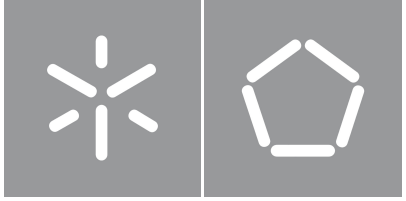


University of Minho
School of Engineering

José Pedro Dias Nascimento Fernandes

Comprehensive study of the energy impact of LSM-based key-value stores

april 2025



University of Minho
School of Engineering

José Pedro Dias Nascimento Fernandes

Comprehensive study of the energy impact of LSM-based key-value stores

Master's Dissertation in Informatics Engineering

Dissertation supervised by
Ricardo Gonçalves Macedo (Supervisor)
António Luís Pinto Ferreira Sousa(Co-supervisor)
Cláudia Vanessa Martins de Brito (Research Center Supervisor)

april 2025

Copyright and Terms of Use for Third Party Work

This dissertation reports on academic work that can be used by third parties as long as the internationally accepted standards and good practices are respected concerning copyright and related rights.

This work can thereafter be used under the terms established in the license below.

Readers needing authorization conditions not provided for in the indicated licensing should contact the author through the RepositóriUM of the University of Minho.

License granted to users of this work:



CC BY 4.0

https://creativecommons.org/licenses/by/4.0/

Acknowledgements

This thesis marks the culmination of my master's degree in Informatics Engineering at the University of Minho, a journey defined by formidable challenges and profound personal growth. I would like to express my sincere gratitude to everyone who has accompanied and supported me along this path.

First and foremost, I owe an immense debt of gratitude to my advisors, Professor Ricardo Macedo, Professor António Luís Sousa, and Professor Claúdia Brito. Their expert guidance, insightful feedback, and unwavering support have been instrumental in shaping the direction and quality of this work. Their dedication not only enhanced my understanding of complex topics but also ignited a passion for continuous learning and discovery.

A special thank you to my colleagues at HASLab for the great company and energising volleyball matches that made this journey more enjoyable. I'm especially grateful to Rúben and Pedro Peixoto for their valuable help and discussions on RocksDB, which helped me jumpstart and gain a deeper insight into this work.

I am deeply thankful to my family and closest friends, whose unconditional love and encouragement provided the strength needed to persevere through every challenge. Their steadfast belief in my abilities has been a constant source of motivation. To Diana, Mariana and Sara, thank you for being by my side through this journey. We've shared both the tough times and the joyful moments, providing constant support and understanding. I will truly miss our thirty minutes of sun basking while exchanging random hypotheses on the canteen stairs after lunch.

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020 through grant 11183/BI-L-EM_B3/2024. This work is co-financed by Component 5 -Capitalization and Business Innovation, integrated in the Resilience Dimension of the Recovery and Resilience Plan within the scope of the Recovery and Resilience Mechanism (MRR) of the European Union (EU), framed in the Next Generation EU, for the period 2021 - 2026, within project ATE, with reference 56 through grants 10371/BII-E_B4/2023 and 11004/BI-L-EM_B3/2024.



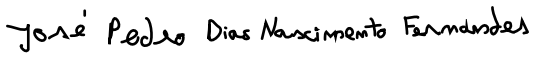
Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, april 2025



José Pedro Dias Nascimento Fernandes

Abstract

The exponential growth in digital data—from mobile devices, social media and **IoT** devices—has created an escalation of information travelling from the edge to the core layer. The core layer plays a crucial and central role in storage, analytics, and service delivery. **Key Value Store (KVS)** as RocksDB, Cassandra, among others, are widely used in the core layer due to their high throughput and low-latency solutions for managing and storing real-time data of large volumes of small-sized data. Recent efforts have been made to reduce the carbon footprint of data centres while improving their energy efficiency. However, due to the end of the Dennard scaling, reducing hardware energy consumption comes at the cost of worse performance. A shift to improving the energy efficiency of data centres through a software-centric approach is expected. As **KVS** support a significant share of services hosted in data centres, it is crucial to evaluate and improve their energy efficiency. Nonetheless, improvements in **KVS** have focused on performance and have neglected the energy impacts of their decisions. To the best of our knowledge, no existing **KVS** is solely designed with energy efficiency in mind, and there are no studies evaluating the energy profile of existing **KVS**. Evaluating the energy of these systems requires access to fine-grained energy monitoring tools, capable of capturing the energy of internal events. Notwithstanding, existing tools cannot capture energy at such fine granularity.

To facilitate the development of energy-aware **KVS**, we present an in-depth experimental study that characterises the energy consumption patterns of RocksDB, a widely used **LSM-Tree KVS**. We also present PowMon, a fine-grain energy monitoring tool capable of monitoring energy at the application, thread, and event levels to support this study's energy readings.

The study shows that energy efficiency is highly dependent on workload, data distribution, and hardware properties, underscoring the need for profiling when choosing the most energy-efficient configuration setup. Furthermore, the insights in this work can be used by **KVS** designers to create a more energy-aware **KVS** and also enable them to compare energy efficiency results against a RocksDB baseline efficiency benchmark.

Keywords Key-Value Store, Fine-Grained Energy Attribution, Energy Efficiency

Resumo

O aumento exponencial da informação digital—de dispositivos móveis, redes sociais e dispositivos **KVS**—tem criado um escalar de informação a ser transportada desde a periferia até à camada central de dados, responsável por serviços críticos de armazenamento, análise e processamento. Por outro lado, esforços recentes estão a ser feitos para reduzir a camada de carbono de centros de dados e melhorar a eficiência energética destas infraestruturas. No entanto, devido ao fim da escala de Dennard, reduzir o consumo energético do *hardware* só é possível com perdas de desempenho. Desta forma é esperado haver uma mudança de paradigma e resolver este problema através de uma perspetiva de *software*. Como **KVS** suportam uma percentagem considerável de serviços hospedados nos centros de dados, é crucial avaliar e melhorar a sua eficiência energética. Não obstante, desenvolvimentos de **KVS** têm se focado melhorias de desempenho e têm negligenciado os seus impactos energéticos. Tanto quanto sabemos, não existem estudos que avaliam o perfil energético destes sistemas de armazenamento. Avaliar a energia destes sistemas requer acesso a ferramentas energéticas de grão-fino, capazes de medir a energia de eventos internos, no entanto, ferramentas existentes não são capazes de medir energia a esta granularidade.

De forma a facilitar o desenvolvimento de **KVS** que sejam conscientes do seu consumo energético, apresentamos um estudo experimental compreensivo que caracteriza padrões de consumo de energia do RocksDB, uma **LSM-Tree KVS** muito utilizada na indústria. Também apresentamos PowMon, uma ferramenta energética de grão-fino, capaz de medir energia ao nível do processo, fio de execução e evento, de forma a suportar os dados energéticos capturados no estudo realizado.

O estudo realizado mostra que a eficiência enegética é muito dependente do tipo de carga e padrões de operações realizados, assim como de distribuição de dados e propriedades de *hardware*, salientando a importância de uma análise prévia para escolher a configuração mais energeticamente eficiente. Além disso, as aprendizagens adquiridas neste estudo podem ser usadas por desenvolvedores de novas **KVS** de forma a criar sistemas mais eficientes do ponto de vista da energia.

Palavras-chave Sistema chave valor, Atribuição de energia grão fino, Eficiência energética

Contents

1 Introduction **1**

1.1 Problem statement . . . 2

1.2 Objectives and contributions . . . 3

1.3 Document structure . . . 4

2 State of the Art **5**

2.1 Background . . . 5

2.1.1 Key-Value Stores: Motivation and Data Structures . . . 5

2.1.2 Log-Structured Merge tree . . . 7

2.1.3 RocksDB Production-grade LSM KVS . . . 12

2.1.4 Energy consumption monitoring . . . 14

2.2 Related Work . . . 20

2.2.1 Studies on LSM-based KVS energy efficiency . . . 20

2.2.2 Energy Monitoring Tools . . . 22

2.2.3 Summary . . . 25

3 PowMon - A fine-grained energy attribution monitoring tool with event correlation **28**

3.1 Design Overview . . . 29

3.2 Energy Monitoring Module . . . 30

3.2.1 Monitor Client . . . 31

3.2.2 Monitor Server . . . 31

3.3 Event Logging . . . 37

3.4 Event-Energy Correlation . . . 37

3.4.1 Event Energy Assignment . . . 38

3.5 Implementation . . . 38

3.5.1 Energy Monitoring . . . 39

3.5.2 Event Logging . . . 42
3.5.3 Event-Energy Correlation . . . 44

4 Energy Consumption in LSM KVS **45**
4.1 Experimental Setup . . . 45
4.1.1 Experimental Environment . . . 46
4.1.2 Experimental Design . . . 47
4.1.3 RocksDB Baseline Configuration . . . 48
4.2 Energy Efficiency Under Different Workloads . . . 51
4.2.1 Energy consumption analysis of different types of events . . . 55
4.2.2 Energy consumption analysis of compaction events . . . 57
4.3 Energy Efficiency Under Different Storage Devices . . . 63
4.4 Energy Efficiency Under Different Memory Configurations . . . 67
4.5 Energy Efficiency Under Different Concurrency Configurations . . . 75
4.6 Energy Efficiency in Compaction Schemes and Durability Configurations . . . 85
4.7 Guidelines for Energy-Efficient KVS Design . . . 91

