Performance Evaluation of In-storage Processing Architectures for Diverse Applications and Benchmarks

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Manas Minglani

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctorate of Philosophy

Dr. David J. Lilja

June, 2018

Acknowledgements

There are many people who have earned my gratitude for their contribution to my time in the graduate school. Firstly, I would like to thank Dr. David J. Lilja and Dr. David Du for their support during my time in the University.

I would also like to thank my parents and my extended family for their untiring support all throughout my life.

I would also like to thank Ashwin Nagarajan and Xiang Cao for working with me. They were my team mates and helped tremendously in completing this dissertation.

And all those people out there who I did not mention, please remember that if your name starts with an ASCII character and could be represented in any format - binary, hexadecimal, decimal etc, you are in my deepest thoughts.

Dedication

I would like to dedicate this work to the scientists who relentlessly work hard without glamor, bring real change to the society and yet, often are unnoticed.

Abstract

As we inch towards the future, the storage needs of the world are going to be massive and diversified. To tackle the needs of the next generation, the storage systems are required to be studied and require innovative solutions. These solutions need to solve multitude of issues involving high power consumption of traditional systems, manageability, easy scaling out, and integration into existing systems. Therefore, we need to rethink the new technologies from the ground up.

To keep the energy signature under control we devised a new architecture called Storage Processing Unit (SPU). For the modeling of this architecture we incorporate a processing element inside the storage medium to limit the data movement between the storage device and the host processor. This resulted in a hierarchal architecture which required an extensive design space exploration along with in-depth study of the applications. We found this new architecture to provide energy savings from 11-423X and gave performance gains from 4-66X for applications including k-means, Sparse BLAS, and others.

Moreover, to understand the diverse nature of the applications and newer technologies, we tried the concept of in-storage processing for unstructured data. This type of data is demonstrating huge amount of growth and would continue to do so. Seagate's new class of drives - Kinetic Drives, address the rise of unstructured data. They have a processing element inside disk drives that execute LevelDB, a key-value store. We evaluated this off-the-shelf device using micro and macro benchmarks for an in-depth throughput and latency benchmarking. We observed sequential write throughput of 63 MB/sec and sequential read throughput of 78 MB/sec for 1 MB value sizes. We tested several unique features including P2P transfer that takes place in a Kinetic Drive. These new class of drives outperformed traditional servers workloads for several test cases.

Finally, large number of these devices are needed for huge amounts of data. To demonstrate that Kinetic Drives reduce the management complexity for large-scale deployment, we conducted a study. We allocated large amounts of data on Kinetic Drives and then evaluated the performance of the system for migration of data amongst drives.

Previously developed key indexing schemes were evaluated which gave important insights into their performance differences. Based on this study we can conclude that efficient mapping of key-value pairs to drives could be obtained. This lead to an understanding of the trade-off between the number of empty drives and mapping of different key ranges to different drives.

In conclusion, in-storage processing architectures bring an interesting aspect where processing is moved closer to the data. This leads to a paradigm shift which often results in a major software and hardware architectural changes. Furthermore, the new architectures have the potential to perform better than the traditional systems but require easy integration with the existing systems.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation focuses on storage technology and systems that could potentially address the future storage needs. The new technology needs to solve multitude of issues before it could be deployed for big high performance systems [1]. These issues primarily are:

- Architecture of Storage Device
- Energy Signature
- Application Level Performance
- Scalability

To satisfy the need for extreme-scale computing the future systems need to be energy efficient while handling massive amounts of data. The systems often have to deal with multiple "walls": the memory wall, power wall, and parallelism wall.

Memory wall happens due to the difference in memory and processor speeds [2]. This lead to engineers designing hierarchical memory and storage. Increased number of gates on a chip leads to power wall. As the operating frequency increase so does the power consumption of the chip. To counter this, small but parallel architectures are created on a chip to reduce the energy signature. However, the energy signature remains significant and this process consumes significant amount of energy and requires superior cooling methodology. This acts as a limiting factor. One major reason for high energy consumption is the constant moving of data back and forth between the permanent storage device and the CPU. Due to the recent analysis it is concluded that the energy to move the data is becoming comparable to the energy consumption due to processing itself [3]. We do not have mechanisms for controlling the data movement and it could potentially hurt the High Performance Computing [4]. New models of computation are immediately required to tackle both energy and performance.

We attempt to understand the memory hierarchy and devise methods to apply the same concept for processing as well. This would distribute the processing across the entire system instead of concentrating the processing capability at one point. Different processing elements spread across would have different power or energy signatures. By placing them at different locations in a system, our systems would have lot more flexibility on data movement and synchronizing the partial results for generating the final output result. However, this would require sufficient know-how of both the processing as well as storage technologies. We choose flash and low power processing units to prove the hypothesis of our research.

Flash technology offers several benefits over contemporary disk based technology. They offer fast random access, high throughput, and low power consumption, over the older technologies. Moreover, flash-based SSDs provide serial I/O interfaces and has wider buses for media transfer [5]. Thus, flash technology is considered for our research.

The next important task is to reduce the data movement for energy and performance efficient processing. We incorporate a low-processor in the flash packages. These coprocessors form level-1 of the computation hierarchy. The much more powerful processors in the SSD controller become the next level in the processing hierarchy. And finally the last level of the processing would be by the Host. This tight coupling of storage and processing is called the Storage Processing Unit (SPU) [6].

Since, we are proposing a new architecture, it requires an exhaustive design space exploration. The exploration is required for application as well as architectural space. First, we choose a wide range of applications with different behaviors. Next, we study these to identify bottlenecks and identify efficient methods to map them to our newly proposed architecture. The behavior of the applications could broadly be categorized as memory bound or compute bound. We use this analysis to resolve the key issues that are observed in the traditional SSD based system. We call it the baseline system or the baseline configuration. We also realized that to exploit the internal features of the SSD and the new hierarchical processing, we need to map applications to the architecture efficiently. Identification of weaknesses of the system and applications also enabled use to execute the optimized versions of the applications with different variants of the architecture. We made several predictions and backed those with our results. We are able to demonstrate that by taking processing closer to the storage and maintaining different flows of data, we achieve significant performance and power gains. This newly proposed architecture gives us a new model for processing and storage as shown in the Figure 1.1.

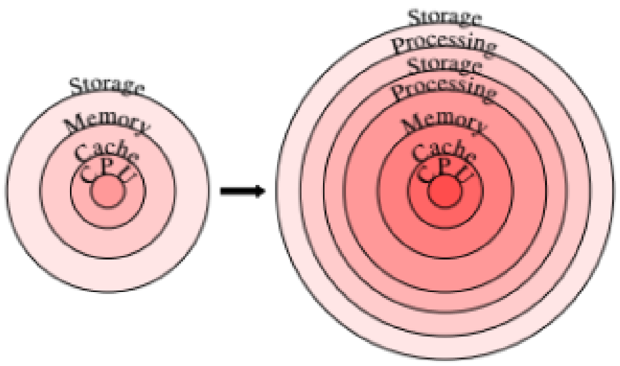


Figure 1.1: Comparison of Traditional vs New Model of Computation

We modeled and simulated the new propsed architecture however, it is important to find how the applications' performance would be impacted on real hardware. We wanted the real system with in-storage processing capability to address the growth of digital data which is growing at 40%-50% per year [7]. Most of this data explosion is due to unstructured data. International Data Corporation (IDC) predicts that in 2017 80% of the 133 exabytes of global data will be unstructured [8].

In the traditional systems data is accessed using the traditional file-based or block-based systems [9]. However, these systems could prove to be inefficient for future technologies. To tackle the unstructured data the storage systems needs to scale out horizontally [10]. Therefore, object storage can overcome the limitations of the file-based systems. They are scalable and could be software defined [11]. The data is usually communicated as objects rather instead of files or blocks. The benefit of object storage is that it stores metadata alongside to the object [9]. Furthermore, the object storage is flat structured and does not let the higher-level applications to manipulate the data at the lowest level. We use unique identifiers called ”key” and it is used to access an object called ”value. This form of storage is called as key-value store (KV store).

In 2015, Seagate announced a new class of drives called Kinetic Drives [11]. These drives have a built-in processor that runs LevelDB based key-value store [12]. Moreover, they do not have a typical SATA or SAS interface. The drives communicate using TCP/IP over Ethernet. Each drives acts a tiny server in itself. Additional feature that sets this drive apart from other hardware is the P2P (peer-to-peer) functionality. These drives can transfer data directly between each other without the interference from the client. This frees the client and it could perform other tasks [13]. This architecture substantially reduces the software and hardware layers for an object storage systems. All a programmer needs is the knowledge of the API that comes with the drives.

Since, these drives came with a brand new concept we wanted to understand the drive’s key functionalities, features, and the performance of the drives [14]. We perform performance evaluation of the Kinetic drives against servers with LevelDB installed. These experiments gave us great insights about the possibility of replacing traditional hard drives with the Kinetic Drives. Moreover, the need for detailed analysis was required because the specification sheets [15] did not provide the much needed details for our desired metrics which are throughput, latency, and other LevelDB related features.

Programmability of the drives is also crucial for wide acceptance of the technology by the industry. Therefore, our significant effort went to understanding the ease of programmability of these drives. The salient feature of this drive is P2P transfer along with ”Get,” ”Put,” and others. Several developed tests helped us understanding the bottlenecks and the limitations of the drives.

In summary, we conduct tests on Kinetic drives to obtain the trends related to throughput and bottlenecks, limits, trade-offs, and inherent characteristics. We also compare the key-value store hardware with servers that run LevelDB on conventional hard drives. We used micro and macro benchmarks such as Yahoo Cloud Serving Benchmark (YCSB). Finally, we share our experience that sheds light on the programmability of these drives.

Final part of the dissertation is one managing a large number of Kinetic drives. To replace the traditional drives with Kinetic Drives, we need to learn how to manage a large number of these drives [16]. We performed a study to understand the distribution and migration of data in relation to the key ids. When a large number of key-value pairs are written to the drives, we have to distribute them with maximum efficiency. Here efficiency needs to be defined and explored. Different distribution methods have different trade-offs associated with them. A metadata server is used that provides indexing of the large number of keys. The idea is if every key-value pair is indexed to different drive then it would make indexing a very difficult job. The indexing in itself would consume significant amounts of data. This is definitely undesirable because it would affect the throughput and latency of the system. Moreover, maintaining the metadata server along with the drives would also become lot more tedious. Therefore, we try to create key ranges and store the data in individual Kinetic drives based on these ranges. Maintaining just the key ranges instead of every key id saves a lot of storage space in the metadata server. Different configurations are considered for efficient data migration. However, we do not want to waste storage devices. We want to utilize as many devices as possible without negatively affecting the performance. Therefore, in this research different data allocation schemes are presented with the target to have a easy maintainability for the metadata server.

In this research, we undertook study to capitalize on the rise of unstructured data. For tackling this massive surge, we need to devise a new storage architecture and understand it from multiple perspectives including manageability, performance evaluation, design, and implementation. In this dissertation, we have attempted to address these issues and made the below given contributions.

The major contributions of this thesis are:

- Design space exploration of a new proposed in-storage processing architecture
- Performance evaluation of a new class of hard drives called Kinetic drives that encapsulate in-storage processing

Remainder of the dissertation is divided in the below given sections.

- Chapter 1 introduces the overarching concept of in-storage processing architectures and benchmarks.

- Chapter 2 briefly presents the previously done research for different branches that emanates from our work.
- Chapter 3 discusses the architecture modeling and simulation of the SSD based in-storage architecture.
- Chapter presents the final conclusion and discussion from our study.
- Glossary gives an introduction on the background of the several underlying concepts used in our study.

Chapter 2

Related Work

Different solutions have been proposed for in-storage processing depending on the storage technology. However, most of these solutions discuss incorporating processing in memory and not in the permanent storage. A detailed discussion on different methodologies is mentioned here.

2.1 Processing Inside Memory

Gokhale *et al* [17] proposed moving processing inside the main memory. Their research was based on using SIMD processing. They integrated SIMD array so applications could benefit from SIMD arrays while others could benefit from the high-performance "carrier" (host). Researchers at Supercomputing Research Center (SRC) incorporated SIMD array closely into the architecture. This resulted in an architecture that could be use either as SIMD processor or an additional conventional memory [17].The authors further discussed that the concept of processing in memory has been there for several years now but with different technologies. However, only few commercial products are available.

2.2 Processing Inside Disks

Earlier attempts have been made to move processing inside the hard disks [18]. In this paper, the authors attempted to use the processors embedded in the individual storage devices for data-intensive applications. The chosen applications were data mining and multimedia databases. The general purpose microcontrollers are incorporated in the high-end commodity disk drives. Authors tried to leverage these for computing. The authors called this disk as Active Disk [18]. However, the proposed solution were quite complicated. Moreover, our methodology is different from theirs because we proposed incorporating low power processors in the SSDs. Our hypothesis resulted in hierarchical processing.

Another research for in-storage processing was conducted by Keeton *et al.* [19]. The authors primarily attempted to address the requirement for decision support systems and data warehousing workloads. The I/O capacity was increasing and with it the associated processing. To manage these high requirements, the authors presented "intelligent disks" (IDISKs) architecture. This architecture also used the low-cost embedded general-purpose processing. They offloaded the computations from expensive desktop processors. Their architecture could also scale out easily.

Acharya *et al.* integrated processing inside the disk drive. Their key idea was to offload bulk of the processing to the disk-resident processors [20]. They used stream-based programming for execution of the disklets safely and efficiently.

Mueller *et al.* used FPGA to accelerate the data processing [21]. They attached an external module for implementation of the system intelligence.

Many of the above mentioned techniques were not successful because the same benefits could be obtained using commodity servers. By spreading the data and partitioning it affectively across the servers efficient performance could be obtained.

2.3 Kinetic Drives

Furthermore, when we attempt to study Kinetic drives as a real hardware platform for our conceptual in-storage processing architecture, understanding of LevelDB was most important. Several key-value stores are being deployed for various tasks at different companies including Dynamo at Amazon [22], Redis at GitHub [23], and RocksDB at Facebook [24]. Dynamo is a fast NoSQL database. It provides single-digit millisecond latency [25]. Furthermore, RocksDB is an embedded database for key-value data. It is derived from LevelDB [26]. It is efficient for solid-state drives. However, our drives use

LevelDB and our research hinges upon understanding LevelDB.

All these systems store ordered {key, value} pairs. Since, the Kinetic drives were a brand new concept we did not find any new research paper on evaluation of these drives. In addition to this work Cao *et al* described managing large scale kinetic drives [16]. Other Object-based Storage Devices have been introduced to manage objects using keys [27] [28]. Several Peer-to-Peer systems have also been proposed [29] [30]. However, Kinetic drives encapsulate all these concepts in one unit.

Chapter 3

Architecture of Storage Processing Unit

3.1 Basics of Solid State Drives

Solid state drives are nonvolatile storage devices that store data persistently on solid-state flash memory. The major components in an SSD drive are:

- Flash Controller
- NAND Flash Package
- NAND Flash Bus
- Channels

A detailed figure is shown below 3.1. Flash controller is primarily responsible for translating the addresses that it receives from the host. In addition, controller also has a Flash Translation Layer (FTL) that performs many house keeping operations for the drive. It has a processor that could also be used for processing tasks instead of just house keeping operations. The major operations performed by SSD are garbage collection, data allocation, wear leveling, error correction etc. It acts as a medium between the host processing and the dies, where the data is actually stored.

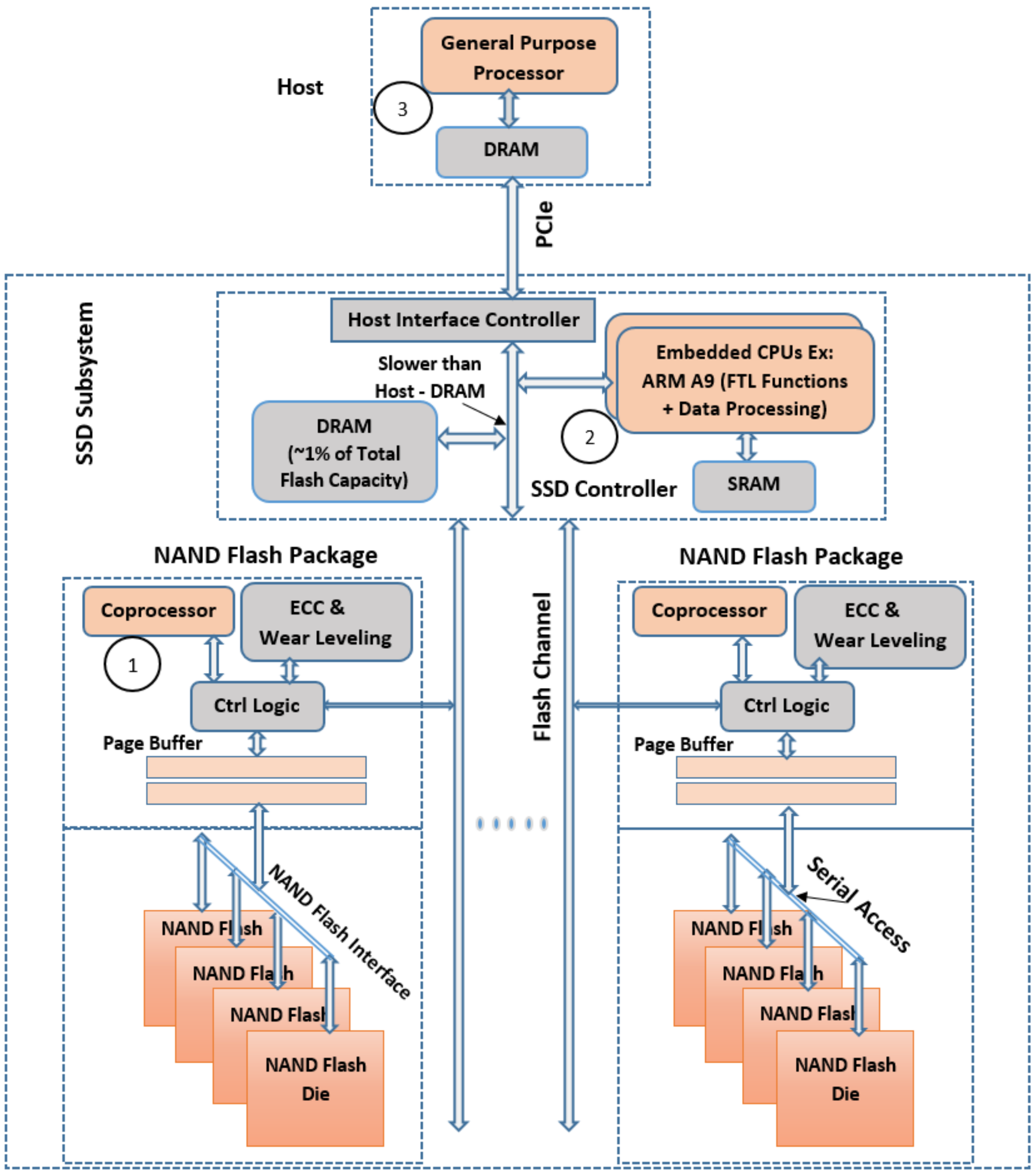


Figure 3.1: Comparison of Traditional vs New Model of Computation

Data is transfered from and to the NAND flash dies through the Flash Channel. It is a bus that helps in moving the data back and forth. Channels are connected to NAND Flash Packages. Each package has multiple dies. This Flash Package already has processing capability for wear leveling, error correction etc. NAND Flash bus connects the channel to these dies. In a pad-limited case, where the number of pins are fixed, manufacturing companies would be packing a fixed amount of silicon. This silicon is used to make gates for storing data. However, for a pad-limited case with fixed number of pins there would be excessive silicon that would go waste, if not utilized. Since, error correction and other house keeping actions could be performed inside a package, this excessive silicon could be used to make a low powered processor[31]. ITRP road map was used to devise the power specifications of this coprocessor.

The number of dies depend on the manufacturing companies' design. Number of dies and the total storage capacity of the drives are directly correlated. Each die has two or more planes. Each plane has multiple blocks. Each blocks consists of pages. Read or write operation takes place at page level. However, delete operation takes place at the block level.

3.2 Modeling with Cofluent

Storage Processing Unit is modeled using CoFluent Studio [32]. This system helps in modeling and simulation of complicated systems with graphical blocks. The user has the flexibility to define the properties of different blocks using internal features or C++ and the simulator generates SystemC code for it. The simulator has primarily three major building models: Time-Behavioral Model, Platform Modeling, and Architectural Model.

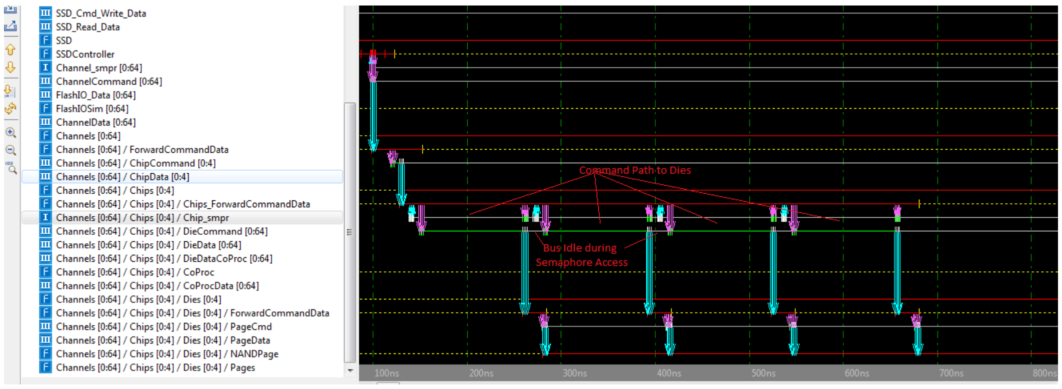

SSD_Cmd_Write_Data
SSD_Read_Data
SSD
SSDController
Channel_smpr [0:64]
ChannelCommand [0:64]
FlashIO_Data [0:64]
FlashIOSim [0:64]
ChannelData [0:64]
Channels [0:64]
Channels [0:64] / ForwardCommandData
Channels [0:64] / ChipCommand [0:4]
Channels [0:64] / ChipData [0:4]
Channels [0:64] / Chips [0:4]
Channels [0:64] / Chips [0:4] / Chips_ForwardCommandData
Channels [0:64] / Chips [0:4] / Chip_smpr
Channels [0:64] / Chips [0:4] / DieCommand [0:64]
Channels [0:64] / Chips [0:4] / DieData [0:64]
Channels [0:64] / Chips [0:4] / DieDataCoProc [0:64]
Channels [0:64] / Chips [0:4] / CoProc
Channels [0:64] / Chips [0:4] / CoProcData [0:64]
Channels [0:64] / Chips [0:4] / Dies [0:4]
Channels [0:64] / Chips [0:4] / Dies [0:4] / ForwardCommandData
Channels [0:64] / Chips [0:4] / Dies [0:4] / PageCmd
Channels [0:64] / Chips [0:4] / Dies [0:4] / PageData
Channels [0:64] / Chips [0:4] / Dies [0:4] / NANDPage
Channels [0:64] / Chips [0:4] / Dies [0:4] / Pages
Command Path to Dies
Bus Idle during Semaphore Access
100ns
200ns
300ns
400ns
500ns
600ns
700ns
800ns

Figure 3.2: Command Flow to Dies in 4 channel, 4 Dies/channel SSD

Any system is modeled using Time-Behavioral Model (TBM) first. In this step different functions and their behavior are defined. This behavior further represents the application-oriented viewpoint of the internal structure of the architecture (It is further discussed by Ashwin *et al.* [33]). There is flexibility to use several pre-defined functions such as events, buses, shared variables, etc to obtain the intended operation of the simulation. Basically, an algorithm is defined using these functional blocks to execute the desired operations. Other features that were really useful for this work were dynamic power, static power, memory, delay, cost etc. These parameters' value is defined based on different specification sheets of different technologies.

When these functions are defined, they should be able to communicate. The functions can communicate with each other through transactions. There are three main methods for functions to communicate: 1) A shared variable, 2) A synchronization or event relation, and 3) data transfer via message queue.

After verifying the behavior of the TBM, the next process is to create a Platform model. In this step the physical architecture of the system is defined. Platform model consists of modeling the real hardware such as processors, memory, SSD, etc. Attributes or the parameters of different blocks in this step could be taken from the specification sheets. These are the design parameters that help in deciding the performance of the simulation and thus impacts the final end result. In summary, functional blocks in the Platform model are created to be associated with the functional blocks defined in the Time-behavioral model.

Final piece of completing the simulation is Architecture model [34] [33]. In this step the components of the Time-behavioral model are mapped to the functional blocks of the Platform model. All the blocks in the platform model inherit the properties from the functional model. As an illustration, when an operation is assigned 1 cycle then this mapping will associate the CPU frequency of 1 GHz, which is defined in the platform model, with that operation.

With this modeling, buses or queues could also be simulated. This is an important feature of this tool. The tool allows to model the behavior of a bus including delay time, transfer time etc. The tool also helps in resolving dependencies and relations such as transfer time dependence and its relation with the amount of data transfer. CoFluent allows this by letting the user define USERDATASIZE.

3.3 Modeling and Simulation

In this section modeling of the SPU and the other relevant components of the system are discussed. First the Instruction Set Architecture (ISA) of the model is defined[1]. The parameter values of different components is also discussed. Modeling and simulation methodology is discussed in detail in this section. Moreover, the different specifications of the modeled system are given in Table 3.1.

