Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Characterizing The Energy Contribution and Energy-Performance Trade-offs of NVMe SSDs in the Linux Storage Stack

Author: Joseph Subash Kanichai (2819675)

1st supervisor:	Tiziano De Matteis
daily supervisor:	Krijn Doekemeijer
2nd reader:	Daniele Bonetta

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

July 30, 2025

Abstract

By the year 2030, it is estimated that more than a yottabyte of data will be stored worldwide. Modern applications such as Machine Learning and AI or Scientific Computing often use NVMe SSDs, because of their microsecond-level low latency and ability to handle millions of I/O operations per second. With the global SSD market being valued at $23 billion in 2024, and estimated to reach $167 billion by 2033, it is clear that demand for high-performance storage is accelerating, driven by applications like AI. As a result, data centers are growing and adopting more NVMe SSDs. This growth, however, comes at the cost of energy, with over 1% of global electricity consumed in data centers, having increased by over 50% in the last two years alone. As a result, reducing power consumption is becoming important as data center operators around the world are working to optimize and reduce their power consumption. However, this should not be achieved at the cost of performance, as most providers have performance storage level agreements (SLAs) to uphold.

In this thesis, we create a benchmarking tool that allows us to collect energy metrics for storage to understand the energy contribution of individual hardware components in various workloads. Our tool measures the energy consumption of the SSD, CPU, and whole server, as well as recording application performance metrics such as average throughput and latency, and system performance metrics like system load, to understand the tradeoffs between performance & energy. This will help us understand how to reduce energy depending on the use case.

Our experiments show that increasing throughput also increases power consumption, and beyond a certain point, the energy savings start to level off. Setting SSDs to lower power saving modes can save energy, but leads to much slower speeds, especially when handling large amounts of data. We found that achieving the best balance between speed and energy use depends on carefully

adjusting various parameters. Based on our findings, we provide a set of recommendations for developers of two general types of applications to help make applications more energy-aware. Our findings show that at least a 25% increase in energy efficiency is possible in specialized benchmarks, and applying our recommendations to a real-world application provided a 13% efficiency gain with only minimal changes to general configuration parameters.

These insights inform practical guidelines for data center operators and application developers seeking to reduce storage energy consumption while maintaining SLAs. The tools and methodologies developed in this thesis lay the groundwork for future energy-aware I/O and storage system design, ultimately contributing to more sustainable, high-performance data centers.

The source code for this thesis are available online at
https://github.com/t348575/energy-benchmark and artifacts available at
https://github.com/t348575/ssd-energy-benchmark-artifacts.

Contents

List of Figures **v**

List of Tables **vii**

1 Introduction **1**

1.1 Problem Statement 2
1.2 Research Questions & Methodology 2
1.3 Contributions 3
1.4 Research Relevance 3
1.5 Societal Relevance 4
1.6 Thesis Structure 4
1.7 Plagiarism Declaration 4

2 Background & Related Work **7**

2.1 Flash storage & SSDs 7
2.2 Flash Translation Layer (FTL) 7
2.3 Non-Volatile Memory Express (NVMe) 8
2.4 Filesystems 9
2.5 Linux Storage Stack 9
2.6 I/O Engines 10

3 Survey of SSD Energy Research **13**

3.1 Survey Methodology 13
3.2 Energy - Performance Characteristics of SSDs 15
3.3 Effects of Internal SSD Design 16
3.4 Linux I/O interfaces – Performance & Energy 18
3.5 Linux I/O Schedulers – Performance & Energy 19
3.6 Power Measurement Techniques for PCIe & Whole System 19

3.7 Effects of CPU Performance on SSDs . . . 20
3.8 Applications & Workloads for SSD Benchmarking . . . 21
3.9 Gaps in Storage Energy Research . . . 21
3.10 Summary . . . 21

4 Benchmark Setup & Tooling **23**
4.1 Experiment Hardware & Software Configuration . . . 23
4.2 Sensors & Data Sources . . . 23
4.3 energy-benchmark Benchmarking Tool . . . 26

5 Microbenchmarks **29**
5.1 General SSD Energy Characteristics . . . 29
5.2 Request Size . . . 35
5.3 I/O Depth . . . 38
5.4 Jobs and Threads . . . 41
5.5 Interrupts and Polling . . . 43
5.6 I/O Engine . . . 45
5.7 Mixed Read-Write Workloads . . . 48
5.8 I/O Schedulers . . . 50
5.9 Request Intervals . . . 52
5.10 Summary & Recommendations . . . 54

6 Application Benchmarks **57**
6.1 Filesystems - Filebench . . . 57
6.2 RocksDB - YCSB . . . 62
6.3 MLPerf Storage . . . 65
6.4 PostgreSQL - TPC-C . . . 67
6.5 Summary . . . 68

7 Conclusion **71**
7.1 Research Questions . . . 71
7.2 Limitations & Future Work . . . 73

References **75**

8 Appendix **83**

8.1 Artifact Checklist 83

8.2 Experiment Setup 83

8.3 Running an Experiment 85

List of Figures

2.1 SSD Internal Layout 8

4.1 Samsung 980 Pro NVMe power states. 24
4.2 Samsung 980 Pro installed with modified PCIe riser to the PowerSensor3, in our server. 24

5.1 GC - 15s write test 31
5.2 GC - 60s write test 32
5.3 APST - throughput & SSD power over time. 33
5.4 Request size experiment power, throughput & efficiency 36
5.5 PS0 Throughput & SSD power - seq. reads 39
5.6 PS0 I/O depth vs. workload type efficiency 40
5.7 Jobs & Threads - Throughput & Latency 42
5.8 PS0 Interrupts vs. Polling efficiency - rand. reads 44
5.9 I/O engine throughput & Efficiency 46
5.10 PS0 I/O engine CPU freq. & load factor - seq. reads 47
5.11 Mixed workload SSD Power & Efficiency 49
5.12 PS0 Scheduler 4 KiB rand. read 51
5.13 Scheduler 4 KiB seq. read efficiency kIOPS/J 52
5.14 Request Intervals EDP & Latency 53

6.1 Filebench fileserver read throughput 59
6.2 Filebench fileserver disk throughput & function call counts 60
6.3 Filebench fileserver PS0 SSD power 61
6.4 YCSB Workload F, PS0 - Insertion 64

LIST OF FIGURES

List of Tables

3.1 Surveyed papers by topic 13

4.1 System hardware configuration 25
4.2 Software stack and tool versions used in experiments. 25

5.1 Summary of workload variable recommendations to maximize efficiency. . . 54

6.1 Filebench workload variables 58

LIST OF TABLES

1

Introduction

Today, nearly every facet of modern life is supported by computing technologies, from on-device processing to local servers, High Performance Computing (HPC) clusters, edge infrastructure, and cloud platforms. Together, these systems form a vast digital ecosystem that powers everything from data analysis and AI to e-commerce and banking. In the Netherlands, data centers and related infrastructure employ over 3.3 million people and contribute 60% of the nation's GDP (1). The cloud computing industry worldwide is valued at $676 billion (2).

Storage systems are an important aspect of these systems and services, and it is estimated that by 2030, more than a yottabyte (10^{24}) of data will be stored worldwide (3). Today, various types of persistent storage media are deployed, with traditional hard disk drives (HDDs) being the most popular, accounting for nearly 80% of data center storage capacity (4). The dominance is primarily due to the cost difference between HDDs and solid state drives (SSDs), with enterprise SSDs costing between 6 and 9 times more than enterprise HDDs (5, 6).

Regardless, the SSD market is valued at $23 billion in 2024 (7), and the increasing demand for high throughput, low latency storage is being driven by large-scale cloud services, scientific computing, and AI to name a few. In the current AI boom, ML training systems require substantial quantities of data to be continuously delivered to high-performance GPUs, not to mention checkpointing and various other tasks in the training process (8).

With data centers worldwide growing, the power consumption of these large data centers is becoming a concern, consuming 1% of worldwide power (9), 6% in the US (10), and increasing over 50% in the last two years alone (11). With a single enterprise SSD consuming up to 25 W (12), and dense storage servers having 24+ SSDs (13), the power quickly

adds up, and as such, it is important to study the energy consumption characteristics and energy-performance tradeoffs of NVMe SSDs.

In this thesis, we create a benchmarking tool to collect energy consumption metrics for the SSD, CPU & whole system. Along with performance metrics, we characterize energy contributions and energy-performance tradeoffs of NVMe SSDs in the Linux storage stack through various host-managed software configuration knobs characterized through microbenchmarks, as well as application benchmarks. We look at throughput, latency, and various other metrics to quantify the energy efficiency of storage systems. Our results show that in controlled microbenchmarks, energy efficiency can be improved by 25% or more depending on workload specifics, and applying our findings to tune real-world applications yielded a 13% energy efficiency increase in RocksDB.s

1.1 Problem Statement

Several studies have looked at the energy consumption of storage, but usually only measure whole system power or SSD power. To our knowledge, no study looks at the power consumption of both the SSD, CPU & whole system simultaneously, diving into the contribution and host configuration effects. All of the aforementioned studies also focus on a single component in the storage stack, and do not analyze the interplay between configuration knobs or components. Therefore, the goals of this thesis are to:

- Characterize the contributions of energy and performance in various components in the Linux storage stack through microbenchmarks and applications, while recording energy of the CPU, SSD & whole server.
- Characterize the energy-performance tradeoffs in modern NVMe SSDs by changing various host-managed software knobs.
- Provide general guidelines to developers, allowing various layers of the storage stack or application to consider energy efficiency during development.

1.2 Research Questions & Methodology

In order to answer the problem statement, we present the following research questions, as well as an overarching research question:

How does the configuration of various host-managed software knobs influence the energy contribution and the energy-performance tradeoffs of NVMe SSDs, CPU & whole system?

RQ1 What is the state-of-the-art & state-of-practice in SSD energy research, what are the methodologies & metrics used for benchmarking, and what are the gaps in research?

M1 Quantitative research: Conducting a survey (14).

RQ2 How do various host - manged configuration knobs derived from *RQ*1 **effect power consumption of the SSD, CPU & whole system?** How do they influence the energy-performance tradeoff & energy contributions?

M2 Experimental research: Designing appropriate microbenchmarks and workload benchmarks to quantify a system (15, 16).

M3 Open science, open source software, community building, reproducible experiments (17).

RQ3 How do various application-specific and general configuration parameters as mentioned in *RQ*2 **affect application workloads?** Do our results from *RQ*2 reflect in these workloads as well? What are the energy contributions and energy-performance tradeoffs? To answer RQ3, we use methodologies M2 (experimental research) and M3 (open science practices).

1.3 Contributions

There are three main contributions in this thesis:

- Energy contributions and energy-performance tradeoffs when varying host-managed configuration knobs
- How do the aforementioned configuration knobs affect energy contribution and energy-performance tradeoffs in applications
- Guidelines & recommendations to developers to consider energy when building systems & applications

1.4 Research Relevance

Considering the evolution of storage technologies in the last two decades, particularly in NVM flash storage, its performance benefits have transformed data centers. However, most existing research has focused on performance, and energy has taken a back seat.

This thesis will help fill this gap in energy research in NVMe storage, as well as the unexplored energy-performance tradeoff gap. Using state-of-the-art PCIe power sensors and a host of benchmarking platforms & tools, we hope to better understand energy-performance tradeoffs and provide insights to improve efficiency and allow energy savings in data centers.

1.5 Societal Relevance

With data centers now accounting for a significant and growing portion of global electricity usage, with data centers consuming over 1% of energy worldwide (9) and 3.3% in the Netherlands (18). With storage being a major contributor to data center energy (10), as well as society's growing needs and reliance on technology that is supported through large data centers, it is important to help reduce the environmental impact of data centers. As highlighted in the CompSysNL manifesto (1), addressing the energy footprint of our computer systems and significant advances in energy-efficient systems are required to make our digital infrastructure sustainable for the future.

1.6 Thesis Structure

In chapter 2, we provide some background on Flash Storage, NVMe, and the Linux storage stack. In chapter 3, we survey SSD energy research to discover known energy-performance characteristics of SSDs, the effects of internal SSD design on energy, the effects of various Linux storage stack components on energy and discuss some of the gaps in current research. In chapter 4, we discuss the experiment setup & configuration as well as the sensors used and our benchmarking tool. In chapter 5, we conduct microbenchmarks using fio to discover energy characteristics and energy-performance tradeoffs, as well as derive a table of recommendations for developers. In chapter 6, we apply our findings from chapter 5 to real world applications like Filesystems, RocksDB, and PostgreSQL and understand the energy-performance tradeoffs. Finally, in chapter 7 we conclude our thesis and answer the research questions and list some limitations & the possible future work. In chapter 8, we describe how to recreate our experimental setup to accurately reproduce this work.

1.7 Plagiarism Declaration

I confirm that this thesis is my own work, and is not copied from any other source (e.g. a person, the internet, or a machine) unless explicitly stated. This work has not been sub-

mitted for assessment anywhere else. I acknowledge that plagiarism is a serious academic offense that should be dealt with if found.

2

Background & Related Work

2.1 Flash storage & SSDs

Flash storage is a solid-state storage medium, comprised of floating gate transistors. These are similar to regular transistors found in CPUs and other microchips, except they retain their state after being powered off. These transistors, referred to as “cells“ are usually arranged together to form a NAND array and further grouped into blocks to form a Flash Package. Together, one or more Flash Packages are connected to a microcontroller, which sends ‘read‘, ‘write‘, and other instructions to the flash packages Figure 2.1. Because SSDs do not contain any moving parts they offer much lower latency than HDDs, and since SSDs can contain multiple flash packages, which are further split into pages, SSDs contain a lot of internal parallelism, and as a result usually offer much higher throughput.

2.2 Flash Translation Layer (FTL)

Hard disks use standard Logical Block Addressing (LBA) to address sectors, and therefore do not usually contain any additional driver or controller logic specific to each HDD manufacturer. However, SSDs do not do this, and as such, each SSD model & manufacturer can use different techniques to map a logical address to a physical address on a Flash Package. This mapping is done by firmware on the microcontroller inside the SSD, and is known as the FTL. FTLs often employ a variety of techniques to maximize parallelism to maximize throughput.

Apart from mapping, FTLs often perform various other tasks such as:

- Wear leveling, to ensure Flash Cells are written to evenly, because they can only perform a limited number of Erase operations before they stop working.

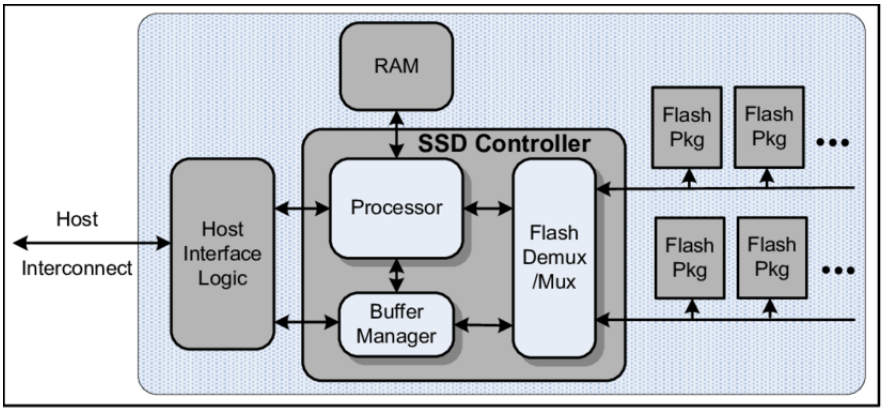


Figure 2.1: SSD Internal Layout

- Garbage Collection (GC), Flash Cells need to be erased before they can be overwritten, so the FTL might temporarily write at one particular location and later move it once the erase operation is complete.

- Managing a DRAM cache, many higher-end consumer SSDs & enterprise SSDs contain a small DRAM cache for quicker reads & writes.

2.3 Non-Volatile Memory Express (NVMe)

NVMe is a protocol and a set of standards designed for non-volatile memory. NVMe can operate over several transports such as PCIe and TCP, and is used in high-performance SSDs. SSDs using this protocol are available in many form factors, such as M.2 and U.2. Unlike older SATA and SAS interfaces, which were limited to $6\,Gbps$ and $12\,Gbps$ respectively, NVMe is only limited by its physical interface, PCIe or the network connection for NVMe-oF (19). NVMe was originally designed to overcome the shortcomings of AHCI/SCSI (older protocols used by SSDs and HDDs), to fully extract the performance available on modern SSDs. NVMe utilizes separate command and completion queues with up to 65536 command queues with 65536 entries each, while AHCI is limited to just 32 entries per device.

Current generation PCIe 5.0 SSDs can provide as much as $14\,GB/s$ of sequential throughput and as much as 2.5 million IOPS of random throughput (20). Modern enterprise NVMe SSDs can consume 25 W (12), with new standards like the Enterprise and Data Center

Storage Form Factor (EDSFF) allowing SSDs to consume up to 70 W (21). While consumer NVMe SSDs consume less energy than hard disks, they have a higher energy density than hard disks.

NVMe Power States. The NVMe specification contains power management features that were originally designed to allow data center operators to limit power for thermal management as well as increase the drive's life (22). Manufacturers can define up to 32 Active Power states, defined as the largest average power in a 10 second window. Each SSD reports a Power Descriptor table Figure 4.1, which lists the maximum power for this state, as well as entry latency, exit latency, and relative throughput & latency. SSDs can optionally also support Autonomous Power State transition (APST), which, when enabled, allows the drive to automatically switch between power states. Some power states are non-operational and intended as sleep states; hence, I/O operations cannot be performed when in these states.

2.4 Filesystems

Filesystems are system software to help organize, store, and retrieve files. Filesystems come with a variety of features, with some modern filesystems offering features like snapshots, encryption, journaling, compression, and many more. The Linux kernel by default offers support for many filesystems: local, networked & virtual. Some important filesystems include:

- EXT2, EXT3, EXT4. This is the classic 'default' Linux filesystem, and hence the most common on standard Linux installations.
- XFS. Designed for scalability, and offers improved performance for more parallel workloads and large files, as well as improved durability because of metadata journaling.
- BTRFS. This is a copy-on-write (CoW) filesystem, with features like volume management that allow running a single filesystem over multiple drives.

2.5 Linux Storage Stack

The Linux storage stack is a set of components that manage I/O from your application till your storage device. At the top of this is the Virtual Filesystem (VFS), which combines

multiple physical filesystems as well as raw I/O (i.e., without a filesystem) into a single interface. Filesystems in the VFS also make use of the OS page cache to cache frequently used data. From the VFS, I/O requests enter the block layer. Here, requests enter Linux's multi-queue block layer, where an I/O scheduler can reorder, merge, and perform other operations on requests to achieve a variety of goals, from providing priority to certain applications to providing fairness. The block layer then dispatches these requests to a hardware dispatch queue, where a device driver then sends the I/O request to the device.

2.6 I/O Engines

I/O engines are software components responsible for managing how I/O requests are handled between user applications and the kernel, before passing through the rest of the Linux storage stack. The choice of I/O engine can have major implications on application design & performance. They are of two general categories: Synchronous & Asynchronous. The former of which can only complete one request at a time, blocking until it completes. This simplifies the design but will result in much lower throughput. Asynchronous I/O engines will allow multiple I/O requests to run simultaneously, overlapping compute and I/O. Not all I/O engines interact with the same kernel interface. Posix I/O engines interact with the VFS, while libaio interacts with the Linux AIO subsystem, and io_uring with the io_uring subsystem.

I/O Interface design. I/O engines are designed with various philosophies. Most I/O interfaces copy data between userspace and kernelspace, vastly increasing CPU & memory cost. A modern interface like io_uring mitigates this by using a shared memory ring, preventing unnecessary copies. Apart from copying, another performance issue can be context switches. To avoid this, another design choice is whether to use polling or interrupts. In polling systems, a dedicated thread (or multiple threads) is used to constantly check for completed I/O, whereas interrupt driven designs rely on both software & hardware interrupts to invoke a method when I/O completes. As a result, interrupt driven designs undergo numerous context switches, which lead to considerable performance costs.

To solve both these issues, fully userspace storage stacks such as SPDK (23) have emerged. These sit fully inside userspace and provide their own components: an I/O interface, a Scheduler, and a device driver. As a result, they deliver much higher throughput & lower latency than any interface that interacts with the kernel. However, these come at the cost of development & maintenance complexity, as well as the inability to use

off-the-shelf solutions for various problems, e.g. standard filesystems cannot be used with SPDK.

3

Survey of SSD Energy Research

Below we look at the findings from our survey on the current state of SSD energy research—the state-of-the-art, state-of-the-practice and key topics.

3.1 Survey Methodology

Table 3.1: Surveyed papers by topic

Topic	**Paper(s)**
General power & performance characteristics	(12, 24, 25, 26)
Effects of internal SSD design	(27, 28, 29, 30)
I/O Interfaces	(23, 31, 32, 33, 34)
Schedulers	(35, 36, 37)
Other considerations & effects	(38, 39)

We conduct a survey to answer RQ1:

> What is the current state-of-the-art & state-of-practice in SSD energy research, what are the methodologies & metrics used for benchmarking, and what are the gaps in research?

We performed a 'snowballing' technique from the following seed papers: (12, 28, 31, 32, 35, 36). From these seed papers, Xie et al. (12) is the most relevant because of its recency (published 2024), and suggests some of the energy-performance tradeoffs when altering NVMe power states section 2.3, and therefore directly influences our research.

Additionally, we scraped popular conferences with storage studies for relevant recent papers: HotStorage, Hotcarbon, Systor, USENIX ATC, OSDI, ACM Transactions on Storage, MSST, CCGRID, IEEE SOC. Finally, we used Google Scholar to search for studies with the following keywords: NVMe, SSD, energy, storage, and power. Studies were sorted by "relevance", because sorting by date flooded the results with studies from unrelated fields outside computer science. Papers with the following subtopics were included in the survey: NVMe, storage energy, and storage interfaces. Finally, the following general inclusion and exclusion criteria were used:

I1 Contained at least one of the keywords: NVMe, SSD, energy, storage, power.