5 Conclusion **93**
5.1 Future work . . . 94

A Cumulative Compaction Analysis **104**

List of Figures

1 Visual representation of two data structure types, a trivial append-only out-of-place structure and a binary tree in-place data structures. 6

2 LSM-Tree structure with level organisation based on the tiered and partitioned levelled merge policies. 8

3 Visual representation of a merge operation across different merge policies. 9

4 Power zones supported by Intel RAPL. 17

5 PowMon architecture 29

6 Average energy efficiency YCSB workloads with default configuration. 52

7 Average energy consumption and throughput of YCSB workloads with default configuration. 52

8 Breakdown of workload A total energy consumption across compaction, flush and foreground threads, under the uniform and Zipfian data distributions. 55

9 Breakdown of workload C total energy consumption across compaction, flush and foreground threads, under the uniform and Zipfian data distributions. 55

10 Average compaction energy across the YCSB workloads, under the uniform and Zipfian data distributions after a warmup period of 300 seconds. 58

11 Average compactions duration across the YCSB workloads, under the uniform and Zipfian data distributions after a warmup period of 300 seconds. 58

12 Read I/O and event context for two compactions to output level 1. 61

13 Read I/O and event context for two compactions to output level 0. 61

14 YSCB workloads average energy efficiency in different disk devices. 63

15 YCSB workloads average energy consumption and throughput in different disk devices. . 63

16 Average energy efficiency for different memory configurations in write-only workload. . . 68

17 Average energy consumption and throughput for different memory configurations in write-only workload. 68

18 Average energy efficiency for different memory configurations in mixed read-write workload. 71
19 Average energy consumption and throughput for different memory configurations in mixed read-write workload. 71
20 Average energy efficiency for different memory configurations in read-only workload. . . 73
21 Average energy consumption and throughput for different memory configurations in read-only workload. 73
22 Average energy efficiency for different concurrency configurations in the write-only workload. 76
23 Average energy consumption and throughput for different concurrency configurations in the write-only workload. 76
24 Average energy efficiency for different concurrency configurations in the mixed read-write workload. 79
25 Average energy consumption and throughput for different concurrency configurations in the mixed read-write workload. 79
26 Average energy efficiency for different concurrency configurations in read-only workload. 81
27 Average energy consumption and throughput for different concurrency configurations in read-only workload. 82
28 Average energy efficiency for different compaction schemes and durability configurations in write-only workload. 85
29 Average energy consumption and throughput for different compaction schemes and durability configurations in write-only workload. 86
30 Average energy efficiency for different compaction schemes and durability configurations in mixed read-write workload. 87
31 Average energy consumption and throughput for different compaction schemes and durability configurations in mixed read-write workload. 88
32 Average energy efficiency for different compaction schemes and durability configurations in the read-only workload. 89
33 Average energy consumption and throughput for different compaction schemes and durability configurations in the read-only workload. 90
34 Cumulative compaction energy across the YCSB workloads, under the uniform and Zipfian data distributions after a warmup period of 300 seconds. 104
35 Cumulative compaction duration across the YCSB workloads, under the uniform and Zipfian data distributions after a warmup period of 300 seconds. 104

List of Tables

1 Related work summary - Energy monitoring tools 27

2 RocksDB baseline configuration - Benchmarking and Statistics 48
3 RocksDB baseline configuration - Memory Management 49
4 RocksDB baseline configuration - LSM Structure 50
5 RocksDB baseline configuration - Durability and File Management 51
6 YCSB core workloads description. 51

Acronyms

API Application Programming Interface.

CPU Central Processing Unit.

DRAM Dynamic Random Acess Memory.

FPGAs Field Programmable Gate Arrays.

GPU Graphics Processing Unit.

HPC High-performance computing.

Intel-RAPL Intel Running Average Power Limit.

IoT Internet of Things.

KVS Key Value Store.

libc Standard library C.

LoC Lines of Code.

LSM-Tree Log Structured Merge Tree.

MemTable Memory Table.

ML Machine Learning.

MSRs Model Specific Registers.

NUMA Non-uniform memory access.

NVMe Non-Volatile Memory Express.

NVML NVIDIA Management Library.

OS Operating System.

POSIX Portable Operating System Interface.

PP0 power plane 0.

PP1 power plane 1.

RAM Random Acess Memory.

RAPL Running Average Power Limit.

SATA Serial Advanced Technology Attachment.

SSTable Sorted String Table.

WAL Write Ahead Log.

YCSB Yahoo! Cloud Serving Benchmark.

Glossary

die Small block of semiconducting material in which a functional circuit is fabricated.

Chapter 1

Introduction

In recent years, there has been exponential growth in the creation and storage of data. Digital data sourced from mobile devices, social media and **IoT** devices have reshaped the global landscape. Businesses leverage data to make better business decisions, optimise internal operations and provide a tailored user experience to gain a competitive advantage. At the same time, users are in a unique position of being able to monitor almost every aspect of their lives, from biometrics to smart house appliances. All the data collected from an individual can then be used to create a personalised experience, from social media interactions to suggestions from media content providers [65]. A big chunk of data travels from endpoint devices—user devices and **IoT** sensors—to the core layer. The core layer plays a crucial and central role in storage, analytics, and service delivery, while also being responsible for storing the majority of data in circulation [66]. Extensive research has been conducted to develop storage solutions for managing and storing large amounts of data in the core layer. Such solutions include distributed filesystems like Google File Systems [28] and Hadoop Distributed File System [68], used for batch processing and analytics in big data environments; Object stores including, OpenStack Swift [56] and Amazon S3 [4] can be used to store large and unstructured data like media; and **Key Value Store (KVS)** like RocksDB [23], and Cassandra [26], among others, provide high throughput and low-latency lookups for real-time storage of large volumes of small-sized data.

Particularly, **KVS** have been used as backend storage for different services, from application data caching [22], stream processing [22], graph processing [67], to event log processing [12]. **KVS** are commonly chosen as the backend for modern services because they offer high throughput and low latency, along with a streamlined and easy-to-integrate **Application Programming Interface (API)**, primarily consisting of fundamental operations such as `Put(key,value)`, `Get(key)` and `Delete(key)`. It is common for a **KVS** to organise its internal structure following an **Log Structured Merge Tree (LSM-Tree)** [55] design. This design allows quick absorption of writes by buffering (random) writes in memory and asynchronously moving them (through a sequential, large write) to a persistent structure containing

several levels sorted by size. It can optimise writes by taking advantage of sequential writes and mitigate read degradation by using caching mechanisms and by keeping recent entries in the smaller levels, making it a good candidate for write-intensive workloads. The **LSM-Tree** is used as the underlying structure in a large number of modern **KVS**, including but not limited to LevelDB [27], RocksDB [23], Cassandra [26], PebblesDB [63], Dostoevsky [19] and Monkey[18], being widely used in cloud computing services and large-scale infrastructure projects. In turn, these **KVS** services serve hundreds of thousands of consumers, requiring the support of a large number of servers located in data centres [20]. The electricity costs required to maintain these data centres and servers are very high; thus, using energy more effectively is one of the most straightforward and cost-effective methods to save money and satisfy rising power demands [38].

1.1 Problem statement

Recently, there has been an effort to reduce the carbon footprint and improve the energy efficiency of data centres. This can be achieved by increasing the data centre hardware resource efficiency or by improving the energy efficiency of software. Increasing hardware energy efficiency while providing the required performance is not sustainable. This can be explained by the end of the Dennard scaling [24]. Essentially, reducing the size of transistors and thus improving performance while maintaining proportional power usage is no longer feasible. To increase performance, it is then necessary to increase energy consumption disproportionately. A shift to improving the energy efficiency of data centres through a software-centric approach is expected [6]. As **KVS** support many services hosted in data centres [22], it is crucial to evaluate and improve the energy efficiency of these backend storage services.