Table 3.1: Specifications

Specifications	Range of Values
CPU Clock Frequency	2 GHz
Number of Cores in Host	4 - 8
CPU Gate Count per Core	200 M
CPU Dynamic/Static power	5.04 W/0.34 W
DRAM Dynamic/Leakage Power	0.44 W/0.09 W
PCIe Interface Speed	24 Gb/s
SSD Controller Clock Frequency	1 GHz
SSD Controller Cores	4 - 16
SSD Controller Dynamic/Static Power	156 mW/1.3 mW
Coprocessor Core Gate Count	1 k - 1 M
CoProcessor Clock Frequency	400 MHz
CoProcessor Cores	1 - 32
CoProcessor Core Dynamic/Static Power	3.12 uW/67 nW
NAND Flash Bus Speed/Bandwidth	2.5 ns per Byte
Number of Channels	4 - 32
Number of Dies	4

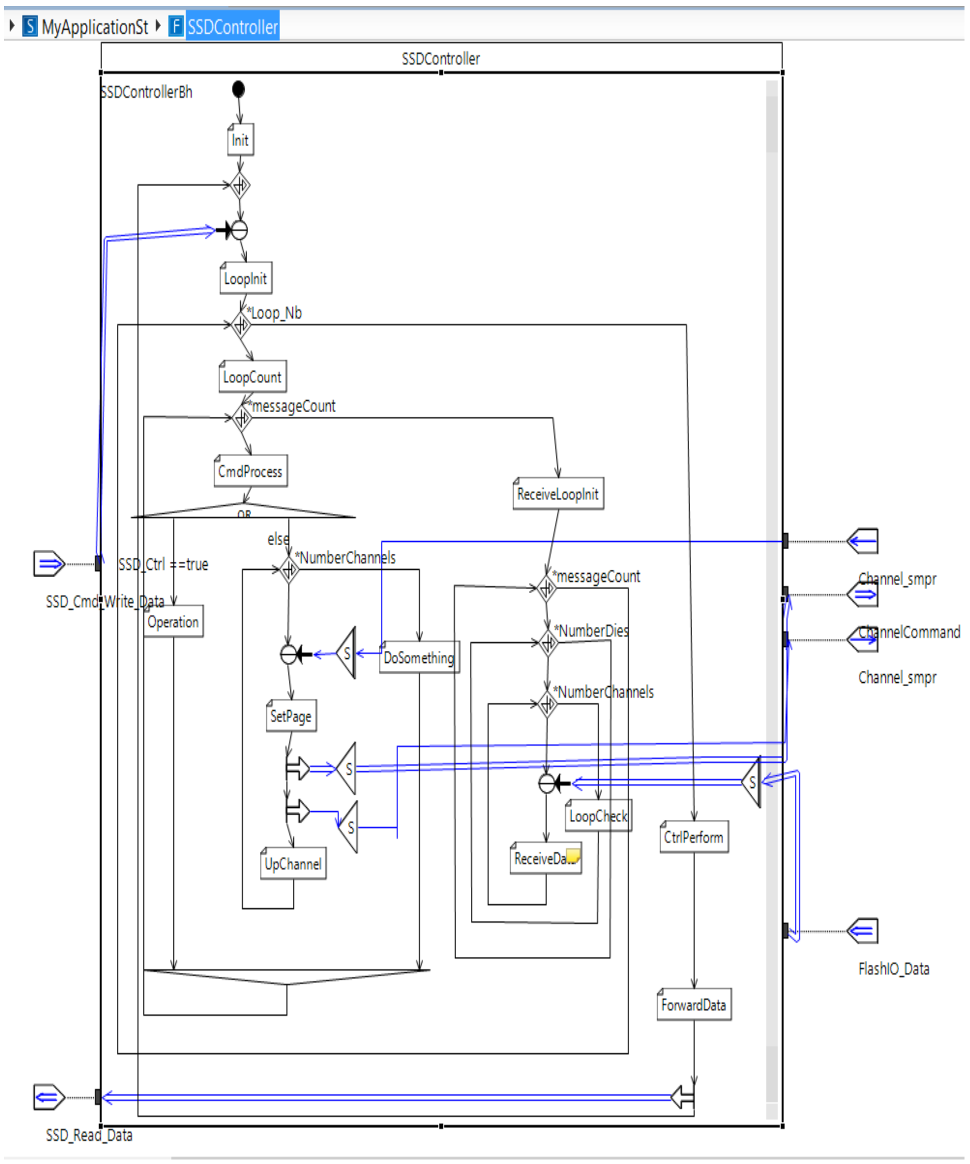


Figure 3.3: SSD Controller Function

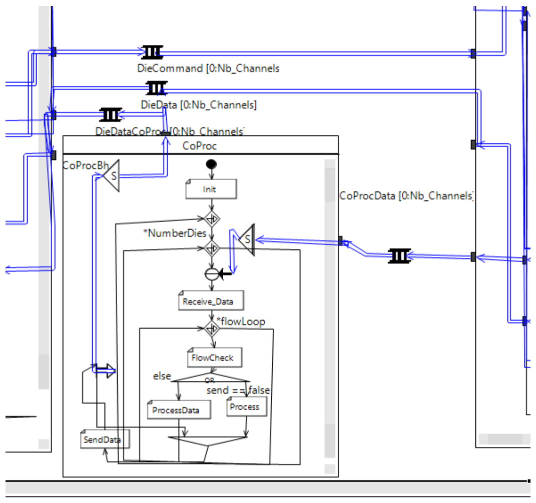


Figure 3.4: Coprocessor Function in TBM

3.3.1 Modeling Storage Processing Unit using CoFluent

The modeling of SSD is discussed in this section. It is particularly important because incorporating a processing inside the SSD is proposed. The NAND Flash dies, Channels, controller, and the coprocessor and other modeled components are described. Each channel is connected to multiple flash packages. Each Package has multiple dies. Coprocessor is modeled as a separate block because of its relevance to the design. It also brought flexibility to the design.

An SSD controller in the SPU performs these given operations: handling page read/write and coprocessor requests from the host system, it coordinates the output result of different coprocessors, and it acts as an interface between the host processor and the coprocessor. A detailed diagram is given Figure 3.5 and the specifications for the modeled system are given in Table 3.1. A further more detailed methodolody is given by Ashwin *et al.* [33].

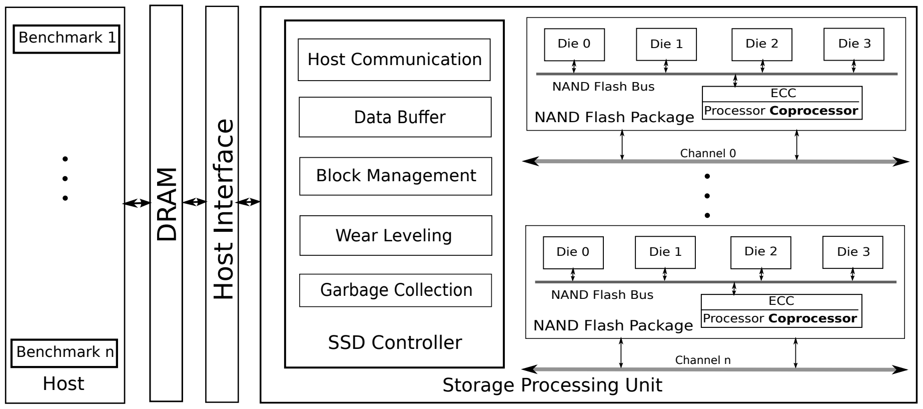


Figure 3.5: Complete System with Storage Processing Unit

Furthermore, commands are passed to the SSD by the Host processor. SSD deciphers that instruction and acts accordingly. It activates or deactivates the quad plane, it also initiates the read or write operation. Also, it instructs the coprocessor to be included or excluded from the data path. This single model for all the processing was a major improvement over the previously developed models by Peng *et al.* [6].

For data allocation, policy devised by Hu *et al* [35] is used. Parallel inside SSD is exploited in this priority order - Channel, Package, Die, and Plane. A detailed explanation of this process is given in the Algorithm 1, which is given in more detail by Ashwin *et al.*[33].

Furthermore, important sections of the models are shown in the Figures 3.3 and 3.4 below. These figures are a part of experiment that was jointly conducted by the students of University of Minnesota. These figures are shared by Ashwin *et al* and Minglani *et al* in the joint project [33]. These could also be found in the submitted thesis of Ashwin Nagarajan. More detailed explanation of the modeling is given by Ashwin *et al* [33].

When the controller detects that a read or write request has arrived and it can not satisfy that request with just one instance then the controller breaks it down to multiple smaller chunks of requests. This is what the functional blocks achieve as shown in Figure 3.3. The controller in the model has to send and receive the data. However, the modeling capability of CoFluent does not allow to do that. Therefore, in the model after every two requests controller switches to receive the data. The processing of the received

Algorithm 1 Pseudo-code for SSD controller Function Page Assignment

```
Input: Number of Channels, Number of Dies/Channel, Page Size, Data Size
Output: Commands to Flash Dies following Page Allocation Policy discussed in [35]
 1: Pages_Iteration = (Number of Channels × Number of Dies × Page Size)
 2: Total_Pages = Data Size / Pages_Iteration
 3: if (Total_Pages > 1)
 4:     Number_Iterations = Total_Pages → Number of iterations of read from all dies
 5: if (Data Size % Pages_Iteration != 0) → Residual data read separately at the end
 6:     Remainder = Data Size - (Pages_Iteration×Number_Iterations)
 7: Dies_Count = Remainder / (Page Size×Number of Channels)
 8: Dies_Remaining = Remainder % (Page Size×Number of Channels)
 9: for (i = 1 to Number_Iterations)
10:     Send → Read Page Size × Number of Dies for all Channels
11:     Move to receive path and return after every alternate Send
12: while (Dies_Count > 0 ) Receive path has similar logic
13:     Send → Read Page Size × (Dies_Count + Dies_Remaining)
14:     Dies_Remaining = Dies_Remaining - 1
```

data is modeled in the block "CrtlPerform" inside the SSD controller. In this block the output results of several coprocessors are received and processed. Depending on the mapping of the application and the architecture, this functional blocks behaves. It can also simply let the data pass to the Host processor if that is how the configuration is set for a particular application.

When a request or command from Host processor comes then it forwards that to the Flash Memory Controller which is located or modeled inside the NAND Flash package. Inside a Flash package, two planes can receive commands in parallel however, only one die can process the commmand at a time. Therefore, we modeled interleaving at the die level. This behavior is modeled using "Semaphores" which is provided by the CoFluent simulator. The time it takes to transfer a command is modeled with the help of NAND Flash bus bandwidth. Here the time taken to transfer the data depends on the bandwidth of the bus, number of write or read requests, data size for each request, and the bus speed. Whenever two blocks are communicating through the bus in that case one of the blocks acquire a semaphore and when it is acquired, the block sends the information across to the next block. Soon after the lock is released.

The processing performed by the coprocessor is done inside the NAND flash package.

Data flow is modeled such that data transfered from the dies goes to the buffer in the flash memory controller via NAND flash bus. Coprocessor reads the data from this buffer processes and then does the next step. The next step could be forwarding its output result to the SSD controller or could write the data back to the dies. The time for a coprocessor to process the data depends on the data size, page size, number of operations, type of operations, and speed of the coprocessor. This time is computed dynamically.

3.3.2 Instruction Set Architecture

This section discusses the Instruction Set Architecture of the model SPU. ISA was devised so that applications could be ported on to this architecture [33]. Different applications or algorithms were chosen and by using an ISA, implemented them on this architecture. Therefore, devising an ISA that could port applications on this model in sufficient depth, was extremely important. Common fields that make an instruction are command name, loops followed by number, and instruction with size of data. Syntax for these instructions is as shown below:

$< commandname > loops < number > instructions < datasize >$

In this format, command name can be one of the commands listed in Table 3.2. A list of all the possible command names is given in the table 3.2. Number in this command simply gives a count. As an illustration a loop instruction is given below.

$ssd\_nand\_loops\ start < loopcount >< loopidentier >< N/A >$

$ssd\_nand\_loops\ end - 1 < loopidentifier > -1$

This instruction has also been discussed in detail by Ashwin *et. al* [33]. Multiple loop instructions could be used together to create nested loop structure. A script is created using these instructions to create programs for different algorithms and applications.

3.3.3 Data Flow Diagrams

Data flow diagrams not only let the designer know about the flow of the data but also helps in validating the model itself. The SPU and Baseline models are validated using 1) different parameters and specifications, 2) data flow, 3) theoretical bounds, 4) literature study, 5) performance and energy trends (cite) (ashwin). In this section, we

Table 3.2: Instruction Set to Run Applications on SPU Model

Command	Description
ssd_nand_read	Reads data from SSD to Host
ssd_nand_write	Writes data from Host to SSD
ssd_nand_perform	Pseudo for other computations
host_dram_add	Add operation in Host
host_dram_mult	Multiply operation in Host
host_dram_perform	Pseudo for other computations
host_dram_write	Write data to DRAM
coproc_cmd_add	Coprocessor performs add operation
coproc_cmd_mult	Coprocessor performs mult operation
coproc_cmd_perform	Pseudo for other computations
ssd_ctrl_perform	SSD controller performs an operation
ssd_nand_loops	Denotes start of a loop
ssd_loops_end	Denotes end of loop
ssd_nand_end	Marks end of program

demonstrate the flow of data in the CoFluent modeling and simulation tool.

In the figure 3.2, the flow of data is shown between different functions. On the left hand side in the figure, different parameters could be seen for example, different instances of channels and dies. The blue and red arrows simply show the flow of the data between different blocks. This figure clearly shows the parallelism across different channels during a page read (cite)(ashwin). It appears that all the dies read in parallel however, there is slight delay in issuing the commands. However, only die transfers at one time. If two dies have to transfer then one dies completes the transfer before the next one starts.

3.3.4 Comparison between Baseline vs SPU

Comparisons of the simulations are made primarily between two configurations - Baseline and SPU. In the baseline configuration. In the baseline configuration, we do not include the coprocessor in the data path [1]. Due to this the configuration acts as the traditional system. In the SPU configuration, coprocessor is included in the data path, which leads to a model that is new and holds the potential to save a energy and give performance gains.

Overall, the simulation records the active times and the time when the devices are idle. This data helps in computing active and idle power for the systems. Whenever a component is active, the simulator would compute its active power and would add it to the overall power in the Post-simulation analysis. This gives the total idle and active power of the system.

3.4 Applications

The chosen applications are diverse in nature and the selection is made after considering the initially proposed ”dwarfs” by Asanovic *et al* [36]. These section is prepared from the publication by Minglani *et al* [1]. These applications differ from each other based on the computation and the pattern of the data movement.

The applications chosen are:

- Sparse BLAS - SpMV and SpMM
- K-Means and kNN
- Graph 500
- Canneal

3.4.1 Sparse BLAS - SpMV and SpMM

The major operations performed for Sparse matrix-vector (SpMV) and Sparse matrix-matrix multiplications (SpMM) are $y \leftarrow y + Ax$ (AXPY) and $y \leftarrow y + AB$. The data formats used are Compressed Storage Row (CSR) and Compressed Storage Column (CSC). Sparse matrix-vector (SpMV) and Sparse matrix-matrix multiplications (SpMM) are considered to be one of the most important iterative-method based computational primitives. In fact, large scale linear algebra problems constitute an estimated 70% of computing cycles in the HPC ecosystem[REF]. Also, dealing with sparse matrix computations offer several challenges to parallel architecture community: memory intensity, indirect memory references, irregular memory accesses for vector and short row lengths. The ratio of computation to communication, inter-processor communication are the other important factors. These features highlight our interest and motivation in mapping sparse BLAS kernels on the SPU architecture.

In CSC, the pages in dies contain the column elements and in the CSR format, the pages in dies contain the row elements of a matrix. Several different data formats have been thouroughly discussed in Several challenges have been discussed. Additionally, they are limited by the memory bandwidth of the system.

We consider Level 2 and Level 3 Sparse BLAS operations which fall into the following two computation kernels:

Matrix-Vector Multiply (SPMV) : y = A * x + y and Matrix-Matrix Multiply(SPMM) : C = A * B + C

where A is a sparse matrix, x and y are dense vectors, B and C could be sparse or dense matrices. Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) storage formats are considered. Matrices and their properties, such as the average number of non-zeros (nnz) in a row or column, used in our simulations.

3.4.2 Graph500-BFS

Graphs are tremendously capable of modeling entities and their interactions with each other in the real-world scenario. Graph traversal algorithms are inherently memory and compute intensive. This application performs Breadth-first search algorithm (BFS) [37]. Graph500 uses BFS for evaluating parallel and distributed architectures because synchronization is a major challenge for efficient parallel implementation [38]. Breadth-first search algorithm (BFS) visits and analyses all the nodes in a graph. A breadth-first tree is constructed by starting with a randomly chosen node. When a new node is discovered then the already discovered node (u) and the new node (v) are added to the tree. The node 'u' becomes the parent of node 'v'. A node can have at most one parent [37]. Graph500 uses BFS for evaluating parallel and distributed architectures and synchronization is a major challenge for efficient parallel implementation [38].

And its intrinsic character is that graph's data is large , lacking locality, cause unbalanced computation and communication workloads. BFS could be benefited from the inherent parallelism available in this architecture. Random memory access flash provides faster random access. We evaluated the energy and performance of the Storage Processing Unit with this primitive.

3.4.3 k-Nearest Neighbor (kNN) and k-means

Both the applications are data mining algorithms and their major operation is to find distance between the input and every other data point [39] [40]. However, the main difference is that k-means is based on the unsupervised learning and kNN is a supervised classification method where the input sample is classified into one of the already defined classes [41].

k-Nearest Neighbor (kNN)

This is classification algorithm doesn't require prior knowledge of the distribution of the data. The training phase of the algorithm involves storing the data points and its labels [41]. Every data point in the training set contains a set of vectors and a label associated with it. Distance between the input and every point in the dataset is computed. The 'k' smallest distance measures corresponds to the elements that influence the classification of the unknown sample. Selecting a suitable neighbourhood value 'k' and the distance metric applied determines the classification performance. kNN might appear to be advantageous when the training data is large as there is minimal cost for the training phase. However, it might suffer from costs in terms of computation and run time if not optimized suitably.

k-means

k-means is a data mining algorithm [39] and is used for partitioning N particles into k clusters where ($k < N$) such that particles or objects in a cluster are as similar to each other as possible [40] [42]. Clustering is a dominant data analysis method in several fields such as Bioinformatics, pattern recognition etc [40]. One of the reasons for the popularity of k-means is its simplicity and efficiency to perform clustering. Furthermore, k-means is a data mining application just as kNN but the difference is that k-means is based on the unsupervised learning as the data points have no external classification.

In every iteration, the data points are divided into 'k' clusters and this process is repeated until the number of data points moving to different data sets is sufficiently small. Whereas, kNN is a supervised classification method where the input sample is classified into one of the already defined classes [41]. The data can be scalars or multidimensional vectors in the feature space. Every data point in the training set contains a set of vectors and a label associated with it. KNN might appear to be advantageous when the training data is large as there is minimal cost for the training phase. However , it might suffer from costs in terms of computation and run time if not optimized suitably.

3.4.4 Canneal

Canneal is a part of the PARSEC application suite which performs simulated annealing for optimizing chip routing costs [43]. Several key issues for these applications, along with the proposed optimization, are listed in Table 3.3.

This kernel is a part of PARSEC application suite which performs simulated annealing for optimizing chip routing costs [43]. The input to this kernel is a circuit netlist where coordinates for every element are specified. At every iteration, two elements are pseudo-randomly picked and the routing cost is computed by swapping the coordinates of the elements. The probability of accepting this swap depends on the temperature parameter. Every time a swap is accepted, the coordinates of elements and the new routing cost are updated. The new values are used in making decisions for the subsequent iterations. Choosing the right annealing schedule for the temperature and the iteration period determines the quality of the solution. This process is generally slow due to its sequential nature and high writing costs.

Table 3.3: Limitations of different applications and proposed optimizations

Primitive	Dwarves	Domain	Limitations	Characteristics	Optimizations
Sparse BLAS	Sparse Linear Algebra	Linar Algebra	Irregular memory access, Low Computation Resource Utilization	Memory Bound	Data Layout/ SSD Cores
Graph 500	Graph Traversal	Graph Algorithms	Communication and Synchronization	Memory Bound	Data Layout
k-means	Dense Linear Algebra	Data Mining	High I/O time, Limited memory for Large Data set	Compute Bound	SPU Cores
kNN	Dense Linear Algebra	Data Mining	Run time increases with large data sets	Compute Bound	SPU Cores/ SPU Communication
Canneal	Structured Grid	Optimization Algorithms	Large number of Updates, Irregular Memory Access	Memory Bound	Caching

3.5 Execution Flow of the Architecture

An example of the execution of Sparse BLAS (SpMV) matrix-to-vector dot product ($y \leftarrow y+Ax$) on the SPU described, in the previous section, can give a better understanding of this architecture [1]. The computations depend on the data format of the stored matrix. As shown in Figure 3.6, for Compressed Column (CSC) format, the coprocessors start processing on their respective columns of the matrix after receiving an instruction from the host. The SSD controller synchronizes the tasks for all the coprocessors and executes the addition operation to complete the AXPY operation.

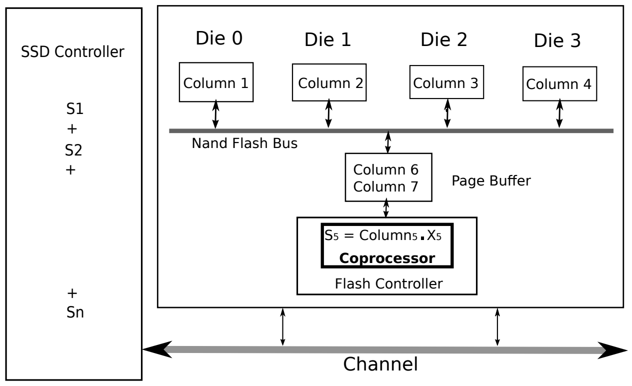


Figure 3.6: Simplified Storage Processing Unit Example with One Flash Package and SSD controller

3.6 Impact of Salient Parameters on the System Performance

In this section, we discuss the design parameters that could potentially affect the performance of the applications and then provide a detailed analysis of performance results. We do this by identifying the application properties, discussing the design trade-offs, and finally providing the optimum architectural configuration.

3.6.1 Different data layouts and their ramifications on the execution flow and energy consumption

Data access patterns and data movement across the system are dependent on the data layout. The order of operations varies with the different data layouts. Therefore, understanding the data layouts is a key to efficient memory management and obtaining performance gains. In the SpMV kernel, the CSR format requires a dot product of the row elements (stored in the dies) and the dense vector. The dense vector could be stored in the SSD controller's memory (CSR-a) or in the dies (CSR-b). In CSR-a, rows are read and the dense vector is fetched from the SSD controller's memory by each coprocessor. Each output element belongs to the final output of the resultant vector.

The CSR-b layout replicates vector elements for each coprocessor to reduce the data movement. Thus, more pages have to be read to perform the dot product. Figure 3.7 confirms that CSR-b consumes more energy than CSR-a due to more reads. The different matrices in these figures are taken from the University of Florida Matrix Collection [44]. Figure 3.8 shows the speedups of the SPU-based system (configured with Channels=32, Dies=4, Page Size=4KB, and Coprocessor CPI=1) over the baseline system for different vector layouts.

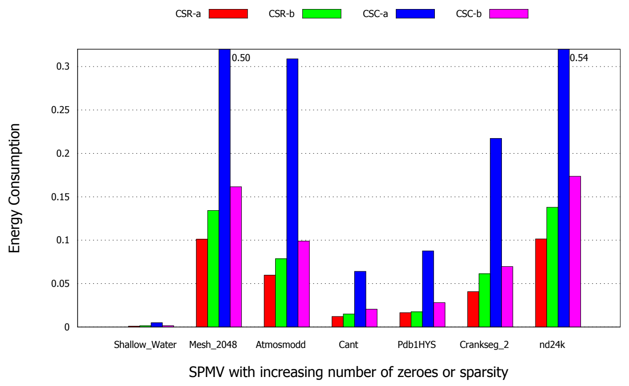


Figure 3.7: Energy Consumption of Sparse Matrix-Vector Kernel with Different Data Layouts for Vector

For the CSC data format, the partial results are spread across coprocessors (Section 3.5). These partial results are to be moved either to the Host (CSC-a) or SSD controller (CSC-b) to obtain the final results. Due to this data movement, the CSC format is 1.5

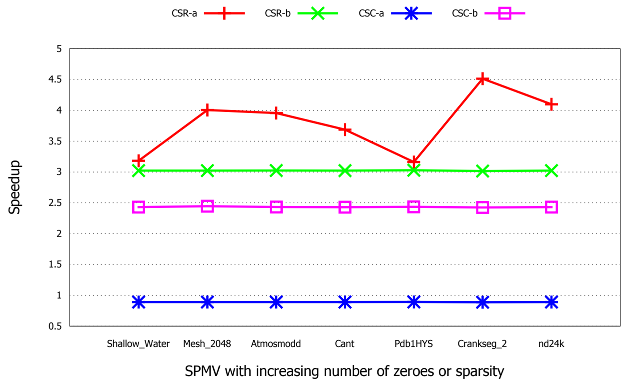


Figure 3.8: Speedup Obtained by Sparse Matrix-Vector Application with Different Layout for Vector

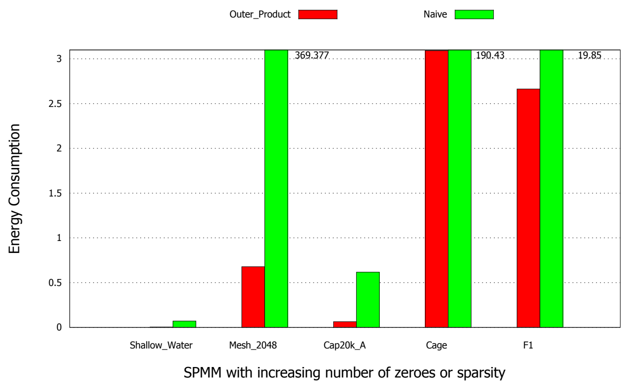


Figure 3.9: Energy Consumption by Sparse Matrix-Matrix Application with Different Layout for Matrix

– 5X less energy efficient than CSR (Figure 3.7).

The other kernel taken from the Sparse BLAS library is SpMM and there are two configurations possible for it. Case 1 is where columns of matrix A and rows of matrix B are stored together in the same die [37]. In Case 2 (naive implementation), one of the two matrices is stored in the dies in row-major format and the columns of the other matrix can be transferred to the coprocessor. The results for SpMM are presented in Figure 3.9 and it can be concluded that the energy consumption is higher in the second case.

In Figure 3.10, it can be clearly seen that when the host is used to perform the

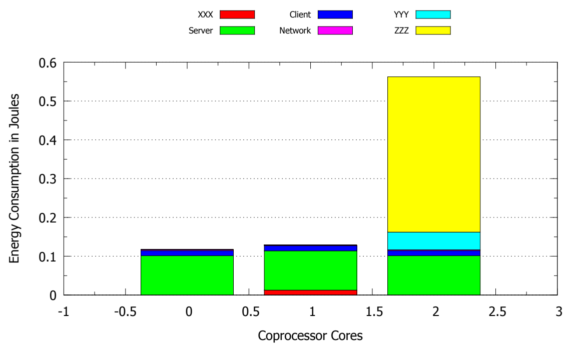


Figure 3.10: Component-wise Split of Total Energy Consumption for Different Configurations for Sparse Matrix-Vector Operations in the SPU Configuration

AXPY operation in the CSC-a configuration then the power consumption is higher than in CSR-a. In the CSR-a and CSR-b layouts, low powered processors in the flash controllers are used alongside the embedded processors inside the SSD controller. Therefore, the host does not have to be used and the total power consumption comes down significantly. Furthermore, CSR-a achieves better energy efficiency than CSR-b. This is because CSR-b involves fetching more pages, therefore using more energy, than the CSR-a configuration. Lastly, it can be observed that by employing the coprocessors with embedded processors the power consumption for PCIe, data transmission power, could be brought down significantly.