I2 Conducted within the last decade (2015-2025).

- SSDs used in studies before 2015 were often very early SSDs with early NAND technologies, and therefore do not reflect the performance expected from SSDs today. Older studies were only included if they provided relevant information not provided by any newer study.

E1 Studies that did not provide energy or performance results for storage.

E2 Studies RAID, or networked storage.

- Omitted for brevity, as these exhibit completely different characteristics and require their own, separate study.

E3 Does not study SSDs.

- Other storage media are not the focus of this study, due to different performance & energy characteristics.

E4 Papers studying interconnects or SSDs over an interconnect (e.g. Thunderbolt, Fibre Channel, RoCE, networks).

In total, 18 papers were selected and categorized based on their primary topic Table 3.1. Below, we will go through the findings from our survey, and finally discuss the current gaps in storage energy research section 3.9.

3.2 Energy - Performance Characteristics of SSDs

In this section, we will discuss the foundational insights into SSD energy-performance such as differences in read and write operations, power state transitions, and finally make some comparisons to energy in HDDs.

Firstly, Xie et al. (12) studied the energy characteristics of SSDs when under sequential and random I/O. They found that SSDs can switch between active power states in just a few milliseconds, and that modern SSDs have a large dynamic power range, often more than 60%, which is considerably more than early SSDs. SSDs also have a noticeable read-write energy asymmetry because writes require an Erase operation first, but write amplification also increases this asymmetry (24). In modern SSDs, Harris et al. (24) have shown that amplification can differ greatly between SSDs, but requires more research to better understand some of these internal effects. In early SSD designs, these effects were greatly exaggerated with writes consuming 18x more power than reads, showcased by writes having a latency of 900 μs while reads took only 50 μs (28). When Xie et al. power-limited their SSD by setting an NVMe power state, Sequential read performance stayed identical at power states with a 50% lower rated power, while Sequential write performance showed a 26% drop in throughput for the same 50% drop in rated power. As expected, lower I/O depths and I/O request sizes consume lower power, because of the reduced internal parallelism (29). Similarly, Bryan et al. (24) show that since NVMe device latencies are low, software pressure tends to increase, and overall energy consumption increases as a result. The general tradeoff is that higher throughput comes at the cost of increased power consumption.

NVMe SSDs can optionally support APST, which, when enabled, allows the drive to switch between power states in a manner predefined by the manufacturer. Hanukun et al. (37) investigated the effects of APST and concluded that it has negligible or no effect on the actual power consumption of the whole system when enabled. This is because APST transitions happen after a host configurable idle time has passed, meaning even a singular I/O or background tasks like GC can force the SSD back into a high-power state.

Unlike SSDs, HDDs take time to spin-up and spin-down, and as a result, implementing power states into HDDs would result in large entry and exit latencies. SSDs have a lower peak power than SSDs, especially with modern SSD standards like EDSFF allowing a maximum of 70 W. However, HDD and SSD idle power can often be similar (37). Understanding idle power and optimizing for idle power is important, since many storage deployments, such as tiered storage solutions, have lots of idle time. Because Flash storage

is generally much more space efficient compared to HDDs, they have a much higher energy density (J/cm^3), which is important, since a major application of NVMe power states in data centers today is thermal management (28).

Studies looking at energy use **IOPS/J and bytes/J as the main two metrics for energy efficiency** (24, 28, 31, 37). The former is better for characterizing workloads with lots of individual I/O: small request sizes, parallel workloads, or high I/O depth, while the latter is usually better for characterizing workloads with large request sizes and continuously high throughput.

3.3 Effects of Internal SSD Design

Here, we will look at various SSD design decisions in terms of both hardware and software, and how these affect energy and energy-performance tradeoffs.

Cho et al (28) & Bjorling et al. (27) studied the internal design SSDs. As discussed above section 2.2, the FTL is a piece of firmware directing and managing I/O requests on a microcontroller on the SSD. The controller is connected via one or more channels to the Flash packages. A channel is like a bus, used to relay information to Flash packages & the controller. Notably, multiple packages could be connected to a single channel in parallel, known as "ways". Depending on the exact setup, the controller might be able to access multiple Flash packages simultaneously, or in an interleaved fashion. Finally there are "planes", which exist inside a NAND Flash package, and allow individual dies inside a package to perform certain operations in parallel (e.g. reads & writes). An Ideal SSD would be able to address every NAND array individually, but various physical as well as software overheads make this unfeasible.

Increasing the number of channels & Flash packages will drastically increase the theoretical maximum throughput. The actual throughput is usually lower than the theoretical maximum because the FTL needs to be able to parallelize the incoming workload for the specific hardware configuration of the SSD. Increasing the number of channels & Flash packages will increase SSD power, which may not be linear and depends entirely on the exact configuration. However, this also presents opportunities to reduce power by turning off Flash packages or channels that are not in use. Different SSDs have different channel, "way" and "plane" configurations, and as a result, can exhibit different power characteristics. Today, however, most SSDs designed for the same use case will have similar configurations, a result of the general maturity in the SSD space. In general, as internal parallelism increases in an SSD, peak power should also increase, Balgeun et al. (29).

In FTL design, there are various techniques for mapping logical addresses to physical addresses, some of the most common ones are:

- **Page mapping.** A one-to-one mapping is done, so every logical address is mapped to a physical address. This has high DRAM usage because every page's mapping needs to be stored, but this allows for maximum flexibility in placing data, and therefore minimizes write amplification. The tradeoff between DRAM energy costs vs. write amplification costs might seem critical, but for just a 512 GB SSD, the mapping table will be 512 MiB, as a result page mapping is usually not a feasible solution.

- **Block mapping.** Instead of mapping pages, blocks are mapped, so even if a single page is updated, the entire block needs to be updated. The simplicity reduces DRAM usage, but increases write amplification and is therefore inefficient in terms of both performance & energy.

- **Demand-based FTL (DFTL).** A hybrid of page & block mapping, with low DRAM usage and similar performance to Page mapping, except when cache locality is poor. This is also more energy efficient, because of lower table management overhead & lowered DRAM usage.

As shown by Cho et al. (28), DFTL is overall the most efficient mapping scheme. However, in practice, FTLs will take various other factors such as increasing parallelism, wear leveling, GC, DRAM caches, mixed NAND Flash types, etc. It is important to consider the effects of GC when benchmarking SSDs, since it can interfere with results but usually is only invoked after write operations (27). An unintended side-effect of mapping schemes is fragmentation, however this is usually good for SSDs, since this helps fragmentation of data across multiple NAND chips & dies will increase parallelism in a somewhat natural manner (30).

Because modern SSDs can consume 25 W, and they have a much higher energy density compared to HDDs (28), SSDs can start to become thermally limited under extreme conditions, which is exacerbated since SSDs are usually passively cooled (30). When thermally limited, SSDs will perform various actions such as reducing the frequency and voltage of the microcontroller, turning off Flash packages, etc. As a result, SSD manufacturers and data center operators need to ensure their SSDs run within their ideal operating temperature even when under sustained loads.

The metrics used in these studies include IOPS/J, J, and latency/J.

3.4 Linux I/O interfaces – Performance & Energy

Here, we look at the implications of I/O interface design: Synchronous and Asynchronous, Interrupts and Polling, Userspace I/O and Kernel I/O, to understand how they affect energy.

As discussed earlier, SPDK is a userspace I/O stack and not just an interface. Removing Linux kernel overhead makes SPDK much more performant than kernel I/O interfaces, as shown by Sunder et al. (32) and consistently achieves the highest throughput (36). However, SPDK also has the highest CPU power among I/O interfaces. SPDK makes various design choice to maximize performance: only runs in an asynchronous polling mode, manages per-thread lockless queues, an inbuilt zero-copy NVMe driver, maintains request data within the context of the same thread to avoid locks (23). SPDK often consumes too much power, reducing its efficiency. In terms of latency, polling in userspace allows SPDK to achieve very low latencies, often 2x lower than other interfaces.

In comparison, io_uring is the last 'modern' I/O interface that has been added to the Linux kernel in version 5.1. io_uring is an asynchronous interface that maintains separate submission & completion queues in the form of a shared ring buffer, allowing access from userspace & kernelspace - eliminating unnecessary memory copies. Didona et al. (34) & Ren et al. (33) studied io_uring performance, showing greater throughput when compared to older Linux I/O interfaces. Unlike SPDK, io_uring can function in an Interrupt mode as well as a Polling mode. Tests show that io_uring requires 2x the CPU cores to match SPDK's throughput.

In general, polling interfaces are more efficient when more requests need to be processed, and can keep the polling thread busy, i.e. small request sizes (4 KiB) and / or high I/O depths. Large request sizes have higher latencies leaving the polling thread idle, which is wasteful and drops efficiency (31, 33). Sunder et al. (32) suggested that a hybrid solution switching between polling and interrupts would allow for a more energy-aware I/O interface.

We found the performance metrics used here are throughput, in MB/s, latency in μs or ms and $IOPS$. The tradeoffs here do not involve the SSD, they come from nuanced specifics of how workloads interact with the I/O interface and affect CPU power.

3.5 Linux I/O Schedulers – Performance & Energy

Similar to I/O interfaces, we find the important performance metrics to be throughput, latency and operations per second. Ren et al. (36) & Whitaker et al (35) studied the impacts of schedulers, and we find that modern NVMe SSDs offer higher performance compared to early SATA/SAS SSDs, to the point where saturating a modern SSD with only a single CPU core can be difficult. By default in Linux, NVMe SSDs use the None scheduler (i.e., unscheduled). At lower queue depths (2 or less) schedulers such as BFQ, MQ-deadline and kyber have broadly similar performance. At higher I/O depths contention can cause a drop in performance (in some storage stacks) (26). As the number of processes increased and the SSD started to become saturated, performance started to drop on all the schedulers, but remained good with None. BFQ & MQ-deadline performance degradation is because they spend a lot of time in locking and synchronization. Of all the schedulers, Kyber provides the best performance. In general, schedulers play less of a role in throughput driven workloads. Also, scheduler performance drops as CPU utilization nears 100%, a clear sign that while overhead is not very large, some overhead still exists.

3.6 Power Measurement Techniques for PCIe & Whole System

There are various techniques used for measuring the power consumption of NVMe SSDs. Power is measured in Watts (W), and energy in Joules (J). Most previous works only had a single power measurement setup.

For measuring the power of the entire system, an off-the-shelf power meter is attached to the system's power supply unit (24, 31, 32, 35, 37). However, there are numerous issues with measuring power using an external power meter. Firstly, power supplies provide a few different voltages: 12 V, 5 V, 3.3 V. As a result of conversion to each of these voltages, as well as conversion from AC to DC, conversion efficiencies hide the true power consumed by these devices. Furthermore, efficiency is not constant but varies based on actual Load, temperature and even the age of the power supply. Most power supplies are rated using the 80 PLUS set of standards, but it does not enforce a specific efficiency curve. Apart from conversion losses, various other issues exist such as Losses due to heat and power factor of the input AC current. Furthermore, whole system power will include other components that consume power and are irrelevant to the study, such as choice of number of CPU & Case fans, other components not studied but connected to the setup like HDDs, GPUs,

etc., Various other features specific to the motherboard used such as extra PCIe switches, IPMI, etc. However, whole system power does still provide useful insight, and will capture trends.

To measure SSD power, the most popular method was adding a shunt (resistor) and then measuring the voltage drop to calculate power. The following studies use this method: (12, 27, 29, 30, 39). As long as high quality components (voltage sensors, ADCs) with a high accuracy rating and sampling rate are used, the power results gathered from this method are good, but not excellent as seen in (37, 38). These imperfections are attributed to heat losses in the resistor, voltage drops induced in the circuit by adding the shunt, as well as EMI interference from PCB traces, wires or other high current components nearby. The studies listed above did not always perform readings in the same way, some used oscilloscopes while others added microcontrollers to perform uniform sampling and reported data via a serial interface.

Apart from physical devices, some studies (25, 39) used various software based power reporting tools to extract CPU power. These tools rely on Intel RAPL, which uses power modeling techniques to accurately estimate CPU & DRAM power (40).

As explained in (12, 27), SSDs are complex devices with various layers of software and hardware, so performing energy consumption or performance simulations can be hard.

No study looks at CPU, SSD & whole system power simultaneously, which is an important gap in research since simultaneous power readings could lend great insight into energy contributions of the CPU & SSD, and what energy-performance tradeoffs exist.

3.7 Effects of CPU Performance on SSDs

With current generation data center SSDs topping 14 GB/s of sequential read throughput, CPUs are sometimes struggling to keep up with SSDs. In systems with multiple NUMA nodes (i.e. multiple CPUs on the same motherboard), which is common in data center and HPC environments, each CPU does not have access to all peripherals on the system as this would lead to contention issues. Instead, each CPU is responsible for a set of peripherals. This includes DRAM and PCIe lanes. If something run on one NUMA domain needs to access a peripheral in another NUMA domain, the requests go through the NUMA domain of the peripheral and introduce extra latency & overhead. Preventing cross NUMA node accesses to an SSD can have as much as a 50% drop in energy (38).

In various non-standard environments such as in low power system-on-chip's (SOCs), the added weight of I/O interfaces & protocols has detrimental effects to both performance &

efficiency (39). While this research was not targeted at traditional storage servers, it can serve as an indicator of poor performance & energy efficiency when the host CPU is fully utilized while also performing I/O.

3.8 Applications & Workloads for SSD Benchmarking

The current state-of-the-practice for benchmarking storage is fio (41). Fio allows controlling many variables such as I/O engine, read-write mix, parallel jobs, queue depth, and many others. This allows us to create identical and repeatable experiments. The majority of NVMe SSD papers today for both energy and performance utilize fio: (12, 23, 24, 25, 31, 32, 33, 34, 35, 36, 38, 39), and studying their experimental configurations for workloads will help us decide on appropriate configurations for our experiments.

3.9 Gaps in Storage Energy Research

No single study records CPU, DRAM, SSD & system power. All studies in our survey stick to a single source for power, either whole system, or CPU or the SSD. Recording all the previously listed metrics can provide useful insight into the interplay between performance & energy of all the above components. Studies also tend to focus on a single element in the software storage stack: Low level components, I/O interface, Scheduler or general energy characteristics. Just like with recording power, a study combining these might help to uncover any energy-performance trade-offs present. There is a lack of application level benchmarks. Most studies focus on micro benchmarks using fio and leave out application level benchmarks.

3.10 Summary

We answer RQ1 as follows: The current state-of-the-art in SSD benchmarking and energy storage research focuses primarily on NVMe SSDs, which offer higher performance than older SATA/SAS SSDs, and display large dynamic power ranges reaching 60%, consuming up to $25W$. Synthetic benchmarks are conducted using fio (41), while higher level application benchmarks are sparse in current studies. Energy efficiency is usually measured using $IOPS/J$, and power measurements are performed on the entire system using off-the-shelf power meters, or custom solutions by placing a shunt and recording voltage drop across it, with most studies only recording power at a single point (e.g., whole system, SSD). I/O interfaces in the kernel or user space have effects on energy efficiency, and techniques like

polling or interrupts also affect efficiency. Read-write energy asymmetry as well as energy differences based on the specific I/O (chunk size, I/O depth, etc.) point to the clear existence of an energy-performance tradeoff. We found that the performance metrics used are MB/s for throughput, μs or ms for latency and $IOPS$. Therefore in our research, we are going to look at Linux stack parameters: Request size, queue depth, threads, random & sequential access, I/O engine, polling & interrupts, I/O scheduler, Idle & GC while varying NVMe device power state.

4

Benchmark Setup & Tooling

Based on the research questions section 1.2 and our survey chapter 3, we explain our experimental setup, as well as how we built a benchmark tool to interact with various sensors & systems to record SSD, CPU, whole system power alongside various metrics.

4.1 Experiment Hardware & Software Configuration

All experiments were run on the same system, with no base software or hardware changed between tests, as shown in Table 4.1 and Table 4.2. The system is running with hyper-threading disabled. The default power management of the system was left enabled, allowing the CPU to reduce power during idle.

The NVMe SSD used for this study is a 1 TB Samsung 980 Pro, a PCIe 4.0 SSD, but our system is limited to PCIe 3.0 with PCIe ASPM being disabled. The SSD features a 1 GiB DRAM cache, and the NAND is split into two dies, consisting of TLC cells. The SSD also features a 114 GB SLC cache (42). It is connected to NUMA node 1. The SSD has 5 NVMe power states, 3 are operational and 2 or non-operational (i.e. sleep states). The maximum power rating, as well as relative read/write latency and relative read/write throughput, and time to enter and exit these states is reported Figure 4.1. In this document, these states will be referred to using their index as listed in Figure 4.1, PS0 being the most powerful and PS4 being the least powerful state.

4.2 Sensors & Data Sources

To record energy & performance characteristics of our benchmarks, we used numerous data sources. Some of these data sources such as CPU frequency and CPU Load, are not

```
Supported Power States
St Op     Max   Active     Idle   RL RT WL WT  Ent_Lat  Ex_Lat
 0 +     8.49W       -        -    0  0  0  0        0       0
 1 +     4.48W       -        -    1  1  1  1        0     200
 2 +     3.18W       -        -    2  2  2  2        0    1000
 3 -   0.0400W       -        -    3  3  3  3     2000    1200
 4 -   0.0050W       -        -    4  4  4  4      500    9500
```

Figure 4.1: Samsung 980 Pro NVMe power states.

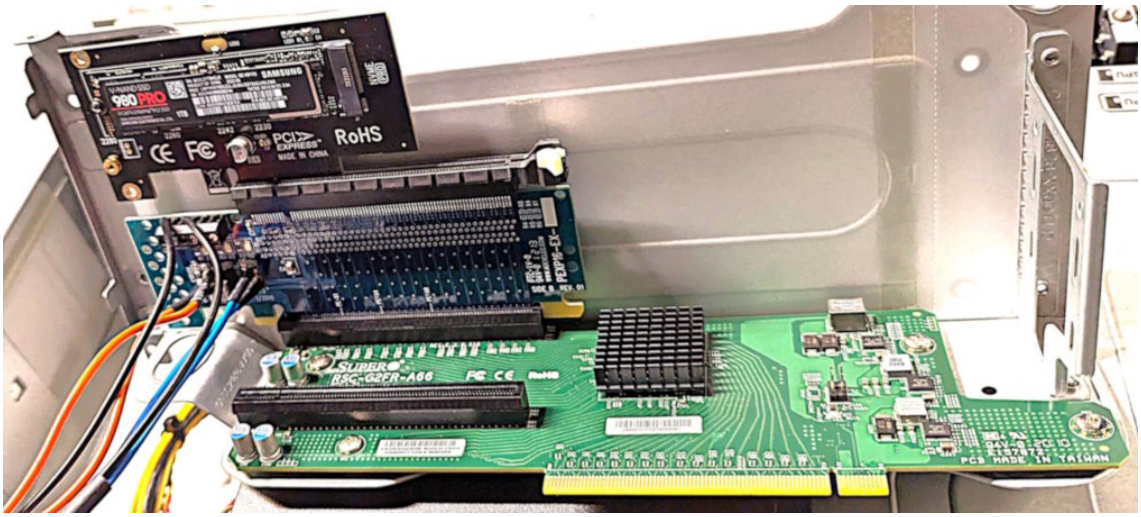


Figure 4.2: Samsung 980 Pro installed with modified PCIe riser to the PowerSensor3, in our server.

Component	Specification
CPU	2 × Intel Xeon Silver 4210R (10 core)
Memory	256 GB (4 × 64 GB) – 2400 MHz DDR4
Test NVMe drive	1 TB Samsung 980 Pro
Motherboard	Supermicro X11DPG-SN

Table 4.1: System hardware configuration

Software	Version (s)
Linux distro	Ubuntu 22.04.5 LTS
Kernel	5.15.0
Docker	27.3.1
fio	3.38
Filebench	1.4.9.1
RocksDB (YCSB)	6.2.2 (0.17.0)
MLPerf	2.0.0b1
Postgres (TPC-C)	16 (git tag 0927c44)

Table 4.2: Software stack and tool versions used in experiments.

performance metrics, but instead can be used to explain or verify observed characteristics, e.g. Lower CPU Load will usually result in lower CPU power.

PowerSensor3. In order to accurately measure our SSDs power with a high precision of 1 ms, we used the PowerSensor3 (43) sensor built by Astron. The 1 ms precision will help us identify energy characteristics that happen at small intervals, as seen in section 6.2. Unlike shunt based power sensors that other studies used, the PowerSensor3 uses a Hall effect sensor and an optically isolated voltage sensor to record current & voltage - making them immune to interference, voltage drops, and heat issues that other power sensors have to deal with section 3.6. Figure 4.2 shows our Samsung 980 PRO NVMe SSD connected through a PCIe riser, with additional wires for the sensor to read the 3.3 V, 12 V PCIe rails. The sensor is connected to the server through a USB cable, and provides a serial interface, and the provided host library can be used to interact with the sensor.

RAPL. Intel's Running Average Power Limit (RAPL) has been used to measure CPU power consumption. This system is limited to Intel CPUs, and uses various power and en-

ergy models to estimate energy consumption, which can then be accessed through Model Specific Registers (MSR). Intel tunes the consumption model specifically for each CPU model to ensure good accuracy. While initial CPU models with RAPL were not very accurate, RAPL provides very reliable energy consumption figures in current CPUs (40). For our measurements, we used the powercap interface provided by Linux, which provides energy readings at `/sys/class/powercap/intel-rapl:*/energy_uj` for the CPU & DRAM in each NUMA domain.

Whole system power. Our server is connected to a Netio PowerPDU 4KS which records power and is accessible via an HTTP interface, which we can poll at an interval of 500 ms, since it updates its counters at this interval.

Sysinfo. To extract other general system information, such as CPU clock speed, system load, and memory usage, the popular sysinfo library was used. On Linux, it utilizes standard statistics populated by the kernel, and are accurate enough for our needs.