Notwithstanding, recent improvements in **KVS**, such as reducing tail latency and higher throughput, have mostly been achieved by higher code parallelisation, while reducing space utilisation and disk I/O has been tackled by compressing data and avoiding performing disk operations when possible [23]. Other works have focused on improving higher write throughput while maximising read performance by improving on the **LSM-Tree** structure [63]. These improvements are focused on performance and could negatively impact **KVS** energy efficiency. To the best of our knowledge, no **KVS** is deliberately designed to maximise energy efficiency, and research around this topic is practically nonexistent. Although new **KVS** alternatives must be designed to be energy efficient, to our knowledge, no existing studies focus on quantifying and describing the energy profile of existing **KVS**. Particularly, these systems are complex and can have a multitude of tuning knobs that significantly affect performance and resource utilisation; however, their

energy impact is unknown.

Measuring the energy consumption of a **KVS** and its internal mechanisms is not trivial. Firstly, existing energy monitoring tools do not provide the required granularity of measurements. The majority of monitoring tools only allow analysis of energy usage at the server or process level. However, this is problematic as **KVS** like RocksDB operate in a multi-threaded environment, where each thread has different responsibilities and energy consumption patterns. Furthermore, even with thread-level consumption metrics, existing mechanisms do not have the ability to associate the energy of the thread with the type of event being executed in the thread. For example, a **KVS** might have multiple threads running and performing distinct tasks. Some threads might be completing foreground requests; others might execute background tasks such as flushes and compactions. Energy usage can be influenced not only by the type of task being performed but also by the context surrounding it; compactions at different levels will have different complexities.

1.2 Objectives and contributions

The main goal of this thesis is to facilitate the development of energy-aware **KVS**, by uncovering energy inefficiencies and understanding the performance and energy consumption impact of specific operations and workloads (e.g., background operations, key access distribution, storage device). This will provide new insights on how to design energy-efficient **KVS**, contributing to the sustainable use of large-scale I/O infrastructures.

To achieve the aforementioned objective, this thesis makes two main contributions:

1. As a first contribution, we present PowMon, a new power monitoring system that enables measuring the energy consumption of **LSM-Tree KVS** with fine granularity, providing energy readings at the application, process, and thread-level. To correlate the energy consumption with events unique to **KVS**, PowMon integrates a logging framework that tracks thread events within the **KVS**, such as flushes and compactions, and correlates them with thread-level energy readings, allowing precise quantification of energy consumption of specific system components.

2. As a second contribution, we present an in-depth experimental study that characterises the energy consumption patterns of RocksDB, a widely used **LSM-Tree KVS** from Meta. The study analyses and evaluates the performance and energy consumption impact of RocksDB across multiple dimensions, including: (1) workload proportions and data distributions, (2) different storage devices, (3) concurrency, (4) memory-related optimizations, (5) durability, and (6) compaction schemes. Our

study identifies 17 insights and draws conclusions for designing energy-efficient **KVS**. To the best of our knowledge, this is the first study that systematically characterises the energy consumption of LSM-based **KVS**.

1.3 Document structure

This document is organized as follows. Chapter 2 lays out the necessary background of **KVS** inner workings and in energy monitoring tools. Moreover, it reviews the existing related work on energy monitoring tools. Following, Chapter 3 introduces PowMon, our monitoring tool, offering design and implementation details. Next, Chapter 4 presents the results and discussion of our comprehensive experimental study, subdivided into five subsections, each evaluating different aspects of **KVS**. Finally, Chapter 5 summarises our findings and insights, considering the broader implications of our work on building more energy-efficient **KVS** and presenting potential roadmaps for future research.

Chapter 2

State of the Art

This chapter covers the necessary background in **KVS**, delving specifically into **LSM-Tree KVS**. It then provides implementation details of RockDB, a highly configurable and widely used **KVS**. Following the background in **KVS**, this chapter will focus on the various approaches and mechanisms for measuring energy consumption in the cloud. Lastly, it will discuss the current state of research on energy efficiency in **KVS** and the tools available for monitoring energy usage.

2.1 Background

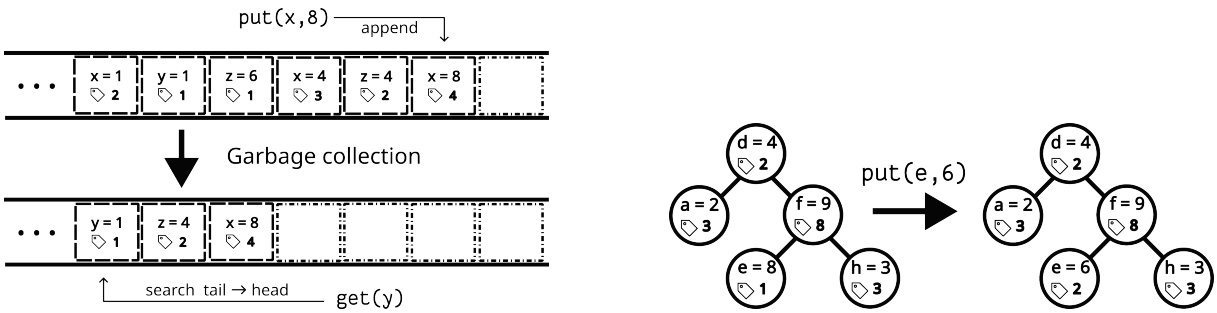
This section begins with a broad introduction of the **KVS** and the existing data structures that usually support these storage systems, namely the B^+ Tree and the **LSM-Tree**, and the trade-offs of using each structure. It then delves into the **LSM-Tree** structure and explains how this structure leverages the basics of an out-of-place structure to achieve better write performance.

Following the fundamentals of the **LSM-Tree**, this section highlights RockDB implementation details. These details range from components and techniques added to the **LSM-Tree** structure to improve durability, read performance, and space utilisation.

Following this background on **KVS**, the chapter will delve into the various approaches and mechanisms for measuring energy consumption. It will provide a background on the available methods for measuring power and energy consumption in the cloud, including their use cases and advantages and disadvantages.

2.1.1 Key-Value Stores: Motivation and Data Structures

KVS support fast ingestion and retrieval of data. They offer a simple **API**, consisting mainly of four operations: `put`, `get`, `delete` and `scan`. They are optimised for fast read/write operations because they do not need to parse complex queries or perform joins due to their limited interface. This means **KVS** can just focus on providing an efficient data ingestion and retrieval service. Furthermore, **KVS** provides



(a) Trivial Out-of-Place Structure

(b) Binary Tree In-Place Structure

Figure 1: Visual representation of two data structure types, a trivial append-only out-of-place structure and a binary tree in-place data structures.

good horizontal scaling as data can be easily partitioned to different servers. One effective example is Dynamo [20], a highly available distributed **KVS** from Amazon. Another aspect that makes **KVS** widely used in the industry is the fact that applications requiring flexibility in their data model can easily evolve without being forced into a fixed schema. This flexibility is possible due to the unstructured nature of **KVS** and big support for different value types. These characteristics make **KVS** a good candidate for the storage backend of different services, ranging from data caching [22], stream processing [22], graph processing [67], to event log processing [12].

Fundamentally, a **KVS** utilises a data structure for the storage of key-value pairs. Data structures can generally be divided into two groups, categorised as in-place or out-of-place, depending on how they update and store their entries. On one hand, an in-place structure does not require extra space complexity besides the original key-value space. Only the latest key version is stored, while the old version is removed from the structure. A trivial example of an in-place structure is a binary tree. A write requires finding the location of the key and then inserting the value into that location. The main advantage of this type of algorithm is the fact that the memory/disk footprint is minimal. The main disadvantage is that it creates a random I/O pattern, reducing the write throughput. On the other hand, an out-of-place structure relies on persisting a value relative to an existing key in a different location from the existing key's value, which inevitably leads to storing multiple versions of key values. A trivial example of an out-of-place structure is an append-only log. Reading the value associated with a key is achieved by performing a search that begins at the tail and moves towards the head of the log, stopping when a key entry is found or the head of the log is reached. Write operations are buffered and written to a sequential location on disk, maximising the disk throughput by performing fewer disk operations. Writing speeds are expected to be higher; however, read operations are slower, especially if a key is non-existent. In this structure, periodic operations must be

performed to delete obsolete versions of key values to prevent disk storage from growing uncontrollably, requiring extensive **Central Processing Unit (CPU)** resources.

The B^+ Tree—a variant of the B-Tree [9]—is an in-place structure and the **LSM-Tree** is an out-of-place structure, and both have long been the two main data structure design choices for **KVS**, each providing different performance trade-offs [36]. The B^+ Tree can achieve better read performance than **LSM-Tree**-based designs, while the latter can achieve an overall better write performance.