Graph500-BFS also demonstrates similar gains in performance and reductions in energy. It employs the breath-first search algorithm and requires the vertex and parent lists to be regularly updated. The limited local memory of the coprocessor is incapable of storing these data structures. Therefore, two alternatives are available for storing the data structures: 1) in the dies or 2) in the DRAM of the SSD controller. If the data structures are stored in the dies, then the regular updates will reduce the lifetime of the flash. Therefore, the natural choice is to store the data structures in the DRAM of the SSD. This process makes the capacity of the DRAM the bottleneck for the application. The maximum speedup and energy efficiency improvement obtained for this application is included in Table 3.6.

So far, the architectural tradeoffs of exploiting the computational hierarchy (host, SSD controller processor, and coprocessor) have been presented. It can be concluded based on the above discussion that data movement is directly related to the data layout for an application. Efficient data layouts will result in lower energy consumption for an application.

3.6.2 Change of application behavior with number of coprocessor cores

The usage of a multi-core coprocessor depends on the behavior of the application. The applications are memory-bound if the number of cycles to transfer the data from the dies to the coprocessor (T_{bw}) is greater than the number of cycles to compute (T_{coproc}) the data of a page. Conversely, for compute-bound applications (T_{coproc}) ¿ (T_{bw}). The cycles needed to transfer the page depend directly on the bandwidth of the NAND flash bus which is an 8- to 16-bit wide bidirectional bus. Applications such as kNN, k-means, and SpMM fall under the compute-bound category and benefit significantly from additional cores in the coprocessor.

Additional cores benefit k-means because significant parallelism is available when finding the distance index of the data points from the cluster points [40]. Also, with more cores the rate of execution increases which results in more energy savings as shown in the Table 3.6 and Figure 3.11. The data is given for 100,000 data points with 8 clusters (k_means_8) or 64 clusters (k_means_64) for the SPU-based system (configured with Channels=32, Dies=4, Page Size=4KB, and Coprocessor CPI=1).

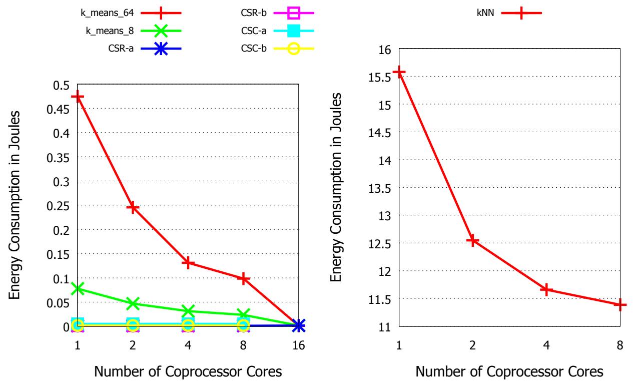


Figure 3.11: Energy Consumption for kNN and k-means with Varying Number of Coprocessor Cores

In SpMM, increasing the core count of the coprocessors does not provide significant gains. This is because the data is moved to the host or SSD controller for completion of the outer product in matrix-matrix multiplication [44]. Consequently, most of the energy consumption in this process is because of the data movement.

For the kNN algorithm, coprocessors compute the distance values and send them to the SSD controller for sorting. With more coprocessor cores, the energy consumption decreases as shown in Figure 3.11. However, the SSD controller can become a bottleneck if its processing rate is slower than the data arrival rate. To mitigate this bottleneck, an optimization is proposed in Section 3.6.4. The input dataset for the kNN algorithm is taken from the OpenStreetMap project [45]. The 2-dimensional dataset contains road network information for 50 states and occupies 6.6 GB.

An important observation is that the energy savings level off with a sufficient increase in the number of coprocessor cores. The primary reason is that with enough coprocessor cores T_{coproc} becomes less than T_{bw} due to less work done by each core. Therefore, the behavior of compute-bound application changes to memory-bound depending on the bandwidth of the bus to deliver the data to the coprocessor. The general rule is that $T_{coproc} - T_{bw}$ should be close to zero to obtain the maximum energy efficiency. The minimum difference ensures that the bus and the coprocessors are busy and not waiting. An increase in either wait time increases the energy inefficiency.

3.6.3 Synchronization at the SSD controller

Significant gains are achieved when the number of clock cycles taken by the SSD controller to process is limited to the maximum of (T_{coproc}, T_{bw}), as this ensures pipelined operation between the coprocessor and SSD controller.

Increasing the number of SSD controller cores to exploit parallelism can help in maintaining the pipelined processing. For example, in the SpMM kernel, the SSD controller needs to add $nnz \cdot nnz \cdot Nb\_Channels$ data elements where nnz is the number of non-zero elements and $Nb\_Channels$ is the number of channels. With an average of 40 non-zero data elements in each row, 1KB page size, and 32 channels in the SPU, the order of clock cycles would be $T_{ssdcontroller}$ ¿ T_{coproc} ¿ T_{bw}. This situation will create a bottleneck. Here, it could be deduced that the required number of SSD controller cores depends on the ratio of nnz to rows in a page. Hence, increasing the number of cores to four would pipeline the operation.

The SSD controller synchronizes the operations of the coprocessors. In the BFS application, coprocessors find new nodes and form a breadth-first tree. Since multiple coprocessors could try to add the same node to their sub-tree, the SSD controller ensures that redundant updates are avoided and only one valid copy of the parent list is maintained.

For the kNN application, once the distance values are computed for all the data points in the training set, the SSD controller sorts and picks the 'k' smallest values. When the number of elements to be sorted increases, the process becomes compute-bound and depends mainly on the sorting time. However, with a sufficiently large number of SSD controller cores, the time taken to merge the different sorted sublists will also be high. This will reduce the energy efficiency of the kNN application.

These applications indicate that it is important to map computations carefully between coprocessor and controller. Furthermore, performance and energy efficiency are determined by the minimum of ($T_{ssdcontroller}$, T_{coproc}, T_{bw}).

3.6.4 Implications of communication between flash packages

For a memory bound application, when $T_{coproc} << T_{bw}$, scaling up the number of coprocessor cores would be energy inefficient because the idle time of the coprocessor would increase. Therefore, sharing previously fetched pages, with fixed NAND flash bus bandwidth, could be efficient. Sharing through the Flash Translation layer (FTL) could make it into a bottleneck. Hence, a light-weight network amongst coprocessors could efficiently share the pages. A detailed analysis of this light-weight network is a part of the future work, but it should not cause excessive writes for the dies.

The above mentioned concept could benefit the kNN application. These distance measures are sorted to obtain 'k' smallest values. Once 'k' values are produced then every other coprocessor sends them to its neighbouring coprocessor. Then the receiving coprocessors sort '2k' elements to produce 'k' sorted values. This process is continued and in 'log k' steps the final 'k' smallest values for all the coprocessors are found.

It could be concluded that by sharing the fetched page, significant energy efficiency could be obtained. However, increasing the number of coprocessor cores beyond a limit, with page sharing enabled, would result in lower energy savings as shown in Table 3.4.

Table 3.4: Energy Consumption for SPU-based System with Communication between the Flash Packages for kNN

Coprocessor Cores	Energy in Joules
2	8.97
4	7.35
8	7.16
16	6.99
32	7.34

3.6.5 Dependence on the limited NAND flash bus bandwidth

As discussed earlier, higher gains are obtained by making the difference, $T_{coproc} - T_{bw}$, smaller. T_{coproc} depends on the computation available in the applications and T_{bw} depends on the NAND flash bus bandwidth. The NAND flash bus is an 8- or 16-bit wide bidirectional bus which transfers the data between the flash controller and the dies as shown in the Figure 3.5. Therefore, the coprocessor depends heavily on the speed of this bus to deliver the data. This bus could be a bottleneck for non-data-intensive applications and when $T_{coproc} << T_{bw}$. The k-means application, being a highly data intensive application, becomes memory-bound at 512 cores for the coprocessor. We found that all applications become memory-bound irrespective of their initial behavioral pattern. As shown in Table 3.5, with $nnz = 250$ in a row, the application will act as compute-bound and energy efficiency will increase with the increase in the bandwidth.

Table 3.5: Energy Consumption in Joules for SPU-based System with Varying NAND Flash Bandwidth for SpMV

BW (ns per byte)	nnz=250	nnz=4
2.5	0.101	0.10333
1.67	0.072	0.10326
1.25	0.057	0.10323
1	0.049	0.10320

3.6.6 Minimal energy change with increased number of channels and page size

An increase in the number of channels increases the throughput, and it leads to an increase in the number of coprocessors in the system. Therefore, the time to complete the processing reduces. However, more channels consume more energy as well. These two aspects offset each other and we observed minimal energy drop with the increase in the number of channels.

The impact of page size on applications depends on their behavior. For data intensive applications, both T_{coproc} and T_{bw} will increase linearly with the page size. Therefore, increasing the page size did not render any significant performance improvements or energy savings.

3.6.7 Reduced lifetime of flash with excessive writes

SSDs have limited write cycles. Therefore, the coprocessor and SSD controller's processor attempts to utilize its own local memory instead of initiating writes for the dies. Moreover, Canneal is not a good application for the SPU architecture. It requires a high number of writes for updating the routing cost. Since the coprocessor's local memory has limited capacity, updates have to be written to the dies. These excessive writes are detrimental to the lifetime of the SSD.

3.6.8 Validation of the Simulation Model

For validation of our model, we compared our experimental results with the results obtained by Raghuwanshi *et al* [46] through curve fitting. The algorithm implemented for k-means by Raghuwanshi *et al* is similar to the algorithm implemented by us. We did regression analysis on the execution time for different numbers of records obtained by Raghuwanshi *et al* and found its $R^2 = 0.999104$ with linear curve fitting. For our data with different numbers of elements we found $R^2 = 1$. These verification results can be seen in Figure 3.12. Since, both the implemented algorithms are very similar, we believe that our model is validated based on the closeness of the R^2 values for both.

In addition, our confidence in our impressive experimental results is bolstered by the fact that Cho *et al* [47] adopted a similar approach and they also achieved performance

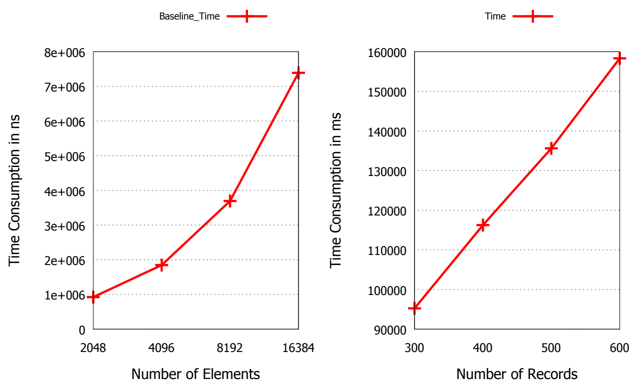


Figure 3.12: Verification of Simulation Model with k-means Application

gains.

3.7 Conclusion

To overcome bandwidth constraints and high energy consumption, we took processing closer to the data to reduce the data movement. Thereafter, we discussed several design parameters that affect the performance and energy use of the applications executed on the SPU-based system and baseline system. Table 3.6 presents some of the best energy savings and performance gains for different applications using the SPU-based system over the baseline system. The results are obtained by executing different scripts in the architectural model generated using the CoFluent simulation and modeling tool. The primary reason for the difference in performance is the behavior of the applications and data movement involved to complete the execution of the application. It is observed that compute-bound applications benefit the most from the SPU-based system. However, memory-bound applications can be optimized and significant performance gains can be extracted. These optimizations are enabled by understanding the trade-offs between the NAND flash bus bandwidth, time to process in the coprocessor, and time it takes to move the data in the baseline configuration. In conclusion, we were successfully able to demonstrate that the hierarchical processing of the Storage Processing Unit (SPU) has the potential to bring positive changes to the storage industry.

Table 3.6: Best Performances for SPU-based System over Baseline System for All Applications

Applications	Energy Savings	Speedups
K-means	423.98	66.17
SpMV Cranskeg_2	39.73	-
SpMV Mesh 2048	-	4.00
SpMM Shallow Water	11.85	6.57
Graph 500	44.34	5.84
kNN	11.25	10.12

Chapter 4

Application Level Performance Evaluation for In-storage Processing Architectures

4.1 Kinetic Platform — Motivation and Architecture

Kinetic drives are the latest class of hard drives from Seagate and have a similar form factor to a conventional 3.5” drive. However, Kinetic drives are fundamentally different from traditional drives in two ways: Ethernet connectivity and an internal processor that converts the storage drive to a key-value store [48] [14].

Each drive has two Ethernet ports (1 Gbps each) to transfer data to the clients or other drives using TCP/IP over Ethernet. The ports are for fault tolerance, and only one should be used at a time. Newer models (2nd generation) are expected to have two 2.5 Gbps ports. However, after inquiring, these newer models are not available for the general public, so we cannot test them. Furthermore, Kinetic drives do not use the SCSI interface and instead have a key-value interface. Applications interact with the drives directly using this interface and a standard Ethernet connection. IP addresses are assigned to the drives by a DHCP server [48]. The use of Ethernet allows the storage system to be expanded without the need to install a separate Storage Area Network. Figure 4.1 is a simplified comparison of the hardware and software stack of a traditional storage system with that of a Kinetic system.

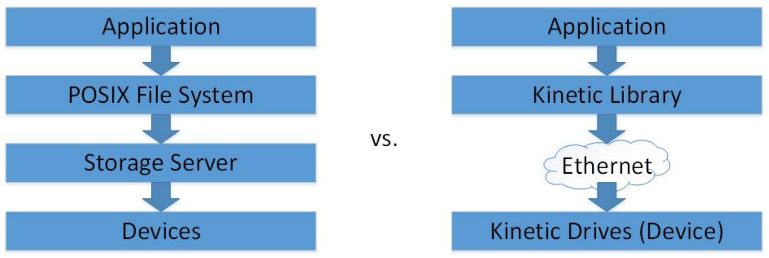


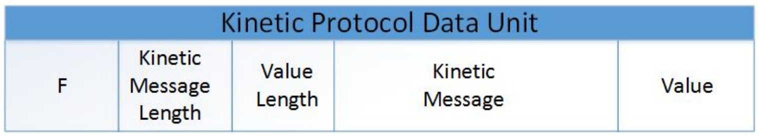
Figure 4.1: Software stack of server with traditional drives vs. Kinetic drives.

The second major benefit of using these drives is that they encapsulate the concept of in-storage processing [1] by running a LevelDB-based key-value store in the drives. There is a single-core System-on-Chip (SoC) processor inside the drives that runs a custom LevelDB and manages the data on the disk itself. This can reduce the software I/O complexity for applications and simplify the storage rack by eliminating the need for separate storage controllers [49].

Table 4.1: Kinetic Drive Limits

Device Feature	Limit
max Key Size	4096 Bytes
max Value Size	1 MB
max Version Size	2048
max Tag Size	128
max Connections	100
max Outstanding Read Requests	30
max Outstanding Write Requests	20
max Message Size	1 MB
max Key Range Count	200
max Identity Count	1000
max Pin Size	200

Seagate provides a software platform to help write programs for these drives. This



Kinetic Protocol Data Unit				
F	Kinetic Message Length	Value Length	Kinetic Message	Value

Figure 4.2: Message protocol for Kinetic drive.

Kinetic API comes with libraries for several popular programming languages, a simulator, installation instructions, some test utilities, and some basic test code [11]. The simulator allows the test and custom programs to run without the actual drives (but we run all our experiments on actual drives). The applications or programs are created using the functions and protocols defined in the API library. The protocols are responsible for the actual connection and communication with the Kinetic drives via messages (Figure 4.2). Moreover, the protocol is responsible for transferring data over TCP-based network. In this protocol Key is used for identifying a certain value [50]. Each message is called a "Kinetic Protocol Data Unit (Kinetic PDU) [50]. IP address of the drive is also used to identify the correct drive. This protocol can support limiting the operations for a user depending on the access rights. A kinetic message consists of a protocol buffer message, metadata, key-value metadata, and the value. In addition, value is not encoded with the message [50]. A message encodes a single command, HMAC of the byte representation. Here each command has type of command, a body which mentions operation-specific information, and status [50]. The Linux Foundation now maintains this Kinetic API [11].

The drives have a built-in capability to support multiple threads and sessions. While the Kinetic API abstracts away most of the internal details of the drives, there are set limits to certain parameters. These limits are listed in Table 4.1. Table 4.2 lists the main API commands used to interact with the key-value store on the drive. In addition, the API allows the programmer to perform write synchronization in several different modes as described in Table 4.3. These modes allow the programmer to force immediate synchronization or allow the drive to control the write operations.

Moreover, the Kinetic drives come with unique features, such as P2P transfer, which set these drives apart from regular hard drives. With P2P transfer, one drive can directly transfer its data to another drive via Ethernet; hence, it could be utilized for direct data replication. The process initiates when a client issues a ”P2P Put” or ”P2P Get” command to a particular Kinetic drive. This drive takes control of the process and automatically (without the client’s involvement) performs ”Put” (write) or ”Get” (read) operations to/from the target drive using Ethernet. After this process completes, both the drives have identical copies of those key-value pairs. These instructions avoid several stack layers typically required to transfer data and can drastically reduce overall system I/O.

In the following sections, we describe how we use this Kinetic API to evaluate the performance trade-offs, limits, and behavior of these sample drives.

Table 4.2: Commands and Functions of Kinetic Drives

Command	Function
put	Writes the specified key-value (KV) pair to the persistent store
get	Reads a KV pair
delete	Deletes the KV pair
getNext	Gets the next key that is after the specified key in the sequence
getPrevious	Gets the previous entry associated with a specified key in the sequence
getKeyRange	Gets a list of keys in the sequence based on the given key range
getMetadata	Get an entry’s metadata for a specified key
secureerase	Securely erases all the user data, configurations, and setup information on the drive
instanterase	Erase all data in database for the drive; this is faster than secureerase
setSecurity	Set the access control list for the Kinetic drive
P2P Get	Kinetic drive reads directly from another Kinetic drive
P2P Put	Kinetic drive writes directly to another Kinetic drive

4.2 Configuration

In this section, we cover the three system configurations we employ to test and evaluate the Kinetic drives and compare them with LevelDB-based servers.

To benefit from the commands described in Table 4.2, we have to leverage the Kinetic API. This API is installed in the client and is used to send commands to different Kinetic drives, which logically act as tiny independent servers as shown in Figure 4.3. The

Table 4.3: Different Write Modes of Kinetic Drives

Kinetic Write Mode	Description
WriteThrough	The request is made persistent before returning; this does not affect any other pending operations
WriteBack	The requests are made persistent when the drive chooses or when a FLUSH is sent to the drive
Flush	All the write requests that are pending and are not written yet will be made persistent to the disk

Kinetic API is designed for several programming platforms including C, C++, Java, and Python. Our testing primarily uses the C library of the API.

The client machine is a Dell R420 running Ubuntu Server 14.04 LTS 64-bit (kernel 3.13) on two quad-core Intel Xeon E5-2407 CPUs @ 2.20 GHz and 12 GB of RAM. This machine uses a 10 Gbps Ethernet PCIe network card to connect to a Kinetic disk enclosure containing four Kinetic sample drives. Internally, this enclosure contains redundant Ethernet switches with 1 Gbps ports for the Kinetic drives and a 10 Gbps uplink port. The Kinetic enclosure also has a separate Ethernet connection to allow access to a web interface that allows basic management of each drive (power cycle the drive, see its IP address, etc). The drives are model ST4000NK001, and we upgraded the firmware to version 03.00.04 using one of the included utility programs. These disks are 4 TB, 5900 RPM, and have 64 MB of disk cache plus another 512 MB of on-board RAM to run the modified versions of Linux and LevelDB on the added Marvell 370 SoC [15].

To compare the Kinetic drives with LevelDB-based servers, we set up two other systems: Server-SATA and Server-SAS. Server-SATA has the same specifications as the Kinetic client system described above plus has LevelDB installed and two additional traditional SATA drives, separate from the OS drive, connected directly to the motherboard via SATA. These two extra SATA drives, Seagate model ST4000VN000, are chosen because they are similar to the Kinetic drives, i.e., 4 TB, 5900 RPM, and 64 MB of disk cache.

Server-SAS also has LevelDB installed but is a more powerful and modern system. This machine runs Ubuntu Server 14.04 LTS 64-bit (kernel 4.1) on two six-core Intel

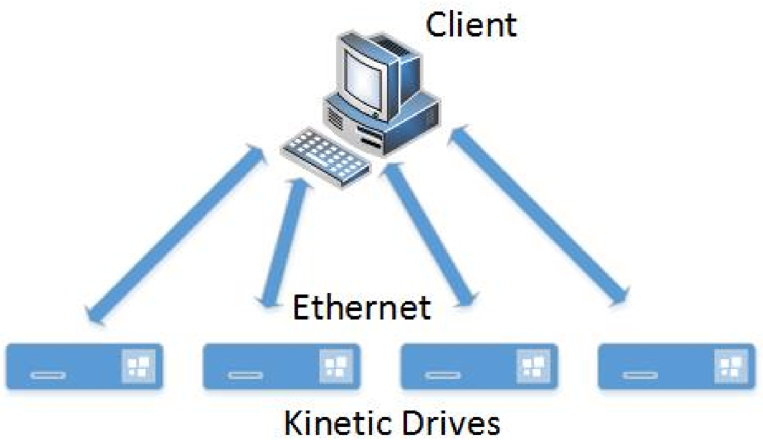


Figure 4.3: Logical view with Kinetic drives as independent servers.

Xeon E5-2620 v3 CPUs @ 2.40GHz and 64 GB of RAM. There is a PCIe LSI MegaRAID SAS-3 3008 adapter connected to an external Dell MD1400 disk enclosure containing 11 Seagate model ST6000NM0034 Enterprise Capacity SAS drives. Each drive is 6 TB, 7200 RPM, and has 128 MB of disk cache.

For both Server-SATA and Server-SAS, LevelDB is installed and db_bench.cc (or a slightly modified version) is run on the various drive configurations from the same machine, i.e., the client benchmark is run against a local server. For tests using two or more drives, Linux software RAID-0 is configured with default *mdadm* settings offered by the GNOME Disk Utility. The target location is then formatted with default ext4 settings and mounted for use. A detailed description of how Server-SATA and the Kinetic client machine are used for YCSB testing is included in Section VIII.

4.3 Methodology and Results for Kinetic Drives

This section details a variety of our tests executed on Kinetic drives. Performance results comparing Kinetic drives with LevelDB-based servers are presented in later sections. Our attempt is to first inform the readers about the raw performance metrics of Kinetic drives. This will help system engineers in designing their systems. We present the results of various random and sequential read and write tests for different value sizes as well as our experience with the Kinetic drive's unique P2P commands.

4.3.1 Definitions

Before delving into the tests, we define important terms which are used throughout the paper.

Sequential Key Order Writes (Sequential Order Writes): Write requests are issued by the client in increasing key ID order, i.e., from key number 1 to key number n.

Random Key Order Writes (Random Order Writes): The client issues write requests in random key ID order.

WriteBack-Flush: This write operation issues $n-1$ commands in the asynchronous WriteBack mode and then the n^{th} command is written in Flush mode to make all the commands persistent. This series of repeated asynchronous WriteBack writes followed by a single Flush command is what we call WriteBack-Flush mode.

4.3.2 Comparison of WriteThrough vs. Flush Mode

There are many applications, e.g., banking, that cannot risk data loss or inconsistency and must immediately have data stored persistently. Therefore, the Kinetic drive's multiple write modes, depending on the application, can be quite useful. There are two different modes for forcing synchronous write operations, WriteThrough and Flush, as mentioned in Table 4.3. Of the two, WriteThrough operations tend to be faster than pure Flush operations. Flush mode must make sure all previously written commands in the asynchronous WriteBack mode are made permanent. However, requests made in WriteThrough mode are made permanent without checking the status of any other command.

Results of a test to illustrate this difference are shown in Table 4.4. In this test there are two variations, one where all writes are issued in WriteThrough mode and one where all writes are issued in Flush mode. In both cases, each write (or put) is made persistent before returning a completion signal to the application. However, for Flush mode, the drives must ensure that there are no pending write requests waiting to be made persistent. For 10,000 key-value pairs and 100,000 key-value pairs, WriteThrough gives higher throughput than repeated Flush mode writes. This illustrates that WriteThrough should be used for synchronous writes, and it is bad programming to repeatedly use Flush mode for multiple writes.

Since throughput depends on the write modes, value size, and number of key-value pairs, we obtain higher throughputs in the following tests than we do in Table 4.4.

Table 4.4: Comparison of Throughputs for Different Write Modes

Value Size	Key-value pairs	Flush	WriteThrough
1 MB	10,000	5.41 (MB/sec)	8.99 (MB/sec)
1 MB	100,000	4.8 (MB/sec)	8.91 (MB/sec)

4.3.3 Comparison of Throughputs for Write and Read Operation

To help programmers understand the basic performance of these drives for various generic workloads, we test read and write performance for KV pairs with different value sizes in either random or sequential key order. In our terminology, Sequential (or Random) Order Reads/Writes mean Sequential (or Random) Key Order Reads/Writes.

Sequential Key Order Writes: This test consists of two experiments. First, after Sequential Order Write (explained in Section 4.3.1), the data is read back from key number 1 to key number n sequentially for Sequential Order Read. Second, after performing Sequential Order Write, the data is read back in random key ID order for Random Order Read.

Figure 4.4 shows the throughput obtained from the Kinetic drive for Random Order Read and Sequential Order Read of up to 1 TB of data sequentially written by key order. Since the write process is the same for both the experiments, it is shown only once in the figure. We vary the size of the value from 120 bytes to 1 MB but keep the key size constant at 16 bytes. We also keep the number of key-value pairs constant at 1,000,000. Thus, the x-axis value-size multiplied by one million will give the amount of data transferred for that particular portion of the experiment (i.e., 120 MB for the leftmost points and 1 TB to generate the points on the right).

