 –12, may 2010.

[16] P. M. Dickens and R. Thakur, "Improving collective I/O performance using threads," in *Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, IPPS '99/SPDP '99, (Washington, DC, USA), pp. 38–45, IEEE Computer Society, 1999.

[17] K. Coloma, A. Ching, A. Choudhary, W. Liao, R. Ross, R. Thakur and L. Ward, "A new flexible MPI collective I/O implementation," in *Cluster Computing, 2006 IEEE International Conference on*, pp. 1 –10, sept. 2006.

[18] S. Nagar, H. Franke, P. W. Y. Wong, B. Pulavarty, J. Morgan, J. Lahr, B. Hartner and S. Bhattacharya, "Improving linux block I/O for enterprise workloads," in *In Proceedings of the Ottawa Linux Symposium*, pp. 390 –406, jun. 2002.

[19] T. White, *Hadoop The Definitive Guide, Second Edition*. O'Reilly Media, Inc., 2011.

[20] L. Ou, X. Chen, X. B. He, C. Engelmann and S. L. Scott, "Achieving computational I/O efficiency in a high performance cluster using multicore processors," in *Proceedings of the* 4^{th} *High Availability and Performance Workshop (HAPCW) 2006, in conjunction with the* 7^{th} *Los Alamos Computer Science Institute (LACSI) Symposium 2006*.

[21] C. P. Ribeiro, J. F. Mehaut and A. Carissimi, "Memory affinity management for numerical scientific applications over multi-core multiprocessors with hierarchical memory," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pp. 1 –4, april 2010.

[22] NVM Express, http://www.nvmexpress.org/, *NVM Express Explained*, 2011. White Paper.

[23] M. Feldman, "IBM specs out Blue Gene/Q chip," *HPC Wire*, Aug. 2011. http://www.hpcwire.com/hpcwire/2011-08-22/ibm_specs_out_blue_gene_q_chip.html.

[24] Fusion-io, Inc., *ioDrive Data Sheet*, 2010. http://www.fusionio.com/data-sheets/iodrive-data-sheet/.

[25] J. He, J. Bennett and A. Snavely, "DASH-IO: an empirical study of flash-based IO for HPC," in *Proceedings of the 2010 TeraGrid Conference*, TG '10, (New York, NY, USA), pp. 10:1–10:8, ACM, 2010.

[26] M. Dunn and A. L. N. Reddy, "A new I/O scheduler for solid state devices." Department of Electrical and Computer Engineering Texas A&M University, Tech. Rep. TAMU-ECE-2009-02-3, 2009.

[27] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau and V. Prabhakaran, "Removing the costs of indirection in flash-based SSDs with nameless writes," in *Proceedings of the 2nd USENIX conference on Hot topics in storage and file systems*, HotStorage'10, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 2010.

[28] E. Seppanen, M. T. O'Keefe and D. J. Lilja, "High performance solid state storage under linux," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1 –12, May 2010.

[29] Y. Wang, K. Goda, M. Nakano and M. Kitsuregawa, "Flash SSD oriented IO management for data intensive applications," in *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST 2011)*.

[30] F. Chen, D. A. Koufaty and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS '09, (New York, NY, USA), pp. 181–192, ACM, 2009.

[31] F. Chen, R. Lee and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 266 –277, Feb. 2011.

[32] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse and R. Panigrahy, "Design tradeoffs for SSD performance," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, (Berkeley, CA, USA), pp. 57–70, USENIX Association, 2008.

[33] X. Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, (New York, NY, USA), pp. 10:1–10:9, ACM, 2009.

[34] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn and D. K. Panda, "Beyond block I/O: rethinking traditional storage primitives," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 301 –311, Feb. 2011.

[35] C. Dirik and B. Jacob, "The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization," *SIGARCH Comput. Archit. News*, vol. 37, pp. 279–289, Jun. 2009.

[36] J. Lee, E. Byun, H. Park, J. Choi, D. Lee and S. H. Noh, "CPS-SIM: configurable and accurate clock precision solid state drive simulator," in *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, (New York, NY, USA), pp. 318–325, ACM, 2009.