The B^+ Tree is a self-balancing tree where all the values are stored in the leaf nodes. The leaf nodes are linked sequentially, allowing range queries to efficiently find the next key in constant time. The internal nodes, however, only store keys and are balanced to guarantee a logarithmic search in a `get` query. Due to the costs of balancing and maintaining the overall structure, these data structures incur a bigger write cost, but generally provide a higher rate of reads. Examples of **KVS** using the B^+ are the WiredTiger [72], a common backend for MongoDB [52], and BerkeleyDB [54].

On the other hand, with a rising number of applications requiring fast ingestion of writes [36]—from stream processing and event log processing, among others—the **LSM-Tree** has become an attractive design solution to support modern **KVS**. Examples of **KVS** using an **LSM-Tree**-based data structure are LevelDB [27], RocksDB [23], Cassandra [26], PebblesDB [63], Dostoevsky [19] and Monkey [18]. We explore the **LSM-Tree** data structure in greater detail in the following subsection.

2.1.2 Log-Structured Merge tree

The basic **LSM-Tree** structure, originally proposed by *O'Neil et al.* [55], mitigates some of the drawbacks of the trivial out-of-place implementation. The structure is divided into two types of device components: a memory component and a group of disk components. The memory component buffers incoming writes, allowing fast ingestion, while the disk components are responsible for persisting data and are structured to facilitate sequential I/O and minimise the negative impacts on read performance.

Most implementations follow a set of common properties. First, disk components are structured across multiple levels, each differing in size. Typically, level N+1 is larger than level N by a configurable factor. Each disk component maintains only a single version of any given key, ensuring that key entries remain sorted within a component. Additionally, if a key has been inserted multiple times into the **LSM-Tree**, the most recent version is always found in a lower-numbered level than the older versions.

These are general guidelines. Some **KVS** implementations may loosely implement these properties in order to achieve better performance. In RocksDB, the first persistent level is usually bigger than the second to accommodate write bursts. Furthermore, at this level, keys can be unsorted, and more than

one key version might exist to reduce the complexity of the memory-to-disk operation.

Each disk component contains entries in an immutable index structure, typically the **Sorted String Table (SSTable)**. The **SSTable** is a persistent data structure optimised for efficient disk storage and fast accesses. The **SSTable** consists of a list of data blocks followed by an index block. The index block maps a subset of keys to their corresponding data blocks. Each data block contains a list of key-value pairs sorted in lexicographical order. These data blocks are optimised for disk storage and are typically similar in size. They are usually configured to be multiples of a page size, ensuring they are stored contiguously.

The **SSTable** offers good read performance by enabling quick searches through the index block mapping. Additionally, because it is an immutable structure, concurrency can be easily implemented around it.

Merge policy

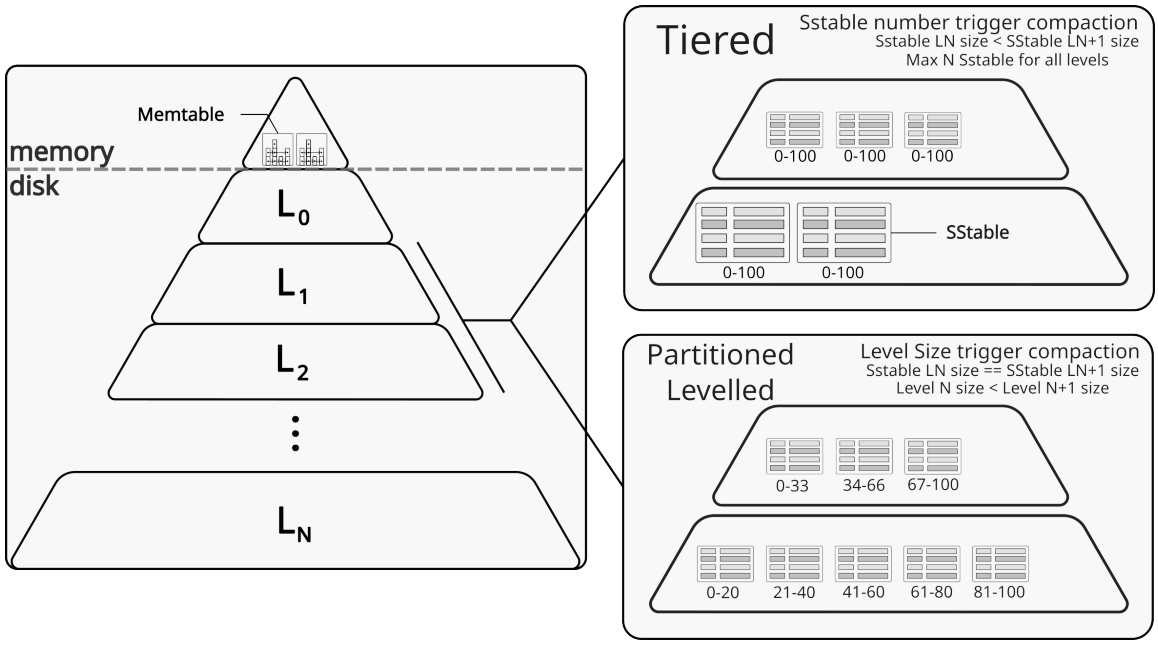
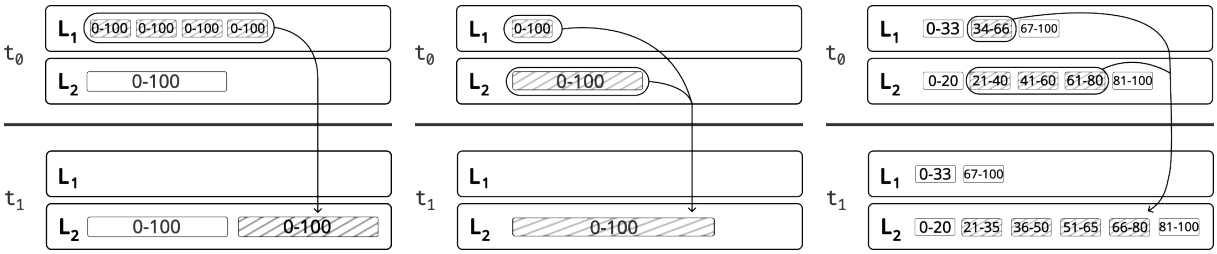


Figure 2: LSM-Tree structure with level organisation based on the tiered and partitioned levelled merge policies.

To minimise the space footprint of the structure, periodic merge operations are triggered. The goal of a merge operation is to move old entry versions to higher-numbered levels, freeing space for new writes in smaller levels. However, it also deletes outdated entries when it is sure it can safely do.

The merge policy determines how **SSTable**s are organized inside a level and how they get moved to the next level when their current level reaches a threshold. Two main merge policies are found in



(a) Tiered Merge (b) Levelled Merge (c) Partitioned-Levelled Merge

Figure 3: Visual representation of a merge operation across different merge policies.

modern **KVS**, the tiered (also referred to as universal) and the levelling strategies [49]. The tiered strategy contains more than one **SSTable** per level. Each level can have up to a threshold number of **SSTable**s of similar size, as can be seen in figures 2 and 3. When the threshold is reached, a merge is triggered. The level's **SSTable**s are merge-sorted to form a bigger **SSTable** on the next level. The input **SSTable**s are removed. If there is more than one entry with the same key in the multiple components being merged, the most updated version is kept; the other versions can be deleted.

The levelled strategy conceptually contains one **SSTable** per level, as can be seen in subfigure 3b. When the **SSTable** reaches a threshold size, a merge is triggered. The level N **SSTable** and the level N+1 **SSTable** are merged-sorted together to form a new **SSTable** in level N+1. The input **SSTable**s from levels N and N+1 are deleted. As in the tiered, if there is more than one key version being merged, only the latest version of the key is persisted on the new **SSTable**. The main advantage of the tiered policy is the fact that it has lower write amplification [18, 19]. Write amplification is a metric representing the amount of writes actually performed for a single intended write. In the tiered policy, each merge operation only requires the level's components and the result is flushed to the next level. On the other hand, in the levelled policy, an entry in the output level might be rewritten several times—increasing write amplification. This scenario often occurs when the output **SSTable** has yet to reach the threshold size limit; until the output reaches the threshold size, entries from the output level will keep being rewritten on the same level without advancing. The application is consuming **CPU** and disk resources by rewriting the same entry repeatedly without advancing its level. Nevertheless, by potentially rewriting the output level **SSTable** several times, obsolete versions get eagerly deleted on lower-numbered levels.

For the aforementioned reasons, the tiered policy is recommended for write-intensive workloads because it has less write amplification; for read-intensive workloads, it generally performs worse because, inside a level, several **SSTable**s have to be read. Furthermore, the tiered policy has a bigger space amplification, requiring more disk space available. By lazily merging, it will keep obsolete versions longer,

creating stale entries that may remain in high-numbered levels for a long time. On the other hand, the levelled policy is more indicated for read-intensive workloads because only one **SSTable** per level is read.