I/O statistics. Actual disk I/O statistics are also recorded using Linux's I/O statistics, which is updated by the kernel immediately on every single I/O, which is useful when applications benchmarks do not report actual disk throughput.

Function call stack traces When running application benchmarks, certain performance & energy characteristics are difficult to understand without intimate knowledge of application internals. In order to explain some of these observed characteristics, we used Linux's bpftrace. Instead of looking for application or even system level function calls, we find call stacks that arrive at the NVMe driver. This is because different applications & filesystems will use different sys calls, and tracing all of these is arduous. We trace and print the call stacks reaching *nvme_setup_cmd*, an internal function in the NVMe driver which is where commands are prepared before being sent to the NVMe device, so all true I/O calls will pass through *nvme_setup_cmd*. We print the entire call stack as well as the number of times this call stack was invoked every second. This trace data can then be parsed and filtered to look for specific calls like *ext4_writepages* or *blk_mq_requeue_work*.

4.3 energy-benchmark Benchmarking Tool

Because of the number of experiments being conducted, and the amount of data being generated, it makes sense that a tool should be built to better coordinate the running of

benchmarks & data collection from sensors as well as cataloging them together - allowing them to be easily interpreted. To do this, we built a tool that integrates & records data from all our sensor sources. While packages exist in languages such as Python to interact with RAPL, the tool was written in Rust for two main reasons: It is compiled to machine code without a runtime, so is just as lightweight as a C/C++ application and as a result will minimize overhead and not interfere with our experiments. Also, its simple package manager and library ecosystem make it very simple to add and use libraries to simplify tasks such as parallelizing tasks during plot generation as well as easily parsing and managing data in JSON & CSV.

The benchmark tool was built with three main components: Benches (benchmarking tools), Sensors, and Plotting tools. Interfaces are used to allow it to be easily extended to add other new sensors or benchmark tools, and a YAML file is used to specify configuration options for the sensors, benchmarks & plots. Once an experiment is started, the tool organizes results into a folder, runs the experiments and collects sensor data, then cleans and generates plots to help understand the data. Plot generation is a mix of native Rust code & python scripts, since plotting libraries are more robust and simpler in Python.

4.3.1 Calculated & Recorded Metrics

From our sensor data, the following metrics are recorded and exported to JSON files for simplifying graph generation. Not all metrics are available for all benchmarks, since application benchmarks do not have all these metrics available, and sometimes use different reporting standards. Metrics used in these benchmarks are specified in their respective sections.

- SSD power & energy (W, J)
- CPU power (Node 0 + Node 1 + DRAM) & energy (W, J)
- Whole system power (W)
- Average latency (ms)
- P99 latency (ms)
- IOPS/J, calculated with SSD, SSD + CPU (W/s)
- bytes/J, calculated with SSD, SSD + CPU (W/s)

- Energy delay product (EDP) using average latency, calculated with SSD, SSD + CPU ($W \cdot s$)
  - EDP (44) is not commonly used in SSD energy research, but we think it helps characterize the cost of latency w.r.t. energy consumed.
- EDP using P99 latency, calculated with SSD, SSD + CPU ($W \cdot s$)
- Average CPU frequency (GHz)
- Average CPU Load
- Function call counts (if tracing is enabled)

5

Microbenchmarks

As described in RQ2, we conduct experiments to measure the impacts of request size, I/O depth, jobs, interrupts, polling, I/O engines, schedulers, access patterns, interval patterns as well as various other characteristics, referred to as microbenchmarks.

When reading this section, unless specified otherwise the main power state discussed is PS0. 'Performance' is used as an overarching term referring to both throughput & latency. Unless otherwise specified, all measurements specified are averages.

For all the experiments below, the following general setup steps are conducted unless specified otherwise. First, `htop` is used to check that no other tenants are running and that the system is idle. Then, our NVMe drive undergoes a standard preconditioning setup: The drive is formatted using `sudo nvme format -s 1 /dev/nvme2n1 -f`. After this, fio is used to fill the entire drive which is verified using `sudo nvme list`. This is done to limit the effects of the DRAM cache & SLC cache on our SSD but also ensures the FTL is not in an optimal situation, and has enough work to do for new I/O. The preconditioning step is run before each experiment.

5.1 General SSD Energy Characteristics

Firstly, we discuss various general observations in energy and performance discovered during various experiments, and provide additional context as well as insights on our particular setup, as well as NVMe SSDs in general. The insights from this section shed light on energy characteristics under certain conditions but energy-performance tradeoffs are the focus here. The energy characteristics will help in answering RQ2.

5.1.1 SSD Energy at Idle in All Power States

SSDs spend time at idle, so it is important to verify idle power in all power states compared to the manufacturer's claims.

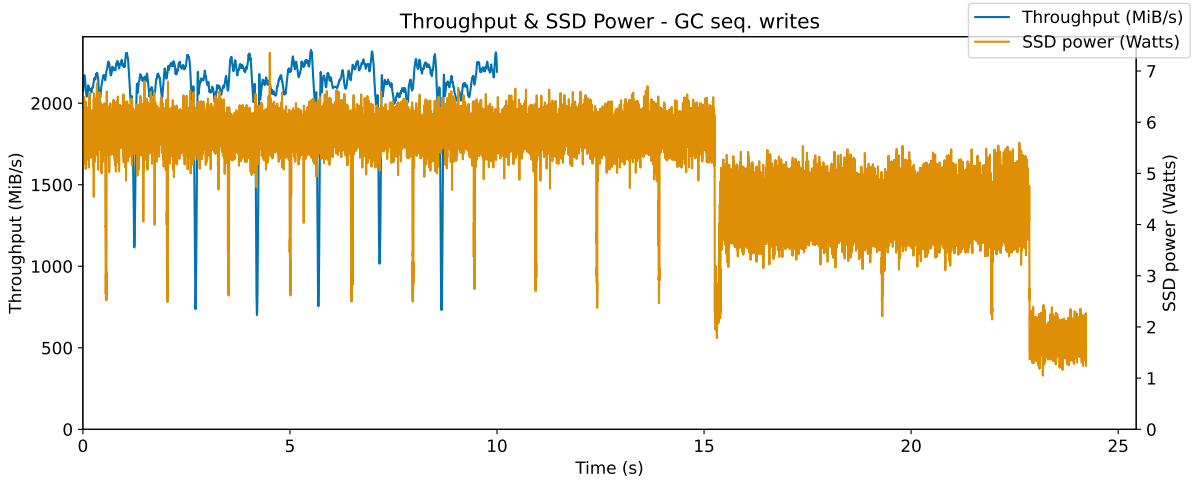
Setup. For this test, we ran the Linux 'sleep' command for 15 s after setting an NVMe power state, allowing the SSD to settle at idle power. All 5 available power states on our drive were tested.

Findings. As stated above in section 4.1, our test SSD has 3 functional power states, and 2 non-functional, sleep states. In our testing, the sleep states (PS3, PS4) consumed 1.2 W, compared to the advertised 0.4 W and 0.05 W. This is likely an issue with the SSD's firmware. At the power state PS0, the SSD consumes 1.75 W at idle. **Note** that the power consumed here is lower because the power states are defined as the largest average power in a 10 second window. The SSD seems to have its own internal power targets and states, as if it were mimicking PS1 & PS2. This is likely performed using power gating (45) & DVFS of the onboard controller, as well as by turning off NAND chips that are not in use. With a peak power shown in other experiments as 6 W, our SSD exhibits a dynamic power range of 79%.

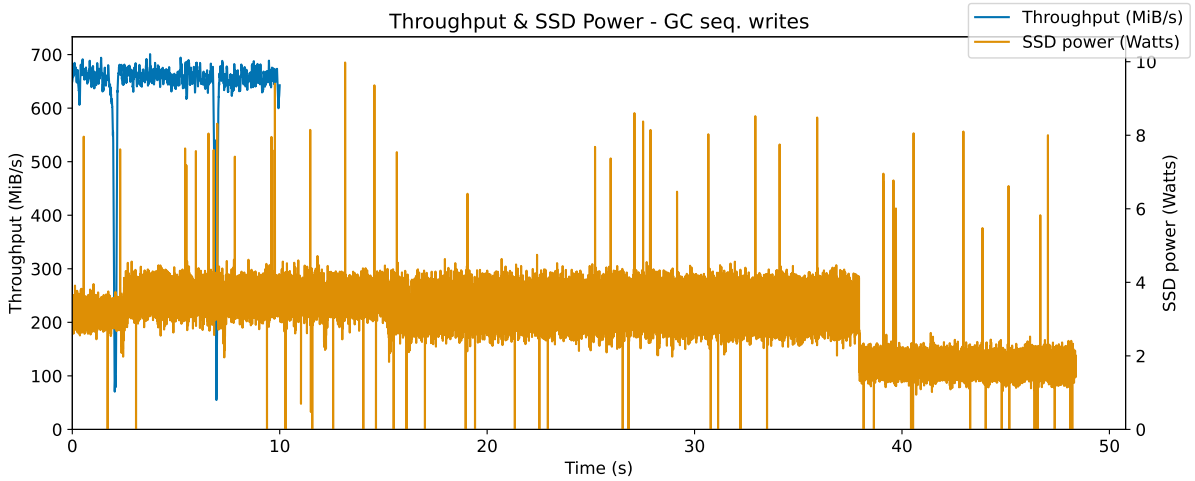
5.1.2 Garbage collection

GC usually only runs after write operations (27), although the specifics will depend on heuristics determined by the manufacturer. Therefore, GC will not be invoked on every I/O, but manufacturers will try to run GC during idle time or periods of reduced activity if possible. The issue is that GC will reduce throughput to the device when running, since it is performing many read & write operations of its own (reducing available bandwidth to actual host I/O), moving data around inside the SSD.

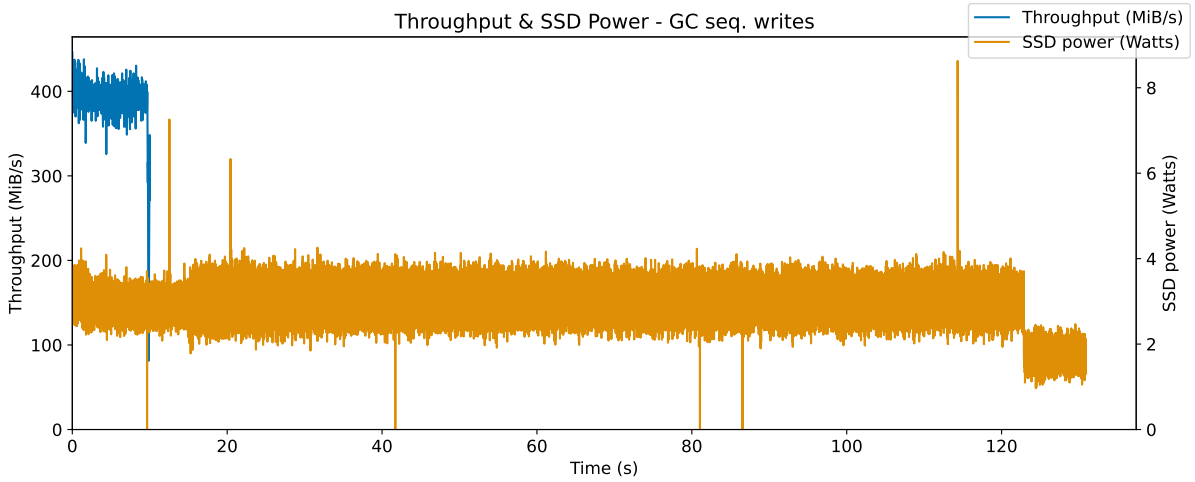
Setup. We know that FTLs usually only perform GC after write operations (28), so to invoke GC we can perform sequential write tests. The tests were conducted in various configurations as described below. The test was run, and once complete we continued to collect power metrics until the SSD power meter returned to the previously measured idle power, signaling that any GC work was complete. A 15 second "maximum throughput" sequential write test was performed by using a request size of 1 MiB and an I/O depth of 32, at all three power states. The same test was also run with a duration of 60 seconds as a sequential write & a random write workload, but only at PS0.



(a) PS0 GC seq. writes (15s)

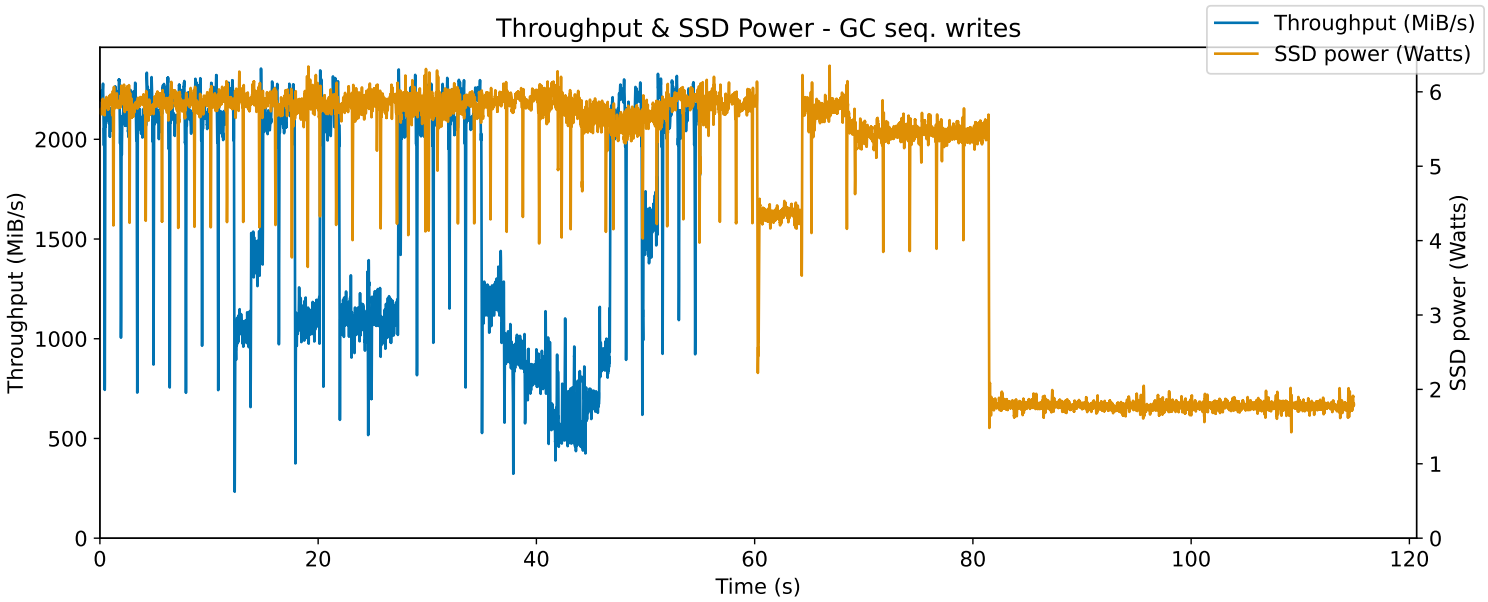


(b) PS1 GC seq. writes (15s)

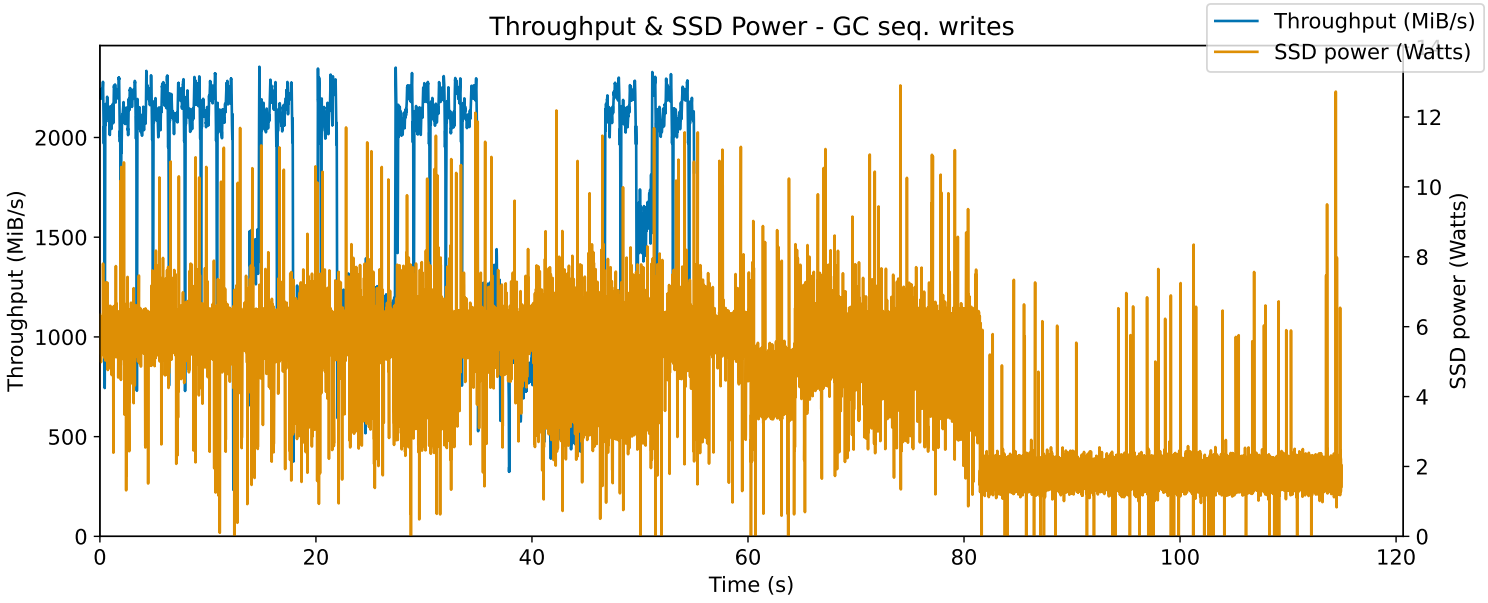


(c) PS2 GC seq. writes (15s)

Figure 5.1: GC - 15s write test



(a) PS0 GC seq. writes (60s)



(b) PS0 GC seq. writes (60s) - without power smoothing

Figure 5.2: GC - 60s write test

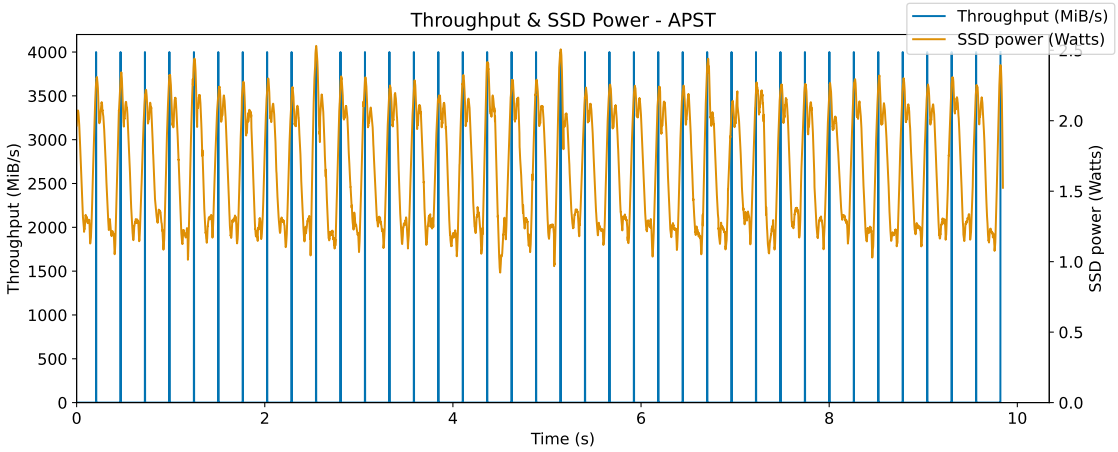


Figure 5.3: APST - throughput & SSD power over time.
Note: throughput & power lines are not aligned perfectly

Findings. What we hypothesize to be GC is especially noticeable in the high throughput sequential tests, because of noticeable performance drops mid test. As seen in 5.1a, GC in PS0 appears to kick in after 15 seconds of writing (30.56 GiB). Because the SSD continues to consume non-idle power for 7.5 seconds rather than throttling down, and CPU remains at idle, we infer this to be GC. Since GC also has to adhere to the active NVMe power state, GC for PS1 & PS2 runs slower, and therefore takes much longer to complete. GC at PS1 5.1b took almost 20 seconds for the same 15 second test, having written just 9.5 GiB during the test. GC runs for more than 100 seconds at PS2 after writing just 5.5 GiB of data during the test 5.1c. GC also appears to run for longer if more data is written. In the 60 second write test performed at PS0, 96 GiB of data was written after which the GC ran for 20 seconds, scaling evenly from the 15 s test 5.2a.

GC also appears to run while the workload is still running, leading to 3 small pauses/slowdowns lasting 2 - 6 seconds. In the first GC invocation, throughput reported by fio dropped by 50 %. After the 40 second mark GC was invoked again, with almost 15 seconds of GC, and throughput dropped by almost 75%. GC appears to be very mild for random writes, running for only 15 seconds after a 60 second test. Based on 5.2b and the other GC tests, it's clear that power usage of our SSD seems to be erratic, jumping between idle power and peak power with a few milliseconds when GC is active.

5.1.3 Autonomous power state transition

Autonomous power state transition (APST) is an NVMe controller feature that specifies the kinds of transitions that can be made between active and idle power states. When enabled, the SSD can then jump between these power states as defined by the transitions, but an idle time to use between transitions can be specified on the Linux host. On our device, 4 rules were present, and all were configured to move from any power state into PS4 after 100 ms of idle time (i.e. no I/O activity).

Setup. To test the effects of APST, a regular sequential read test was conducted with 4 MiB and 4 KiB request sizes. These two request sizes were selected to see the actual power saved by APST in a minimal I/O setting and a larger I/O setting, where the device might have to activate many more channels and Flash packages to complete an I/O size of 4 MiB. Then, using 4 MiB request size, fio's 'thinktime' and 'thinktime_spin' options were set at 250 ms. This causes fio to sit idle, effectively not issuing any I/O for 500 ms. The aim was to use an idle time above the APST's configured idle time - providing an artificial ideal scenario to test APST.