We observe that the throughput increases with the value size. To understand this trend, we have to understand the capability of the Kinetic drive's internal processor, the client's processing capability, and throughput computation. The client sends requests to the Kinetic drive as fast as possible and can easily saturate the drive with messages. Due to the Kinetic drive's limited internal processing capability and the overhead to handle each request, it can process only a certain maximum number of read or write requests per second. When small value sizes are used, overhead dominates the Kinetic CPU and results in low throughput. As the value size is increased, the ratio of wasted overhead to data sent is improved and so is throughput. As is expected for hard disk technology, we also observe that Random Order Read throughput is lower than Sequential Order Read, because the drive's head has to be repositioned on the platter more often for Random Order Reads than Sequential Order Reads.

To save space, results for Random Order Reads after Random Order Writes on Kinetic drives are incorporated into the LevelDB server comparison in Figure 4.16.

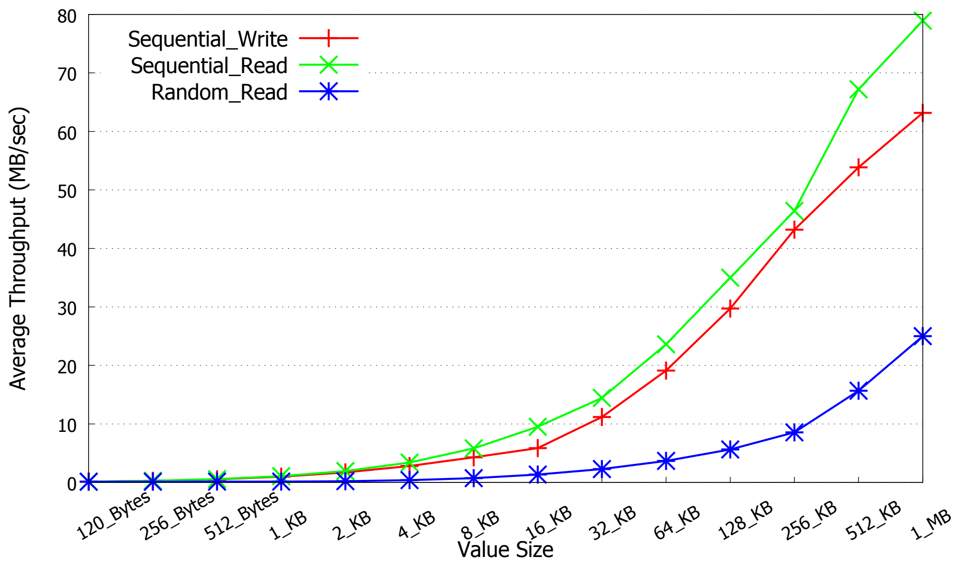


Figure 4.4: Average throughputs of Sequential Order Write followed by Random Order Read or Sequential Order Read for 1,000,000 key-value pairs with varied value size and 16 byte key size.

Random Order Writes: This testing also consists of two sets of experiments. This time, the client issues write requests in random key ID order for Random Order Write. After the data is successfully written in WriteBack-Flush mode (as explained in Section 4.3.1), key-value pairs in one experiment are sequentially read back. In the other set of experiments, KV pairs are randomly written and then read back randomly. The value size and other settings for Random Order Write tests are the same as the Sequential Order Write tests.

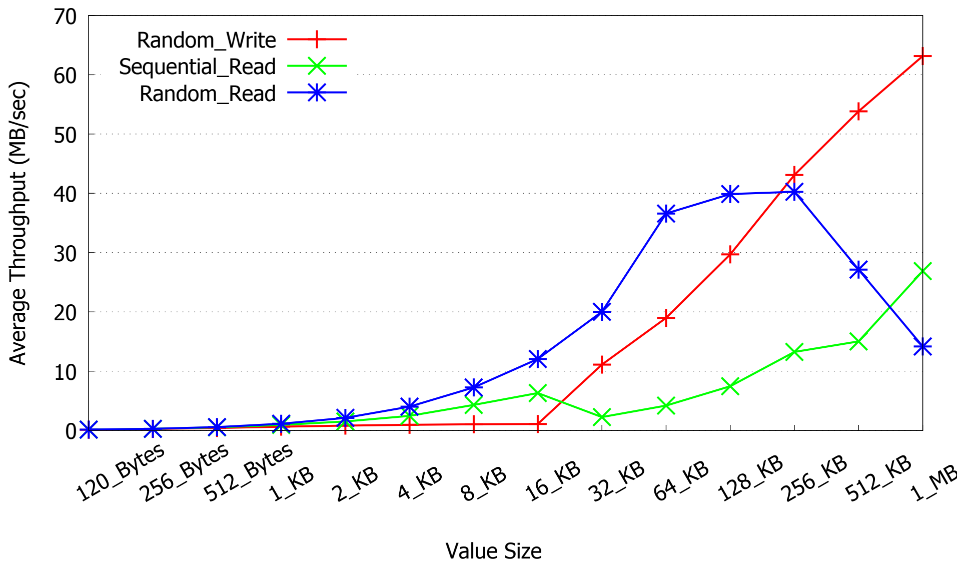


Figure 4.5: Average Throughputs of Random Order Write followed by Random Order Read or Sequential Order Read for 1,000,000 Key-Value Pairs with Varied Value Size and 16 Byte Key Size

This testing also consists of two sets of experiments. This time, the client issues write requests in random key ID order for Random Order Write. After the data is successfully written in WriteBack-Flush mode (as explained in Section 4.3.1), key-value pairs in one experiment are sequentially read back. In the other set of experiments, KV pairs are randomly written and then read back randomly. The value size and other settings for Random Order Write tests are the same as the Sequential Order Write tests.

The throughput comparisons are shown in Figure 4.5. Again, write is only plotted once because the results are the same for both sets of experiments. We see that Sequential Order Reads after the Random Order Writes in this figure show low throughputs similar to the Random Order Reads following Sequential Order Writes in the previous figure. Meanwhile, the randomness penalty is almost cancelled for some values showing high throughput with Random Order Reads after Random Order Writes. The two figures also show that the throughputs of Random Order Writes and Sequential Order

Writes are similar.

These trends occur because there is no relationship between the key ID and the location at which the drive stores the data on the physical platter. Instead, the drive receives write requests, does a minimal amount of batching and sorting, and writes them to the next available physical location on the platter a few megabytes at a time. This is a log-structured approach to writes because LevelDB (running in these drives) is based on the Log-Structured Merge-Tree (LSM-Tree)[51]. Later, during compaction, LevelDB will sort the key-value pairs based on the keys. We assume Seagate has optimized and customized LevelDB for this on-disk environment, but the general log-structured principles remain. In contrast, on a traditional hard drive where the Logical Block Address (LBA) typically maps directly to a Physical Block Address (PBA), we would expect that the throughput of random writes would be lower than sequential writes due to excessive seek times (head movements).

4.3.4 Maximum Write Throughput for Various Value Sizes

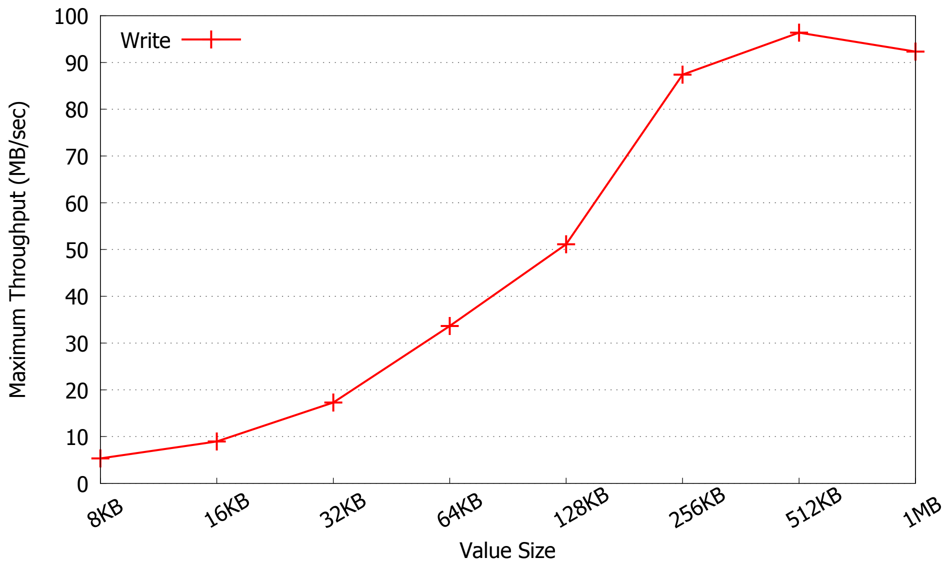


Figure 4.6: Maximum write throughput for various value sizes.

While the previous subsection measured average throughput, application designers may often require the maximum performance that can be extracted from these drives.

Based on the peak performance of the individual drives, the system engineers can estimate and try to optimize the raw performance that an entire system will be able to deliver. In this test we find the maximum write throughput that can be obtained for different value sizes with each key size equal to 16 bytes. Each data point on the x-axis in Figure 4.6 is a different experiment. For each experiment, we vary the number of key-value pairs and perform Sequential Order Write as explained in Section 4.3.1. The number of key-value pairs required to achieve the maximum throughput is different for all the experiments. Maximum write throughput increases with value size as shown in Figure 4.6. This trend is again because the CPU-bound (as demonstrated in Section 4.3.6) Kinetic drive wastes fewer resources on overhead with larger value sizes.

4.3.5 Impact of Key Size on Throughput Performance

So far we have varied the value size of the key-value pairs, but now we vary the size of the keys themselves. This gives application programmers an understanding of what to expect as the number of keys reaches its maximum or if they wish to use large keys to embed metadata or otherwise organize the KV pairs.

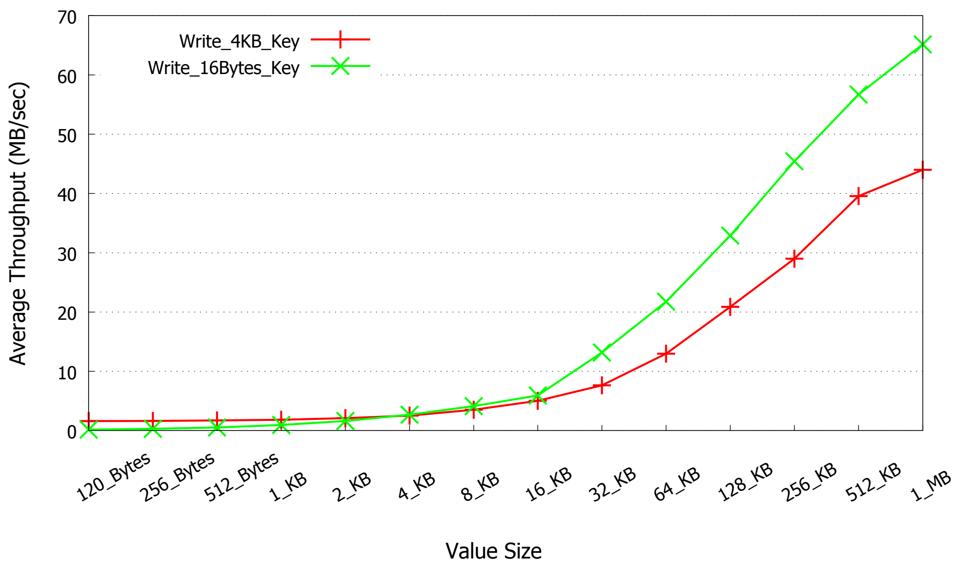


Figure 4.7: Average throughputs for minimum and maximum key size.

Figure 4.7 depicts the impact different key sizes have on write throughput. The comparison is made when KV pairs are written sequentially in WriteBack-Flush mode (as explained in Section 4.3.1) with two different key sizes, 4 KB and 16 Bytes. 1,000,000 key-value pairs are sequentially written for each value size from 120 bytes to 1 MB (the maximum value size).

While Seagate has a modified implementation, based on our understanding of LevelDB, we assume that the better throughput for smaller keys depends on the drive's internal RAM. Larger keys occupy more of the drive's internal RAM. The drive only has 512 MB of RAM for the entire onboard OS and LevelDB system. In this RAM, LevelDB keeps track of the largest and smallest keys of each SSTable. Each SSTable, which is a data structure that stores sorted KV pairs on disk, is 2 MB by default in stock LevelDB. As the value size of the KV pairs increases, fewer KV pairs fit in each fixed-size SSTable. As the number of SSTables increases to hold the larger values, there are more largest/smallest keys to maintain. When the key size is very large, this large number of largest/smallest keys no longer fits in the available memory. This introduces more disk activity which slows the overall throughput.

4.3.6 Throughput for P2P Transfer

Table 4.5: Throughput for P2P Transfer

Value Size	Read	No. of KV pairs	Internal CPU
1 KB	0.43 (MB/sec)	512	99.01%
1 MB	22.45 (MB/sec)	512	98.02%

This section presents the results for a unique feature, P2P transfer, which allows data transfer between two disks without extra network traffic or hardware burden on a separate machine. This feature is especially useful for propagating fault tolerance copies such as for replication. For example, if you want three separate identical copies of some key-value pairs in case one drive fails, you can have the Kinetic drives use P2P transfer in the background to spread these redundancy copies directly to other drives.

The client issues a P2P Get or P2P Put command to a Kinetic Drive (KD-Init) along with the IP address of the target Kinetic Drive (KD-Target). KD-Init establishes a connection over Ethernet to KD-Target and initiates put/write or get/read operations based on the command sent from the client. Once the operation is successfully completed, both the drives have these same key-value pairs. This data copying process takes place without interference from the client. The client only issues the initial instruction to begin the P2P transfer.

In Table 4.5, we show the throughput and internal Kinetic CPU load while using this feature for small and large values. First, we write 512 key-value pairs to KD-Target. After that, we perform the P2P Get operation in which KD-Init copies 512 key-value pairs, each with 1 KB or 1 MB value size, from KD-Target. We see in Table 4.5 that the internal processor's utilization is almost 100% when P2P transfer is taking place.

Figure 4.8 shows the results of a similar experiment using the P2P Put command to write 25,000 KV pairs for various value sizes. We can see that non-P2P Put from the client machine offers better throughput. However, if the client first has to read data from a different Kinetic drive before writing, P2P is more efficient and less burdensome for the client and the network.

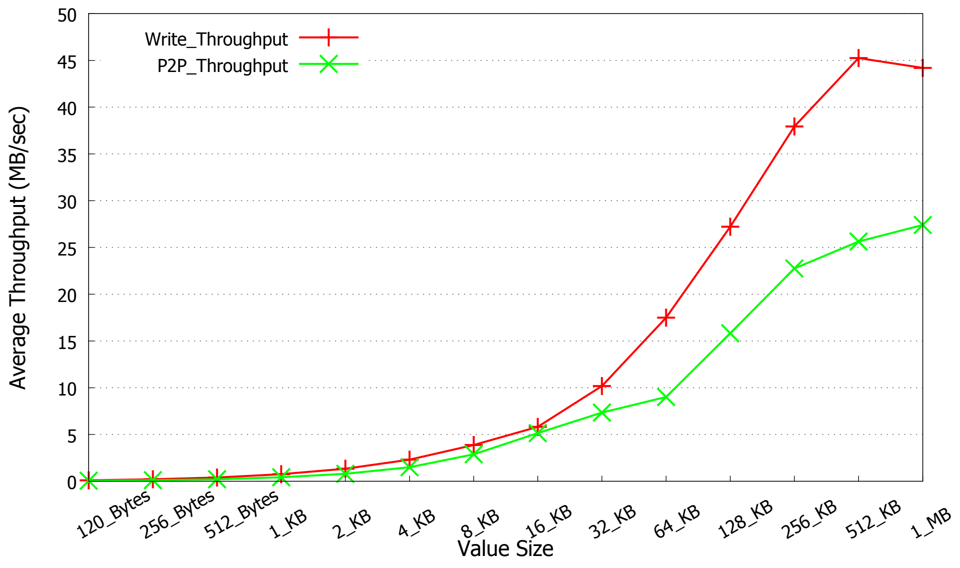


Figure 4.8: Throughputs of client-based Kinetic write vs. P2P put for 25,000 KV pairs.

4.3.7 Multi-threaded Throughput

This section presents the throughput obtained when two or more threads perform the write operation. Our write method for each thread count (two, four, and six threads) is similar to that described in Section 4.3.1. In Figure 4.9, we observe that the cumulative throughputs for multiple threads are similar to that of a single thread (i.e., the write plot in Figure 4.4). Again, this is likely because the internal Kinetic CPU is the limiting factor, and it cannot process multiple threads any faster than a single thread.

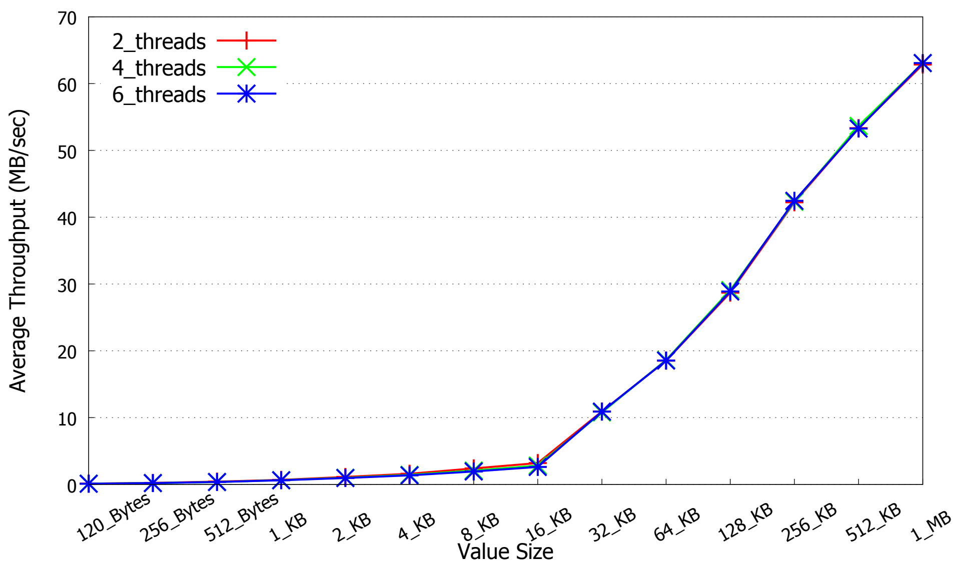


Figure 4.9: Aggregate Throughputs for Multi-thread Performance

The drives also have two Ethernet ports, but when we attempt to use them simultaneously, some of our test programs crash. Since 1 Gbps is not the bottleneck, these these ports are designed for fault tolerance rather than parallel performance.

4.3.8 Performance of Get Previous Command of Kinetic Drives

Figure 4.10 compares the read performance of reading key-value pairs in reverse order. Here, we test the "getPrevious" command that comes with the Kinetic API. As an illustration, if we give key n to the Kinetic drive using this command, the Kinetic drive returns the key-value pair with the key $n - 1$.

First, we perform sequential writes as described in Section 4.3.1, and, after completion, we use the previous command of the API to read the key-value pairs in reverse order, i.e., from key number n to 1. We compare this test with a custom created program that issues read commands from the n^{th} key-value pair to the first key-value pair. It can be observed that the performance is comparable. Therefore, this feature only brings convenience to the users, not improved performance. It can also be concluded that the "getPrevious" command and the regular get commands are handled similarly by the Kinetic drive.

To summarize all the results in this section, we conclude that the throughput is dependent on the value size of each key-value (KV) pair and that the utilization of the internal processor remains high with small or large value sizes. Therefore, we believe that the Kinetic CPU acts as a bottleneck and prevents throughputs from increasing beyond a certain range. Seagate is working on upgrading this processor for future Kinetic models.

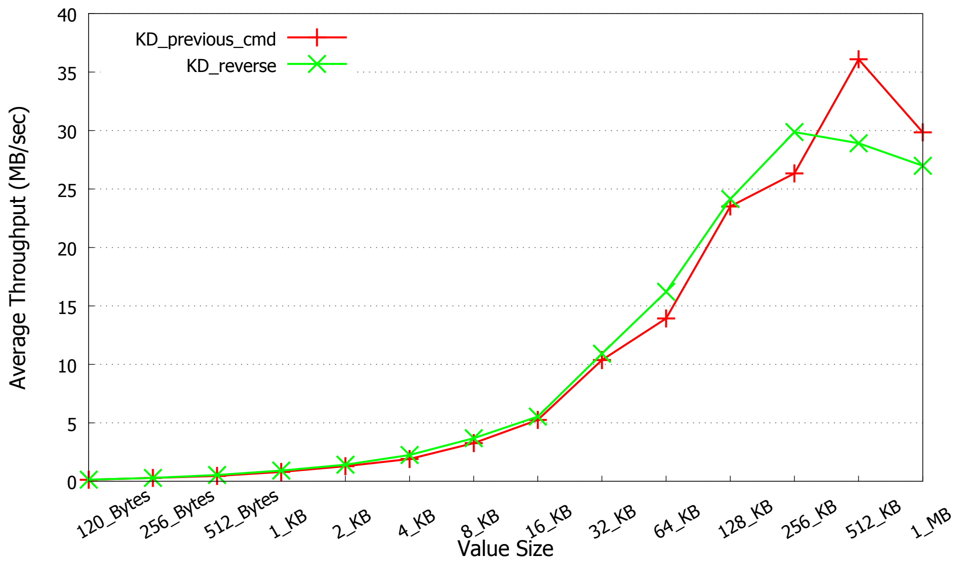


Figure 4.10: Comparison of Throughputs for Kinetic getPrevious Command (KD_previous_cmd) and Custom Backwards Read Program (KD_reverse)

4.3.9 Kinetic Drive Key Not Found Search Times

In this section, we study the time it takes when the searched keys are "not found" in a Kinetic drive. This aspect is important when a large number of drives are used to create a system. There is a strong possibility that users search for key-value pairs and may not find them in a particular Kinetic drive. Therefore, the users will have to search other Kinetic drives. The time it takes to perform the operation of searching key-value pairs in a particular server or Kinetic drive can affect the entire system's performance.

With 3.7 TB of Data Written: For the first experiment, we write 3.7 TB of data in sequential order in the form of 3,700,000 KV pairs with keys of 16 bytes each and value size of 1 MB. All the keys are unique and "kinetic-c-util" (an included test utility) is used to generate data for each of the 1 MB values. Following that, we request 100,000 non-existent key-value pairs from the Kinetic drive in sequential order. We ensure that the requested keys do not exist in the Kinetic drive. For each request, we record the time it takes for the Kinetic drive to return the status that the requested key is not found in the drive. We plot the "not found" response time of each request in Figure 4.11. We see that nearly all (99.916%) of the responses occur in one millisecond or 0.001 seconds, while a small number of responses take two or eight milliseconds. This implies there is mostly a network response time delay and only occasional disk seek latency, if any.

We next repeat the same operation; however, we request nonexistent KV pairs in random order. Again, the results look the same and are omitted. We also test the difference when nonexistent keys are inside the range of the written keys (e.g., write key 1 and key 3 and search for key 2 instead of key 4), but we see the same extremely low response time plot. Finally, we experiment with a smaller value size than 1 MB, but the results are generally the same. Small value sizes, such as 4 KB, show a few more requests high enough to be disk activity, but the overwhelming majority are still 1 ms. Because all these tests provide similar results, we cannot determine the effect of read or write randomness in searches for keys that do not exist. However, we can conclude that the Kinetic drive is able to quickly resolve unavailable key requests from its memory even when the drive is full.

These results make sense when you consider that LevelDB stores KV pairs in sorted key order inside 2 MB SSTable files. For each SSTable, the in-memory MANIFEST stores the file name, the largest key, and the smallest key stored in that SSTable. Because we are using a 1 MB value size in our KV pairs, each SSTable can only store two KV pairs. Thus, for 1 MB values, the MANIFEST acts as an in-memory index of every key stored, and requests for nonexistent keys can be processed immediately no matter the order of the initial writes or later reads. If the MANIFEST is not as helpful in determining if the requested key is stored, e.g., when we use smaller value sizes and search for keys in an overlapping range, the next layer is a Bloom filter. Only in the rare instance that the Bloom filter is unable to correctly determine that a key is not stored (i.e., a false positive) does LevelDB actually have to read the SSTable from the disk to check. We believe that some of the 2 ms read times are Bloom filter checks, and only the 8 ms searches involve actual disk accesses.

In summary, we can conclude that the time spent fulfilling a request for each key not found is significantly less than performing a successful read operation in the Kinetic drive when using 1 MB value sizes. When a key is requested that does not match the previously written keys, the in-memory MANIFEST and a fall-back Bloom filter can rapidly report the key as not found without the need for very many actual disk accesses.

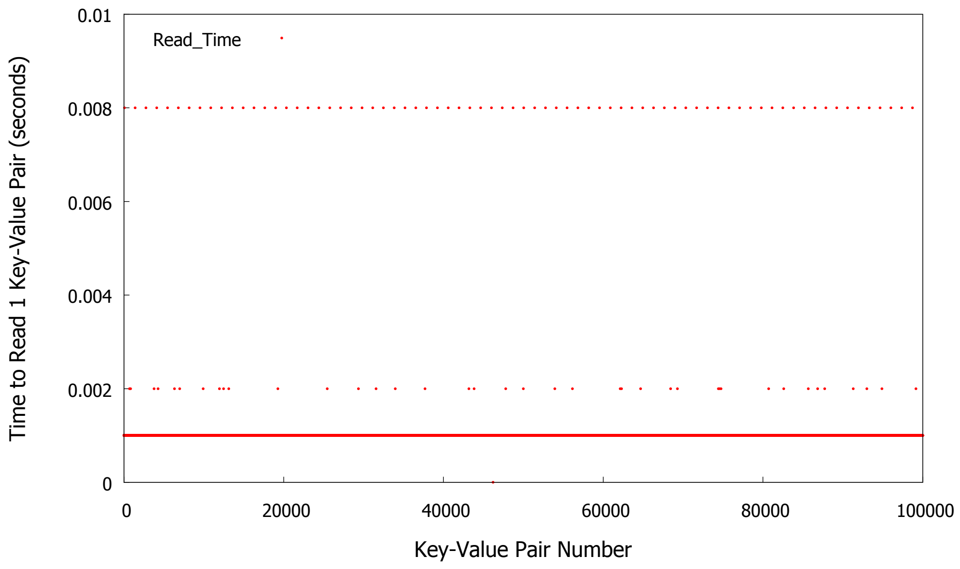


Figure 4.11: Keys Not Found - 100k KV Pairs Searched in Random Order With 3.7 TB of Data Written in Sequential Order

4.4 Throughput Comparisons of Kinetic Drives and LevelDB Servers

This section compares the throughput of Kinetic drives against traditional servers running LevelDB (Server-SATA and Server-SAS). We believe it is important to understand the possible implications of replacing traditional hard drives with Kinetic drives for various workloads and disk configurations. As described in Section 4.2, we have two different LevelDB-based server systems. Server-SATA has 12 GB of RAM and two extra 5900 RPM SATA disks with similar properties to the Kinetic drives, and the faster Server-SAS has 64 GB of RAM with several 7200 RPM enterprise drives.