The partitioned levelled policy splits the **SSTable** found in each level into several range-partitioned, non-overlapping, **SSTable**s. When a merge is triggered, a **SSTable** is selected from level N. All the **SSTable**s with overlapping key ranges from level N+1 get selected, too. The selected **SSTable**s get merge-sorted to form one or more **SSTable**s in level N+1. This strategy has the advantage that two merge operations with distinct key ranges can happen concurrently. Furthermore, each individual merge operation requires fewer resources because fewer records are loaded and merged-sorted.

Throughout this thesis, the partitioned levelled policy will be implicitly used. This preference is based on the fact that it is the default strategy used by several production-grade systems, including RocksDB [23] and LevelDB [27].

Foreground tasks

In contrast to the merge operations discussed, some operations are only performed to query the **LSM-Tree** and do not directly modify the structure. These operations are typically performed by the end user and are called foreground tasks; they can be directly mapped to the **KVS API** operations: `put`, `get`, `scan`, and `delete`.

A `put` operation involves inserting a key-value pair into the **MemTable**. The insertion is in place, meaning that if a pair with the same key is present, it will be replaced. If, after the insertion, the **MemTable** becomes full, a merge operation is triggered. To ensure durable writes, the key-value pair is also inserted in a **Write Ahead Log (WAL)**, which is a common component added to **KVS**, whose job it is to mirror the insertions to the **MemTable**, minimising the loss of information in case of system crashes. In this scenario, both **MemTable** and the **WAL** are written to in this operation.

On the other hand, a `get` operation involves just a key, and the process of retrieving the associated value is more complex. A `get` request starts by searching for an entry with the key in the **MemTable**. When the key is present, the value is returned immediately. Otherwise, the search continues through the disk component levels. The search starts at the lowest-numbered level. In each level, the goal is to find a candidate **SSTable** that might contain the key-value pair. As mentioned before, as a general rule, the keys are sorted throughout the level, meaning only one table can be the candidate. By finding the **SSTable** whose range of keys intercepts the searched key, it is then possible to query that table for the existence of the key. If the key is present, then the value associated is the latest version and is returned; otherwise, the search continues to the next level. If the key is not found on the last level, then the key was

never inserted into the **LSM-Tree**. In the case where a level's **SSTable**s have overlapping key ranges and are not sorted, as in RocksDB's first persistent level, then all tables with overlapping ranges with the searched key are candidates and need to be searched—increasing the read amplification.

The `scan` operation can be implemented as a `seek` to the closest provided key, returning an iterator object. The iterator can have two functions: a `key` operation returning the current iterator key and a `next` function returning the next key's iterator object. Because the **LSM-Tree** is sorted chronologically, a key can be in any sorted run of the structure, including the **MemTable** and all levels' **SSTable**s [75]. For this reason, the `seek` operation needs to perform a search through all sorted runs to identify the smallest key that is greater or equal to the lower-bound key. In each run, the smallest key satisfying this condition will be marked by a cursor. The `next` function can be implemented as a priority queue, where the priority is defined by $priority(key, version) = (-key, version)$, where the version will serve as the deciding factor in the event of a tie. Essentially, the `next` operation finds the smallest key from the list of cursors and advances that cursor to the next key of the sorted run.

The `delete` operation has a similar path as the `put` function; however, rather than involving a key-value pair, it only involves a key. The key is attached to an antimatter value, also referred to as a tombstone. This pairing will be treated as any other key-value pair would, with some exceptions. One exception is when a merge operation occurs in the highest numbered level, and the tombstone entry is present in the input **SSTable**. In this scenario, if the key is present in the output level, then both entries are deleted, contrasting the usual case where the latest version is kept. Another exception is regarding the read and scan paths. Whenever a tombstone is found in a `get` operation, it returns immediately, concluding the search with no value found. In the case of the `scan` function, it is similar; however, if the `next` key is a tombstone, it is skipped.

Background tasks

Merge operations are considered background tasks. However, some **KVS** implementations—as is the case in the RocksDB—have a lightweight version of the merge operation when the **MemTable** contents are transferred to the first disk level. This operation is called a *flush*. The *flush* occurs whenever the **MemTable** is full and can no longer accept incoming writes from users. To minimise the downtime, the *flush* directly moves the contents of the **MemTable** to an **SSTable** in the first persistent level without requiring the merge-sort step of the merge operations. This is the reason the first persistent level is generally allowed to be unsorted and have overlapping key ranges, so this transfer can be as quick as possible and not stall incoming user writes. In reality, the number of **MemTable** is configurable, as it

is not recommended to completely stop the flow of writes when the **MemTable** is full. For this reason, there is usually more than one **MemTable** to buffer incoming writes when a *flush* is occurring.

On the other hand, the merge operation between disk components is also commonly referred to as a compaction job. The rate at which compaction jobs are executed can affect the latency of writes. If the rate is not fast enough, causing the first persistent level to be full, a *flush* cannot occur, causing a cascading effect inevitably causing delays in user writes. To try to maximise the compaction rate, some **KVS** implementations, such as RocksDB [23], PebblesDB [63] and Cassandra [26] support concurrent compactions. Compactions can run simultaneously if they operate on different sets of files. For instance, if one compaction is merging a specific range of keys within a level, another compaction on the same level can execute concurrently as long as its key range does not overlap with that of the first compaction. Additionally, while key ranges in non-adjacent levels may theoretically overlap, the compactions running on those levels work with different files, so they do not interfere with each other.

2.1.3 RocksDB Production-grade LSM KVS

The RocksDB **KVS** is a robust and customisable **KVS** built by Meta in 2012, aiming for performance on a wide range of use cases. Built as a fork of Google's LevelDB, RocksDB improves many of the shortcomings of the previous foundational **KVS**.

At the heart of Meta's motivation was the industry shift to modern storage hardware, such as **SATA** and **NVMe** devices. LevelDB's design could not capitalise on the reduced latencies and bigger bandwidths provided by these devices, and at the core of the problem was an inefficiency to fully utilise multi-core **CPU**s by only allowing single-threaded compactions. By introducing multi-threaded background tasks, RocksDB was able to improve performance and better utilise these new devices. Besides the performance aspects, RocksDB evolved to become a versatile store, allowing fine control of various configurations, ranging from compaction styles, different support for **MemTable** and **SSTable** structures, and control of different memory configurations [11]. RocksDB's unique characteristics contribute to its high performance across various use cases, which is why it is widely adopted in the industry. This is also the reason it was chosen as the backend **KVS** for the benchmarks conducted in this thesis. The insights gained from this research can be directly applied to a wide range of users.

The following subsection gives a brief explanation of optimisations and features built upon the aforementioned **LSM-Tree**, present in RocksDB.

Durability

Fundamentally, the **LSM-Tree** does not ensure the durability of the operations stored in the **MemTable**. If a **KVS** wants to minimise the loss of information, it needs to implement those features.

A **WAL** is a file that keeps a durable log of all the operations that are added to the **MemTable** so they can be recovered after a system crash. After the **MemTable** is flushed, the **WAL** file becomes obsolete and can be archived and eventually destroyed.

Writing only to the **WAL** does not guarantee that data is durably stored [64]. When data is updated, the changes are initially held in memory by the kernel's page cache. The kernel determines when these in-memory pages are flushed to disk, so without an explicit synchronisation operation (like `fsync()` or `fdatasync()`), there is no assurance of when the data will be persisted.

To circumvent this problem, RocksDB has the option to issue a `fsync()` for each **WAL** write performed; as a consequence, the performance will be significantly affected.

Bloom filter

RocksDB uses Full filters, a type of Bloom filter, to optimise read operations in the **LSM-Tree**. A Full filter is a space-efficient probabilistic data structure that can answer membership queries, for example, if a key might exist in an **SSTable** in constant time [1].

The primary purpose of this data structure is to minimise disk reads when a key is not present in an **SSTable**. In the case of a `get` operation, without Bloom filters, each candidate **SSTable** has to be searched in logarithmic time, even if the key is not present. The Bloom filter checks if the key exists in constant time and saves the logarithmic cost of searching when it is not present.

Bloom filters ensure that there are no false negatives, meaning that if a key is present, it will not be incorrectly identified as absent. However, there is a possibility of false positives, which means that a key may be indicated as present even though it is not. In such cases, when attempting to search for the key, it will become clear that it does not exist. Therefore, the worst-case scenario is similar to not using a Bloom filter at all.