Findings. In the sequential test, the SSD behaved just as if APST was not enabled. APST never selected a lower power state, even when running at 4 KiB, where the resulting throughput could be attained at PS1 and PS2. When testing with the 'thinktime' variable set, the SSD throttles down to PS4 when idle, before shifting back up to maximum power when the next set of requests arrive Figure 5.3. Disabling APST led to almost identical results to not using APST in this test, since as described above in subsection 5.1.1, the measured PS4 power was very close to PS0 idle. This suggests that APST does not help save any more power than the SSD already does on its own. Enabling APST only provided a 3 % drop in average power, and consumed a mere 7 joules less in total in this highly artificial test.

5.1.4 NUMA

As discussed in (38), NUMA domains can affect the SSD. The exact performance gap can vary based on the exact scenario, but at 4 KiB with a high I/O depth, we saw a 9% drop in throughput and a 7% rise in latency when running fio on the NUMA domain the SSD was not connected to. i.e. the SSD was connected on NUMA domain 1, but in this test fio was locked to NUMA domain 0.

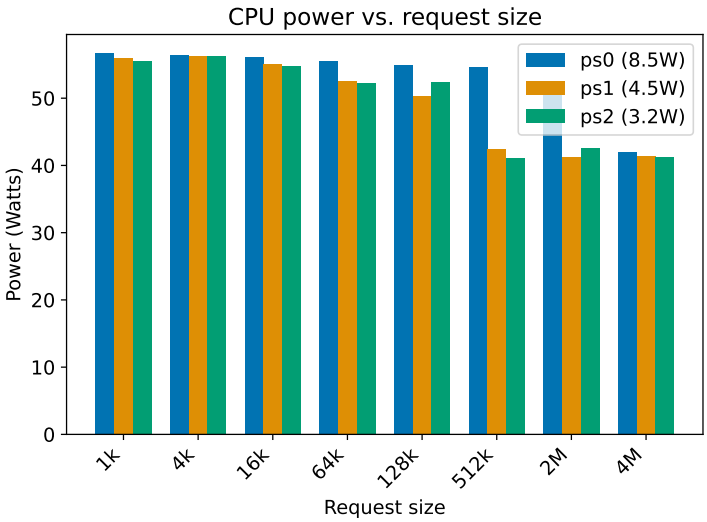
5.2 Request Size

In this section, we look at the effects of request size on performance, energy and the resulting tradeoffs. From the survey, we know that request size can impact performance, and as shown in Xie et al. (12), we hypothesize that SSD energy usage will increase with request size, since the device will be doing more work, answering RQ2.

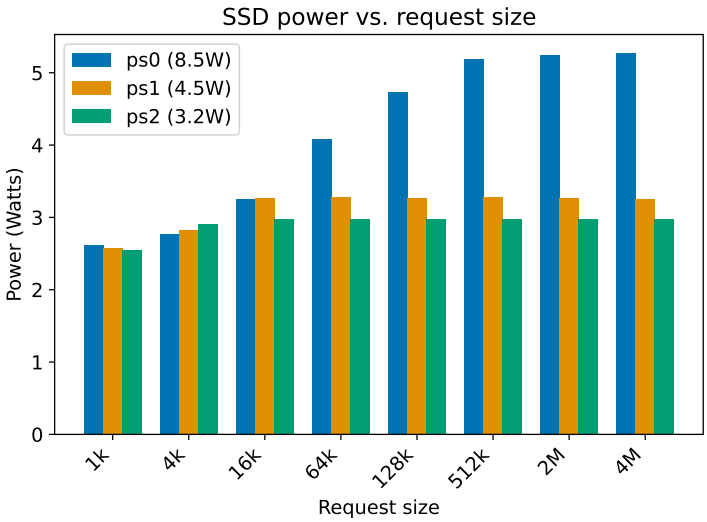
Based on previous works, I/O request sizes depend on the particular application. Large sequential workloads (lasting a second or more) such as copying a file will use large request sizes (128 KiB+), and a database might only read in 4 KiB chunks to minimize latency. For performance, we expect to see throughput increasing with request size until it reaches saturation, although the specifics of the curve will differ based on the type of operation (sequential read or write, random read or write). Latency should see a substantial growth as throughput saturation is reached. These are known characteristics. For energy, we expect higher throughput to use more energy (SSD), since more NAND chips will get activated, and also since the controller onboard the SSD will have to do more work (28). Unlike the SSD, we expect CPU power to come down, since the CPU will have to interact with the SSD less frequently, as larger request sizes will have a higher latency, allowing the CPU to down clock and save power. Based on Xie et al. (12), we also expect read throughput not to degrade as much as writes, when running the SSD in a lower power state.

Setup. A range of request sizes are tested 1 KiB, 4 KiB, 16 KiB, 64 KiB, 128 KiB, 512 KiB, 2 MiB, 4 MiB. The non-uniform scaling was used, since these are common request sizes used in various applications & studies (46), (47). io_uring was used, with an IO depth of 1, to prevent its effects from interfering with the experiment. Sequential read, write and random read, write were then tested at all 3 SSD power states. All combinations of the above mentioned variables were tested.

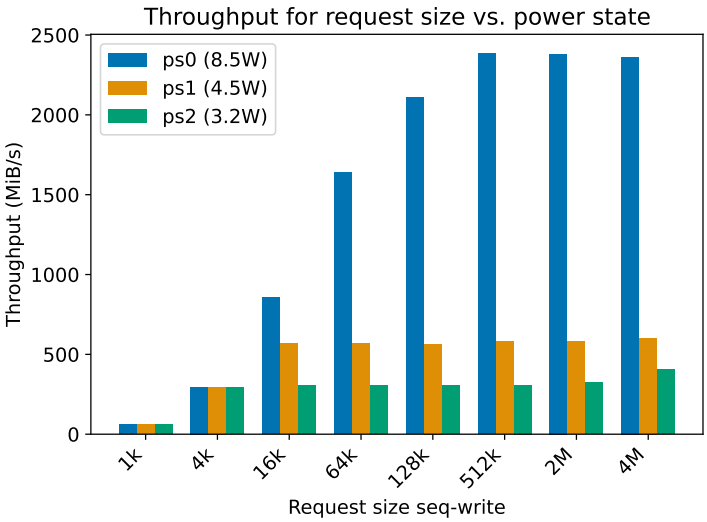
Findings. We find that SSD power increases as throughput increases 5.4b, as expected. However contradicting Xie et al. (12) throughput is significantly degraded at PS1 and PS2 5.4c. This is likely due to the design of our SSD, being a consumer NVMe drive, its lower power states might be designed to save a lot more power (for portable devices) unlike the NVMe enterprise drives used by Xie et al. We also find a steady drop in CPU power 5.4a as request size increases, an almost 27% decrease. This is further explained by looking at CPU data. The CPU spent 652 % more time processing requests at 1 KiB than 4 MiB, and



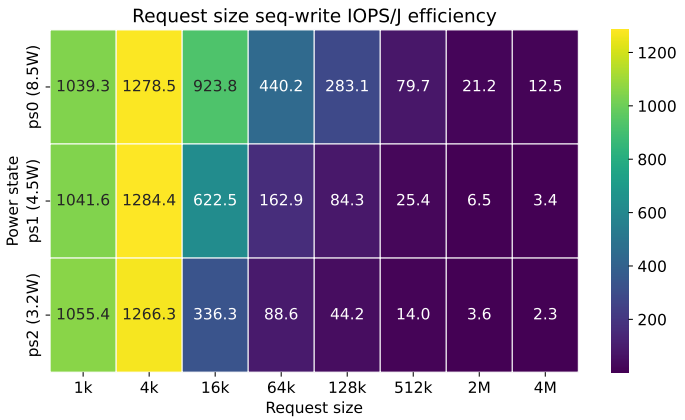
(a) CPU power - seq. writes



(b) SSD power - seq. writes



(c) Throughput - seq. reads



(d) PS0 CPU + SSD efficiency - seq. writes

Figure 5.4: Request size experiment power, throughput & efficiency

performed 3619 % more context switches at 1 KiB, but context switches per I/O operation is nearly 1 for both, which is expected because io_uring is running in its default interrupt mode for this test.

Average latency takes a large rise from 2 MiB in PS2 (it stays below 1 ms for smaller sizes), with an average latency of 12.4 ms at 4 MiB. This rise in average latency is after the MDTS size (512 KiB), suggesting that PS2 has trouble handling higher I/O depths (confirmed in section 5.3). For sequential writes, PS0 provided an efficiency of 37 MiB/J, while PS1 only had an efficiency of 9.7 MiB/J. This is explained by writes having a higher peak power of 5.3 W - 5.65 W, with other operations only taking 4.1-4.5 W. which can be attributed to write amplification as well as the increased time spent to erase and write a NAND flash cell compared to just reading it, as well as any additional FTL logic that gets triggered while writing, such as GC work.

Looking at whole system power, the readings do not align with our CPU & SSD power readings. At 1 KiB, our RAPL reported power was 61 W, while our whole system power was 94 W. This is expected because of power used by fans and the motherboard. However even at 4 MiB, the power was similar at 91 W, while CPU power was only 46 W. In other words, CPU contributes 50% to total system power at 4 MiB, and SSD contributes 6%, while at 4 KiB CPU contributes 63% and SSD 3%. ***F1.*** The gap between CPU power and system power reduces as the system load increases, and showcases some of the losses in power supplies, as showcased earlier section 3.6.

F2. The general tradeoff we find is that more throughput requires more power, with the efficiency reducing as you get closer to the minimum power of the system, but in this test it comes at the cost of P99 latency, taking a 2000% increase from 1 KiB to 4 MiB (although this is still only 1.5 ms).

Recommendations. Based on the findings, we can say that applications that focus on IOPS should stick to the most efficient (IOPS/J) request size, which was 4 KiB in our setup, and smaller sizes such as 1 KiB give worse efficiency because of the increased CPU cost 5.4d.

For applications prioritizing throughput, use as high a request size that saturates device throughput, while staying at the highest power state, thereby maximizing bytes/J. If the application only uses limited bandwidth, under the maximum deliverable under PS1 and PS2, applications should throttle down to these power states, if the increased latency is acceptable.

5.3 I/O Depth

In this section we look at the effects of I/O depth on energy performance tradeoffs. I/O depth refers to the number of requests queued at the same time at the top application layer. Based on Xie et al. (12), we can find the energy performance tradeoffs present at various I/O depths answering RQ2, and we expect SSD energy to increase as I/O depth increases because of increased device utilization.

As I/O depth increases, throughput quickly saturates, and latency will also quickly saturate. Studying I/O depth is crucial, since high throughput tasks might queue many operations together, as an easy way to maximize throughput. Finding if SSD efficiency starts to drop beyond the throughput saturation point is also important, since many applications send much more I/O than the device can handle (in terms of hardware queues) like filesystems as seen below in section 6.1. Based on the previous experiment, we expect I/O depth to yield different results to Xie et al. (12), because the consumer NVMe SSD being used, these enterprise SSDs are known for having significantly high IOPS ratings when compared to consumer SSDs. In terms of power, we expect the results to be similar to the request size experiment, because at the end the SSD can only handle requests at its block size, 512 bytes. CPU power might increase as I/O depth increases, but it would depend on the specifics of the SSDs performance, which would dictate how quickly the CPU has to process incoming & outgoing data.

Setup. A range of I/O depths were tested: 1, 4, 8, 16, 64, 256. Based on the results of the previous request size experiment we set Request Size to 16 KiB which will let us find the saturation point of I/O depth, since the SSD does not saturate at smaller sizes. Sequential read, write and random read, write tests were conducted, with the default configuration of io_uring, as well as all 3 SSD power states. All combinations of the above variables were tested.

Findings. We find that for sequential ops, the SSD power curve jumps up, with a relatively small increase in power until saturation Figure 5.5. Sequential writes start to saturate at an I/O depth of just 4, and increase by only 100 MiB/s when I/O depth is 256, while sequential reads have a 750 MiB/s gap for the same I/O depth range. This is a general characteristic of all modern SSDs. The limited growth in SSD power is likely an indicator of internal parallelism limits (ability of the FTL to parallelize) for sequential

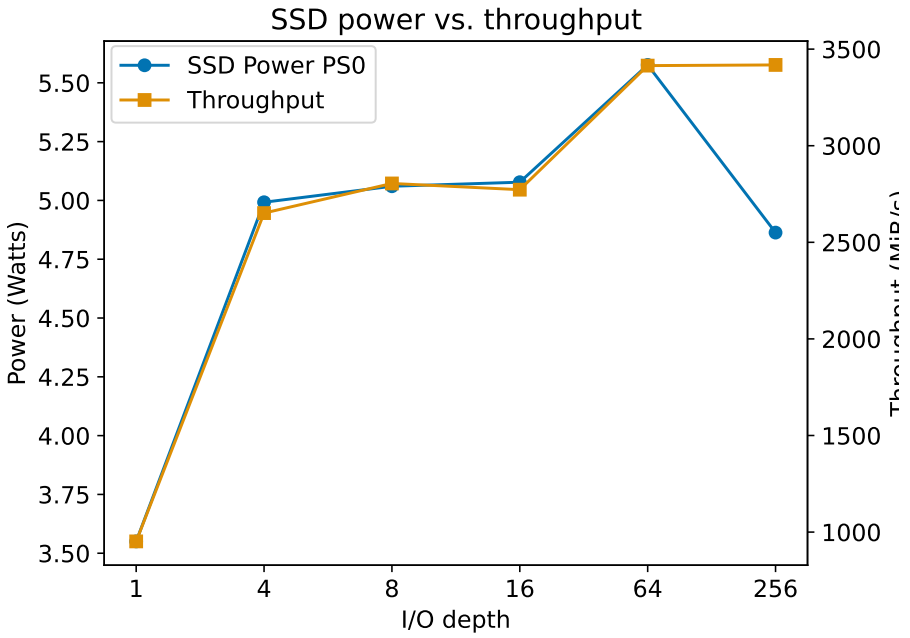


Figure 5.5: PS0 Throughput & SSD power - seq. reads

tasks. Even when throughput is below the saturation limit for PS1, IOPS/J efficiency is 38% worse than PS0.

In random ops, SSD power has a gradual increase until the saturation point. Unlike in sequential ops, bytes/J efficiency is significantly skewed to a higher I/O depth. This is seen in random writes, where going from an I/O depth of 4 to 256, there is a 511% increase in bytes/J efficiency, while sequential writes bytes/J efficiency only increases by 5%, again a result of parallelism Figure 5.6. As a result the energy-performance tradeoff for sequential ops, especially for writes is less significant than with request size. This tradeoff was not seen in the previous request size test, because at any given point the SSD only handled a single request (I/O depth was 1 for the test). Similar to the request size test, P99 latency for all operations remains low and within 1 ms at all I/O depths at PS0, but its effect at PS2 is drastic: ***F3.*** comparing "equivalent" sizes, at a request size of 4 MiB and 16 KiB at an I/O depth of 256, P99 sequential read latency is 352% higher at the high I/O depth, with EDP being 1975% higher, showcasing PS2's extremely poor latency.

As expected, CPU power has a mild increase with I/O depth increase due to the additional work in issuing and processing the I/O, shown by the increased CPU time from 4.6 s to 18.8 s, the increased context switches, as well as an increased average CPU load

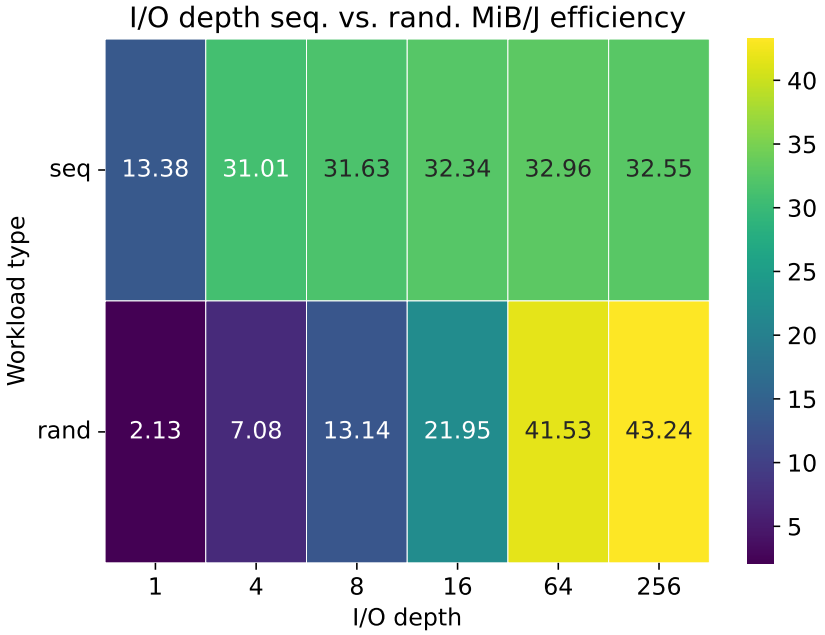


Figure 5.6: PS0 I/O depth vs. workload type efficiency

from 5 to 12, ***F4.*** In general "equivalent" sizes show a higher efficiency for higher request sizes, thanks to the large CPU power decrease when compared to I/O depth.

5.3.0.1 Recommendations

Based on the findings, we can conclude for our setup, that applications should stay at PS0 for SSDs with similar power state configurations to ours, since the latency cost is too large, especially for PS2. This also holds for throughput, even in throughput conditions below the PS1 saturation point. Applications focusing on IOPS/J should try to queue at high I/O depths, up to their acceptable latency target, especially if the workload is random. Applications using throughput with bytes/J as the metric, high I/O depths until device saturation can be used, but must only be done after tuning request size. The additional CPU power & load should also be considered, as it can take considerable CPU resources, e.g. our 20 thread CPU had a load factor of 12 at I/O depth of 256, which is more than 50% of CPU time spent issuing and receiving I/O.

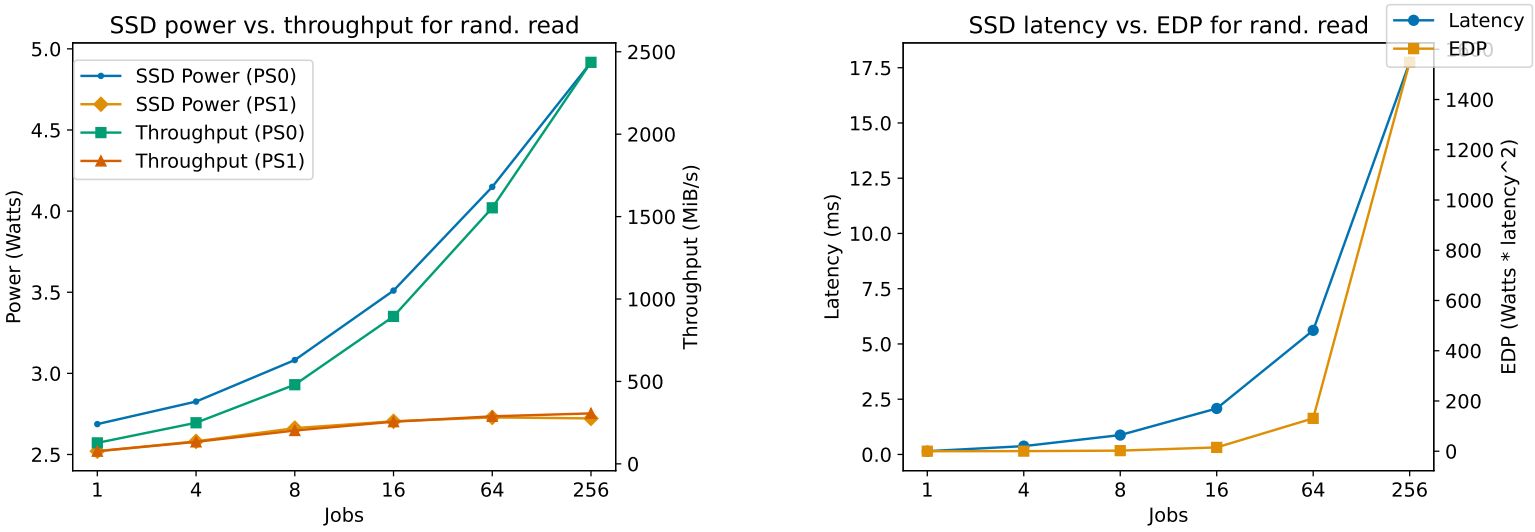
5.4 Jobs and Threads

In this section we look at the effects and energy-performance tradeoffs of multiple jobs or threads used for submitting I/O. Based on Ren et al. (36), we hypothesize that since throughput increases, SSD power will increase, but overall efficiency might not change much due to the CPU cost, answering RQ2.

As jobs increase, throughput will quickly saturate - a similar effect to I/O depth. However unlike I/O depth, there will be an increase in CPU power due to the additional tasks, as well as the increased contention in the kernel for managing I/O. Understanding how jobs affect energy-performance tradeoffs is important, since in many scenarios, especially in the cloud, multiple applications might be sharing an SSD. Previous studies that conducted tests in multi-thread or multi-process scenarios (36), (33), (35), (39), clearly show the increased CPU costs as well as contention as I/O threads increase. SSD power, throughput and latency are expected to be similar to I/O depth section 5.3, a general increase in power, throughput & latency as jobs increase.

Setup. The following job counts were tested: 1, 2, 4, 8, 16, 32, growing in powers of 2, until we exceed the number of CPU cores on our system, 20. A "job" refers to a fio job, where increasing the job count spawns more processes. We tested both jobs & threads. Using the `-thread` flag creates threads instead of processes. The experiment was conducted at the above mentioned job counts, in both job & thread mode, with a request size of 16 KiB and an I/O depth of 1, and io_uring to avoid external variable interference. Sequential read, write and random read, write workloads were run at all 3 SSD power states. All combinations of the above mentioned variables were tested.