4.4.1 Sequential Order Write and Sequential Order Read for Single and Multiple Drives

In this set of tests, we compare the sequential key order throughput of the three systems (Kinetic drives, Server-SATA, and Server-SAS) for reads and writes with various value sizes. As in the previous figures, each point on the x-axis represents different experiments. For example, 120_B in Figure 4.12 represents 1,000,000 KV pairs, each with value size of 120 bytes, first written in sequential key order for one experiment and then read back in sequential key order for another experiment. The total data transfer of each run is 120 MB. Similarly, 256_B represents another 1,000,000 KV pairs written and then read back, each with a value size of 256 bytes (total data transfer is 256 MB per run). It is also important to note that the y-axis is in log scale so that high throughputs do not overwhelm the plots. This happens because some of the server read experiments with smaller value sizes fit mostly in the server RAM and avoid many of the slower disk accesses. For read tests, the LevelDB benchmark, db_bench.cc, first loads some data to read but does not purge any caches before measuring the reads.

Having conducted simultaneous parallel experiments on all four Kinetic disks to verify no individual performance degradation, we extrapolate the plots for multiple Kinetic drive tests by assuming the load is balanced evenly across the independent disks and the performance will scale linearly until the Kinetic enclosure's 10 Gbps network link becomes a bottleneck.

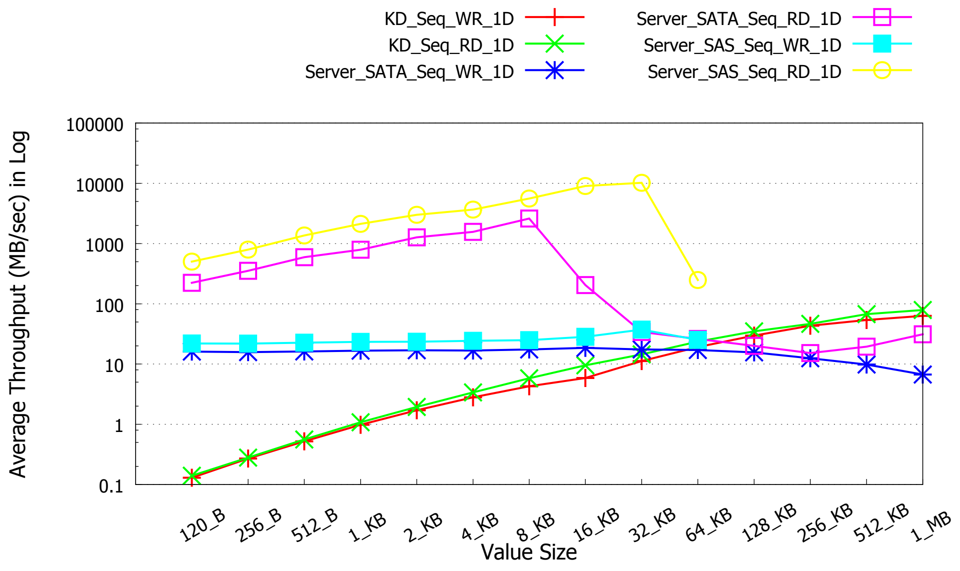


Figure 4.12: Sequential (Seq) Read (RD) and Write (WR) Throughput Comparison of a Kinetic Drive (KD), Server-SATA, and Server-SAS using One Drive (1D)

The most interesting observation is that the Kinetic drive can sustain higher throughput than a LevelDB server for large enough values sizes. For small value sizes, however, the servers with their large RAM size can easily outperform a Kinetic drive. Overall, even ignoring the small experiments that mostly fit in server RAM, we can also conclude that read operations achieve higher throughput than writes for all of the three systems. During reads, the drives simply have to deliver the data that has already been sorted by the systems. When there are many writes, LevelDB has to do compaction and level management that introduces write amplification.

We extend these experiments by adding drives, striped in software RAID-0, to Server-SATA and Server-SAS as shown in Figures 4.13, 4.14, and 4.15. Only Server-SAS has enough disks for four and eight drive tests. While the LevelDB servers do show throughput increases as we add more disks, their performance does not scale up as much as expected. For example, sequential order writes on Server-SAS stop improving beyond four drives.

Moreover, because Server-SAS has faster and far more RAM, it shows significantly higher read throughput than the Kinetic drives or Server-SATA for small value sizes.

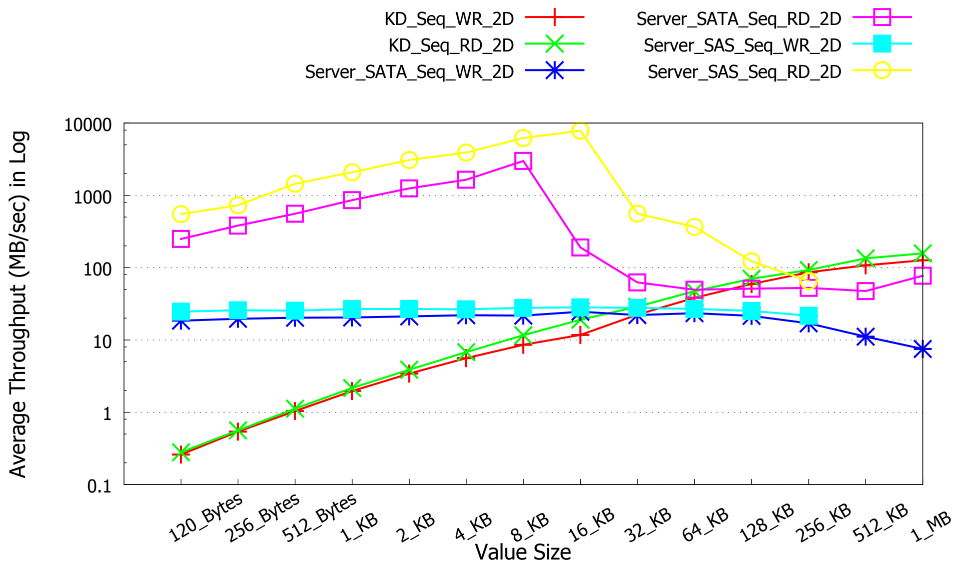


Figure 4.13: Sequential (Seq) Read (RD) and Write (WR) Throughput Comparison of Kinetic Drives (KD), Server-SATA, and Server-SAS using Two Drives (2D)

However, Server-SATA and Server-SAS both reflect a "memory cliff" trend in read throughputs. After the read throughputs increase continuously, they suddenly drop beyond a certain value size. As the amount of data transferred in each experiment increases with value size, the system memory cannot continue to hide slower disk accesses and throughput drops. At these points, the Kinetic drives usually take over and start giving better throughputs than either of the servers.

4.4.2 Random Order Write and Random Order Read for Single and Multiple Drives

We make comparisons when data is written and read back in a random key order (sequential results are omitted for space). We use db_bench.cc, which comes with LevelDB, to perform random key order reads and random key order writes with various value sizes on a single drive as well as multiple drives.

Every point on the x-axis in Figure 4.16 shows several different experiments. For example, 120_B represents 1,000,000 KV pairs, each with value size of 120 bytes and key size of 16 bytes, first written in random key order for one experiment and then read

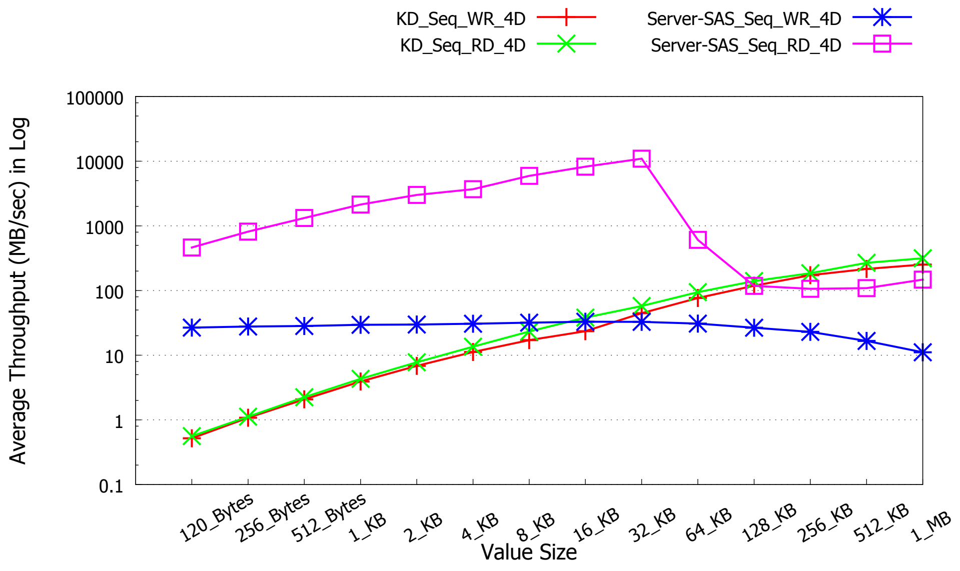


Figure 4.14: Sequential (Seq) Read (RD) and Write (WR) Throughput Comparison of Kinetic Drives (KD) and Server-SAS using Four Drives (4D)

back in random key order for another experiment. The total data transfer of each run is 120 MB. Similarly, 256_B represents another 1,000,000 KV pairs written and then read back, each with a value size of 256 bytes (total data transfer is 256 MB per run). It is also important to note that the y-axis is in log scale so that high throughputs do not overwhelm the plots. This happens because some of the server read experiments with smaller value sizes fit mostly in the server RAM and avoid many of the slower disk accesses. For read tests, the LevelDB benchmark first loads (writes) some data to read but does not purge any system caches before measuring the reads. It is likely that some (or, for small tests, all) of these previously written/loaded key-value pairs already exist in the server's cache when read performance is measured. Significant modifications to the LevelDB benchmark or adjustments to default settings, to hinder the servers by eliminating this caching, are beyond the scope of this research.

Looking at Figure 4.16, we notice that as the value size increases, the server's random write throughput decreases below 1 MB/sec. The larger value sizes were so slow that we were forced to abandon those tests. As the value size increases, so does the amount of data per experiment, the number of LevelDB files, the amount of key-value pair

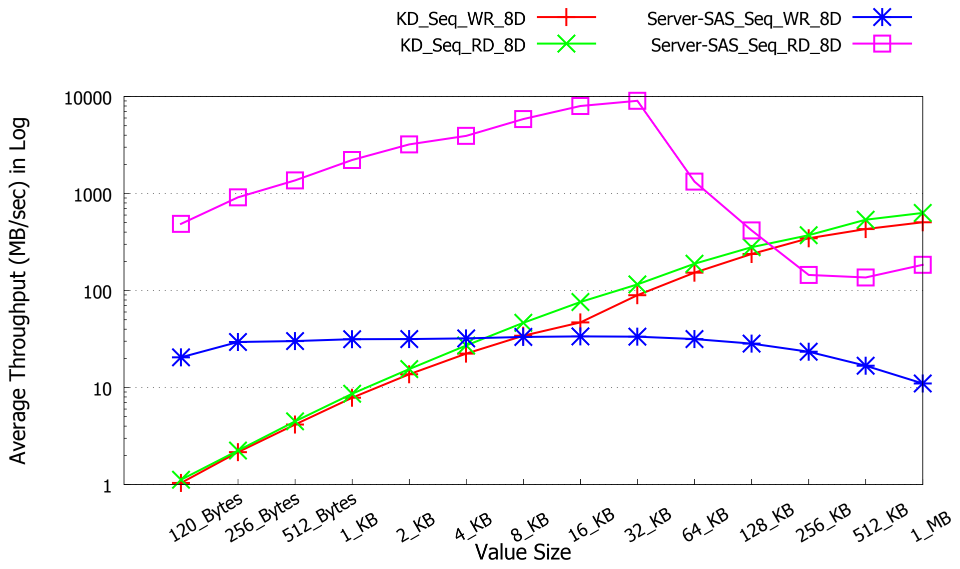


Figure 4.15: Sequential (Seq) Read (RD) and Write (WR) Throughput Comparison of Kinetic Drives (KD) and Server-SAS using Eight Drives (8D)

sorting, and overall disk activity. Moreover, irrespective of different RAM sizes, the write throughput for Server-SATA and Server-SAS both decline at similar rates as we increase the value size. When key-value pairs are written, they initially are written to a 4 MB memtable. Subsequently, the memtable is flushed to the drive in the form of 2 MB SSTables. Since such a limited amount of data is written in each memtable before flushing, the RAM of Server-SATA and Server-SAS has a limited impact on write performance.

The most interesting observation is that the Kinetic drive (Figure 4.16) can sustain higher throughput than a LevelDB server for large values sizes. Unlike the open-source API, the Kinetic drives themselves are closed-source black boxes. Therefore, the following hypothesis to explain their relative performance is difficult to verify. We assume that there is more write amplification during random key order writes for the default LevelDB configuration on the servers than for the optimized LevelDB in the Kinetic drive [52]. It is likely that the memtable and SSTable size is larger for Kinetic drives compared to LevelDB's default. This increased table size would create fewer files thus reducing read and write overhead. However, for small value size experiments, the servers

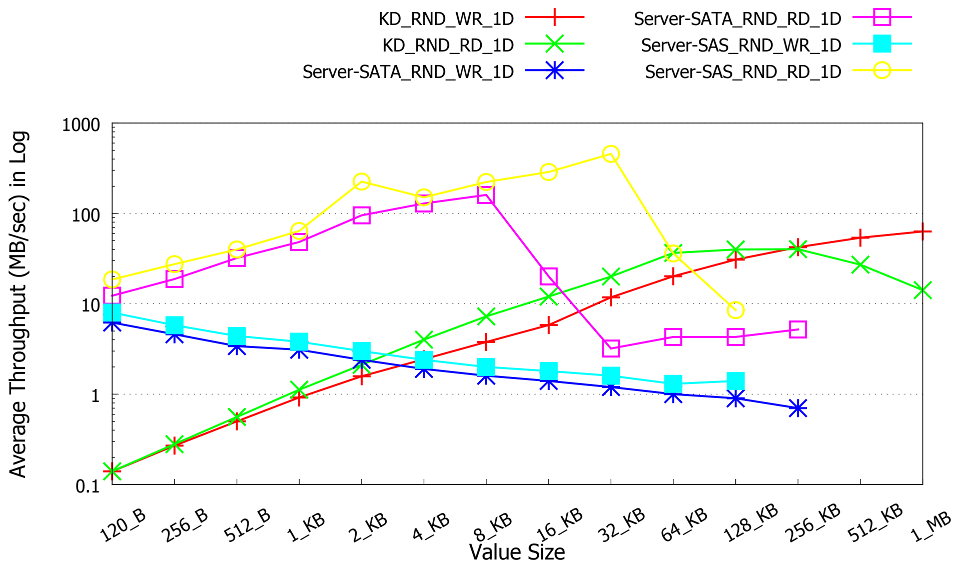


Figure 4.16: Random Order Key (RND) Read (RD) and Write (WR) throughput comparison of a Kinetic Drive (KD), Server-SATA, and Server-SAS using One Drive (1D).

with their large RAM size for reads and faster CPUs for writes can easily outperform a Kinetic drive. Overall, even ignoring the small experiments that mostly fit in server RAM, we can also conclude that read operations achieve higher throughput than writes for all of the three systems. During reads, the drives simply have to deliver data that has already been sorted by the systems. When there are many writes, LevelDB has to do compaction and level management that introduces write amplification affecting the throughput.

We extend these experiments by adding drives, striped in software RAID-0, to Server-SATA and Server-SAS as shown in Figures 4.17, 4.18, and 4.19. Only Server-SAS has enough disks for four and eight drive tests. The multiple drive Kinetic results in these figures are a linear extrapolation of the single drive results in Section 4.3.3. By conducting simultaneous parallel experiments on all four Kinetic disks, we verified that there is no individual performance degradation. Therefore, we assume the load is balanced evenly across the independent disks and the performance will scale linearly until the Kinetic enclosure's 10 Gbps network link becomes a bottleneck. While the LevelDB servers do show throughput increases in some cases as we add more disks,

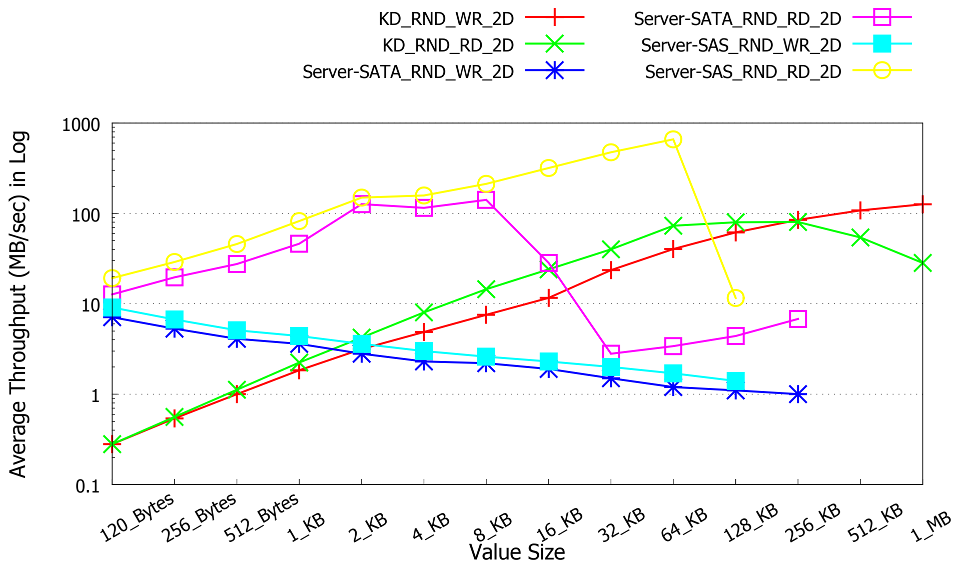


Figure 4.17: Random (RND) Read (RD) and Write (WR) throughput comparison of Kinetic Drives (KD), Server-SATA, and Server-SAS using Two Drives (2D).

their performance does not scale up as much as expected.

In these figures, the larger memory size of Server-SAS allows it to demonstrate significantly higher read throughput than the Kinetic drives or Server-SATA for small value sizes. However, Server-SATA and Server-SAS both reflect a ”memory cliff” trend in read throughputs. After the read throughputs increase continuously, they suddenly drop beyond a certain value size. As the amount of data transferred in each experiment increases with value size, the system memory cannot continue to hide slower disk accesses and throughput drops. At these points, the Kinetic drives usually take over and start giving better throughputs than either of the servers.

4.5 Latency Tests of Kinetic Drives and LevelDB Servers

Latency is a measure of the delay between when data is requested and when it actually arrives. This is critical for Quality of Service (QoS) of time sensitive applications. Moreover, latency is an indicator of the response time of a server. Users often dislike long wait times for their applications.

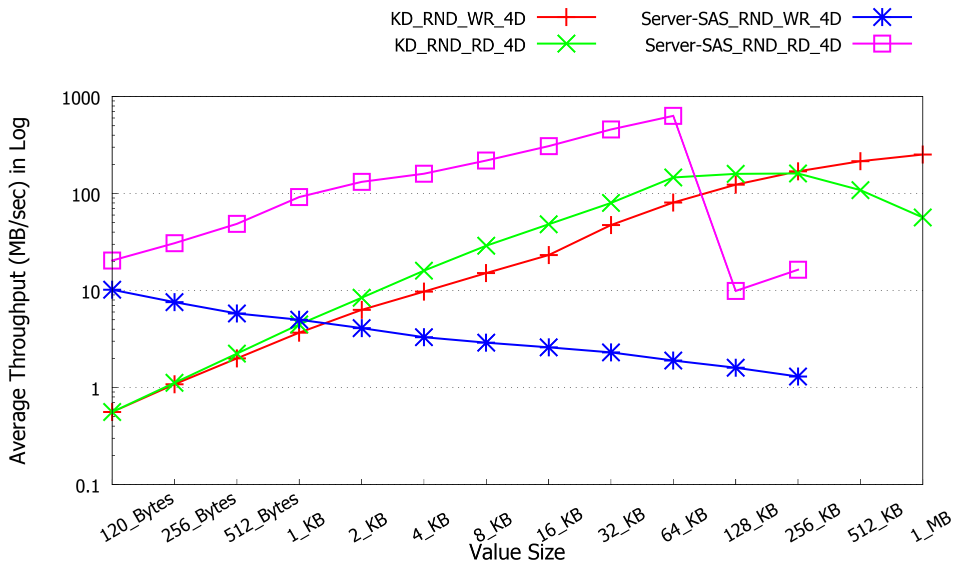


Figure 4.18: Random (RND) Read (RD) and Write (WR) throughput comparison of Kinetic Drives (KD) and Server-SAS using Four Drives (4D).

Read Latency of Kinetic Drives Following Random and Sequential Order Writes: Here we obtain and compare latency results for random key order reads following random key order writes or sequential key order writes (described in Section IV-A) on the Kinetic drives, Server-SATA, and Server-SAS. There are three different sets of tests conducted:

- Random Read Latency for Kinetic Drives after 100 GB or 3.7 TB of Random Order or Sequential Order Writes
- Random Read Latency for Server-SATA after 100 GB of Random Order or Sequential Order Writes
- Random Read Latency for Server-SAS after 100 GB of Random Order or Sequential Order Writes

Figure 13 presents a histogram of our Kinetic results. We first perform the 100 GB Kinetic sequential key order write operation as described in Section IV-A and then record latency as we read 100 GB in random key order by requesting 100,000 key-value pairs (writes and reads use value size of 1 MB and key size of 16 bytes). Having

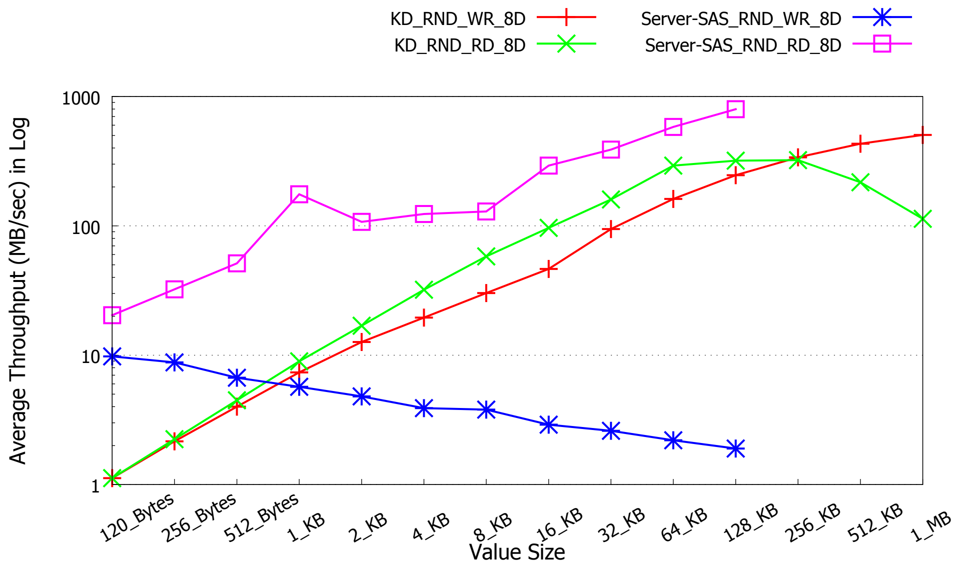


Figure 4.19: Random (RND) Read (RD) and Write (WR) throughput comparison of Kinetic Drives (KD) and Server-SAS using Eight Drives (8D).

recorded the time it takes to successfully complete each random read request, we put these latencies in different histogram bins to determine a count, or frequency of accesses, in that latency range. We obtain relative frequencies by dividing the number in each bin by the number of key-value pairs (100,000 in this case). We repeat this process for various write patterns to complete Figure 13.

Looking at Figure 13, we can observe that the read latencies are higher when the Kinetic drive is full, and reads following a drive filled with random order writes show the highest latency. In the figure, we group all latencies above 100 ms, which account for approximately 4-5% of the reads, into the right-most bin. For a read request, LevelDB may have to read files from multiple levels before it finds the correct KV pair. Reads satisfied by early levels like L0 or L1 will have lower latencies than requests that still have to access files in early levels but are not satisfied until L5, for example. The more data we write to the drive, the more data resides in higher/later levels of LevelDB, and reads from these higher levels cause the higher latencies.

This trend is due to the same optimizations we describe in the previous section that reduce read and write amplification in the Kinetic drive compared to the default

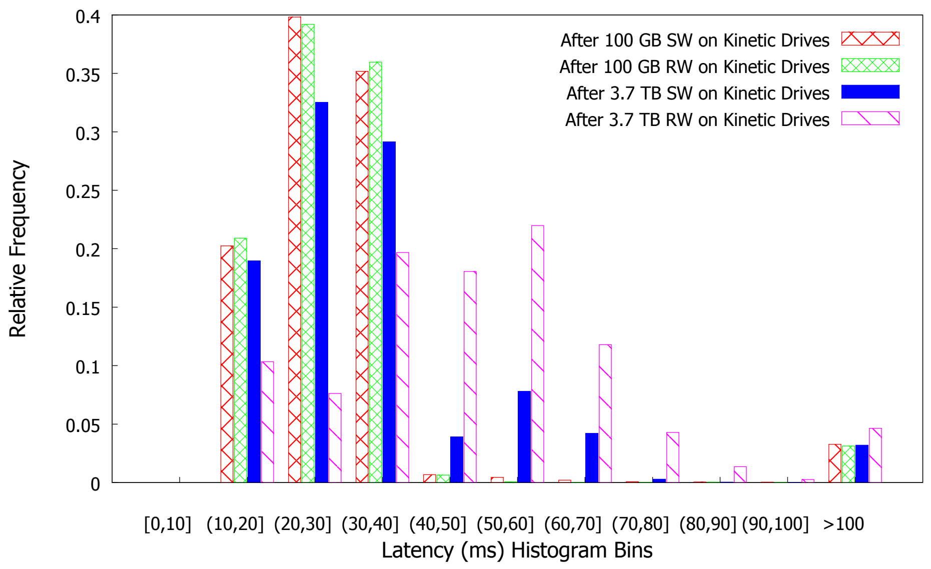


Figure 4.20: 100 GB Random Read after Sequential Order Writes (SW) or Random Order Writes (RW) on Kinetic Drives

LevelDB server. Seagate's Kinetic optimizations allow it to outperform Server-SATA, which has a similar hard drive, while the obvious hardware advantages of Server-SAS allow it to dominate over the other systems.