Bloom filters introduce a small but non-zero overhead in both storage and computational resources. In write-heavy workloads, where data is frequently written but rarely read, this overhead can negatively impact performance. The filters need to be constructed during the writing process of an **SSTable**, utilising **CPU** resources, even though their main advantage—speeding up read operations—is not realised in such scenarios. As a result, Bloom filters are not ideal for use cases that are either write-only or heavily dominated by writes.

Block cache

The block cache is an in-memory mechanism designed to store data blocks from **SSTable**s that are accessed exclusively during read operations (such as `get` and `scan`). By serving data blocks from the block cache, the system can significantly improve the performance of the read path. When a data block is requested, the block cache is checked first; if the block is not found, it is subsequently loaded from disk and then added to the cache.

The block cache is highly configurable, with tuning parameters that include the overall cache size, the eviction policy, and the number of shards used to reduce lock contention. In addition to data blocks, the cache can also be configured to store index and filter blocks—or even to ensure that index and filter blocks from the first persistent level remain in the cache. This can have a significant impact because these blocks must be loaded before the corresponding data blocks can be located, and the first persistent level typically contains the largest number of **SSTable**s that need to be searched.

However, caching these additional blocks increases the number of cache insertions and evictions. Moreover, in write-heavy workloads, the block cache may not be fully utilised because frequent writes tend to invalidate cached entries, leading to cache trashing.

2.1.4 Energy consumption monitoring

In any electrical system, two fundamental quantities—voltage and current—play a key role in energy consumption monitoring. Voltage is the difference in electrical potential between two points, acting like a pressure that pushes electric charges through a circuit. Current, on the other hand, measures the rate at which charge flows past a given point. Together, the product of these metrics determines the power used in the system.

Power is instantaneous; energy consumption is the sum of all power measurements performed over infinitely small intervals. In theory, energy consumption is obtained by integrating the instantaneous power over time [33]. Given that power $P(t)$ is the product of voltage $V(t)$ and current $I(t)$, the energy E consumed over the time interval from t_0 to t_1 is expressed as:

$$E = \int_{t_0}^{t_1} P(t)\,dt = \int_{t_0}^{t_1} V(t) \cdot I(t)\,dt. \tag{2.1}$$

A power meter fundamentally needs to utilize voltage and current sensors that operate at a high sampling rate. This is the basic method used to determine the energy or power consumption of any electrical circuit.

In trivial circuits with multiple components, such as several machines connected to a power supply in parallel, a power meter for each machine can be installed to monitor its power consumption. This approach allows for more detailed and finer-grained power and energy usage measurements. These methods work well in straightforward setups; however, the situation becomes more complex in intricate and microscopic circuits, such as those found in hardware components. Integrating power meters into such sophisticated circuits is extremely challenging in these cases. Nonetheless, some hardware vendors have developed special units to measure the power consumption of their components, exposing the results through drivers, such as **Intel Running Average Power Limit (Intel-RAPL)** from Intel [16], **NVIDIA Management Library (NVML)** from NVIDIA [15] and AMD `amd_energy` [5, 2].

With the growing demand for cloud services [66], managing power budgets in data centres has become increasingly challenging, making energy monitoring an essential part of data centres operations. Energy monitoring can provide insights to improve the energy efficiency of data centres. This is critical to adhering to strict energy constraints while ensuring acceptable service performance guarantees. With the end of the Dennard Scale scaling [24], a shift to improving the energy efficiency of data centres through a software-centric approach is expected [6]. To achieve these efficiency gains, it is essential to first monitor the software's energy usage and identify opportunities for improvement. Software-based tools that use coarser-grained energy metrics—such as those from physical power meters or energy drivers—can be employed to estimate or attribute the software's energy consumption.

Each method of energy acquisition and monitoring has its own unique trade-offs regarding accuracy, ease of deployment, scalability, and granularity. These methods can be broadly categorized into three groups: instrument-based methods, dedicated acquisition systems (such as **Intel-RAPL**, **NVML**, and `amd_energy`), and software-based monitoring tools [47]. The following subsection overviews each method, highlighting its advantages, disadvantages, and typical use cases.

Hardware Instrumentation Mechanisms

These methods of energy acquisition are based on a power meter device being installed in an electrical circuit or a subcircuit. These methods can be divided into external and internal devices. On the one hand, external devices, such as Watts Up Pro [70], are installed upstream of the power distribution unit, while internal devices, such as PowerMon [10], are installed inside the machine, generally requiring manual intervention to install the device.

External devices only provide machine-level granularity, while internal devices, depending on where the internal device is installed, can achieve component-level granularity, such as **CPU**, **DRAM** and **GPU** power

measurements. These instrumentation devices are an excellent solution for supporting multiple types of devices while maintaining a high level of accuracy. A good use case for using hardware instrumentations is a system administrator who needs to monitor the energy usage of various devices within a facility. However, for applications that require more detailed energy metrics—such as component-level measurements—the internal device method can become too complex to scale for production use.

Dedicated Acquisition System Mechanisms

Unlike generic hardware instrumentation methods, dedicated acquisition systems are custom-designed to be soldered directly onto a specific circuit, making them unsuitable for use with other hardware. However, more manufacturers are implementing these built-in power meters in hardware components, making them a more reliable form of energy acquisition.

Examples of dedicated acquisition systems are AMD's `amd_energy`, NVIDIA's **NVML** and Intel's **Intel-RAPL**. Following is a brief overview of each system, although an in-depth discussion is provided for **Intel-RAPL** due to its widespread adoption in energy measurement studies, and it is the acquisition method used for the source of energy measurements for the experiments performed in this thesis.

AMD energy. Since AMD's 17h **CPU** family model [2], `amd_energy` [5] has been available as a kernel driver allowing users access to **CPU** components energy. Although extensive research has not been conducted on the accuracy of these measurements, AMD supports several levels of granularity, reaching energy measurements up to the individual core level. To the best of our knowledge, no other vendor can achieve this level of granularity.

NVIDIA NVML. NVIDIA's **NVML** [15] is a C-based interface that is designed for monitoring and managing NVIDIA **GPU** devices. It provides real-time access to various performance metrics, such as utilisation, temperature, and power consumption.

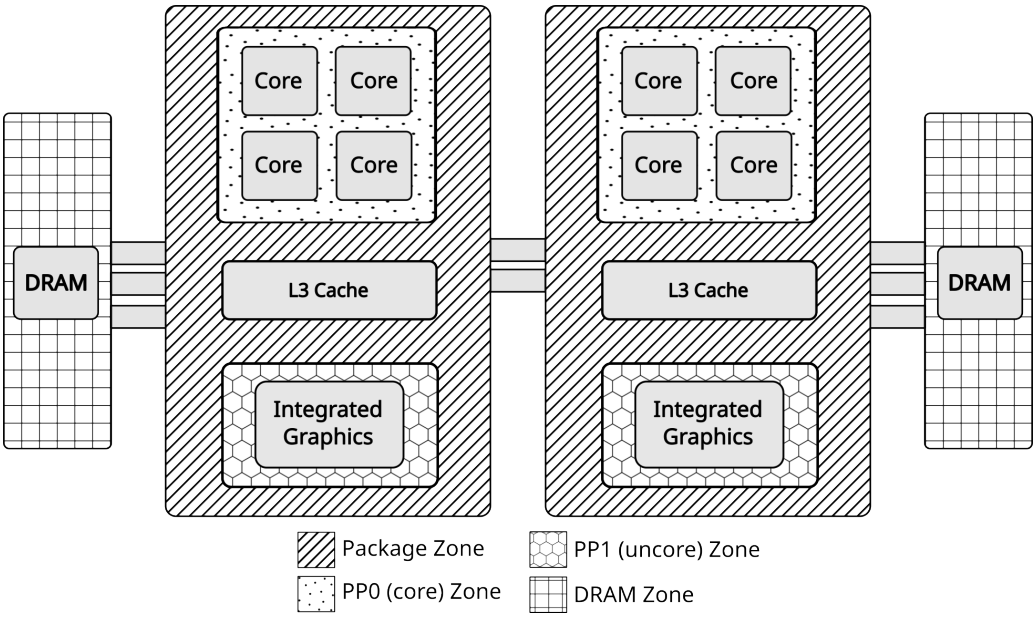


Figure 4: Power zones supported by Intel RAPL.

Intel RAPL. Lastly, in the case of Intel, starting from the Sandy Bridge model, all processors are built with the **Intel-RAPL**, which allows reading accumulated energy values of various zones of the processor. These readings are estimated through various hardware sensors and provide accurate measurements. **Intel-RAPL** is provided as a low-level interface through **Model Specific Registers (MSRs)**. **MSRs** are control registers that can be used to interface with the **CPU**. By using these registers, it is possible to toggle **CPU** features, but it can also be used to monitor the **CPU**. Some registers are read-only and provide specific information about the state of the **CPU**. Energy consumption measurements are retrieved through the MSR_ENERGY_STATUS register; the unit of the retrieved value is provided in the MSR_RAPL_POWER_UNIT register. These registers are available for different domains: Package, **DRAM**, **power plane 0 (PP0)**, **power plane 1 (PP1)**. These different domains map to a subset of the **CPU** architecture. Following, a brief overview of the **CPU** hardware architecture is given.