Findings. Overall, threads vs. jobs had little to no difference in throughput & latency. Due to the additional jobs/threads, CPU power increases by $\tilde{2}5\%$ going from 1 to 32 jobs. Throughput scaling in this test was not identical to "equivalent" values in the I/O depth or request size experiments. Firstly, sequential read and write tests showed roughly identical throughput from 1 to 8 jobs, around 930 MiB/s, after this it spiked up, but this throughput is significantly lower than in other tests 5.7a. At 8 jobs, throughput is 930 MiB/s, while the "equivalent" I/O depth (16) yields 2700 MiB/s and "equivalent" request size (128 KiB) yields 2300 MiB/s. ***F5.*** scaling using jobs is around 66% less efficient (bytes/J) when compared to request size scaling or I/O depth scaling at lower job counts (until 16). This changes at a job count of 32, where throughput is matched, but around 27% less efficient



(a) Throughput vs. SSD power - rand. reads

(b) Latency vs. EDP - rand. reads

Figure 5.7: Jobs & Threads - Throughput & Latency

(bytes/J) when compared with "equivalent" I/O depth and request size, due to the increased CPU power. Similar poor scaling is present in random workloads as well. Another note is that while performance was identical between jobs & threads, IOPS/J efficiency was consistently 0.15 - 1% higher when using jobs compared to threads.

F6. Unlike the latency scaling observed in the request size or the I/O depth experiments, latency is noticeably worse when scaling throughput using jobs/threads 5.7b. Average latency was 0.17 ms for an I/O depth of 16, but 2.52 ms for 16 jobs, a 1382% increase. However, EDP between I/O depth and jobs/threads is almost the same - suggesting that this is an acceptable loss in latency if efficiency is more important for a workload. While PS0 latencies and P99 latency stayed within "acceptable" margins for an SSD (under 2 ms), at PS1 and PS2, P99 latency ballooned reaching 250 ms and 500 ms respectively, which is well outside the latency range expected from a modern SSD. Similar to our findings in the I/O depth experiment it is clear that lower power states cannot handle much parallelism.

Recommendations. As all the above findings show, the tradeoffs from trying to maximize throughput by increasing jobs/threads are not a good idea due to poor scaling at small job counts, up to 16. The worsened latency due to the added jobs is a severe CPU tradeoff for applications requiring good latency - usually ones that focus on IOPS as a metric. Our recommendation is to use thread and job parallelism in a later stage of opti-

mization/design and prefer using request size and I/O depth before trying to saturate the device using multiple threads. Lowering power states is also not recommended.

5.5 Interrupts and Polling

In this section we will look at the effects of interrupt & polling on energy, performance and any tradeoffs that arise. From Harris et al. (31) we know that interrupts are more efficient than polling for larger request sizes, and polling is more efficient for small request sizes. However, this study only looked at whole-system power. Interrupt vs. polling is an important design choice when building I/O software, and understanding its energy-performance tradeoffs helps answer RQ2.

An interrupt is essentially an action performed once a previously started task is ready/complete. In other words, when an interrupt is triggered when I/O is complete and data read from the SSD can be returned to the caller. Polling is checking if I/O is complete at a constant interval, which significantly increases CPU work, so the energy performance tradeoffs here are likely to come in the form of additional CPU costs.

Setup. io_uring was used since it can run in interrupt and polling modes. For polling, we need to set the NVMe driver `poll_queues` option, as well as fio's "hipri" option, but also enable the polling thread for checking io_uring's ring buffer using "sqthread_poll". We also have the "sqthread_poll_cpu" option to pin the polling thread to a CPU core. NVMe poll queues are also enabled, as without this the actual NVMe device command does not run in polling mode. We tested it using the request sizes 4 KiB (representing a small request size), 16 KiB (an intermediate request size), 128 KiB (a large request size, limited because our io_uring module crashed the kernel when running at higher request sizes in polling mode), with an I/O depth of 4. Sequential read, write and random read, write workloads were run, at all 3 SSD power states. All combinations of the above mentioned variables were tested.

Findings. Some of our results were similar to Harris et al. (31), with polling more efficient only at 4 KiB - that too only under certain types of workloads, something not identified by Harris et al. Polling is 32% more IOPS/J efficient than interrupts for sequential workloads at 4 KiB at an I/O depth of 4. Random workloads at the same I/O depth are 4% less IOPS/J efficient, but increasing the I/O depth to 32 yields an IOPS/J efficiency 14% higher than interrupts, equalizing roughly at an I/O depth of 8. At 16 KiB,

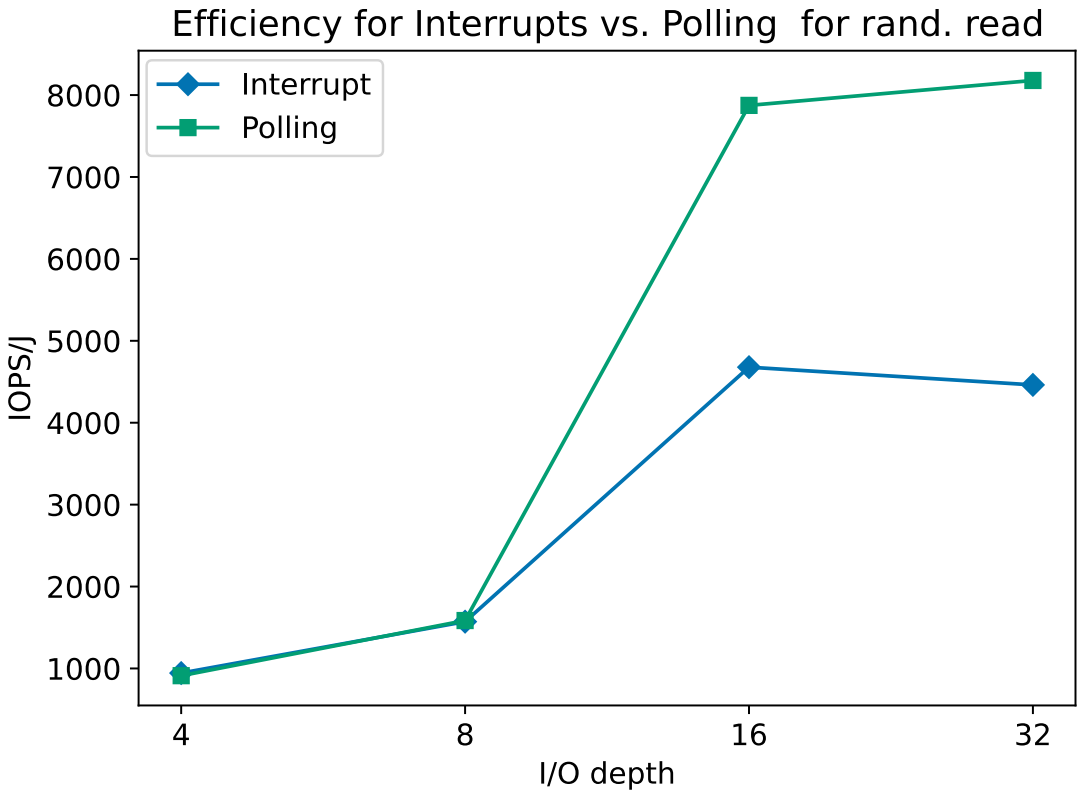


Figure 5.8: PS0 Interrupts vs. Polling efficiency - rand. reads

only at an I/O depth of 1 for sequential ops polling is more IOPS/J efficient by 15%. In all other workloads at 16 KiB and all workloads at 128 KiB, interrupts are more efficient, up to 17% in sequential ops. As stated in Harris et al. polling is more efficient in workloads where I/O has a very low completion time. ***F7.*** I/O in random workloads does not complete in a fast and predictable manner to keep the polling thread busy, wasting CPU cycles resulting in worse efficiency. This is why only at an I/O depth of 8 does the efficiency of random workloads equalize Figure 5.8, showcased by random workloads at an I/O depth of 4 having identical average latency to sequential workloads.

F8. In all configurations, polling always consumed more CPU power than interrupts, between 5% - 17%. This is because the polling thread is constantly consuming CPU cycles, preventing it from down-clocking to save power, the polling tests spent 70% more CPU time and double the CPU load in the worst case. SSD power matched our previous findings here.

In all configurations, polling throughput was greater than or equal to interrupt through-

put, and in the most efficient configuration (4 KiB, 32 I/O depth) where polling was 100% more efficient, throughput was 115% greater and P99 latency 45% lower, this is due to the block layer merging I/Os - with 89% of I/Os being merged. Clearly a sign that the number of interrupts (at an IOPS of 330,757 while polling managed 712,825 IOPS) was too much for our system to handle. At PS2 & PS3, the increased latency meant that in no configuration was polling more efficient than interrupts.

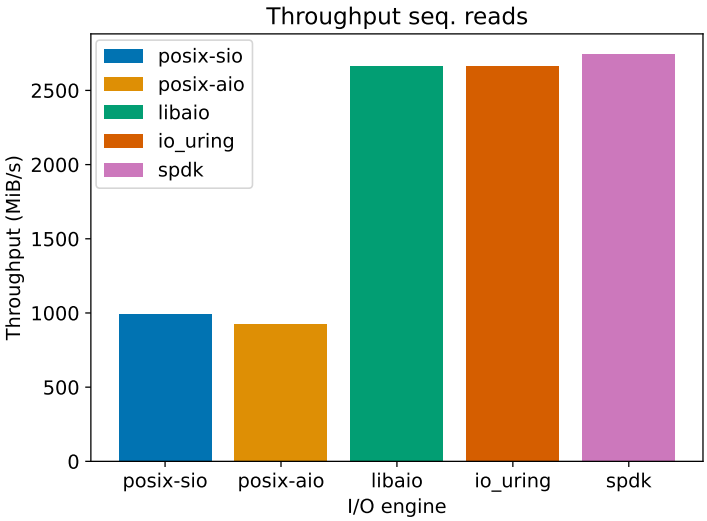
The tradeoff here is for the CPU and is more of a line - only applications with a high I/O depth at 4 KiB stand to gain any significant efficiency gains, with the largest gains in sequential operations - but applications requiring high throughput sequential I/O should stick to interrupts and instead tune request size and I/O depth, the bytes/J efficiency for polling is always lower than using interrupts with a large request size like 1 MiB.

Recommendations. Based on the findings, we can recommend using polling for applications that require the lowest possible latency (at 4 KiB request size) or for workloads that have continuous I/O at a high I/O depth at 4 KiB. All other applications should stick to using interrupts for maximum efficiency. Polling should also be avoided at lower power states if they exhibit higher latency than that provided by the highest power state.

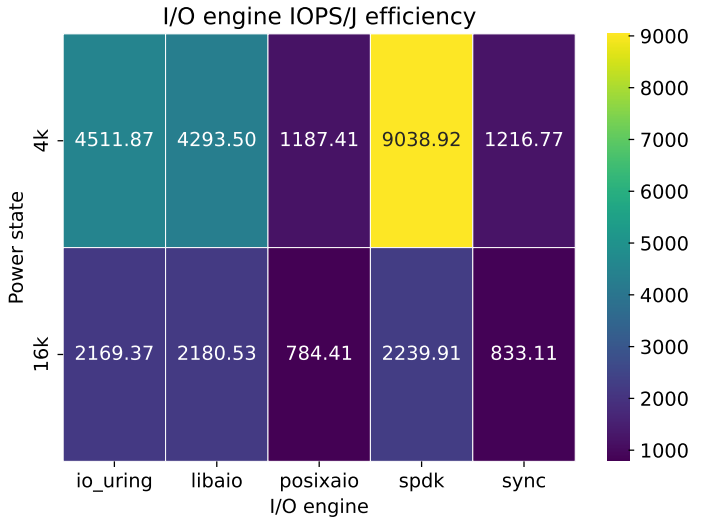
5.6 I/O Engine

In this section we look at the energy-performance tradeoffs in different I/O engines. An I/O engine is a software abstraction used to interface with lower level components in the storage stack. Different I/O engines employ different design and performance philosophies and present different performance and usability tradeoffs, and these differences should result in differing energy-performance tradeoffs. Some previous studies have looked at performance (33), (34) as well as some energy characteristics (32). Since I/O engines are a critical interaction point for applications, examining the energy and performance tradeoffs will help answer RQ2.

Setup. Popular I/O engines based on the survey are posix-sio (sync), posix-aio (async), libaio, io_uring & SPDK. The above engines were tested at 4 KiB, 16 KiB request sizes with an I/O depth of 4 and 32 to test a range of throughput up to the SSDs saturation point. io_uring was tested in its default interrupt mode. SPDK took additional work to configure & benchmark, since during SPDK initialization, all volatile NVME controller settings are reset, including power states. So for our benchmarks, we needed to modify



(a) PS0 I/O engine throughput - seq. reads



(b) PS0 CPU + SSD efficiency - seq. writes at 4 KiB, I/O depth 32

Figure 5.9: I/O engine throughput & Efficiency

the SPDK fio plugin to set power states after SPDK device initialization was complete by calling 'spdk_nvme_ctrlr_cmd_set_feature'. A CLI flag was also added to aid with this. Sequential read, write and random read, write workloads were tested, as well as all 3 SSD power states. All combinations of the above mentioned variables were tested.

Findings. From the findings, at 4 KiB we see that libaio is most efficient in random ops, followed by io_uring then spdk, posix-aio and posix-sio. The lower efficiency for random ops reflects the earlier finding in section 5.5, since SPDK uses polling mechanisms internally (23). And just like in the aforementioned experiment we see SPDK being more efficient in sequential ops, followed by the others in the same order. The gap in SPDK efficiency is 17% in random ops, and 24% in sequential ops. Continuing the trend, SPDK managed higher throughput in random and sequential ops where it delivered 30% more throughput compared to libaio, and at an I/O depth of 32, SPDK is 100% more IOPS/J efficient than libaio 5.9b. These clearly show that the SPDK provides large performance gains by bypassing the kernel and operating in userspace, and show that getting rid of the kernel massively exacerbates the characteristics displayed when we tested io_uring in polling mode.

At 16 KiB, the large gap in throughput between SPDK and the other engines reduces to roughly 5%, and as a result SPDK is only 2% more efficient than libaio here. Unlike

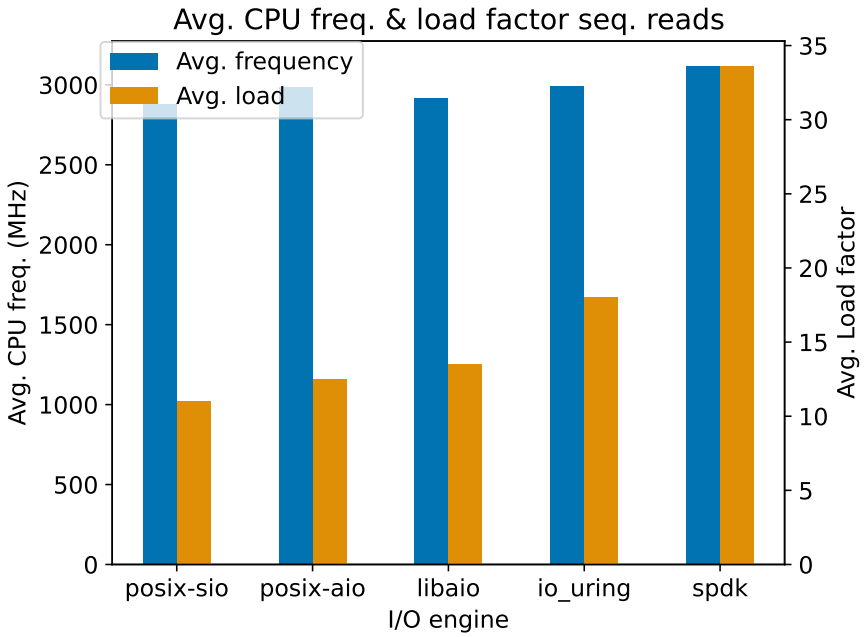


Figure 5.10: PS0 I/O engine CPU freq. & load factor - seq. reads

previous studies, most of our findings suggest libaio to be the ideal choice as it outperformed io_uring in efficiency in every test, due to the lower CPU power. In random ops, at libaio consumed 30% less CPU power while delivering similar performance to io_uring Figure 5.10. Increasing the request size to 512 KiB shows libaio's better efficiency, 10% more than io_uring and 24% more than SPDK in random ops. ***F9.*** Modern engines like io_uring and SPDK deliver great performance 5.9a - but their energy efficiency is poor under many conditions: They are best suited for highly specific, sequential workloads.

It is important to note that posix-sio and posix-aio had terrible results in all tests, at best 60% worse efficiency compared to libaio. Notably, posix-aio's P99 latency was 200% higher than its competitors in all tests except at 512 KiB.

Similar to our previous experiment on polling, SPDK provides substantial efficiency gains but is suitable only for highly specific and counterintuitive: sequential workloads using small request sizes at high I/O depth. In every other situation libaio is the more efficient I/O engine due to its lower CPU power and up to 25% lower CPU clock while delivering similar throughput & latency.

Recommendations. Our recommendation is to use libaio at PS0 for all applications, thanks to its reduced CPU utilization & power, while still providing comparable perfor-

mance to io_uring. Other power states provide 60% lower efficiency, and therefore are not useful. SPDK is recommended only for high I/O depth small request size sequential I/O, and comes with higher development complexity.

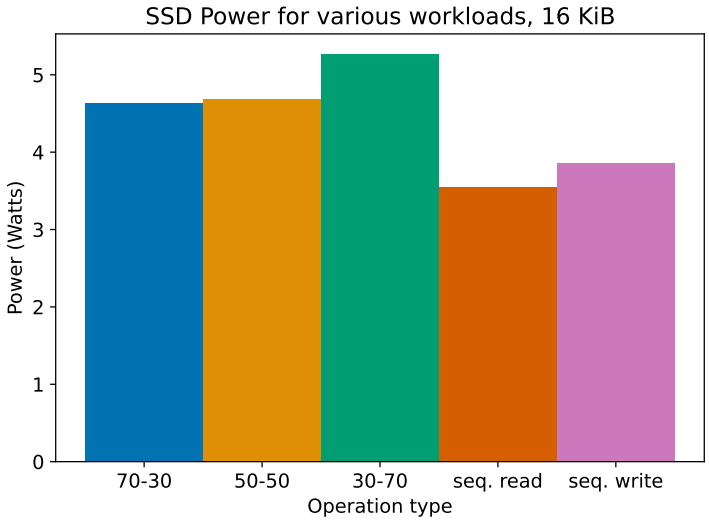
5.7 Mixed Read-Write Workloads

Unlike the workloads run in previous experiments, mixed workloads have a mix of both read and write operations. This is an important type of workload to benchmark since many real world workloads are mixed. This experiment is therefore important to understand any energy performance tradeoffs that arise in such conditions, to help answer RQ2.

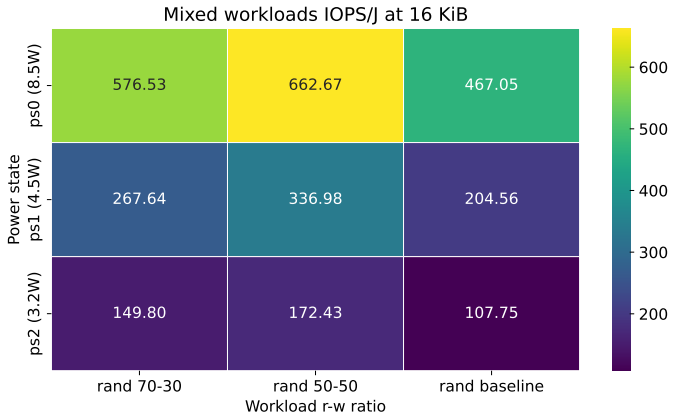
Unlike in hard disks, the internal parallelism in SSDs can lead to performance characteristics different from regular workloads, because of locking as well as numerous other reasons. E.g.,z an array of NAND chips needs to be locked to perform a write op, so read ops to the array will need to wait. Due to the nature of how NAND flash packages are designed, it is important to know that the granularity of locks in the FTL is likely not on a per-block basis. We hypothesize that performance will be degraded in most tests, because of the contention created in locking. **In this experiment, workloads that are only read ops or only write ops are referred to as "regular" workloads.**

Setup. Common representations for mixed workloads include 30 - 70, 50 - 50, and 70 - 30 splits of read-write, based on (48) as well as the standard workload representations in YCSB (49). Sequential and random operations were conducted for the above mixed workloads, with io_uring, an I/O depth of 4, and with request sizes 4 KiB, 16 KiB & 64 KiB, at all 3 SSD power states. All combinations of the above variables were tested.

Findings. Firstly, in sequential workloads we find the throughput to be lower than any regular workload. i.e. For 70 - 30 (read heavy), 50 - 50 and 30 - 70 (write heavy), throughput was lower than both regular sequential read & sequential write workloads, with a throughput drop of 27% for read heavy, but only 2.6% for write heavy. ***F10.*** Increased NAND contention reduces efficiency. This showcases how just a few write ops can cause enough contention drop throughput. Contention from writes is significant because write ops take 95% more time than read ops (28). CPU power is the same comparing sequential mixed workloads to regular mixed workloads, and therefore the lower throughput means that read heavy sequential mixed workloads have 27% lower IOPS/J efficiency.



(a) Seq. workloads SSD power



(b) Mixed random workload efficiency

Figure 5.11: Mixed workload SSD Power & Efficiency

Looking at SSD power readings, we can see the clear expected increase in SSD power in write heavy workloads, with an 11 - 12% power difference between read heavy and write heavy workloads. Similarly, write heavy workloads have a 60% higher EDP at 16 KiB, which increases to 70% at 64 KiB.