We can speculate that this is due to some poorly timed LevelDB compaction, but we are not certain why this happens. To rule out network latency, we tested the network ping time from the client to the Kinetic drives and found that it is usually around 0.3 ms. The ping is never above 1 ms, even when the Kinetic drive is at 100% CPU load. The latency also depends on the distribution of the searched key-value pairs, so it is possible that a few of the randomly chosen keys are requested in an order that is particularly difficult to process.

Overall, we can conclude that the amount of data written and the amount of randomness in that data affect the latencies for subsequent read operations on these Kinetic drives.

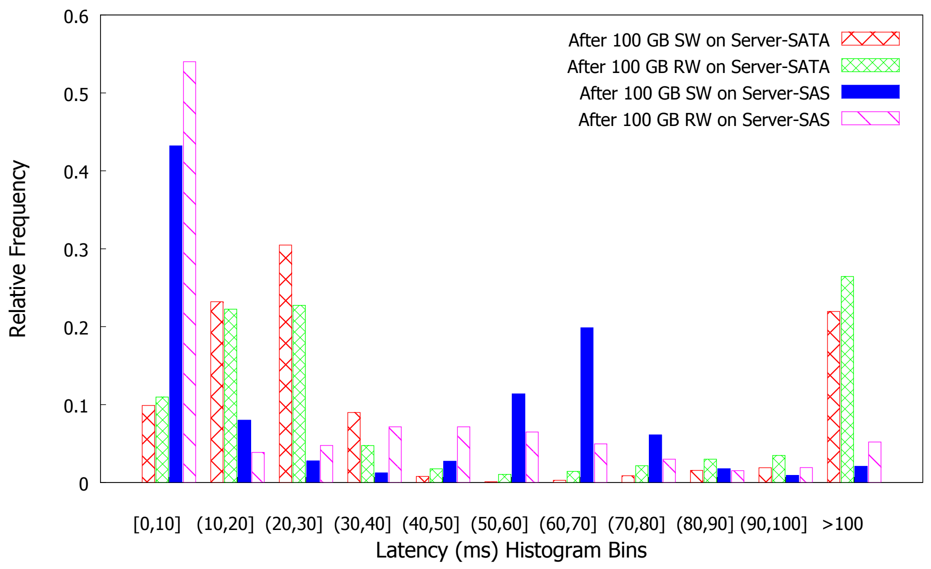


Figure 4.21: 100 GB Random Read after Sequential Order Writes (SW) or Random Order Writes (RW) on Server-SATA and Server-SAS

Confidence Interval

In an attempt to better understand these read latency results, we will examine them over time. Rather than show the cluttered scatter plots, we obtain the mean, along with 99% confidence intervals, of the latencies and use those to generate Figures 4.22, 4.23, 4.24, and 4.25.

We calculate the mean with confidence intervals of the latency data using the method described in [53]. We find a sample mean of every 1,000 KV pair read latency data points and use this mean to calculate the 99% confidence interval. We plot 100 data points from the original data showing 100,000 KV pairs and repeat for the four tests.

The most striking feature of these plots is the significant transition from higher latency to lower latency at some point during the progression of each experiment. Similar mean latency plots of the corresponding random and sequential Kinetic tests (added to these plots for easy comparison) are flat. We suspect that the high to low latency transition that appears in the server data is related to file system's buffer/cache, LevelDB compaction, and possibility that db_bench does not always generate unique keys or unique data for its random tests. We can say, however, that the transition from

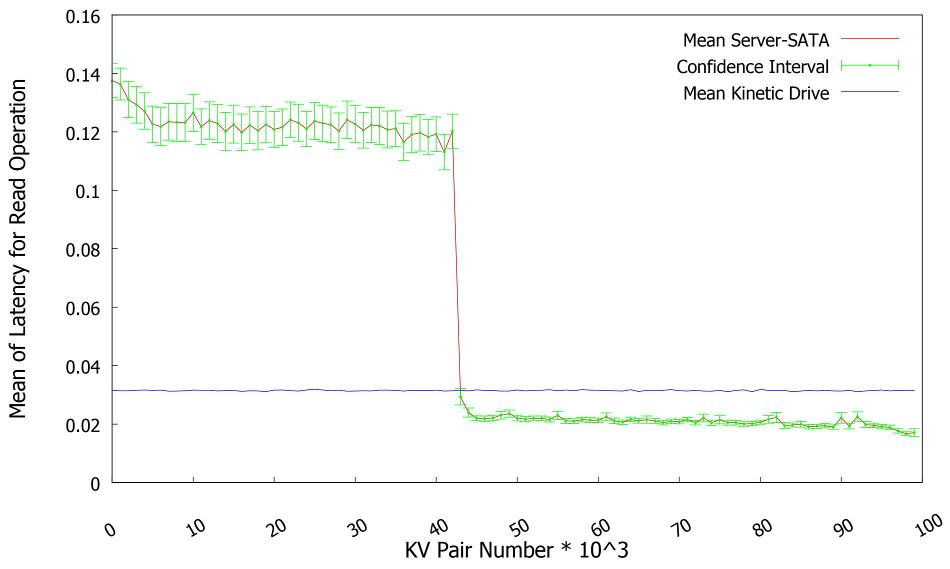


Figure 4.22: Mean Latency with 99% Confidence Interval of Reading 100,000 KV Pairs in Random Order After Writing 100 GB in Random Order with 1 MB Value Size and Key Size 16 Bytes on Server-SATA

high latency to lower latency happens sooner when reading the data that is written in sequential order.

4.5.1 Latency Tests for Server-SAS with Multiple Drives

To conclude this section, we repeat the read after random write Server-SAS LevelDB tests, but this time we use multiple hard drives in RAID-0. Again, we use the included LevelDB benchmark to write 100 GB in random order. The key size is still 16 bytes and value size is 1 MB. Afterwards, 100,000 KV pairs are read back and each latency is recorded. Figure 4.26 demonstrates the latencies for two drives, four drives, and eight drives on Server-SAS. As more drives are added, the overall read latencies reduce. Unlike the server results in the previous subsection, the mean latency plots (omitted for space) are flat.

Based on all the latency tests in this section, we conclude that there are significant differences between the Kinetic drive's customized LevelDB and other internal optimizations and the stock version of LevelDB we installed on the two servers. While

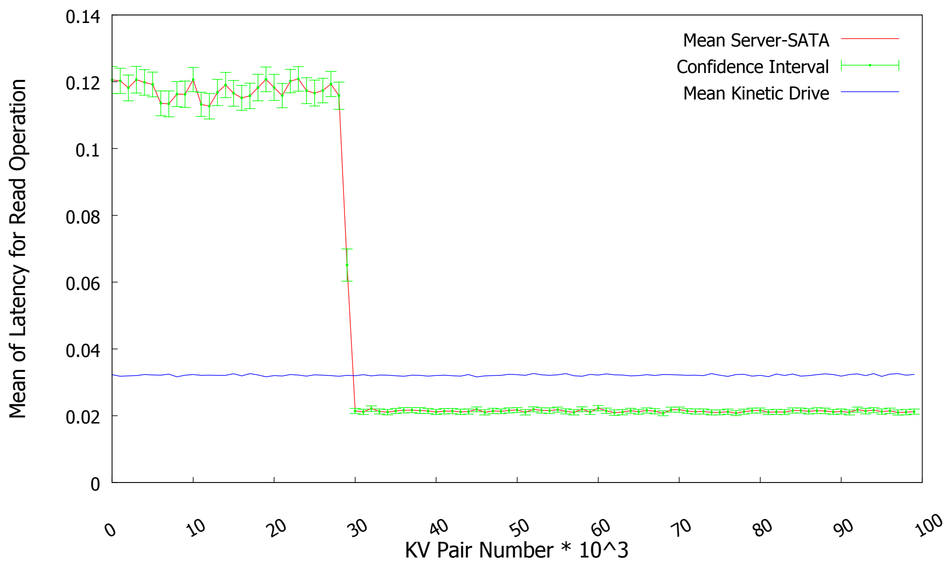


Figure 4.23: Mean Latency with 99% Confidence Interval of Reading 100,000 KV Pairs in Random Order After Writing 100 GB in Sequential Order with 1 MB Value Size and Key Size of 16 Bytes on Server-SATA

the powerful Server-SAS can generally show comparable or lower latency than Kinetic drives, the mid-range Server-SATA with a default LevelDB installation on a default file system is sometimes substantially outperformed by the Kinetic drive.

4.6 Keys Not Found — Trends for Kinetic Drives and LevelDB-Based Servers

In this section, we study the time it takes when the searched keys are ”not found” in a Kinetic drive or in a server. This aspect is important when a large number of drives are used to create a system. There is a strong possibility that users search for key-value pairs and may not find them in a particular Kinetic drive. Therefore, the users will have to search other Kinetic drives. The time it takes to perform the operation of searching key-value pairs in a particular server or Kinetic drive can affect the entire system’s performance.

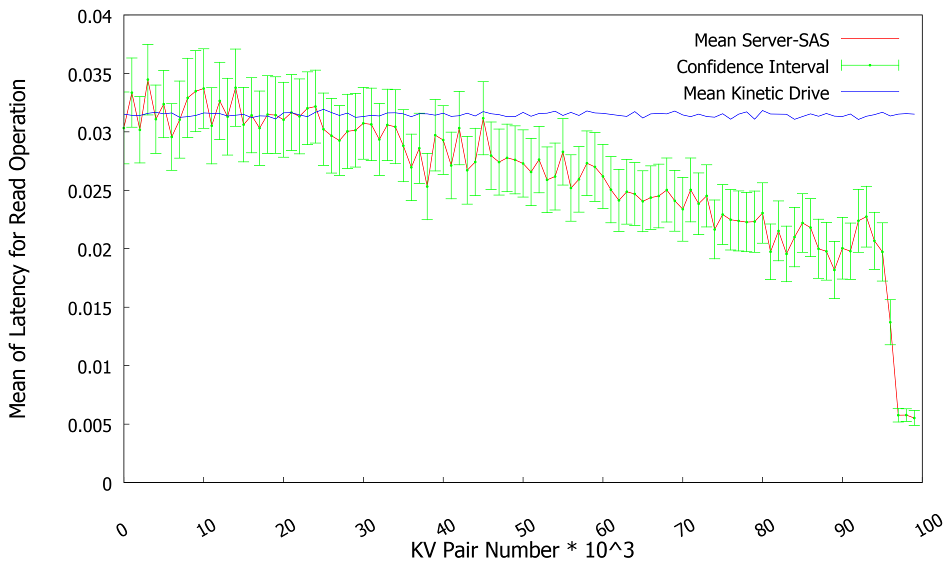


Figure 4.24: Mean Latency with 99% Confidence Interval of Reading 100,000 KV Pairs in Random Order After Writing 100 GB in Random Order with 1 MB Value Size and Key Size of 16 Bytes on Server-SAS

4.6.1 Kinetic Drive Key Not Found Search Times

With 100 GB of Data Written

For this test, we perform the write operation in sequential order, as explained in Section 4.3.1, to write 100 GB of data in the form of 100,000 KV pairs each with value size of 1 MB. Following that, we request 100,000 key-value pairs from the Kinetic drive in sequential order. We ensure that the requested keys do not exist in the Kinetic drive. For each request, we record the time it takes for the Kinetic drive to return the status that the requested key is not found in the drive. We plot the ”not found” response time of each request in Figure 4.27. We see that nearly all (99.916%) of the responses occur in one millisecond or 0.001 seconds, while a smattering of responses take two or eight milliseconds. This implies there is mostly a network response time delay and only occasional disk seek latency, if any.

To find out if the drives treat random order search differently from sequential order search, we perform the above experiment again. However, this time we request the 100,000 KV pairs in random order after writing 100 GB of data in sequential order.

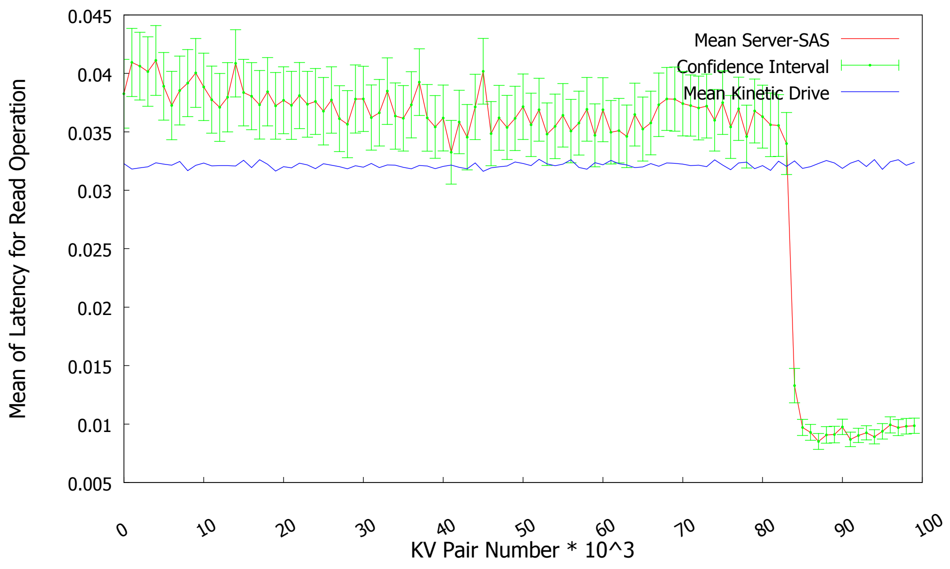


Figure 4.25: Mean Latency with 99% Confidence Interval of Reading 100,000 KV Pairs in Random Order After Writing 100 GB in Sequential Order with 1 MB Value Size and Key Size of 16 Bytes on Server-SAS

This plot, which we omit for space considerations, looks almost identical to Figure 4.27.

Based on our experience from the previous latency-based experiments, we introduce more randomness into the process. We next write 100 GB of data in random order and search for known missing keys in random order as well. The plot, again omitted, looks very similar to the previous test.

Because the results all depict a very similar and regular pattern, we conclude that the Kinetic drives are searching a data structure in their local memory (the LevelDB MANIFEST), not the physical disk platters, to determine that the keys are not stored.

With 3.7 TB of Data Written

In this subsection, we conduct similar experiments but observe the impact of completely filling the drive.

For the first experiment, we write 3.7 TB of data in sequential order in the form of 3,700,000 KV pairs with keys of 16 bytes each and value size of 1 MB. All the keys are unique and ”kinetic-c-util” (an included test utility) is used to generate data for

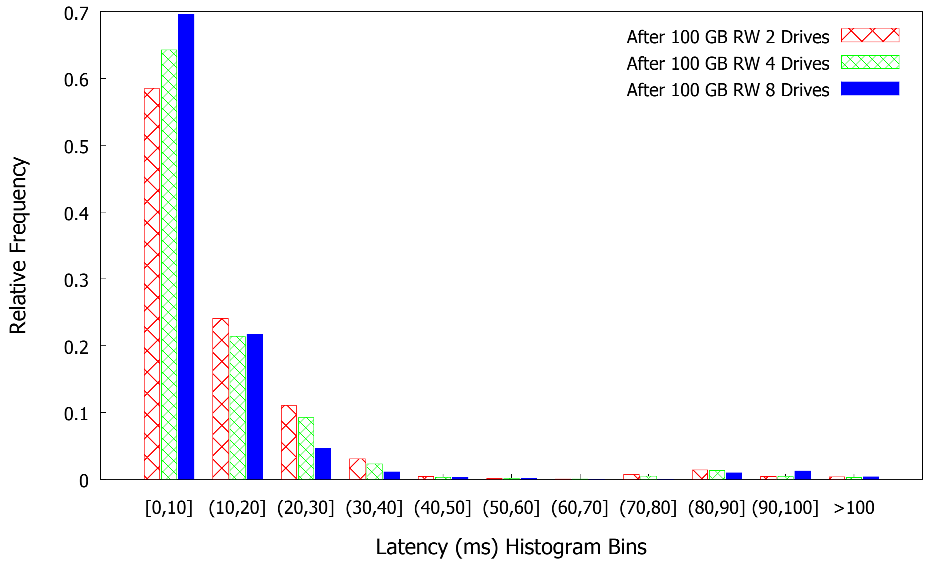


Figure 4.26: 100 GB Random Read after Varied Writes on Server-SAS with Multiple Drives

each of the 1 MB values. A plot showing response times of 100,000 sequential searches for nonexistent keys is similar to the previous plot, so we omit it. We next repeat the same operation; however, we request nonexistent KV pairs in random order. Again, the results look the same and are omitted. We also test the difference when nonexistent keys are inside the range of the written keys (e.g., write key 1 and key 3 and search for key 2 instead of key 4), but we see the same extremely low response time plot. Finally, we experiment with a smaller value size than 1 MB, but the results are generally the same. Very small value sizes, such as 4 KB, show a few more requests high enough be to disk activity, but the overwhelming majority are still 1 ms. Because all these tests provide similar results, we cannot determine the effect of read or write randomness in searches for keys that do not exist. However, we can conclude that the Kinetic drive is able to quickly resolve unavailable key requests from its memory even when the drive is full.

These results make sense when you consider that LevelDB stores KV pairs in sorted key order inside 2 MB SSTable files. For each SSTable, the in-memory MANIFEST stores the file name, the highest key, and the lowest key stored in that SSTable. Because

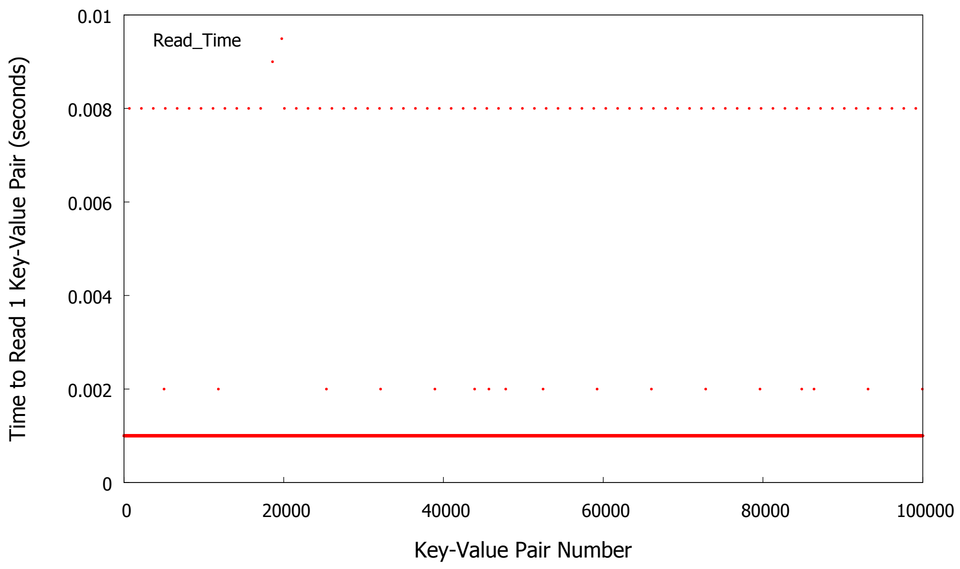


Figure 4.27: Keys Not Found—100,000 Nonexistent Key-Value Pairs Searched in Sequential Order after 100 GB of Data Written in Sequential Order

we are using a 1 MB value size in our KV pairs, each SSTable can only store two KV pairs. Thus, for 1 MB values, the MANIFEST acts as an in-memory index of every key stored, and requests for nonexistent keys can be processed immediately no matter the order of the initial writes or later reads. If the MANIFEST is not as helpful in determining if the requested key is stored, as an illustration when we use smaller value sizes and search for keys in an overlapping range, the next layer is a Bloom filter. Only in the rare instance that the Bloom filter is unable to correctly determine that a key is not stored (i.e., a false positive) does LevelDB actually have to read the SSTable from the disk to check. We suspect that some of the 2 ms read times are Bloom filter checks, and only the 8 ms searches involve actual disk accesses.

In summary, we can conclude that the time spent fulfilling a request for each key not found is significantly less than performing a successful read operation in the Kinetic drive when using 1 MB value sizes. When a key is requested that does not match the previously written keys, the in-memory MANIFEST and a fall-back Bloom filter can rapidly report the key as not found without the need for very many actual disk accesses.

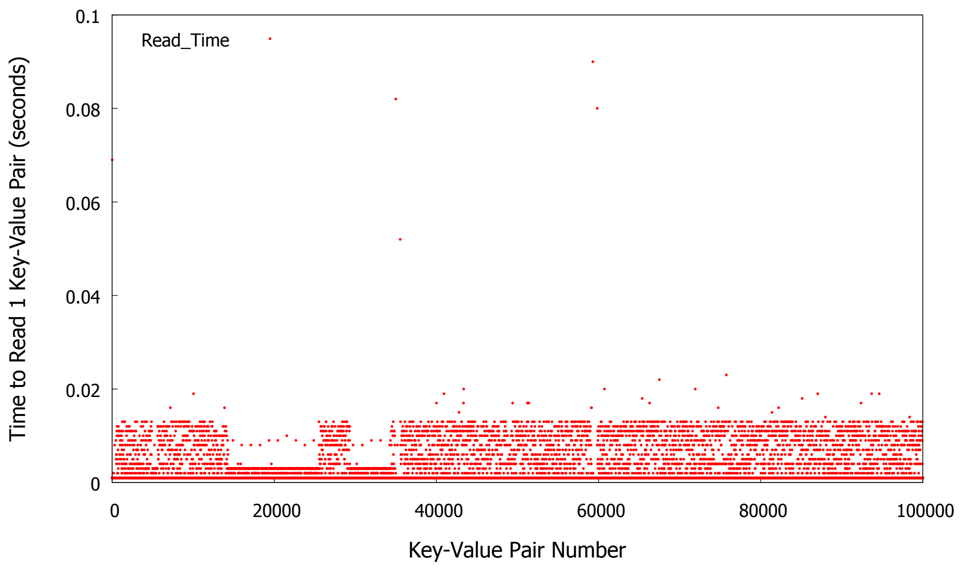


Figure 4.28: Keys Not Found—100,000 KV Pairs Searched in Random Order Seven Days After 3.7 TB of Data Written in Random Order

4.6.2 Effect of Long Term Storage of KV Pairs Before Searching for Keys Not Found

Now we further explore the on-disk memory behavior of the Kinetic drives. When read and write operations are performed, some of the KV pairs and other LevelDB data remain in the memory of the Kinetic drives. Therefore, if we read the data immediately after writing, the Kinetic drives may give us higher throughput or our searches might respond faster. In this experiment, we write 3.7 TB of data to the Kinetic drive in random order and then keep the drive idle for a week. After waiting for about a week, we randomly search for 100,000 KV pairs that cannot be found in the drive and plot Figure 4.28. The pattern is somewhat different from the previous plots. We finally start to see higher response times that imply there is actual disk platter data access needed to search for these missing keys. However, though it is difficult to see in the figure, 95.145% of the missing key search response times are still one millisecond (i.e., satisfied from the Kinetic drive's onboard RAM).

4.6.3 Impact of LevelDB Server-SATA's System Memory on Performance when Keys Not Found

In this subsection, to test the effect of LevelDB-based Server-SATA's memory on searches for unavailable keys, we perform two different sequences of read and write operations:

1. Random Write 100 GB, Search for Missing Keys (Figure 4.29)
2. Random Write 100 GB, Random Read 100 GB, Sequential Read 100 GB, Search for Missing Keys (Figure 4.30)

In these and the remaining plots in this section, it is critical to note that a large percentage of the key not found response times are reported as taking zero seconds. They obviously do not take zero time, but they are faster than the benchmark is able to report. As we also observe with the Kinetic tests, the overwhelming majority of the key not found requests are satisfied from the in-memory LevelDB MANIFEST. We suspect that the points on the plot showing between zero and approximately five milliseconds are Bloom filter searches, and the remaining points showing higher latency are false positives that required actual disk access to search an SSTable for a key that does not exist.

For the first sequence, we use the db_bench.cc program to randomly write 100,000 key-value pairs, each with a 1 MB value size. Afterwards, we immediately search for 100,000 missing keys and obtain Figure 4.29. 77.423% of the points show zero latency, as described above, and the rest of the points have relatively high latency.

For the second sequence, we also start by writing 100,000 key-value pairs of 1 MB value each. After successful completion of these initial writes, however, we perform additional reads, first randomly, then sequentially, of all 100 GB of the stored KV pairs. After these reads, we then search for keys that are not stored in Server-SATA to generate Figure 4.30. 88.195% of the points show zero latency.

Comparing these two figures shows that the latencies substantially reduce when we have multiple read operations performed across all the data before searching for nonexistent keys. This latency decline is likely due to some of the LevelDB SSTables being cached in the server's memory during the reads and available for fast search during

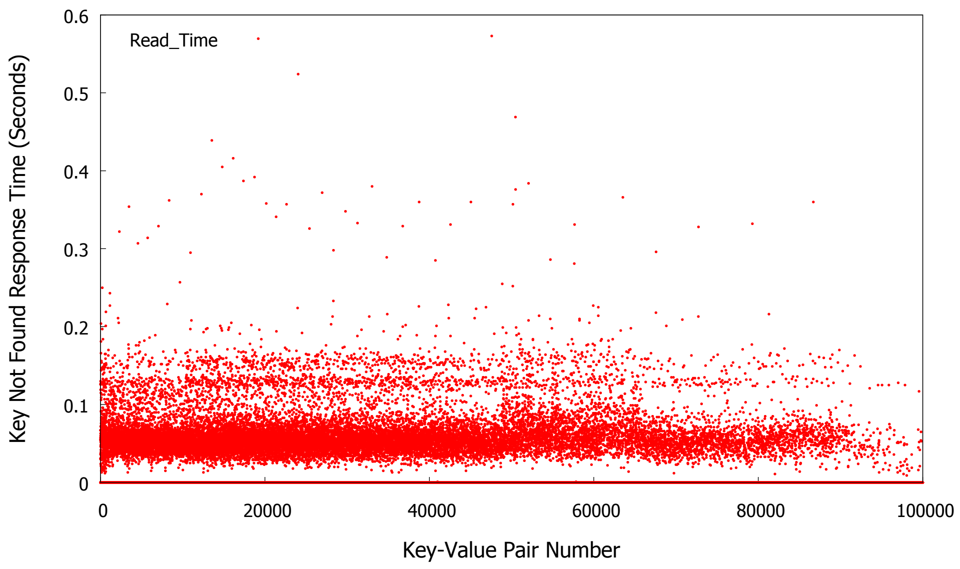


Figure 4.29: Key Not Found Response Times for LevelDB-based Server-SATA After Write Alone

the key not found requests. We do not know why the in-memory MANIFEST does not perform as well in Server-SATA as it does in the Kinetic drives.