A motherboard can have one or more **CPU** sockets. A CPU socket provides the connection between the motherboard and a **CPU** package. A **CPU** package, or processor **die**, can contain one or more cores displaced in a matrix. A package can also contain other components, such as a memory controller, inter-core cache and I/O interfaces. In fact, some recent chips can also contain an integrated graphic engine. Finally, a package is connected to the **Random Acess Memory (RAM)** through one or more memory channels, connecting the memory controller of the package with the **RAM** [14].

The **Intel-RAPL** Package domain directly maps to the processor **die**. The **DRAM** domain corresponds to the directly-attached **DRAM**. The **PP0** is generally relative to the processor cores, while **PP1** is relative

to the uncore part of a processor [16]; if the chip includes integrated graphics, the energy readings from this domain will account for it. Although not officially disclosed by **Intel-RAPL** documentation, previous studies state that the **PP0** and **PP1** domains are a subset of the Package domain [41, 62].

There is extensive research validating **Intel-RAPL** energy values. Accuracy variability between processor models is expected. Models starting from Haswell with onboard voltage regulators are more accurate than previous models [41]. The **DRAM** measurements are actually based on accurate power measurements [21].

Discussion. These dedicated acquisition methods offer the potential for high-accuracy measurements and provide an **API**, simplifying the integration and manipulation of readings within user-developed software, such as monitoring tools. Furthermore, this approach eliminates the need for manual installation and effortlessly scales to accommodate large infrastructures.

Despite its advantages, this approach has some limitations. It is limited to measuring energy consumption at the hardware component level, which means it cannot monitor software energy usage. Additionally, it depends entirely on the capabilities of the underlying hardware, so the accuracy and resolution of measurements can differ among various vendors and models. Consequently, implementing this method on a larger scale requires specific hardware choices, limiting vendor selection or models and necessitating thorough planning.

Software-based monitoring mechanisms

The primary objective of software energy monitoring is to estimate energy consumption by modelling the power function based on system state information. This includes metrics such as cache misses, ALU operations, and resource utilisation, among others [47]. This approach allows for more detailed readings by breaking down a system's energy usage into several components. For instance, if the aim is to measure energy consumption at the thread level, metrics from thread execution can be used to assess each thread's contribution to overall energy consumption.

Software-based power models can be organised into two main strategies. On the one hand, trained power models use coarse-grained metrics for training and require only performance metrics during the inference phase. On the other hand, attribution models leverage readily available **API**s, such as **Intel-RAPL**. The latter utilises accurate data from the **API**s and incorporates system state metrics to estimate the contribution of a task to the overall energy consumption.

Following are examples of a trained power model and an attribution model. Each type of model is

further characterised by its granularity and its advantages and disadvantages.

Trained Power Models

This class of power models is trained using empirical data to create a formula based on performance metrics. During the training phase, the model is adjusted by comparing its predictions with high-accuracy coarse-grained measurements. Once the training phase is complete, the model enters the inference phase, which no longer requires access to energy metrics. Instead, it relies solely on performance metrics to estimate energy consumption.

The following model predicts energy consumption by resorting to busy hardware **CPU** cycles in SmartWatts [25], a trained power model.

$$P = \beta_0 + \beta_1 \cdot C_{\text{busy}} \tag{2.2}$$

where:

- P: Estimated power consumption.
- β_0: Intercept term (idle power when $C_{\text{busy}} = 0$).
- β_1: Regression coefficient.
- C_{busy}: busy **CPU** cycles.

Coefficients β_0 and β_1 are obtained in the training phase using a regression method that analyses matches between **CPU** busy cycles and power usage at the time the **CPU** metric was recorded.

This model estimates a coarse-grain power measurement and then distributes the power through fine-grained components such as processes or threads. The advantage of using these types of models is that the inference is fast, and sourcing performance metrics can be done at a faster rate than energy from dedicated acquisition systems, which in the case of **RAPL MSRs** they typically update every millisecond. Nevertheless, the primary issue with these models is their low accuracy when compared with hardware and dedicated acquisition mechanisms.

Attribution Models

This type of software-based model uses an accurate coarse-grained measurement in real-time, usually from dedicated acquisition systems, and then attributes a portion of the total energy to a finer-grained metric.

The following model is an example of an attribution model that collects energy from **CPU**, **DRAM** and **GPU** from dedicated acquisition systems, then based on each process resource's utilisation, calculates the process total power consumption [32].

$$P_{total} = \frac{PUE \sum_{process}(p_{dram}e_{dram} + p_{cpu}e_{cpu} + p_{gpu}e_{gpu})}{T} \tag{2.3}$$

where:

- P_{total}: process energy consumption.
- PUE: constant factor (used to account for external factors).
- $p_{resource}$: resource utilisation.
- $e_{resource}$: accumulated energy in the last T seconds.

Attribution models can achieve application and thread-level readings while achieving better accuracy than trained power models; however, when compared, this method cannot achieve the high sampling frequency of trained power models. Furthermore, this approach is limited by the hardware support of these acquisition systems when using these tools.

2.2 Related Work

This section describes and discusses the current state of the art in energy monitoring systems and studies focused on analysing the performance and energy costs of **KVS**.

2.2.1 Studies on LSM-based KVS energy efficiency

As the demand for increased energy efficiency in data centres grows, recent research has evaluated the energy efficiency of software across different data storage systems, ranging from file systems to database management systems. However, to the best of our knowledge, no comprehensive study has examined the energy efficiency of **LSM-Tree KVS**—even though these systems are widely deployed in cloud environments and data centers, where electricity consumption is a significant cost factor.

The following subsection is divided into two categories: studies investigating the energy efficiency of different storage systems and existing work focused specifically on **KVS** studies.

Studies on storage systems energy efficiency. Existing studies focusing on the energy efficiency of storage systems range from I/O subsystem components to DBMS query energy efficiency analysis.

With the availability of ultra-low latency disks—capable of less than 10 μs device latency—Whitaker et al. [71], *Harris et al.* [31] and *Sundar et al.* [69] investigate the energy efficiency of different aspects of the I/O subsystem.

Whitaker et al. argue that when using these disks, the processing time of I/O schedulers becomes a considerable portion of the I/O request time. They evaluate various I/O schedulers, including the option of not using one, focusing on their performance and energy efficiency in different benchmarks.

In the 2022 study *Harris et al.* also make a case that better energy efficiency can be achieved by not performing interrupt-based I/O completion. In low-latency disks, the overhead of switching and handling interrupts can lead to more inefficiencies than continuously querying the device's completion queues—polling completion mechanism. They evaluate an interrupt I/O completion mechanism with classic polling and hybrid polling.

Regarding *Sundar et al.* study, they examine the energy implications of various I/O interface designs, contrasting kernel-bypass solutions like `spdk` with kernel-based mechanisms such as `io_uring`, `libaio`, and POSIX interfaces.

On the other hand, other works focus on other energy implications of using ultra-low latency devices. In a 2020 study *Harris et al.* [30] investigated the energy and I/O performance impact of using different disk mediums, including an ultra-low latency disk, in different workloads generated in `fio`.

Katsaragakis et al. [39] focused primarily on the energy impact of using ultra-low latency disks on different indexing data structures. However, as an additional contribution, they evaluate the energy consumption and performance of LevelDB **KVS** in this storage medium while comparing the results with a **SATA** storage device. This evaluation consisted of varying the storage medium between the ultra-low latency disk and the **SATA** device and observing the performance and energy consumption when the number of foreground threads increased. This experiment was repeated under two workloads, a read-only and a write-only workload.

Although *Koutsoukos et al.* [42] did not focus on the energy impact of using ultra-low latency disks in storage systems, they provided a thorough evaluation of the performance impact of using different disk configurations in SQL systems and RocksDB. They evaluated the performance impact of using different internal configurations in PostgreSQL[60] and MySQL [57] while also varying the mode in which these disk devices can operate. Furthermore, they ran different **YCSB** workloads in RocksDB while increasing the number of foreground threads and varying the mode at which the ultra-low latency disk operated.

Regarding work on the energy efficiency of database systems, *Lella et al.* [44] and *Guo et al.* [29] have focused on analysing the energy efficiency of different query operations. *Lella et al.* investigated query operations in different SQL and NoSQL database management systems, while *Guo et al.* evaluation focused only on Oracle database [58]. Aside from evaluating the efficiency of different queries in the Oracle database, they also performed an analysis of the power impact of using different memory and cache configurations in this database management system.