Random mixed workloads at 16 KiB exhibit an unexpected behavior: They are more efficient than their regular counterparts. Read heavy workloads are 18% more IOPS/J efficient compared to a regular random read workload, and write heavy workloads are 36% more IOPS/J efficient compared to a regular random write workload 5.11b. ***F11.*** Random mixed workloads are often more efficient than their regular counterparts. At 64 KiB however, regular random workloads become more efficient than mixed random workloads, but lowering the size to 4 KiB results in the same effect, with 17% IOPS/J efficiency for mixed random over regular random. Increasing the I/O depth from 4 to 16 results in an IOPS/J efficiency of 21%.

Recommendations. Based on our findings, we recommend applications with read heavy sequential mixed workloads to try and split read and write I/Os, and issue them separately to maximize efficiency. In random mixed workloads, applications with small request sizes such as 4 KiB are encouraged to interleave read and write I/Os, to minimize any NAND contention, thereby maximizing throughput & efficiency.

5.8 I/O Schedulers

On linux, by default NVME SSDs do not use a scheduler, i.e. the "none" scheduler. If only a single application makes use of the SSD, there is usually no need to consider the using a scheduler as shown by Zebin et al. (36). Schedulers for NVME devices are considered in multi-tenant scenarios, when there are multiple applications that might be fighting for resources, and a scheduler can help provide fair access to the SSD, or help maintain some other access criteria when required. Two popular schedulers are mq-deadline and kyber. mq-deadline delivers worse throughput, but kyber delivers throughput close to "none". Since some situations require such schedulers, it is important to benchmark their energy-performance tradeoffs to help answer RQ2.

Mq-deadline is the multi-queue version of the deadline scheduler, that maintains two FIFO queues for read and write ops, giving read ops priority, as well as giving expiry times to requests. Kyber was originally designed by facebook and uses multiple queues, and also sets time limits for requests.

Setup. After setting the scheduler for the drive, benchmarks are run with io_uring, at an I/O depth of 1, since the best way to test multiple apps is by increasing the number of fio jobs, like in section 5.4. The experiments were run at 4 KiB from 4 jobs to 256, increasing in powers of 2. Sequential read, write and random read, write workloads were tested. After this, mixed workloads were also used to see the effects of split read and write queues in the scheduler. A 50 - 50 and a 70 - 30 split of read-write were tested. The tests were run at all 3 SSD power states. All combinations of the above mentioned variables were tested.

Findings. For regular workloads (not mixed), Kyber matched None performance almost perfectly in sequential ops. In random ops, kyber has poor scaling after 32 jobs never crossing 2500 MiB/s in both random read & write ops, while None peaked at 3400 MiB/s already signaling kyber will be less efficient than None at peak throughput Figure 5.12. Intuitively, the lowered throughput signals lower CPU load & power but Kyber has a 7% higher load, and 2.3% higher power. Because of the above factors, at peak throughput (256 jobs), Kyber is 25% less efficient than None in random ops, and only 1.1% less efficient in sequential ops.

Examining CPU load values showcases different results. At 32 jobs or below, Kyber matches None efficiency or beats it by 1 - 2% Figure 5.13. After 32 jobs, the additional

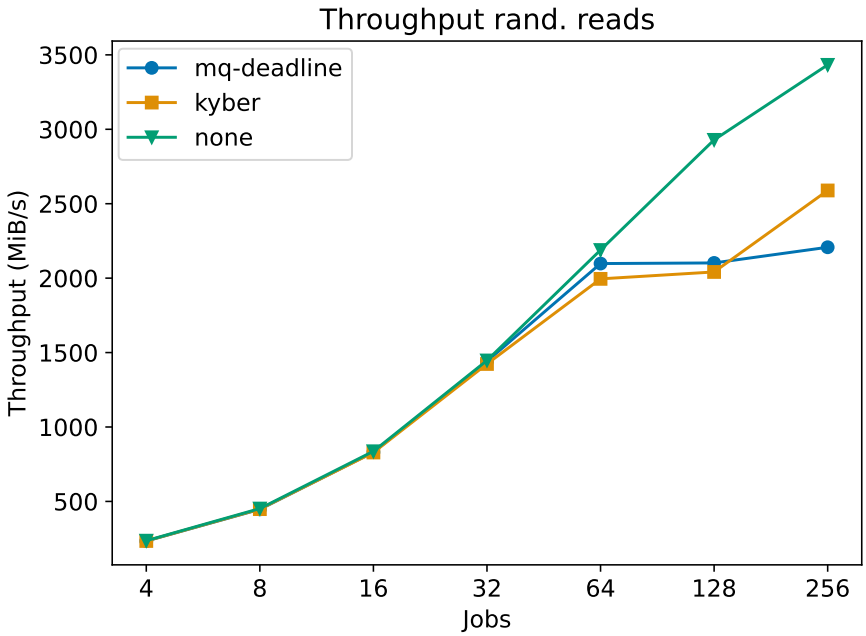


Figure 5.12: PS0 Scheduler 4 KiB rand. read

CPU load severely hurts scheduler performance, similar to what Ren et al. observed (36). mq-deadline efficiency is slightly lower than Kyber & none until 64 jobs, where it levels off. In Kyber and None, CPU frequency takes a small dip at 32 jobs in all workload types, and increases again at 64 jobs. This is unexplained because neither CPU power nor Linux load takes a small dip at 32. Looking at EDP, Kyber consistently is up to 3% less efficient at 32 jobs, and 38% less efficient at 256 jobs, due to the CPU saturation. mq-deadline is 60% less EDP efficient at 256 jobs. In mixed workloads, the results mostly matched the findings from above section 5.7, with slightly improved sequential efficiency throughout for a 70 - 30 read-write split, with no difference between kyber & None. ***F12.*** We can conclude that modern schedulers like Kyber are lightweight in terms of CPU and hence are very energy efficient, and add little overhead.

Recommendations. Use None for most situations, since it provides the best throughput & latency while also matching or beating the efficiency when using a scheduler. Kyber is the only scheduler recommended if you need a scheduler.

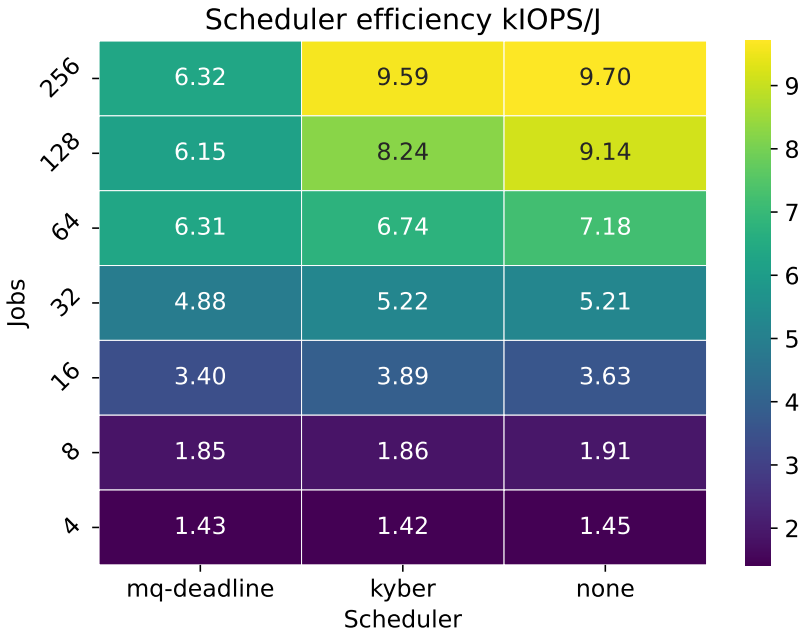


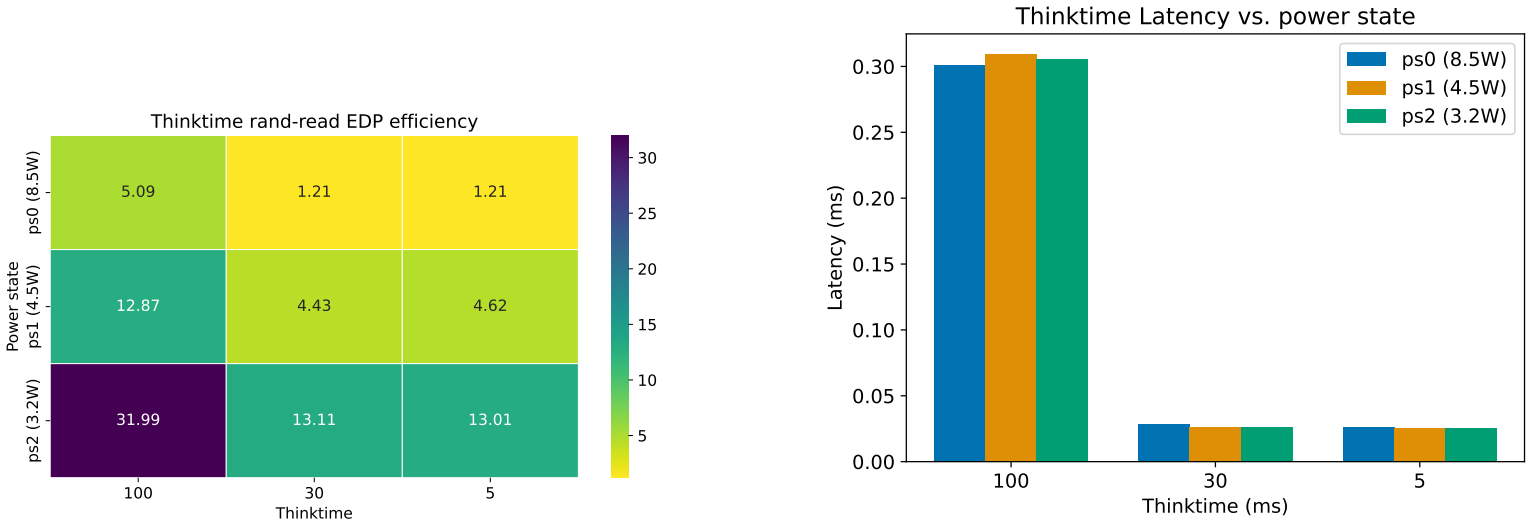
Figure 5.13: Scheduler 4 KiB seq. read efficiency kIOPS/J

5.9 Request Intervals

In most situations, data is not being written or read constantly - some kind of processing or action is done using the data, and even in simple systems that do not process data such as a file server, requests will not constantly arrive. To simulate this, while staying within the realm of micro benchmarking, we simulate a ‘think time‘ in the benchmark, allowing us to better understand performance & energy in scenarios taking processing time and queuing into account, helping answer RQ2.

We hypothesize that tests run with thinktime will yield slightly higher efficiency than a standard workload without thinktime - thanks to down throttling, reducing average power. As a result, we hypothesize that lower thinktimes of 5 ms will yield poor efficiency because of the lowered throughput but 5 ms might not be enough time for the SSD to perform internal down throttling. We could also see increased P99 latency because of the SSD waking up components after sleeping for large idle times.

Setup. Here, we will use fio’s ‘rate_process‘ variable with the value ‘poisson‘ to simulate requests being dispatched with a Poisson process (default is linear). Next we use ‘thinktime‘, ‘thinktime_spin‘ and ‘thinktime_blocks‘ to tell fio to simulate processing time.



(a) EDP - rand. reads

(b) Latency - seq. reads

Figure 5.14: Request Intervals EDP & Latency

These tests were conducted with io_uring, an I/O depth of 4 at a request size of 16 KiB. Thinktime values of 5 ms, 30 ms, 100 ms, 200 ms were tested with 'thinktime_blocks' set as 8. Sequential read, write and random read, write were tested at all 3 power states. All combinations of the above variables were tested.

Findings. In our tests, at 16 KiB, we find that the SSD fails to throttle down to reduce power at 5 and 30 ms. At 100 ms, the SSD spends a little less than 0.08 ms at idle power between requests, before swiftly increasing to over 2.5 W.

Unlike in other experiments, throughput and efficiency cannot be measured based on IOPS and MB/s, since the artificial rate we create through 'thinktime' limits throughput over unit time meaning we cannot compare IOPS/J or IOPS between different thinktimes. Instead, we know that in all tests the same amount of data is written per I/O issued so instead we focus on latency and EDP.

Looking at EDP using average latency, and both CPU and SSD power, a thinktime of 100 ms in random ops results in a 76% higher EDP, and 53% higher latency 5.14a. ***F13.*** The internal throttling performed by the SSD has an undocumented exit and entry latency. This additional latency cost is absent at 5 ms and 30 ms. In sequential ops, EDP is over 100% higher at 100 ms, with latency jumping by 1400% 5.14b. Changing request sizes did not affect the observed startup latency, testing at 4 MiB, latency values reflected a similar

average startup latency, ranging from 80 μs to 100 μs. Using a Poisson process did not have any effects on energy or performance.

Recommendations. In very sparse workloads, a small additional latency cost is incurred on SSDs. In these spare conditions lowering SSD power states will not lead to any noticeable efficiency increase but will definitely increase latency and EDP, and should therefore be avoided.

Table 5.1: Summary of workload variable recommendations to maximize efficiency.

Variable	**IOPS driven**	**Throughput driven**
Request Size	Small sizes like 4 KiB	Large sizes like 4 MiB
I/O Depth	As high an I/O depth until latency targets are hit	Increase to throughput saturation point
Jobs	Parallelize as a last effort, because of increased latency cost	Parallelize tothe throughput saturation point
Interrupts	Use polling only at 4 KiB, if workloads are continuous and without idle time, otherwise use interrupts	Always use interrupts
I/O Engine	Always use libaio	
Mixed Workloads	In read heavy sequential workloads, attempt to split and issue reads and write separately. In random mixed workloads, interleave read and write I/Os where possible	
Scheduler	Use `none`. Only use Kyber if you need a scheduler	
Intervals	Because of latency costs, stick to the highest SSD power state	
NUMA	Try to run application on same NUMA domain as SSD	
GC	Keep SSD in high power state after writes to aid GC	

5.10 Summary & Recommendations

From the above microbenchmark experiments, we can conclude that numerous energy-performance tradeoffs exist, and that depending on the specific use case, they can be optimized to adhere to SLAs while saving energy.

We find that variables that could be interchangeable in theory such as I/O depth and Request Size - in fact exhibit different efficiency and EDP. Using a request size of 4 MiB can be thought of as using a request size of 16 KiB with an I/O depth of 256, but this is

not the case. Using a 4 MiB request size yields 50% greater bytes/J. The skew is so large, that to match an I/O depth of 256 at 16 KiB, a request size of 64 KiB is enough. While SSD power is similar, it is the CPU power that makes the difference here, which is 25% lower, because fewer I/O requests need to be handled. Similarly, comparing multiple jobs & I/O depth, we see 60% lower bytes/J when using multiple jobs to simulate I/O depth. This is because of increased contention (for write ops.) and the added CPU usage from having more processes.

The more general take-away from the experiments is that the SSD appears to be significantly more efficient at its highest power state, and its energy-performance curve is skewed heavily towards its highest energy point. We can also definitively state that our high level microbenchmark variables do not directly affect SSD power - rather **the only governing factor for SSD power seems to be throughput**. In other words - trying to squeeze as much throughput as possible out of the SSD is the most energy efficient scenario. Workloads that do not fully stress the SSD are incredibly wasteful, because of the skewed energy-performance curve. Because of this, CPU efficiency needs to be looked at closely, because of how much power it consumes. In our setup, our CPU running light tasks consumed as much power as 8 SSDs, and the entire system power was equivalent to 15 SSDs. The lack of a direct relation between our microbenchmark variables and SSD power is understandable since a lot of software exists between our microbenchmark & the NAND flash, each stage acting as a black-box. Our findings are categorized into two types: IOPS driven workloads, and throughput driven workloads, and what should be done to maximize energy efficiency in both scenarios Table 5.1. It should be noted that the recommendations should only be applied to the point of throughput saturation of the device. Because the split between IOPS/J & bytes/J is a gradient and not a clear line, the exact point at which an application defines itself to be IOPS or throughput driven can change, which could have large implications on efficiency.

6

Application Benchmarks

As described in RQ3, we conduct experiments to understand and measure the energy-performance tradeoffs of various popular applications using relevant benchmarks, aimed to reflect real world workloads.

Similar to our microbenchmarks in chapter 5, we will use IOPS/J, bytes/J and EDP to measure the efficiency of I/O here. Some applications do not report the same metrics, so suitable alternates are used where applicable, such as transactions/J. It is also important to keep in mind that unlike in our application benchmarks, real work is happening with the data being read and written instead of just being raw I/O and therefore CPU power and utilization will reflect this, and cannot be blindly included in all efficiency metrics like was done in our microbenchmarks.

General setup. Before running any benchmark, the SSD is formatted with the required filesystem using 'mkfs', after which a large 700 GiB file is created using fio. This is done to fill up the SSD as part of preconditioning like was done for each microbenchmark. However, running 'mkfs' also signals an NVMe format, marking all cells as free. A single large file is used to minimize any filesystem effects while also filling up as much NAND as possible, leaving the remaining space for useful data for each benchmark. We also do not try to achieve 100% disk capacity (on top of the data for each benchmark), since the benchmarks need some free space on the drive to conduct various tasks.

6.1 Filesystems - Filebench

For conducting filesystem benchmarks, the current standard filebench (50) was used. Filebench comes with a set of standardized benchmark configuration files that are de-

signed to emulate I/O of various applications. We used fileserver, varmail & webserver since they are among the most popular used in other studies (51, 52, 53). The benchmark configuration files are written in filebench's own "Workload model language", where the operations for the benchmark, like threads, read and write I/O operations, etc. are specified. Filesystems were chosen since most applications and workloads use a filesystem, a few applications perform raw I/O without a filesystem. Therefore the aim of using this experiment is to find any energy-performance tradeoffs arising in standardized workloads on filesystems, helping answer RQ3.

We hypothesize that based on our microbenchmarks, a larger file size should improve efficiency, thanks to more sustained throughput to the device. Increasing the number of files should reduce throughput and increase latency, showcasing filesystem overhead.

It is important to note that filebench reports read and write throughput separately, as a 'per-operation' breakdown. Operations include `readfile`, `openfile`, etc. Throughput in MB/s, operations/s and latency are reported. Throughput values discussed below will be specified as read or write throughput. If unspecified, the actual throughput reported by Linux stat files is used. All efficiency results shown are using the combined read + write throughput reported by filebench.

Table 6.1: Filebench workload variables

Workload	Mean file size	# Files	Threads
fileserver	128 KiB, 512 KiB, 1 MiB	50000, 125000, 300000	5, 10, 20
varmail	16 KiB	950000	8, 16
webserver	16 KiB	5000, 50000	20

Setup. First, the general setup is run section 6. The variables available for each benchmark Table 6.1 are set using the filebench console, after loading the appropriate workload file. The variables were tested by isolating each variable, but not all combinations were tested. For varmail and webserver, the default filesize was used since this is an essential characteristic of those benchmarks, whereas a wide variety of file sizes could exist on a fileserver. All 3 power states were tested for each combination run. We tested EXT4, XFS & BTRFS filesystems. To try and prevent host level page caching from causing too many issues with our results, we try to clear system page caches after creating our benchmark dataset as described in (54). `system sync`, and `system "echo 3 > /proc/sys/vm/drop_caches"` are run to free cache. This ensures the entire initial file-set

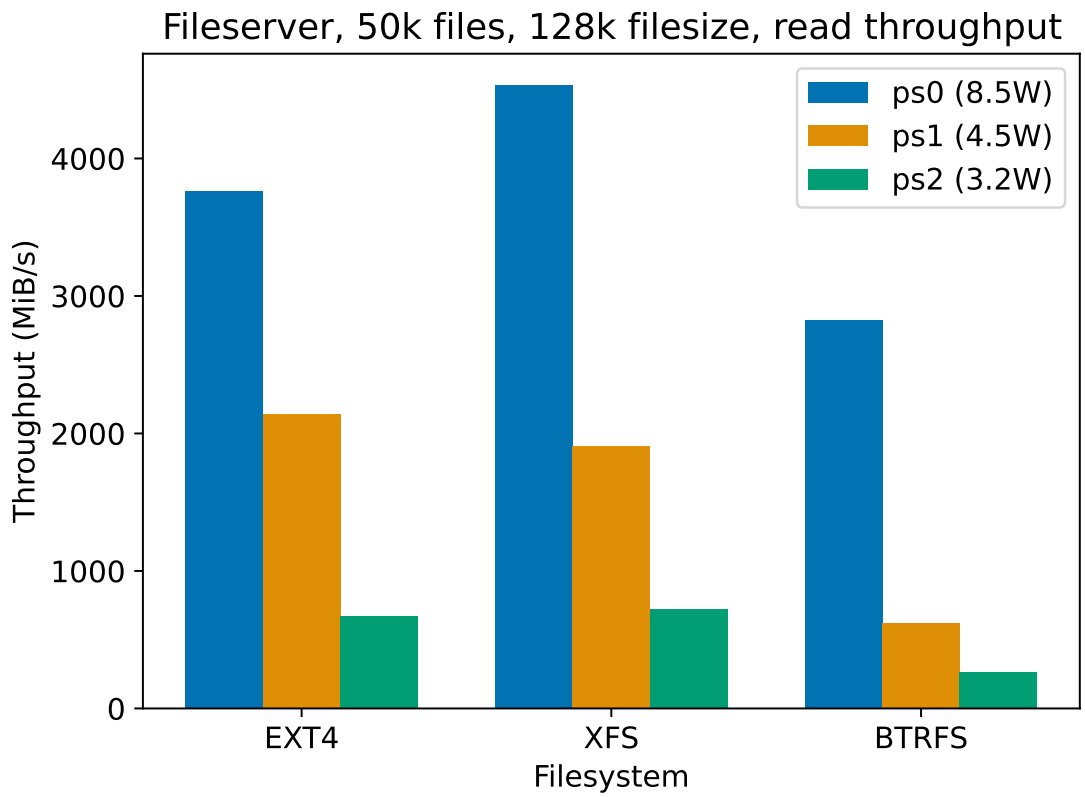


Figure 6.1: Filebench fileserver read throughput

is flushed to disk, and also ensures that any page cached filesystem blocks are cleared. The benchmark variables are also designed where possible to ensure sufficiently large datasets are used to prevent caching.