4.6.4 Keys Not Found on Server-SAS With Multiple Drives

We now discuss the performance implications when the requested keys are not available in the more powerful LevelDB-based Server-SAS. We conduct these tests on one, two, four, and eight drives.

One Drive Tests

For the first experiment, we use db_bench to randomly write 100,000 KV pairs, each with a 1 MB value size and 16 byte key size. The LevelDB data storage location is a single SAS drive mounted with default ext4 settings. After writing, we randomly request 100,000 keys which are not stored in the system and create Figure 4.31. 98.374% of the response times are memory searches reported as zero seconds, and a handful are slow enough to be disk accesses.

Next, we perform a similar test; however, this time we write 100 GB of data in

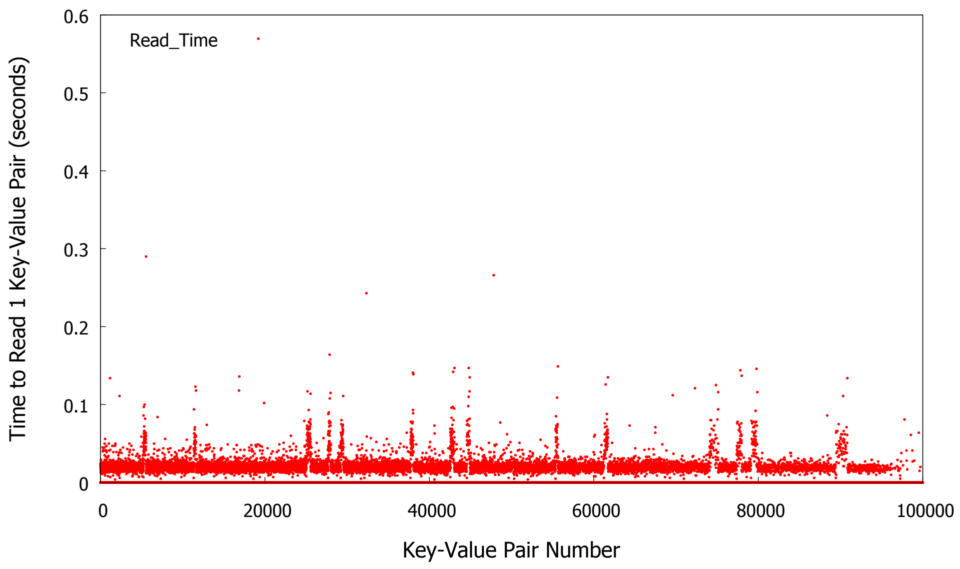


Figure 4.30: Keys Not Found Response Times for LevelDB-based Server-SATA After Write and then Two Reads

sequential order to generate Figure 4.32. Again, most of the data points are reported as zero seconds, but this time the few searches that take longer are all less than three milliseconds. Writing the data in sequential order appears to have eliminated all slower disk accesses for these key not found searches.

Two Drive Tests

We next use two drives in RAID-0 in Server-SAS to repeat the previous tests and obtain two plots. The random searches following random writes plot is shown in Figure 4.33. Again, only a small fraction of the requests show non-zero search times (less than 1% here), but this time they appear to be grouped toward the earlier part of the test. The search after sequential write plot looks almost the same as Figure 4.32, so it is omitted.

Four and Eight Drive Tests

Repeating these tests with four and eight drives again shows mostly zero milliseconds results from the system memory with a handful of slightly higher latencies from the

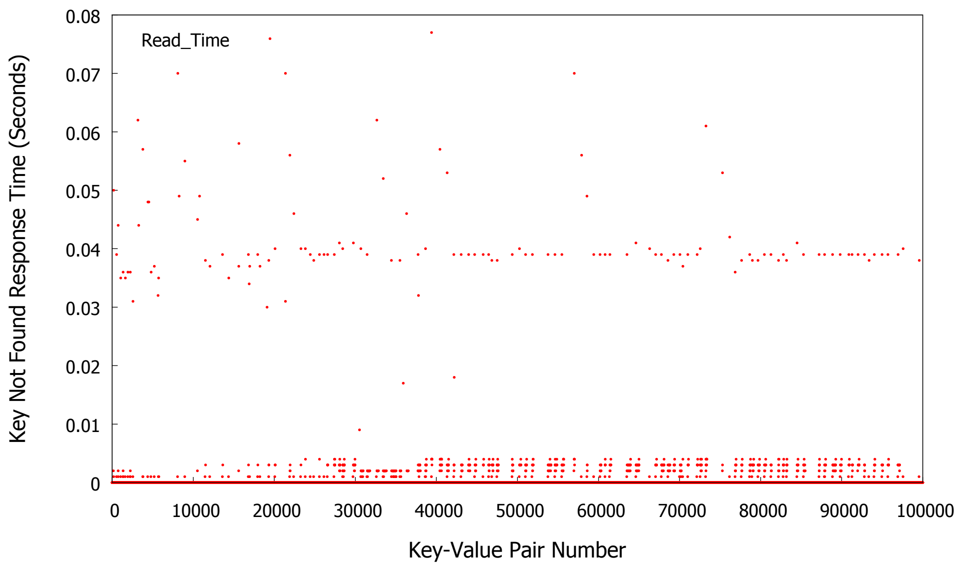


Figure 4.31: Key Not Found Response Times for Random Key Searches of LevelDB-based Server-SAS After Random Order Write on One Drive

occasional disk access. Plots are omitted for space. The non-zero times show a few different patterns, which we assume are related to the randomness generated by db_bench and how many of the keys are not unique. If the keys are not unique, the file system buffer/cache may be able to assist. However, many of the tests showed over 99% in-memory response times, so these differences are insignificant.

We again repeated the above mentioned experiments, however this time we employed 4 drives in the Server-SAS. The patterns obtained are shown in the Figures 4.35 and 4.36.

8 Drive Tests

Results for latencies when keys are not found for Server-SAS with 8 drives are presented in this section. Figures 4.37 and 4.38 present our findings.

To summarize our keys not found experiments, we can say that the expensive Server-SAS and the Kinetic drive perform quite well at detecting and quickly reporting that certain searched keys are not found. Mid-range Server-SATA shows more latency due to needing more disk activity to determine that the requested keys do not exist.

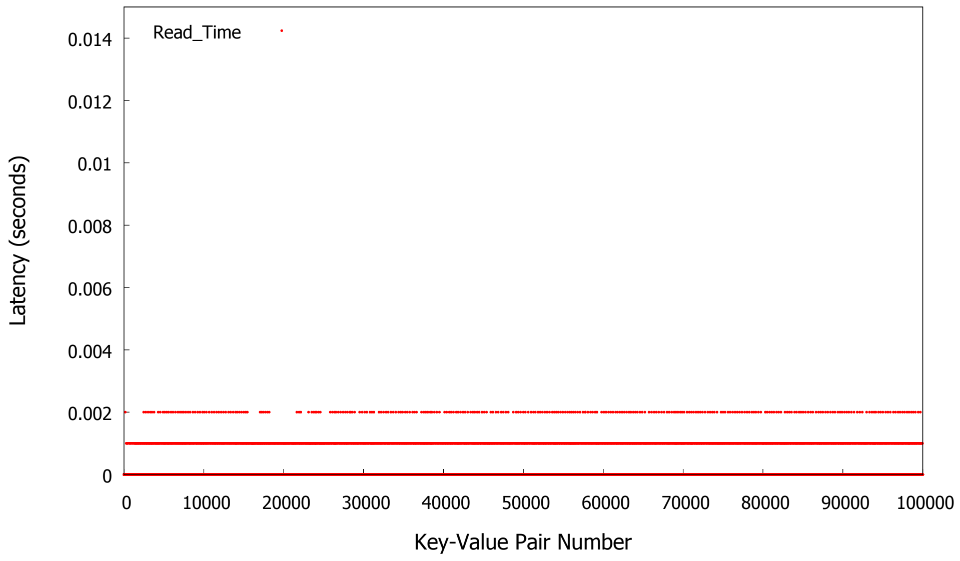


Figure 4.32: Key Not Found Response Times for Random Key Searches of LevelDB-based Server-SAS After Sequential Order Write on One Drive

4.7 Yahoo Cloud Serving Benchmark

In addition to the micro benchmarks discussed in the previous sections, we choose a popular macro benchmark, Yahoo Cloud Serving Benchmark (YCSB) [54], to simulate several more realistic application-level workload mixes. Because there is no direct LevelDB interface to YCSB, we initially tried to use Mapkeeper [55] as the key-value store with LevelDB as the storage backend. Unfortunately, Mapkeeper is no longer supported by YCSB. After much effort, we were only able to make it work with an older version of YCSB, but even then there were file locking issues after the YCSB load phase that prevented the run phase from working correctly.

Another key-value store, Riak KV [56], can use an optimized version of LevelDB as its storage backend. Riak is under active development and works with the newest version of YCSB. Our YCSB testing for a normal hard drive uses a single Riak node configured on Server-SATA to use LevelDB as the backend and the 4 TB SATA drive as the data storage location. Because we use only one node, we set the number of data copies from the default of three to only one. All other parameters are left as their defaults. We first run the Riak tests with the YCSB client running locally on the Riak node itself. Then,

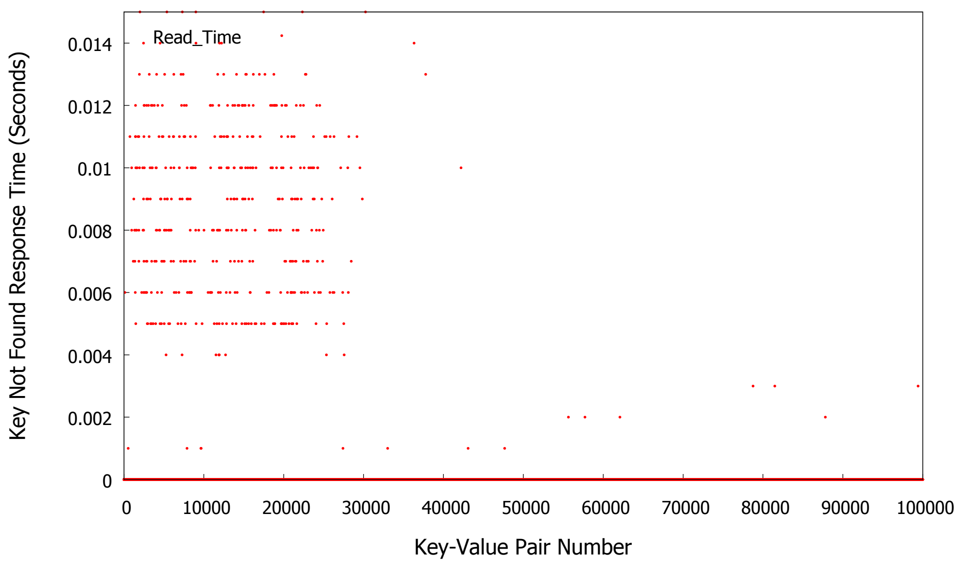


Figure 4.33: Key Not Found Response Times for LevelDB-based Server-SAS After Random Order Write on Two Drives

in order to add a network layer similar to that used by the Kinetic drives, we repeat the experiments with the YCSB client sending data over a Gigabit Ethernet network to the Riak node from an identical machine (i.e., the same specs as Server-SATA). These tests use YCSB version 0.12.0 and Riak version 2.2.0. For YCSB tests on the Kinetic drives, we use the version that comes with the Kinetic Java Tools software [57]. Both the normal HDD and the Kinetic YCSB testing use the same workload parameters for a fair comparison.

YCSB includes the following six workload patterns: Workload A is a heavily updated workload with a 50/50 read/write ratio. Workload B is a read-mostly workload with a 95/5 read/write ratio. Workload C is 100% reads. Workload D has 5% updates and 95% reads, but the recently updated records are the ones that are read the most; thus, it is a read-latest workload. Workload E queries short ranges of records, but the scan command is not yet implemented on the Kinetic drives, so we do not use this workload. Workload F is half reads and half read-modify-write commands where a record is read, updated, and written back.

The YCSB load phase of Workload A is 100% writes, and we follow that with a run

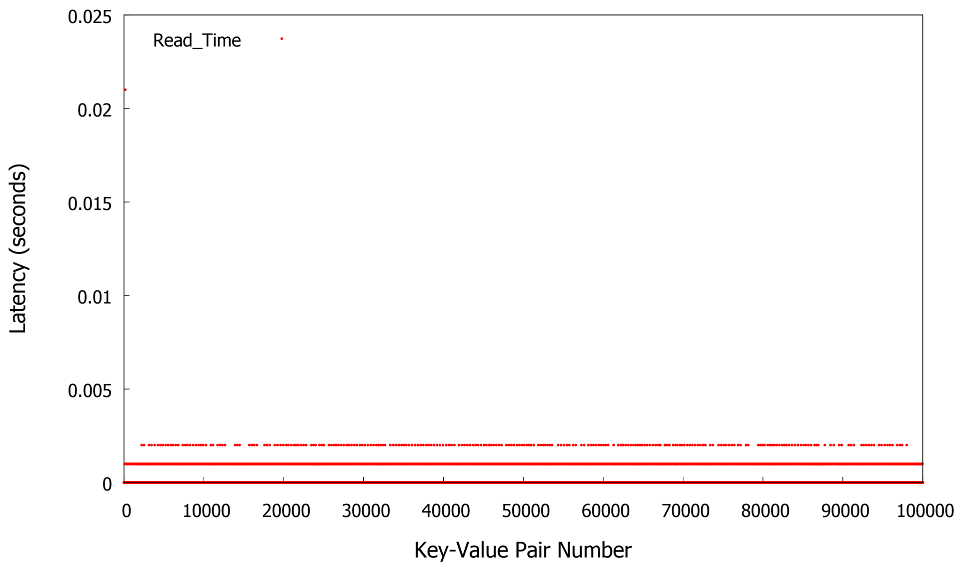


Figure 4.34: Time for Searching Keys Which Are Not Found In The LevelDB Installed Server-SAS with Sequential Order Write and Random Order Search for Keys with 2 Drive

of Workloads A, B, C, D, and then F. We insert a pause of five minutes between each workload to allow the system some idle time for LevelDB compaction or other internal processes.

Figure 4.39 shows the average throughput while loading and running these workloads with a value size of 1 MB. For each workload, 50 GB of data is transferred. In this case we see that a Kinetic drive is able to consistently outperform the Riak/LevelDB node for both the load and run of Workload A as well as for the run of Workload F. For runs of Workloads B, C, and D, the Kinetic drive shows throughput similar to or slightly worse than that of the Riak node when its YCSB test is run locally. When YCSB is run over the Gigabit network to the Riak node, its performance declines, and the Kinetic drive shows similar or slightly better throughput for those workloads. The Riak node shows poor performance in workloads with large amounts of updates/writes like Workloads A and F.

Figure 4.40 displays the results for a value size of 4 KB. In order for the experiment to complete in a reasonable amount of time, the data transferred per workload was reduced to 5 GB. Here we see that the Kinetic drive becomes I/O bound due to the

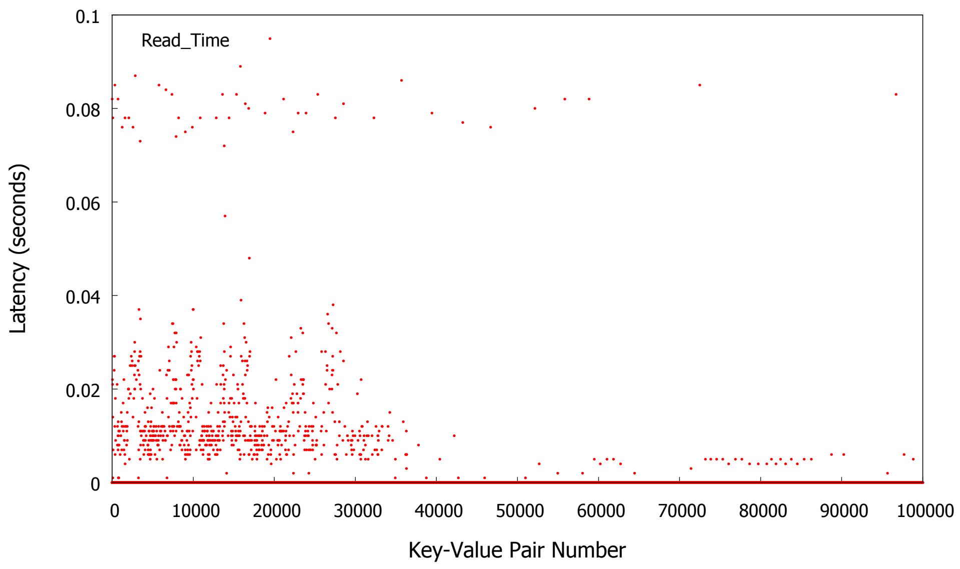


Figure 4.35: Time for Searching Keys Which Are Not Found In The LevelDB Installed Server-SAS with Random Order Write and Random Order Search for Keys with 4 Drive

large number of small accesses while the Riak server, with its larger amount of RAM, is able to better organize these small transfers for more efficient disk access. However, the performance of both the Kinetic drive and the Riak node is far lower with 4 KB values than with 1 MB values. Our findings with YCSB confirm our findings of the Section 4.3.3. This poor performance with small I/Os is typical of rotating mechanical media and is due to the disk seek time. Again, the Riak tests suffer a slight drop in performance when run over the network compared to when run locally.

In summary, the YCSB tests use a more realistic configuration to reaffirm the earlier micro benchmark results. For smaller value sizes, the full server is easily able to show higher performance; but, for larger value sizes, the Kinetic drive can perform surprisingly well, especially in write-intensive workloads where it outperforms the server for similar reasons as discussed in Section 4.4.

4.8 Prior Work on Management of Large Scale Drives

The final part of the dissertation is about managing a large number of drives with the capability of in-storage processing. One key aspect of large number of drives is the

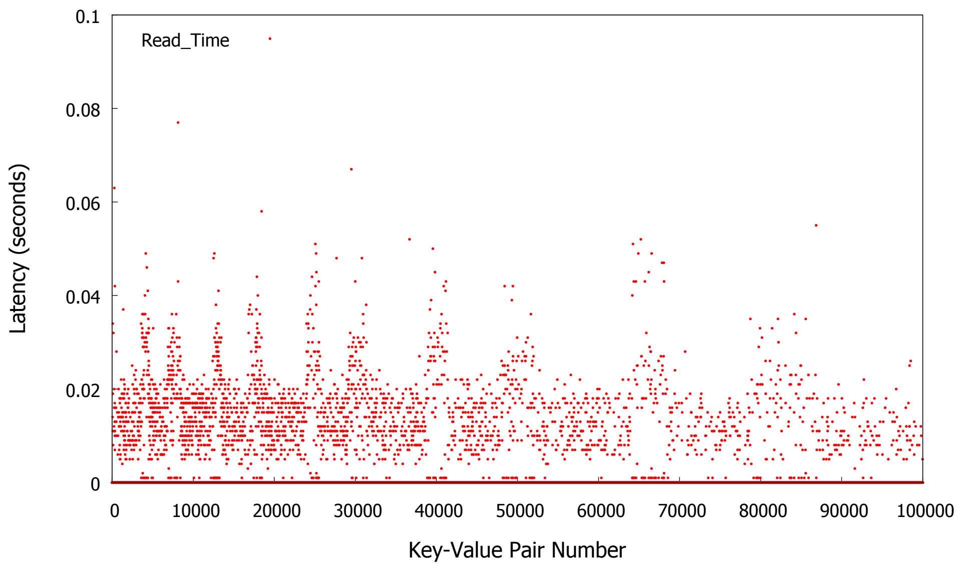


Figure 4.36: Time for Searching Keys Which Are Not Found In The LevelDB Installed Server-SAS with Sequential Order Write and Random Order Search for Keys with 4 Drive

ability to manage the drives. Without this methodology, industry would find it very challenging to adopt a new technology. At data centers enormous number of drives are used and for an efficient management system a low turn around time or less down time is extremely important to maintain service times for the requests. In addition, data could often be migrated among the drives and we need to ensure their is not much migration. In this chapter, we discuss data allocation and migration schemes for Kinetic drives for a possible large scale deployment of these drives.

As mentioned before, Kinetic drives could be accessed using the IP addresses and do not need a dedicated server. However, to organize a large number of drives, we would require a metadata server that keeps a track of where the data could be found. Users can not have the IP addresses of all the drives in the system for security purpose. Users also won't have the latest information on where the data resides. Data is often migrated between the drives and needs to be tracked. Therefore, a metadata server is absolutely required.

In Kinetic drives data is stored in the format of (key,value) pair. Without a key, data can not be accessed. When a user or client requests for data access, a server would

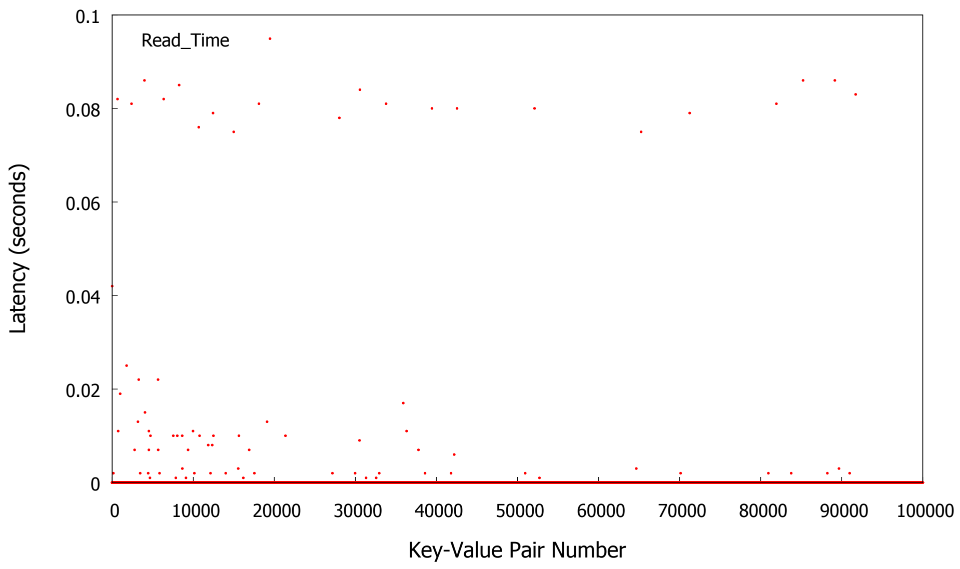


Figure 4.37: Time for Searching Keys Which Are Not Found In The LevelDB Installed Server-SAS with Random Order Write and Random Order Search for Keys with 8 Drive

give the user the latest information on the drive, where the data is stored. The drive has to keep a track of all the locations for the data. Metadata server provides indexing scheme for the data allocation. This is shown in the Figure 4.41. In sharp contrast to the previous configuration, the users would have to contact with the storage server and then subsequently the server would have to fetch the data from the drives and service the request as shown in the Figure 4.42. These figures are obtained from work done jointly with Cao *et al* [16] [1] .

A major concern for a metadata server for the Kinetic drive would be to keep a track of all the key-value pairs. So, all these key-value pairs are stored in key ranges. We want to reduce the number of drives on which mapping of key ranges take place. However, we also want to increase the disk utilization. And in this research work, we attempted to find the trade-offs. In this research indexing scheme is discussed to map the key-value pairs to the drives, migration policies for efficient data transfer, and different design

[1] Figures 4.41, 4.42, and 4.43 are reproduced with permission in this dissertation to illustrate the performance of possible key-indexing schemes that could be used in the metadata server. The key-indexing schemes are not specific, new contributions of this dissertation, however, these ideas were previously published in Cao, Minglani, and Du [16]

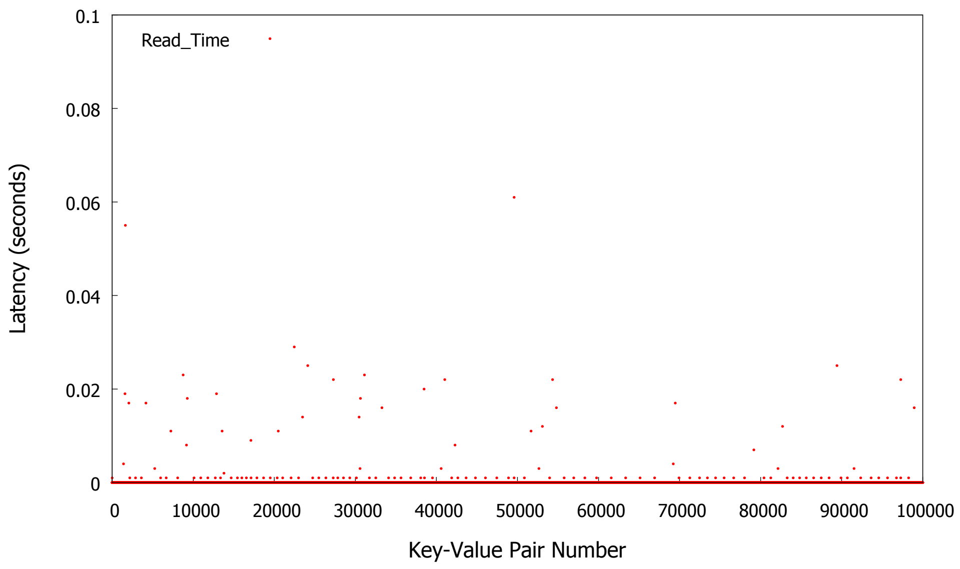


Figure 4.38: Time for Searching Keys Which Are Not Found In The LevelDB Installed Server-SAS with Sequential Order Write and Random Order Search for Keys with 8 Drive

factors [16].

We have already shown in the previous chapter that Kinetic drives can scale really well. We observed that the results of one drive could conveniently be multiplied by a factor to get the performance of multiple drives. Another factor before we designed these schemes were when a request is made then the metadata server should not return too many IP addresses of the drives. In addition to that that number of non-empty drives should be minimum. Several policies as proposed by Cao *et al* [16].