Findings. In the fileserver benchmark, BTRFS throughput lagged behind XFS & EXT4 by as much as 40%, but equaled them in conditions more favorable to higher throughput (1 MiB or 300k files). Similarly, in most benchmarks, EXT4 managed slightly higher throughput than XFS, showcasing a clear performance line from EXT > XFS > BTRFS in fileserver. Comparing efficiency at the reported EXT4 throughput around 3000 MiB/s at 1 MiB filesize, to similar conditions in our microbenchmarks section 5.3 we see that SSD energy efficiency is almost identical, and combined efficiency is 2% lower in microbenchmarks, but note that recreating an identical scenario using fio is not possible. Here we encountered an issue: In situations where the dataset size was smaller (128 KiB, 50k files), throughput exceeded limits previously observed, such as more than 1000 MB/s at PS1, as

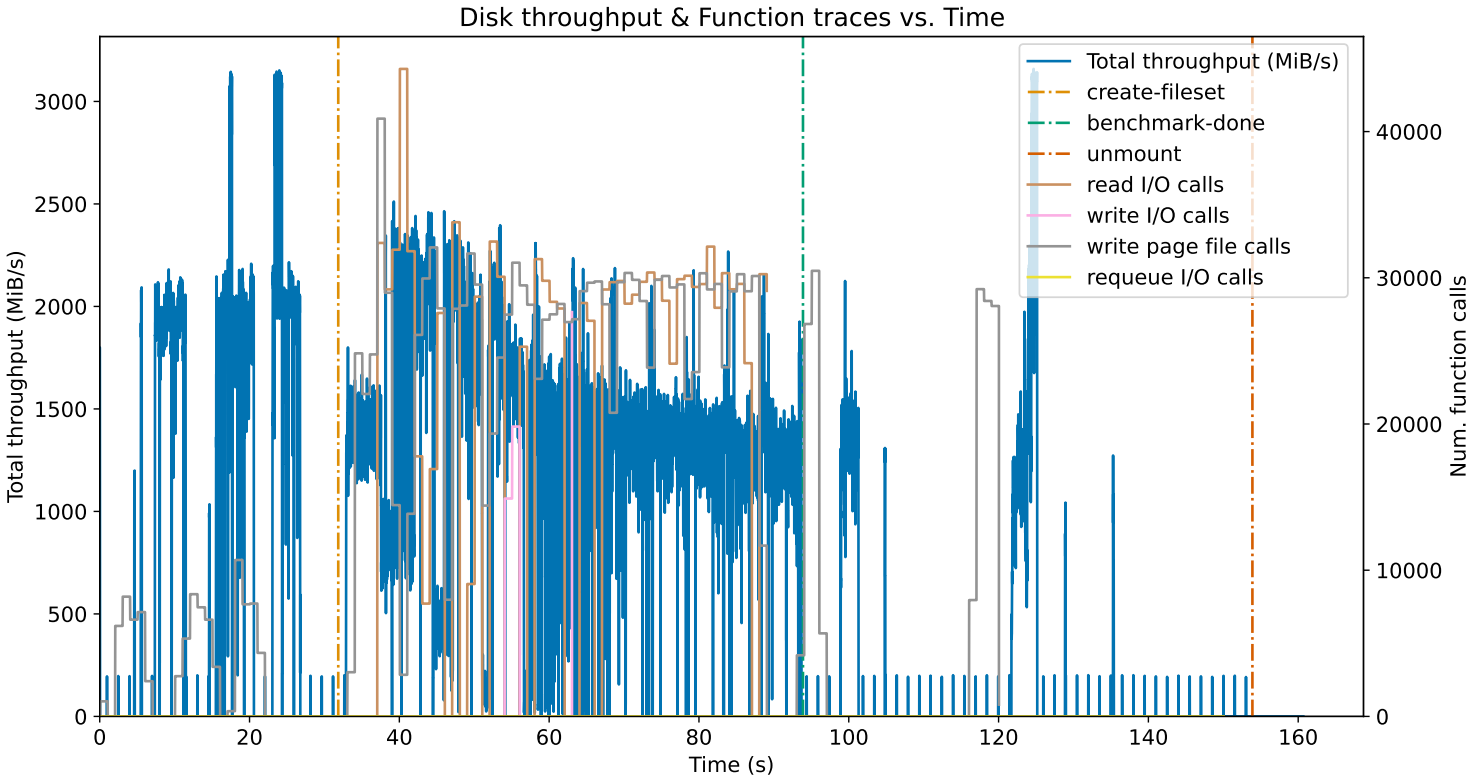


Figure 6.2: Filebench fileserver disk throughput & function call counts

well as theoretical limits of over 4 GB/s since our SSD is connected toa PCIe gen 3 4x slot Figure 6.1. ***F14.*** Page cache effects can result in inaccurate throughput results, and can even double the actual throughput & efficiency.

When looking at power, increasing filesize had the expected result of reducing CPU power - but an unexpected SSD power tradeoff is visible, a non-monotonic power profile (128 KiB $<$ 1 MiB $<$ 512 KiB) is visible Figure 6.3. This is likely an effect of both the benchmark itself and the request size. Based on the function call count graph Figure 6.2, we can see that around 60k I/O functions (read + write) are called every second at 128 KiB, in an attempt to saturate the drive. At 512 KiB the I/O function call count is around 25k requests per second, and 12k requests per second at 1 MiB. This translates into how many requests the SSD is having to handle at any given time. At 1 MiB, the drive likely has more time to perform internal power gating and DVFS but 512 KiB hits the perfect spot to keep the drive more engaged - which also makes sense because the maximum data transfer size (MDTS) for our drive is 512 KiB, so the drive will have to spend more time just to transfer data back to the host at 1 MiB.

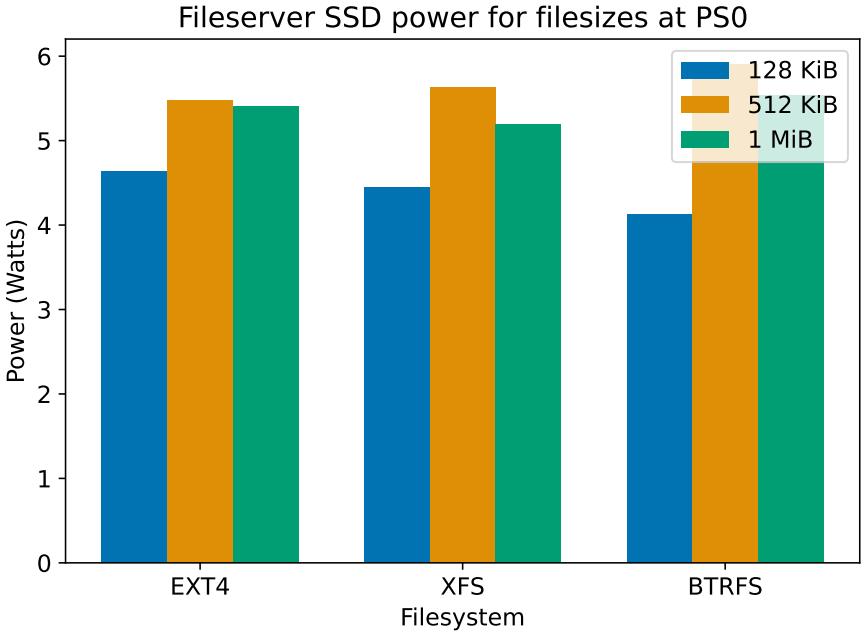


Figure 6.3: Filebench fileserver PS0 SSD power

As observed in Figure 6.2, ***F15.*** Filesystems make thousands of I/O function calls every second something not very efficient. Instead, a more complex design that properly asserts back-pressure onto the SSD while adhering to our recommended I/O efficiency guidelines Table 5.1 will bring large efficiency gains. This design, without consideration of back-pressure is clear when taking a closer look at function call data - the Linux multiqueue block layer is constantly having to re-queue I/O to the SSD because the SSD's hardware queues were full.

In the varmail benchmark, a benchmark with a heavy focus on filesystem metadata performance because of deeply nested directories, throughput was very poor peaking at just 17 MB/s (reads) and as a result only a 5% power difference between PS0 and PS2 in this test, and just 18% lower IOPS/J between PS0 and PS1 in EXT4 with 16 threads. This is the smallest gap in efficiency between PS0 and PS1 in all experiments conducted so far. The webserver benchmark with a smaller dataset (5k files) showcased the effects of caching, with all 3 power states having identical throughput. We also find that increasing thread count to parallelize has negligible effects, roughly a 3% increase in bytes/J efficiency when going from 10 to 20 threads.

Recommendations. Based on our findings, it's clear that smaller datasets can show inflated performance & efficiency compared to actual I/O. Parallelizing I/O to different files does not scale well as seen with threads in microbenchmarks. Our recommendation is therefore to stick to EXT4, and always stick to PS0 due to the poor efficiency in other power states, and the poor efficiency of other filesystems, though XFS efficiency was within 5% of EXT4 efficiency. The tradeoffs in filesystems draw many parallels to our microbenchmark tests, with larger file sizes and smaller numbers of files yielding better efficiency while having a large number of files can reduce efficiency because of filesystem metadata overheads.

6.2 RocksDB - YCSB

RocksDB is a popular embedded key-value store used by numerous companies (55). In order to benchmark RocksDB, we decided to use the industry standard tool YCSB (49). YCSB provides realistic and standardized workloads to benchmark NoSQL database platforms. We chose workloads denoted as A, B, C and F based on studies such as (56). These workloads have the read-update splits of 50 - 50, 95 - 5, and 100 - 0. Workload F performs read - modify, write at 50 - 50. Studying energy contributions and energy-performance tradeoffs for RocksDB will help answer RQ3.

RocksDB uses log-structured merge (LSM) trees to store data on disk. When a write is performed, it is first written to a write-ahead log (WAL) for durability and fills the in-memory data structure. Once this data structure is filled, it gets flushed to disk. At preset sizes, the logs are compacted into 7 layers, they increase in size and age, with the oldest data in the largest layer (L7). This compaction helps save space and improves read performance, but it is a costly operation to perform.

YCSB does not report disk throughput, only operations/s, so our efficiency results use the ops/J metric. CPU power is not included in our efficiency calculations here, because it was identical across all tests.

Setup. First, the general setup is run section 6. We used EXT4 for our tests. First, YCSB's "load" command is run for the selected workload, to create the dataset. After this, we flush page caches using `sync`, and `echo 3 > /proc/sys/vm/drop_caches`. Once this is complete we run the workload using the YCSB "run" command. The workloads were run with 10 million records and 100 million records in the database. Only PS0 and PS1

were tested, since PS2 has already been established as a poorly performing and inefficient power state.

Findings. Firstly, workload 'A' runs at half the throughput of other workloads at PS0, which is in line with our microbenchmark findings for 50 - 50 mixed workloads. When looking at insertion performance, it is constant around 120k ops/s at 10m records, but drops to just 55k at 100m records, due to compaction. Based on our microbenchmarks, we would expect read ops to provide the highest throughput, such as in workload 'C', but instead workload 'F' provided the highest throughput at 100m records at 237k ops/s. At first look, it appears to showcase our other microbenchmark finding, that interleaving reads and writes can yield great throughput and efficiency thanks to reduced contention. However, examining our actual throughput to disk results shows average read throughput of 250 MiB/s for 'F' and only 178 MiB/s but a CPU load factor of more than 80, suggesting that the CPU overhead in RocksDB operations is too high to fully exploit the performance of these SSDs. While average throughput was low, there were sudden bursts crossing 2000 MiB/s. Thanks to the low average disk throughput, in all workloads except 'A' SSD power was 3.2 W. Since our CPU was saturated in all workloads, CPU power was nearly identical in all workloads. Workloads 'B' and 'C' were 210% more efficient than workload 'A'. Workload 'A' was so inefficient, that even workload 'B' at PS1 was 100% more efficient. ***F16.*** The effects of read-write interference are significantly worse in RocksDB when compared to isolated microbenchmarks. Optimizing RocksDB to prevent interleaving of read and write requests would improve performance & efficiency here.

Examining the subsection I/O graph for insertion Figure 6.4, we see that a lot of time is not performing any disk I/O, with gaps of 200 ms or more between each burst of I/O. This leads to the continuous up-down power bands. Such cycling in power, especially the idle time spent not issuing I/O is extremely inefficient, which is why insertion efficiency is 67% worse than the actual benchmark for workload 'B' (as well as other workloads). Insertion efficiency is roughly 50% lower at 100m records compared to 10m records. Efficiency at PS1 is roughly half that of PS0 in all workloads, because runtime roughly doubles due to the increased latency at PS1, something that was not considered in microbenchmarks since microbenchmarks do not process data and hence was never identified to be a serious issue for real world workloads, rendering PS1 unusable in application workloads as well.

Taking a look at whole system power, this is our first experiment with 100% CPU usage and as seen earlier section 5.2, the gap between CPU power and whole system power reduces with increased CPU load. This finding reflects here as well, with the gap now

Figure 6.4: YCSB Workload F, PS0 - Insertion

reduced to 20 W, compared to nearly 40 W at CPU idle. The system power seen earlier section 5.2 is 96 W with CPU low CPU load and a CPU power of 46 W. Comparatively in this experiment, system power is at 116 W while CPU power is 90 W. Based on studies that look at PSU efficiency graphs (57), we can state that this is a result of moving into the ideal efficiency range for the PSU.

Effects of no write-ahead log. As mentioned above, RocksDB has a write-ahead log for durability. In the event of a crash, the write-ahead log is used to recover data, since all operations are first committed to it. RocksDB allows skipping the write-ahead log altogether for systems that implement their own recovery logic or want to improve performance in scenarios that do not require durability. This is done by setting `WriteOptions::setDisableWAL`.

We decided to look at workload ‘A‘ since this had the most to gain. At 10m records, workload ‘A‘ efficiency improved by 22%, but insert ops without the WAL provided a 76% higher efficiency, but only a 30% higher average throughput to the disk. Removing the WAL from RocksDB yields clear efficiency gains, but update efficiency does not improve much. This shows that microbenchmarks findings might not translate perfectly to applications due to their complexity.

Tuning for improved efficiency. RocksDB has a variety of options to tune it for better performance in various settings. To try and apply some of our microbenchmark recommendations here to extract efficiency. We tuned `write_buffer_size`, `max_total_wal_size`, `db_write_buffer_size`, `enable_pipelined_write`, but also experimented with other options that proved ineffective. With these options, the aim was to increase throughput and reduce the number of small I/O operations in favor of large sequential I/O to maximize efficiency. Tuning and testing with workload 'A' resulted in 13% efficiency gain for insert ops, but a 3% efficiency drop in read & update ops. Again, this reflects our statement from earlier ***F17***. Microbenchmark findings do not always translate to efficiency gains in complex real world applications using configuration parameters or options.

6.3 MLPerf Storage

MLPerf is a suite of benchmarks built by MLCommons, a collective that conducts machine learning research. Among their benchmarks is the MLPerf storage benchmark (58). This benchmark aims to emulate data ingestion, training and inference of machine learning models, and thanks to emulation, GPUs are not required for the benchmark. The benchmark lets you choose the number of accelerators to emulate choosing either the Nvidia A100 or H100, the maximum DRAM for the benchmark, as well as the hosts/nodes. MPI is used as the backend to execute the benchmark. While the official results provided by the MLPerf team show that the benchmark is intended to run on very large systems providing hundreds of gigabytes of throughput, it should still provide useful insights using a small number of accelerators on a single node. Understanding energy-performance tradeoffs in the popular space of ML will help answer RQ3.

We hypothesize that the type of model and the type of data will vastly change read characteristics and therefore efficiency. Small training files will require lots of I/O ops, lowering performance but large sequential I/O for large training files will yield better efficiency.

For this benchmark, latency is not a key metric. Throughput is recorded as mean samples/s, as well as MB/s. Accelerator utilization was also recorded, but this will not be a focus because GPU power characteristics for the emulation do not exist and are not the focus of this study.

Setup. We ran all three ML models provided with MLPerf - U-Net3D, ResNet-50 and CosmoFlow on EXT4. The general setup section 6 is not fully followed, because dataset

sizes for the benchmark are large enough, often covering most of the drive's capacity, negating the need for a "fill" file. Memory was configured as 230 GiB (does not matter since our accelerator count is low), using the A100 preset. For each experiment we first run one accelerator, then estimate peak throughput and run a range below and above this estimate to find the peak throughput configuration for each model, to maximize SSD throughput. Before training, the dataset was first generated using the MLPerf tool, but only a small subset of the dataset required for each model was generated, since the complete datasets for all the models are far larger than our device capacity. This is not a problem, since the benchmark runs for multiple epochs covering at least half of the drive capacity in each epoch.

Findings. Firstly, we identified the ideal number of accelerators as 6 for U-Net3D and CosmoFlow, and 35 for ResNet. ResNet and U-Net managed throughput over 3 GiB/s with a bytes/J efficiency of 29 and 26 respectively, while CosmoFlow managed only 2.2 GiB/s and an efficiency of 19. Efficiency is relatively poor compared to microbenchmarks, where a similar throughput yields an efficiency of 40, which is a 50% increase. Our CPU was completely pinned in all tests, with a 50% higher CPU power when compared to RocksDB. As a result, efficiency calculated only using SSD power is excellent, with the efficiency of U-Net 3.4% higher than the best efficiency achieved in our microbenchmarks at the same throughput. Examining the throughput & power graph, we see constant ups and downs in actual disk throughput. This is seen with all three models. Unlike with the up-down throughput we noticed in our filesystem experiment section 6.1, the gap between I/O is too small preventing the SSD from throttling down power gating, which as we saw earlier is very inefficient. ***F18.*** We hypothesize the reason this is more efficient when compared to a microbenchmark is the timing: Just enough of a break so the controller on the SSD does not consume enough power. ResNet had a larger number of training files, 200000 compared to just 4000 in the other two, yet it managed to maintain over 3 GiB/s of throughput a clear indication that the MLPerf storage benchmark is well designed to extract performance in sequential read I/O which is why it almost matches microbenchmark performance.

Accelerator utilization for ResNet and CosmoFlow was around 80%, but U-Net only 34%. Even models that do not best utilize the GPU appear to saturate current single SSD performance, suggesting that SSDs are fast enough for GPUs during training currently.

6.4 PostgreSQL - TPC-C

PostgreSQL is a very popular open source SQL database used by many large companies. It is very extensible and is used today for numerous other use cases apart from being a traditional relational database. Addons allow it to operate with geo-spatial data, time series data and even as a vector database for AI embeddings.

The Transaction Processing Performance Council (TPC) is an organization run by industry giants, and provides standardized benchmarks for transaction processing, big data, AI, and many more. TPC-C is their Online transaction processing (OLTP) benchmark. It consists of 5 transaction types, 9 tables and a variety of data sizes. TPC-C is not a tool, but only a specification for the benchmark.

The popularity of PostgreSQL makes it an ideal application to understand energy-performance tradeoffs to help answer RQ3.

Here, the main metrics are transactions per minute - C (TPMC), throughput in requests/s, and 'efficiency'. This 'efficiency' is not energy efficiency, but a calculation of how well the available resources on the machine are utilized and is returned by TPC-C. We will therefore express efficiency as TPMC/J.

Setup. The TPC-C PostgreSQL implementation we used had scripts for simple setup through Docker (59), so this experiment was fully run on Docker. First SSH keys are generated for each Docker container, and provided to the TPC-C benchmark helper scripts. These are used to upload the benchmark to the Docker containers and set them up. After this, a host helper script is used to perform load & run actions. The benchmark has a host, a client, and the PostgreSQL database. The entire general setup was not followed section 6, but a 500 GiB prefill file was used. The data directory for the PostgreSQL container was mounted using a regular Docker host mount. Performance in Docker host mounts is identical to bare-metal performance. The benchmark was run with 1000 & 2000 warehouses, at PS0 and PS1. Only one client was used, since initial testing showed it was able to saturate throughput by itself. The default benchmark configuration for order ratios, warm-up time and batch size was used.

Findings. At 1000 warehouses, PS1 TPMC/J efficiency is just 7.8% lower than PS0. This makes sense since the difference in TPMC was just 1%. TPC usage efficiency at PS0 was 100% and 98% at PS1. When looking at the I/O graphs, there are two large write operations near maximum throughput (3 GiB/s) at the start, followed by throughput

varying between 50 MiB/s and 150 MiB/s for the entire benchmark duration, as well as 1 - 3 second short bursts around 1000 MiB/s. At 2000 warehouses, TPMC/J efficiency is 45% greater than at 1000 warehouses, and PS1 is no longer viable being 30% less efficient. I/O characteristics are very similar to 1000 warehouses, except with higher throughput peaks for most operations. Looking at actual disk I/O, bytes/J efficiency only considering SSD is terrible, around 70 to 90, around 600% lower than the peak SSD only efficiency observed throughout all experiments. ***F19.*** Complex applications like databases do not seem to optimize for energy efficiency, since similar poor efficiency in earlier microbenchmarks was seen at 4 KiB with an I/O depth below 4.

Tuning PostgreSQL for improved efficiency. To improve efficiency based on our microbenchmark findings Table 5.1, we configured and tuned various parameters using the PostgreSQL configuration file. Tuned parameters include: `shared_buffers`, `work_mem`, `effective_io_concurrency`, `wal_buffers`, `max_wal_size`. Like in RocksDB, PostgreSQL also uses a WAL log, but this cannot be disabled. Tuning provided a 13% increase in TPMC/J efficiency, as well as a 36% increase in actual disk I/O SSD only efficiency, however there was still a 420% gap to the peak SSD only efficiency observed. As discussed with RocksDB section 6.2, these high level applications are too complex to extract larger efficiency gains just by tuning options. A large architectural change & redesign would likely be required to maximize energy efficient I/O.

We also wanted to see the effects of EXT4 tuning on efficiency. To do this, we mounted the EXT4 filesystem using the following options string: `defaults,commit=60,data=ordered`. The main component is the ‘commit‘ option, which only writes filesystem metadata changes to disk every 60 seconds, which should reduce the frequency of small low I/O depth random writes & reads made when updating filesystem metadata blocks. This change by itself improved efficiency by 15%.

6.5 Summary

From our above findings, we summarize that application workloads that perform work like RocksDB and TPC-C are very complex and have not been designed with efficiency in mind, as reflected in their poor I/O efficiency results - while simple filesystem benchmarks exhibit better efficiency when compared to our microbenchmark results. We clearly see how different applications and workloads have distinct energy consumption patterns.

Filesystem benchmarks using Filebench show large variations in throughput and both CPU & SSD power depending on the filesystem type and configuration, with various factors like the page cache playing a role. We find strange non-monotonic power results in our SSD: 128 KiB $<$ 1 MiB $<$ 512 KiB, a result of I/Os issued to the drive and MDTS resulting in overheads. We also find that not all variables from our microbenchmarks translate to filesystems, with parallel operations having poor scaling resulting in poor efficiency due to increased CPU power, especially when compared with microbenchmarks.

In RocksDB, we see that workloads with read-write interference can severely worsen efficiency, resulting in the lower PS1 power state of a different workload without interference to have higher efficiency - something completely unexpected considering the terrible efficiency of PS1 seen in microbenchmarks. Here we see the applications like RocksDB are far too complex to achieve great efficiency improvements without a change in design, as tuning RocksDB options results in a 13% efficiency gain for insert ops, but a 3% drop in efficiency for read & update ops. We also see that removing the WAL greatly improves efficiency by removing the need for extra writes.

MLPerf Storage was the only application workload that closely resembled our high throughput microbenchmark results, with SSD efficiency being greater than or equal to our microbenchmark SSD efficiency.

In TPC-C, actual disk throughput is very low, showing how PostgreSQL performs small I/O, which might have low latency but leads to poor efficiency, around 600% lower than the peak SSD efficiency that the drive is capable of. Tuning PostgreSQL using various options improved efficiency, but it was still far from the peak SSD efficiency observed in microbenchmarks. Just like in RocksDB, we see that it is too difficult to tune complex applications for efficiency.

The general process in modifying application configurable or parameters to improve efficiency is based on our microbenchmark recommendations Table 5.1. For example, increasing buffer size parameters and concurrency parameters.

7

Conclusion

In this thesis, we have conducted a comprehensive study into energy contribution and energy-performance tradeoffs while varying numerous host I/O knobs. We began by conducting a survey into SSD energy research to understand its current state, using which we decided on the appropriate I/O knobs to conduct experiments on, which we hypothesized would yield insights into energy-performance tradeoffs. After this, we built a tool to record & integrate results from our numerous data sources & sensors with automated plotting to help in drawing insights from all our data. We analyzed this data to derive a set of recommendations to help developers in creating more energy-aware systems. Finally, we explored the energy-performance tradeoffs of various real world applications to understand and test the applicability of the recommendations from our previous tests.

In this section, we succinctly provide answers to our research questions, and then discuss some of the limitations and ideas for future work to expand further on SSD energy research and energy-aware I/O.

7.1 Research Questions

RQ1 **What is the state-of-the-art & state-of-practice in SSD energy research, what are the methodologies & metrics used for benchmarking, and what are the gaps in research?**

In chapter 3, we identified the state-of-the-art in & state-of-the-practice in SSD energy research by firstly, examining known energy & performance characteristics of SSDs such as read-write asymmetry and the key metrics for efficiency: **IOPS/J** & **bytes/J**. Next looked at the effects of Internal SSD design that can explain various energy & performance characteristics. Then we determined the effects of various

parts of the Linux I/O stack such as schedulers. Finally we identified the gaps in SSD energy research: No study looked at CPU, SSD & whole system power to derive insights.

RQ2 **How do various host - manged configuration knobs derived from $RQ1$ effect power consumption of the SSD, CPU & whole system?**

In chapter 5, we studied 8 I/O knobs and determined configurations that provided performance, and configurations that provided efficiency. In order of relevance: Request Size, I/O Engine, Mixed workloads, and I/O depth were determined as some of the most important I/O knobs that can be used to maximize energy efficiency, with individual knobs providing up to a 25% increase in efficiency. We also determined that CPU power, being much larger than SSD power is a large contributor to energy and the largest efficiency gains come from optimizing CPU & therefore whole system power. Whole system power reflects general power trends across experiments but is less reactive to CPU power changes due to power supply efficiency losses, adding between 20% to 68% on top of CPU & SSD power in our setup, with the gap reducing as CPU load increases. Finally using all our findings in this chapter we derived a table of general recommendations to maximize energy efficiency.

RQ3 **How do various application specific and general configuration parameters as mentioned in $RQ2$ effect application workloads?**

In chapter 6, we examined the energy & performance of filesystems and 3 relevant real world applications. In filesystems, we determined that overhead from filesystem metadata operations hurt efficiency & performance. We also determined that certain variables such as Threads do not scale well in filesystems, in comparison to their scaling in chapter 5. We also realized that system page caching effects can interfere and inflate results if experiment configurations are not carefully adjusted to account for them. In our other real world applications, we applied our recommendations from chapter 5, and discovered that tuning configuration parameters alone may not yield efficiency gains in all applications - due to the inherently complex nature of these applications. However we managed to extract a 13% efficiency gain in RocksDB. We also determined that relaxing filesystem metadata durability rules removes performance bottlenecks - leading to a 15% efficiency gain.

7.2 Limitations & Future Work

The first and arguably largest limitation of our study is that we only tested a single NVMe SSD. As discussed in the survey section 3.3, numerous factors differentiate SSDs and can lead to different performance & therefore energy characteristics, with increased skew under various workloads although the general findings should remain the same. Conducting this study with multiple SSDs, especially from different manufacturers, especially with data center SSDs is important. Today, various manufacturers market and design SSDs to excel at specific types of workloads, it is important to test and characterize energy in these specialized SSDs.

Our test setup had a single SSD. However SSD storage servers often have 10 or more, and as many as 24 SSDs in a single system. Such systems might exhibit different CPU & SSD power characteristics, with many SSDs sitting idle. Such a study would have to come up with more elegant ways to integrate PCIe power measurement tools into their server chassis, since such systems often do not have much leftover space for extra components.

Most data center storage solutions work with complex networked systems, as NAS or SAN / JBOD deployments. Many storage systems also use RAID for redundancy. In modern SSD RAID deployments, the CPU is often the bottleneck, being unable to compute parity calculations fast enough, meaning many of the SSDs sit at very low throughput for long periods. Understanding energy efficiency in these systems is also important to apply this research towards saving energy in data centers.

In our SSD, it was clear that the lower power states were designed for power saving, and were very inefficient compared to the highest power state. Manufacturers of SSDs, especially in data center settings should work to design power states that are more useful in gating performance & while achieving great efficiency.

Finally, the findings from our study and other studies into SSD energy should be used to build an energy-aware I/O stack. In the process of this study, we tried to build an energy-aware I/O scheduler but were time limited - and the complexity of working with the internals of the Linux block layer, with very limited public facing documentation as well as no documentation for Linux I/O scheduler design & internals proved difficult. Linux I/O schedulers are somewhat limited in their power to control or redirect I/O, greatly limiting what is possible using just a Linux I/O scheduler. As a result, we think that an entire I/O stack designed from the ground up to allow for energy-aware operations, integrating NVMe power state control is likely to yield large power savings.

References

[1] Alexandru Iosup, Fernando Kuipers, Ana Lucia Varbanescu, Paola Grosso, Animesh Trivedi, Jan Rellermeyer, Lin Wang, Alexandru Uta, and Francesco Regazzoni. **Future Computer Systems and Networking Research in the Netherlands: A Manifesto**, 2022. 1, 4

[2] **Cloud Computing Market Size, Share & Industry Analysis** [cited 2025-06-25]. 1

[3] **Huawei Computing 2030** [cited 2025-07-20]. 1

[4] **A Balancing Act: HDDs and SSDs in Modern Data Centers** [cited 2025-06-25]. 1

[5] **SSD vs HDD for Enterprise Servers: Performance Guide** [cited 2025-07-26]. 1

[6] Rich Kenny and Jonmichael Hands. **Data Explosion, Circular Solution: How Refurbished Drives Can Bridge the Storage Supply Gap**. *European Journal of Business Management and Research*, **9**:71–75, 11 2024. 1

[7] **Data Center SSD Market Size, Share & Trends Analysis** [cited 2025-06-25]. 1

[8] Cong Xu, Suparna Bhattacharya, Martin Foltin, Suren Byna, and Paolo Faraboschi. **Data-Aware Storage Tiering for Deep Learning**. In *2021 IEEE/ACM Sixth International Parallel Data Systems Workshop (PDSW)*, pages 23–28, 2021. 1

[9] Ana Radovanovic, Bokan Chen, Saurav Talukdar, Binz Roy, Alexandre Nobrega Duarte, and Mahya Shahbazi. **Power Modeling for Effective Datacenter Planning and Compute Management**. *IEEE Transactions on Smart Grid*, **13**:1611–1621, 2021. 1, 4

[10] **Sustainable performance in the datacenter** [cited 2025-03-07]. 1, 4

[11] **How data centers and the energy sector can sate AI's hunger for power** [cited 2025-07-01]. 1

[12] Dedong Xie, Theano Stavrinos, Kan Zhu, Simon Peter, Baris Kasikci, and Thomas Anderson. **Can Storage Devices be Power Adaptive?** In *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '24, page 47–54, New York, NY, USA, 2024. Association for Computing Machinery. 1, 8, 13, 15, 20, 21, 35, 38

[13] **SuperMicro Ultra A+ Server AS -2124US-TNRP** [cited 2025-02-19]. 1

[14] Yair Levy and Timothy Ellis. **A Systems Approach to Conduct an Effective Literature Review in Support of Information Systems Research**. *International Journal of an Emerging Transdiscipline*, **9**:181–212, 01 2006. 3

[15] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques For Experimental Design, Measurement, Simulation, and Modeling, NY: Wiley.* 04 1991. 3

[16] **System Benchmarking Crimes** [cited 2025-07-01]. 3

[17] **Open Science Training Handbook** [cited 2025-07-01]. 3

[18] **Dutch Data Center Statistics** [cited 2025-07-01]. 4

[19] **NVMe** [cited 2025-03-10]. 8

[20] **Samsung PM1743 E3.S 15.36TB** [cited 2025-03-10]. 8

[21] **Kioxia EDSFF** [cited 2025-07-26]. 9

[22] **NVMe Power Management** [cited 2025-03-10]. 9

[23] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. **SPDK: A Development Kit to Build High Performance Storage Applications**. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161, Dec 2017. 10, 13, 18, 21, 46

[24] Bryan Harris and Nihat Altiparmak. **Ultra-Low Latency SSDs' Impact on Overall Energy Efficiency**. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. USENIX Association, July 2020. 13, 15, 16, 19, 21

[25] Satoshi Imamura and Eiji Yoshida. **Reducing CPU Power Consumption for Low-Latency SSDs**. In *2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 79–84, 2018. 13, 20, 21

[26] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. **Towards an unwritten contract of intel optane SSD**. In *Proceedings of the 11th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'19, page 3, USA, 2019. USENIX Association. 13, 19

[27] Matias Bjorling, Philippe Bonnet, Luc Bouganim, and Björn Pór Jónsson. **uFLIP: Understanding the Energy Consumption of Flash Devices**. *Bulletin of the Technical Committee on Data Engineering*, **33**(4):48–54, 2010. 13, 16, 17, 20, 30

[28] Seokhei Cho, Changhyun Park, Youjip Won, Sooyong Kang, Jaehyuk Cha, Sungroh Yoon, and Jongmoo Choi. **Design Tradeoffs of SSDs: From Energy Consumption's Perspective**. *ACM Trans. Storage*, **11**(2), March 2015. 13, 15, 16, 17, 30, 35, 48

[29] Balgeun Yoo, Youjip Won, Seokhei Cho, Sooyong Kang, Jongmoo Choi, and Sungroh Yoon. **SSD Characterization: From Energy Consumption's Perspective**. In *3rd Workshop on Hot Topics in Storage and File Systems (HotStorage 11)*, Portland, OR, June 2011. USENIX Association. 13, 15, 16, 20

[30] Jie Zhang, Mustafa Munawar Shihab, and Myoungsoo Jung. **Power, Energy, and Thermal Considerations in SSD-Based I/O Acceleration**. In *USENIX Workshop on Hot Topics in Storage and File Systems*, 2014. 13, 17, 20

[31] Bryan Harris and Nihat Altiparmak. **When poll is more energy efficient than interrupt**. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '22, page 59–64, New York, NY, USA, 2022. Association for Computing Machinery. 13, 16, 18, 19, 21, 43

[32] Sidharth Sundar, William Simpson, Jacob Higdon, Caeden Whitaker, Bryan Harris, and Nihat Altiparmak. **Energy Implications of IO Interface Design Choices**. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '23, page 58–64, New York, NY, USA, 2023. Association for Computing Machinery. 13, 18, 19, 21, 45

[33] Zebin Ren and Animesh Trivedi. **Performance Characterization of Modern Storage Stacks: POSIX I/O, libaio, SPDK, and io_uring**. In *Proceedings of the 3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*, CHEOPS '23, page 35–45, New York, NY, USA, 2023. Association for Computing Machinery. 13, 18, 21, 41, 45

[34] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. **Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring**. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, SYSTOR '22, page 120–127, New York, NY, USA, 2022. Association for Computing Machinery. 13, 18, 21, 45

[35] Caeden Whitaker, Sidharth Sundar, Bryan Harris, and Nihat Altiparmak. **Do we still need IO schedulers for low-latency disks?** In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '23, page 44–50, New York, NY, USA, 2023. Association for Computing Machinery. 13, 19, 21, 41

[36] Zebin Ren, Krijn Doekemeijer, Nick Tehrany, and Animesh Trivedi. **BFQ, Multiqueue-Deadline, or Kyber? Performance Characterization of Linux Storage Schedulers in the NVMe Era**. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, ICPE '24, page 154–165, New York, NY, USA, 2024. Association for Computing Machinery. 13, 18, 19, 21, 41, 50, 51

[37] Hankun Cao, Shai Bergman, Si Sun, Yanbo Albert, Zhou, Xijun Li, Jun Gao, Zhuo Cheng, and Ji Zhang. **Answering the Call to ARMs with PACER: Power-Efficiency in Storage Servers**. In *Proceedings of the 38th IEEE International Conference on Massive Storage Systems and Technology (MSST)*, 2024. 13, 15, 16, 19, 20

[38] Junjie Qian, Hong Jiang, Witawas Srisa-An, Sharad Seth, Stan Skelton, and Joseph Moore. **Energy-Efficient I/O Thread Schedulers for NVMe SSDs on NUMA**. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 569–578, 2017. 13, 20, 21, 34

[39] Pablo J Pavan, Ricardo K Lorenzoni, Vinícius Machado, Jean Bez, Edson Padoin, Francieli Zanon Boito, Philippe Navaux, and Jean-François Méhaut. **Energy Efficiency and I/O Performance of Low-Power Architectures**. *Concurrency and Computation: Practice and Experience*, August 2018. 13, 20, 21, 41

[40] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. **RAPL in Action: Experiences in Using RAPL for Power Measurements**. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, **3**(2), March 2018. 20, 26

[41] **fio** [cited 2025-03-04]. 21

[42] **Samsung 980 Pro 1TB** [cited 2025-06-25]. 23

[43] Steven van der Vlugt, Leon Oostrum, Gijs Schoonderbeek, Ben van Werkhoven, Bram Veenboer, Krijn Doekemeijer, and John W. Romein. **PowerSensor3: A Fast and Accurate Open Source Power Measurement Tool**, 2025. 25

[44] Kaijie Fan, Biagio Cosenza, and Ben Juurlink. **FLEXDP: flexible frequency scaling for energy-delay product optimization of GPU applications**. In *Proceedings of the 19th ACM International Conference on Computing Frontiers*, CF '22, page 177–180, New York, NY, USA, 2022. Association for Computing Machinery. 28

[45] K. Agarwal, H. Deogun, D. Sylvester, and K. Nowka. **Power gating with multiple sleep modes**. In *7th International Symposium on Quality Electronic Design (ISQED'06)*, pages 5 pp.–637, 2006. 30

[46] **SQL Server I/O Size** [cited 2025-07-11]. 35

[47] Kai Shen, Ming Zhong, and Chuanpeng Li. **I/O system performance debugging using model-driven anomaly characterization**. In *Proceedings of the*

4th Conference on USENIX Conference on File and Storage Technologies - Volume 4, FAST'05, page 23, USA, 2005. USENIX Association. 35

[48] **Understanding Workload and Solution Requirements For PCIe Gen 4 SSDs** [cited 2025-07-20]. 48

[49] **Yahoo! Cloud Serving Benchmark** [cited 2025-05-13]. 48, 62

[50] **filebench** [cited 2025-03-10]. 57

[51] Jiaxin Ou, Jiwu Shu, and Youyou Lu. **A high performance file system for non-volatile main memory**. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery. 58

[52] Jian Xu and Steven Swanson. **NOVA: a log-structured file system for hybrid volatile/non-volatile main memories**. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, page 323–338, USA, 2016. USENIX Association. 58

[53] Guoyu Wang, Xilong Che, Haoyang Wei, Shuo Chen, Puyi He, and Juncheng Hu. **Boosting file systems elegantly: a transparent NVM write-ahead log for disk file systems**. In *Proceedings of the 23rd USENIX Conference on File and Storage Technologies*, FAST '25, USA, 2025. USENIX Association. 58

[54] Vasily Tarasov, Erez Zadok, and Spencer Shepler. **Filebench: A Flexible Framework for File System Benchmarking**. *login Usenix Mag.*, **41**, 2016. 58

[55] **RocksDB Users** [cited 2025-07-01]. 62

[56] Krijn Doekemeijer, Zebin Ren, Nick Tehrany, and Animesh Trivedi. **ZWAL: Rethinking Write-ahead Logs for ZNS SSDs with Zone Appends**. In *Proceedings of the 4th Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*, CHEOPS '24, page 9–16, New York, NY, USA, 2024. Association for Computing Machinery. 62

[57] Faisal H. Khan, Thomas D. Geist, Baskar Vairamohan, Brian D. Fortenbery, and Eric Hubbard. **Challenges and Solutions in Measuring Computer Power Supply Efficiency for 80 PLUS® Certification**. In *2009 Twenty-Fourth Annual IEEE Applied Power Electronics Conference and Exposition*, pages 2079–2085, 2009. 64

[58] **MLPerf storage** [cited 2025-03-04]. 65

[59] **TPC-C-Postgres** [cited 2025-06-25]. 67

8

Appendix

In this appendix we describe how to setup the experiment to execute and reproduce the results.

8.1 Artifact Checklist

- energy-benchmark tool: https://github.com/t348575/energy-benchmark
  Result & configuration files: https://github.com/t348575/ssd-energy-benchmark-artifacts.
- Compilation: Rust (benchmark tool), Python (plots).
- Publicly available?: Yes.
- Code licenses: Apache License 2.0.

8.2 Experiment Setup

To reproduce this study, our benchmarking tool is required. More extensive documentation on configuration specifics (available options) and design is available in the source repository.

1. Setup the hardware configuration as described in Table 4.1.
   - When setting up the PowerSensor3, ensure that the 3.3 V and 12 V PCIe rails are connected.
   - When installing the SSD, ensure it is installed into NUMA node 1, otherwise all our experiment configuration yaml files will need to be adjusted to NUMA node 0.

2. Clone the source repository https://github.com/t348575/energy-benchmark.
3. Install the rust toolchain https://www.rust-lang.org/tools/install.
4. Install Python 3 for plot generation https://www.python.org/downloads/.
5. Create & activate a Python virtual environment at the repository root.
6. Run `pip3 install numpy pandas matplotlib seaborn scipy`.
7. Setup all software dependencies as described in Table 4.2.
    - **Note:** TPC-C Postgres, MLPerf Storage, YCSB require their repositories to be cloned, and passed appropriately in the workload configuration files.
8. Install the NVMe CLI tool https://github.com/linux-nvme/nvme-cli
9. For SPDK setup:
    (a) Clone and set up the SPDK repository and its submodules https://github.com/spdk/spdk.
    (b) Replace `app/fio/nvme/fio_plugin.c` from the SPDK repository with our modified version available in our source repository at `benches/fio/fio_plugin.c`.
    (c) Build SPDK as described in their README.
10. Fill the `setup.toml` file with the following entries:
    - `benches = ["fio", "filebench", "cmd", "ycsb", "mlperf", "tpcc-postgres"]`
    - `sensors = ["rapl", "powersensor_3", "sysinfo", "netio-http", "diskstat"]`
    - `plots = ["fio-basic", "filebench-basic", "ycsb-basic", "mlperf-basic", "tpcc-basic"]`
    - `[filebench.features] = ["prefill"]`
    - `[ycsb.features] = ["prefill"]`
11. Run `cargo build`, which populates dependency `Cargo.toml` files.
12. Run `cargo build -p energy-benchmark -release` to build the tool.

8.3 Running an Experiment

1. Write a workload configuration file, or select one from the artifacts.
2. Perform the following workload configuration file checks:
   - If NUMA node is set, correct NUMA node & NUMA memory node is set.
   - Required benchmark application paths set under `bench_args` (fio, ycsb, tpc-c, filebench).
   - Correct NVMe device (e.g. `/dev/nvme2n1`) and correct NVMe device root (e.g. `/dev/nvme2`) are set under `settings`.
   - Netio PowerPDU 4KS IP address set under `sensor_args`.
   - `DiskStatConfig`'s `device` set to the NVMe device (e.g. `/dev/nvme2n1`).
3. Run the experiment using
   `sudo target/bench/release/energy-benchmark bench -c path_to_config_yaml`