4.8.1 Indexing

There are two primary methods to design an indexing scheme - 1) Disjoint Key Ranges and 2) Joint Key Ranges. Disjoint or non-overlapping key ranges mean that drives store key-value pairs with different non-overlapped key ranges. On the other hand, for overlapped key ranges different drives would contain the overlapped key ranges. In this scenario multiple drives are needed as discussed by Cao *et al* [16].

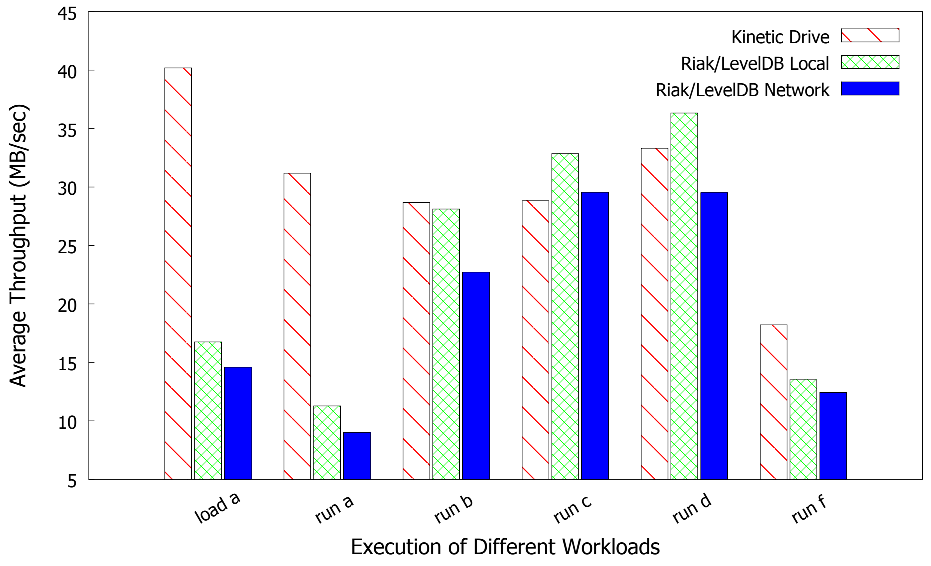


Figure 4.39: Average throughput for loading phase and running phases of different YCSB workloads using 1 MB value size on Kinetic drive vs. Riak server with LevelDB backend.

Search operation for keys is convenient in non-overlapped scenario. However, overlapped key ranges could lead to multiple drives being used. This leads to higher throughput. In the overlapped scenario, data migration could be limited. Once the drives fill up in non-overlapped scenario, data has to be migrated and this could be a huge bottleneck. An illustration of mapping of keys for non-overlapped scenario is shown in Table 4.6.

Table 4.6: Mapping of Keys on Drives

Key Range	IP Address of the Drive
0... to 001...	138.32.211.4
010... to 011...	138.32.211.5
100... to 101...	138.32.211.6

Moreover, we initialize the system by mapping keys to the drives as described in Cao *et al* [16]. With the passage of time more key-value pairs will be added to the drives. When a drive fills up then the data needs to be migrated. We transfer the

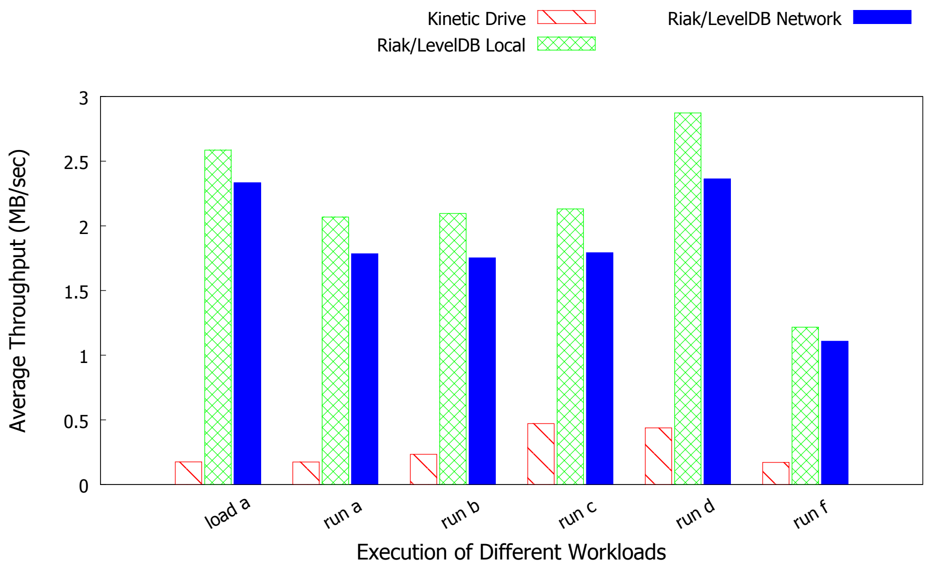


Figure 4.40: Average throughput for loading phase and running phases of different YCSB workloads using 4 KB value size on Kinetic drive vs. Riak server with LevelDB backend.

data to different drives using different schemes. Two schemes for data distribution are discussed below.

- OneToAll
- Prefix-All

4.8.2 OneToAll

In this approach, initially, all the keys are assigned to the drive 0. As the drives fill up, we need to migrate the data. We transfer half the data from drive 0 to drive 1 and split the key index into two drives. The policy is to divide the data in half and transfer it to another drive. The keys are chosen based on the ascending order of the key ids [16]. Kinetic drives can easily perform this transfer using "P2P" operation and "GetPrevious" operation. The keys starting from 0... would remain in drive 0 and the keys starting from 1... will be migrated to drive 1. Any new key that arrives and starts with 0 will be mapped to Drive 0. Moreover, if drive 1 gets filled then the data from that drive will be split and migrated to another disk. This scheme leads to large number of empty drives in the beginning. However, as the drives fill up the data gets distributed.

4.8.3 Prefix-All

This scheme attempts to reduce the data migration by evenly distributing the data across the drives from the beginning as well. In this scheme prefix of a key is used and the drive is using the mechanism described in [16] [16]. The length is decided by the number of drives in a system. For N drives in a system, we use $\log_2 N$ bits of a key as the prefix. An illustration is given below.

$$0000000...- > Drive0$$

$$0000001...- > Drive1$$

$$0000010...- > Drive2$$

In this approach, the data is separated in the beginning itself based on prefixes. The data is spread in the beginning itself and in the "OneToAll" approach, the data expands gradually. When the drives fill up, we simply adopt greedy approach to transfer the data. The drive with the most available storage space is chosen. The destination drive is filled with half the source drive's data. In this scheme the data is uniformly distributed.

4.8.4 Prefix-Half

This section considers the case when key distributions are not uniform [16]. "Prefix-Half" approach [16] introduces the trade-offs between percentage of non-empty disks and data migration amount. This approach first half drives are used to store the data. Initially, half the drives are empty. Data migration happens disks begin to fill up. When disks fill up, half the data is moved to the spare drives and key indices are split into two. This is the similarity that this technique shares with the previous technique. At the end of this process, all the drives would occupied but might not be completely filled. At this point if a drives fills up completely then the disks' contents are merged together to create free disks. The drives are chosen which have the least amount of data on them and whose key ranges are also adjacent to each other. This step is followed by updating the index table as well. This creates an empty disk and gives an opportunity to collect together different key ranges. If disks can not be found for merger then new drives are needed for the system. This technique reduces the number of non-empty disks.

4.8.5 Prefix-Half-2Drives

This technique was developed to reduce the data migration. However, this technique leads to overlapped key ranges [16]. When a disk "A" becomes full then the data movement is delayed. Instead of transferring the data right away. The new incoming data for disk A is transferred to disk B. At the end of this process the key ranges of Disk B and Disk A are overlapped. Another significant consideration is not to search too many disks. Therefore, attempt is made to have only 2 disks for searches. This approach is developed based on "Prefix-Half" approach.

4.8.6 Prefix-Half-Unknown

Final case is when the key distribution is not entirely known at the time of data distribution. In this method instead of equally dividing the data amongst the drives, the data is distributed based on approximation.

4.8.7 Results

Comparative results for all the above mentioned is shown in Figure 4.43 [16]. To make the experiment realistic, data is intentionally made non uniformly distributed. Data amount follows a normal distribution [16]. Standard deviation is used for data amount.

It could be observed that "OneToAll" approach gives the largest data migration. The other methods follow the predictions. The other approaches so reduce the data movement.

4.9 Summary and Conclusion

Based on our techniques and methodologies, it could be concluded that "OneToAll" approach has the maximum amount of migration. Other devised techniques do lead to reduction in data movement. And with the usage of normal distribution for some of the techniques, the study is able to incorporate realistic workloads.

When we compared the performance results of the Kinetic drives with SATA and SAS based servers we found Kinetic drives are CPU-bound but they give an average sequential write throughput of 63 MB/sec with average sequential read throughput of 78 MB/sec for 1 MB value sizes. It could be observed that RAM has a big impact on the performance on the server. More RAM usually translates to better performance for traditional servers.

Moreover, we were able to confirm the results of micro benchmarks with YCSB. The results confirmed that for smaller value sizes, the server performs better however, for larger value sizes, the Kinetic drives perform better as discussed in Section 4.4.

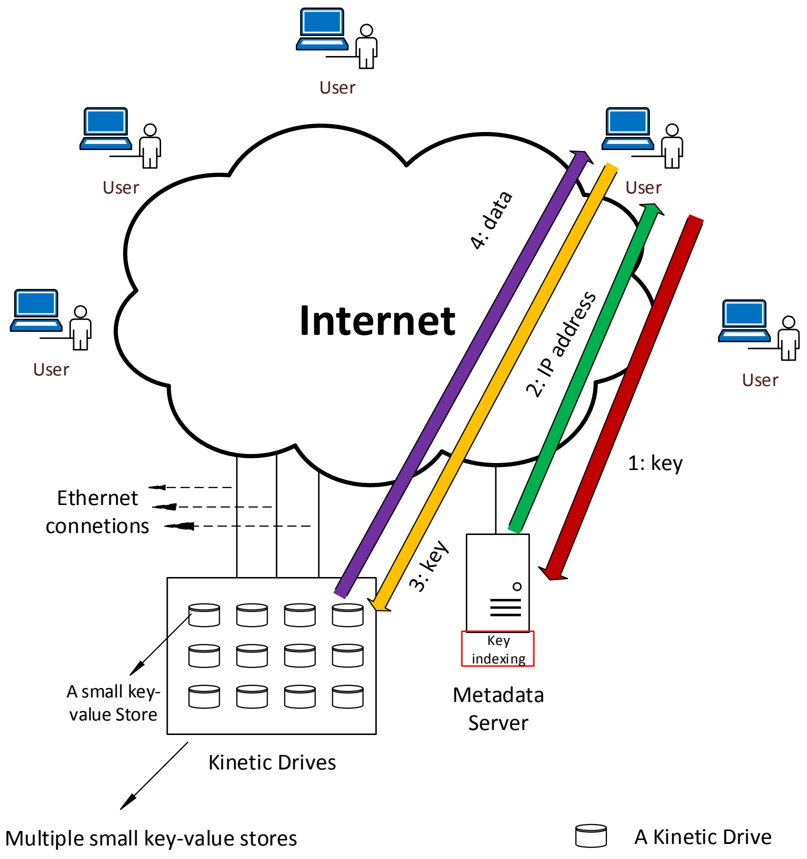


Figure 4.41: New Kinetic Key-Value Store System (Reproduced from [16] with permission)

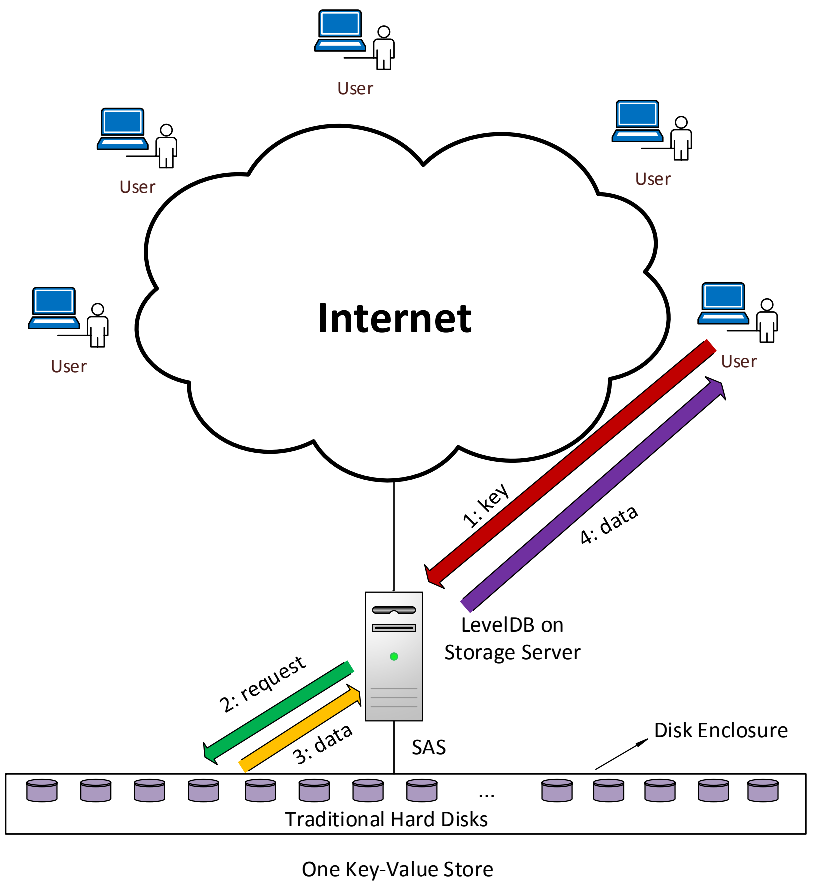


Figure 4.42: Traditional Key-Value Store System (Reproduced from [16] with permission)

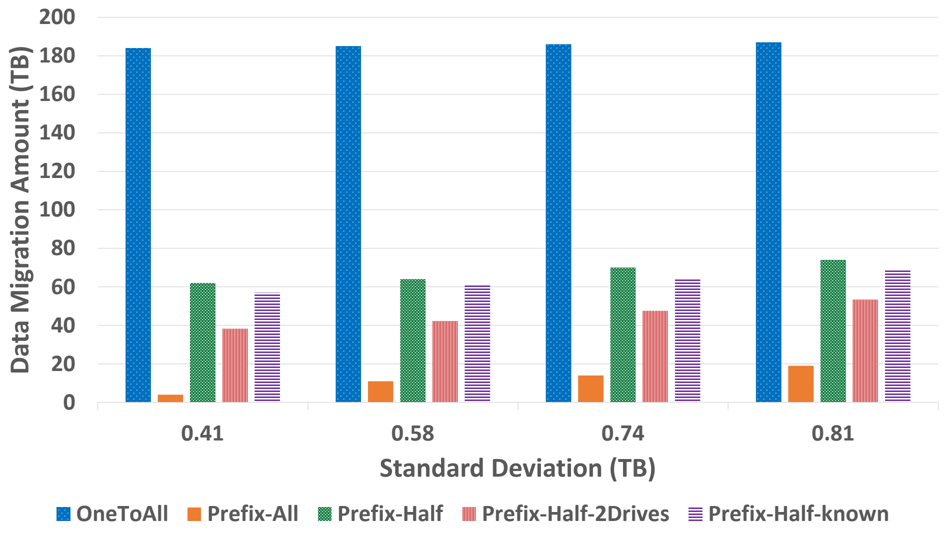


Figure 4.43: Amount of Data Migration (Reproduced from [16] with permission)

Chapter 5

Conclusion and Discussion

Based on our study of in-storage processing architectures, it can be concluded that these systems have the potential to give massive power and performance improvements. For large scale deployment they require efficient indexing schemes for mapping data to drives and for data sharing amongst these drives. Any company, desiring to choose these drives, would need to extensively study the impact of integrating these drives with their existing systems. Therefore, an API or other mechanism is required to handle or program these devices. Without easy programmability and smooth integration with the existing software and hardware platforms, acceptance rate for this technology would be low. Therefore, the companies making the hardware would have to do brainstorming before releasing any products. For products or architectures that do not blend easily with existing platforms would find severe resistance. This primarily happens because upgrading existing software is extremely expensive and time consuming. Changing several hardware components would also be extremely difficult. Samsung's attempt to include programming models in their Flash Translation Layer did not produce desired outcomes. However, the technology again proved to be promising for data intensive applications be it for machine learning algorithms or unstructured data.

When we move towards the architectural aspects of these devices, we often find that internal processors are low powered. Almost all of these devices could use improved processing capabilities. The reason for incorporating low powered processors is to bring the energy signature down. However, once low powered processors are used a usual feed-back is to increase their capability. This presented a conundrum for the manufactures to whether have a low power processor or have a high powered processor.

RAM and Cache sizes at any level of the hierarchy of the system is extremely important. Often the performance of the system was dictated by the amount of cache or RAM in the system. More RAM and cache were often translated to better performance.

Maintainability of drives is also an important issue. New devices often beg the question that would large number of drives could be maintained without loss of data? Different system architectures are required to be studied to make the hardware suitable for any particular application. Analysis of workload remains the quintessential significant task for any study. Without understanding of benchmarks, systems would not deliver the desired performance. Next comes the topic of fault tolerance and data failure. Disks are required to recover from any data failure. RAID has been used to recover from the data corruption. For fault tolerance, often multiple drives are used. However, identifying which drives have gone bad amongst thousands of drives in a data center and having a quick down time to replace these bad drives is the key. Without advanced IT knowledge, technology would not be able to deliver in a data center environment.

Security is another big concern that is required to be addressed. Drives that could easily be accessed by IP addresses could be under threat. Efficient methodology is required to create a secure system for deployment. This aspect has not been addressed for Kinetic drives by Seagate. More work is required to secure the drives from attacks.

We also found that hardware with new concepts are often interesting academic projects and bring new insights but many times industry would simply optimize the existing systems. This would be done to save time, effort, money, and often the prior knowledge of the existing know how. These factors results in slow acceptance of Kinetic or Object storage devices in the market.

In conclusion, for storage processing unit observed energy savings from 11 423X and performance gains from 4 — 66X for applications including k-means, Sparse BLAS, and others. Our sample Kinetic drives, that encapsulate the concept of in-storage processing, are CPU-bound, but they still average sequential write throughput of 63 MB/sec and sequential read throughput of 78 MB/sec for 1 MB value sizes.

References

[1] M. Minglani, A. Nagarajan, S. Deshapande, L. Everson, and D. J. Lilja. Design space exploration for efficient computing in solid state drives with the storage processing unit. In *2015 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 87–94, Aug 2015.

[2] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[3] Devesh Tiwari, Simona Boboila, Sudharshan Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 119–132, San Jose, CA, 2013. USENIX.

[4] William Harrod. A journey to exascale computing. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1702–1730, Nov 2012.

[5] Cagdas Dirik and Bruce Jacob. The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization. *SIGARCH Comput. Archit. News*, 37(3):279–289, June 2009.

[6] Peng Li, Kevin Gomez, and David J. Lilja. Exploiting free silicon for energy-efficient computing directly in NAND flash-based solid-state storage systems. *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, September 2013.

[7] Michael Walker. Structured vs. unstructured data: The rise of data anarchy. `http://www.datasciencecentral.com/profiles/blogs/structured-vs-unstructured-data-the-rise-of-data-anarchy`, 2012. Accessed: 2017-6-27.

[8] John Mallory. Save your data deep storage considerations. `http://web.stanford.edu/group/dlss/pasig/PASIG_March2015/20150313_Presentations/Mallory_Isilon.pdf`, 2014. Accessed: 2017-6-27.

[9] Derek Gascon and Greg White. Managing unstructured data in object-based storage. `http://www.dell.com/downloads/global/power/ps4q10-20100472-white.pdf`, 2010. Accessed: 2017-6-27.

[10] Joe Arnold. Kinetic motion with seagate and open-stack swift. `https://swiftstack.com/blog/2013/10/22/kinetic-for-openstack-swift-with-seagate/`, 2013. Accessed: 2017-6-27.

[11] Kinetic open storage project. `https://www.openkinetic.org/`, 2016. Accessed: 2017-6-27.

[12] Google. Leveldb - lightweight database library by google. `https://github.com/google/leveldb`, 2016. Accessed: 2017-6-27.

[13] Mayur Shetty. Seagate kinetic open storage platform. `http://www.snia.org/sites/default/files/MayurShelty_Seagate-Kinetic.pdf`, 2014. Accessed: 2017-6-27.

[14] M. Minglani, Jim Diehl, Xiang Cao, Bingzhe Li, Dongchul Park, D. J. Lilja, and David H.C. Du. Kinetic action: Performance analysis of integrated key-value storage devices vs. leveldb servers. In *2017 IEEE International Conference on Parallel and Distributed Systems*, 2017.

[15] Seagate. Seagate kinetic hdd. `http://www.seagate.com/www-content/product-content/hdd-fam/kinetic-hdd/en-us/docs/100764174b.pdf`, 2014. Accessed: 2017-6-27.

[16] X. Cao, M. Minglani, and D. H. C. Du. Data allocation of large-scale key-value store system using kinetic drives. In *2017 IEEE Third International Conference on Big Data Computing Service and Applications (BigDataService)*, pages 60–69, April 2017.

[17] M. Gokhale, B. Holmes, and K. Iobst. Processing in memory: the terasys massively parallel pim array. *Computer*, 28(4):23–31, Apr 1995.

[18] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 62–73, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[19] Kimberly Keeton, D Patterson, Joseph Hellerstein, John Kubiatowicz, and Katherine Yelick. A case for intelligent disks (idisks). *Database, 9(6 S 5)*, 1998.

[20] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. *SIGPLAN Not.*, 33(11):81–91, October 1998.

[21] Rene Mueller, Jens Teubner, and Gustavo Alonso. Data processing on fpgas. *Proc. VLDB Endow.*, 2(1):910–921, August 2009.

[22] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[23] How we made github fast. `https://github.com/blog/530-how-we-made-github-fast`. Accessed: 2017-6-27.

[24] Rocksdb. `http://rocksdb.org/`. Accessed: 2017-6-27.

[25] Amazon's Dynamo. *https://aws.amazon.com/dynamodb/*.

[26] RocksDB. *https://en.wikipedia.org/wiki/RocksDB*.

[27] M. Mesnier, G. R. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8):84–90, Aug 2003.

[28] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran. Object storage: the future building block for storage systems. In *2005 IEEE International Symposium on Mass Storage Systems and Technology*, pages 119–123, June 2005.

[29] B. Y. Zhao, Ling Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, Jan 2004.

[30] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, Feb 2003.

[31] Peng Li, K. Gomez, and D.J. Lilja. Exploiting free silicon for energy-efficient computing directly in nand flash-based solid-state storage systems. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6, Sept 2013.

[32] Intel-cofluent studio, 2014.

[33] Ashwin Nagarajan. Modeling and Design Space Exploration of Storage Processing Unit for Energy Efficiency. Master's thesis, University of Minnesota, Twin Cities, Minnesota, 2015.

[34] Intel cofluent studio virtual systems: Architecture exploration, 2014.

[35] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Chao Ren. Exploring and exploiting the multilevel parallelism inside ssds for improved performance and endurance. *Computers, IEEE Transactions on*, 62(6):1141–1155, June 2013.

[36] Krste Asanovic, Bryan Christopher Catanzaro, Kurt Keutzer, and Katherine A Yelick. The Landscape of Parallel Computing Research : A View from Berkeley. Technical report, 2006.

[37] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, second edition, 2003, http://epubs.siam.org/doi/pdf/10.1137/1.9780898718003.

[38] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007, http://www.worldscientific.com/doi/pdf/10.1142/S0129626407002843.

[39] Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, GeoffreyJ. McLachlan, Angus Ng, Bing Liu, PhilipS. Yu, Zhi-Hua Zhou, Michael Steinbach, DavidJ. Hand, and Dan Steinberg. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1–37, 2008.

[40] Jing Zhang, Gongqing Wu, Xuegang Hu, Shiying Li, and Shuilong Hao. A parallel k-means clustering algorithm with mpi. In *Parallel Architectures, Algorithms and Programming (PAAP), 2011 Fourth International Symposium on*, pages 60–64, Dec 2011.

[41] Sadegh Bafandeh Imandoust and Mohammad Bolandraftar. Application of K-Nearest Neighbor (KNN) Approach for Predicting Economic Events : Theoretical Background. *Int. Journal of Engineering Research and Applications*, 3(5):605–610, 2013.

[42] Kantabutra Sanpawat and Alva L. Couch. Parallel k-means clustering algorithm on nows. *NECTEC Technical Journal*, pages 243–247, 2000.

[43] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[44] Erik Saule, Kamer Kaya, and Ümit V. Çatalyürek. Performance evaluation of sparse matrix multiplication kernels on intel xeon phi. *CoRR*, abs/1302.1078, 2013.

[45] Openstreetmap, 2014.

[46] Shailendra Singh Raghuwanshi and PremNarayan Arya. Comparison of k-means and modified k-mean algorithms for large data-set. *International Journal of Computing, Communications and Networking*, 1(3), 2012.

[47] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R. Ganger. Active disk meets flash: A case for intelligent ssds. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 91–102, New York, NY, USA, 2013. ACM.

[48] SwiftStack Seagate, Supermicro. Reference design: Scalable object storage with seagate kinetic, supermicro, and swiftstack. `https://swiftstack.com/wp-content/uploads/2015/05/KineticReferenceDesign-SwiftStackSupermicroSeagate.pdf`, 2015. Accessed: 2017-6-27.

[49] Seagate's kinetic will impact object storage. `https://go.forrester.com/blogs/seagates-kinetic-will_impact-object-storage-and-data-driven-applications`, 2013. Accessed: 2017-9-27.

[50] Seagate. Kinetic open storage project. `"https://www.openkinetic.org/technology/kinetic-protocol`, note = Accessed: 2018-1-10, urldate=2018, year=2018.

[51] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[52] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *Trans. Storage*, 13(1):5:1–5:28, March 2017.

[53] David J Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, New York, NY, USA, 2000.

[54] Kevin Risden. Yahoo cloud serving benchmark. `https://github.com/brianfrankcooper/YCSB/wiki`, 2014. Accessed: 2017-6-27.

[55] Mapkeeper. `https://github.com/m1ch1/mapkeeper/wiki`, 2014. Accessed: 2017-6-27.

[56] Riak. http://docs.basho.com/riak/kv/2.2.0/, 2014. Accessed: 2017-6-27.

[57] Kinetic java tools. https://github.com/Seagate/kinetic-java-tools, 2014. Accessed: 2017-6-27.