Studies on KVS energy efficiency. Studies focusing solely on the energy efficiency of **KVS** are scarce, and to the best of our knowledge, there has not been a comprehensive study on **LSM-Tree KVS**.

Stella et al. [50] is a study comparing the energy efficiency of different processor architectures while only evaluating **KVS**. They compare four servers with different configurations, one ARM-based server and three x86 servers under different **YCSB** workloads in two distinct **KVS**, RocksDB and Kreon [59]. Although they test two different **KVS**, they do not make assumptions about the energy efficiency between the two.

2.2.2 Energy Monitoring Tools

In recent years, several tools have been released to measure energy consumption in different contexts. Solutions range from providing high-level consumption metrics, such as energy consumption at the machine level, to providing energy usage at the software level.

When restricting existing work to software-level capable tools, two main approaches are observed: while some regularly collect precise coarse-grained measurements and interpolate a finer-grained measurement, others train a model against coarse-grained precise measurements and then regularly predict energy usage based only on performance metrics.

The following subsection lists the main challenges existing systems tackle and their respective contributions to solving those problems. We first list the works focusing on tools providing high-level resource energy monitoring and then discuss solely tools capable of software-level monitoring.

High-level component energy monitoring. In some monitoring use cases, energy accounting does not require to be fine-grained. In the context of **Machine Learning (ML)**, the energy consumption of a single **Graphics Processing Unit (GPU)** can be retrieved with good accuracy, and since **ML** training requires large amounts of time, one training instance can monopolise the **GPU**. Several tools take advantage of this specific context to provide estimations of carbon emissions and energy costs of training a **ML** model, including Carbontracker [7] and an emissions calculator from *Lacoste et al.* [43].

Carbontracker gathers the energy from the **GPU**, **CPU** and **DRAM** to calculate a machine-level consumption of the **ML** training. It then gathers public data on carbon emissions in geographical locations and, based on that, predicts the carbon footprint of the **ML** training. The work from *Lacoste et al.* follows an even higher-level approach. They cross-reference the provided geographic location with public local grid energy and emissions data, and based on the type of GPU and training time, they interpolate an emission value for the training.

Although not focused on **ML** training, CodeCarbon [17] has a similar workflow to Carbontracker by gathering energy metrics from the **GPU**, **CPU** and **DRAM** to predict a carbon emission metric.

Regarding **High-performance computing (HPC)** environments, these systems usually work with specialised components and varying vendors, making it difficult to monitor the infrastructure's energy consumption as it requires interfacing with the different components and vendors. PAPI [37], PMT [13], LIKWID [46] and perf [51], are examples of libraries bridging this gap. PAPI is a coherent and operating-system-independent interface for accessing performance counters from diverse hardware and software components. PMT follows a similar objective as PAPI by also providing a unified library **API** for application use but focuses solely on energy metrics. Additionally, PMT offers energy metrics for **FPGAs** circuits, a feature that other tools do not provide, albeit it supports just one specific vendor. LIKWID serves a purpose similar to PAPI in that it provides most performance counters and energy metrics, and it is highly compatible across architectures and vendors. However, it is exposed to the user as a command-line tool suite. Perf is a low-level interface available through the kernel **API** and can also be used through a command-line tool; unlike PAPI, PMT and LIKWID, it only offers **CPU** energy metrics.

Aside from the existing works presented, other systems provide coarse-grained level metrics while their main goal is on providing finer-grained metrics [34, 53, 3, 35]. These works do not support as many architectures and vendors as the PAPI, PMT, LIKWID, and perf tools.

Software-level energy monitoring. With the push to increase energy efficiency by creating more efficient software, more and more systems have been proposed to measure the energy consumption of applications. Since no precise and granular energy metric is available to monitor software's energy, these tools require an additional step to calculate the energy contribution of software. There are two main methods for calculating a software's energy usage: either a tool trains a model based on precise and coarse-grained measurements, and the inference phase only requires performance counters or a portion of a coarse-grained measurement is attributed to the software.

Furthermore, orthogonal to the type of model used, these tools can be further categorised by the grain of measurement they provide, from virtual machine, container, application, process, to thread level.

While application-level measurements can be a direct match to process-level measurements in single-process applications, when an application is multi-process, most applications cannot accurately calculate the energy of the entire application; thus, the distinction.

Regarding trained power models at the software level, these systems are a good fit for measuring the energy of applications in high-frequency sampling rates and can also be used in systems where **RAPL** readings are not available. However, they do not provide the best accuracy since the inference is based on performance counters and not actual energy metrics. Kepler [3] and SmartWatts [25] are examples of trained power models. Kepler is a framework for measuring energy at the process, container and Kubernetes pod level. It collects performance metrics, such as **CPU** time, cache utilization, and number of instructions, to generate a power model. It provides several pre-trained power models based on the architecture and processor used, not requiring **RAPL** availability. However, when **RAPL** is available, it will use an attribution method to calculate the energy consumption [40]. Smartwatts differs from Kepler in that it performs real-time self-calibration of the power model. It uses hardware counters to infer an energy reading for **DRAM** and **CPU** energy values at the virtual machine, container and process levels. After a predefined period, a self-check is performed by checking the power model readings with **RAPL** backed readings. If they differ by a certain threshold, a calibration of the model is done with **RAPL** accurate readings.

On the other hand, energy attribution models can achieve better energy precision since they are based on real-time coarse-grained energy metrics. Systems using this type of model include: Scaphandre [34], Wattmeter [61], an energy estimator by *Henderson et al.* [32], Powerjoular [53], HaPPy [74] and EnergAt [35].

Scaphandre, PowerJoular and the system proposed by *Henderson et al.* follow a similar strategy to measure fine-grained energy. They use the time a process used a resource to assess a share of resource consumption. Scaphandre is a metrology agent dedicated to monitoring the energy usage of several services through an easy-to-use dashboard. It can provide **CPU** energy metrics at the virtual machine and process levels by using the **CPU** time of processes to assess a share of total energy consumption. Powerjoular is a command-line tool that can monitor the **CPU** energy at the process level. The system proposed by *Henderson et al.* differs from Scaphandre and PowerJoular by providing process-level measurements while accounting for **GPU**, **CPU** and **DRAM** energy measurements.

Wattmeter was proposed as a monitoring tool to allow energy-aware process scheduling in Linux. Unlike other energy attribution tools, it tries to directly measure the energy usage of a process by its strict presence on **CPU**. More concretely, by having high-frequency **RAPL** readings, it assigns those fast

sampling readings to the processes present on the **CPU**. All these small attributions can be summed up to determine a process's energy consumption over a time interval. To collect energy metrics from **RAPL**, Wattmeter created an eBPF program. The eBPF is a kernel technology that can run sandboxed programs while safely and efficiently extending the capabilities of the kernel. By collecting energy inside the kernel layer, they can reduce the latency to read **RAPL** counters when compared to other **RAPL** interfaces and increase the frequency of energy readings.

HaPPy is a hyperthread-aware energy attribution model that can attribute **CPU** energy at the application level. Actually, HaPPy attributes energy to a core, which requires an application to be pinned to a group of cores and associate their energy with the application. It does not use **CPU** time but actually retrieves busy **CPU** cycles to determine core usage. The core usage is a ratio between weighted core cycles compared to the sum of all the system's cycles. The tool predicts how much of a core's total cycles were done when more than one thread was running and multiplies that cycle share by a constant to achieve the weighted metric.

Lastly, EnergAt is a fine-grained energy attribution model that can attribute **CPU** and **DRAM** energy at the application, process and thread levels. They differ from existing work by providing thread-level measurements and being **NUMA**/multi-socket conscious. Over a sampling period, they find a thread residency rate for each socket—the fraction of time the thread was scheduled in each socket. The calculation of thread energy begins by determining the thread usage for each socket. This involves multiplying the total **CPU** busy time of the thread by the residence rate in the socket, which provides the thread's **CPU** time for that specific socket. Next, this **CPU** time is divided by the total **CPU** busy time of the cores present within the socket to calculate the thread's usage for that socket. Finally, this usage is used to compute the energy consumption of the thread in each socket. The total energy consumed by the thread across all sockets represents its overall energy usage.

2.2.3 Summary

Several studies have explored the energy efficiency of storage systems, particularly in the context of I/O subsystems, ultra-low latency disks and database management systems.

Although their primary focus is not evaluating the energy efficiency of **KVS**, some studies have benchmarked these stores with varying degrees of detail [39, 42, 50]. However, from these studies, no study attempted to analyse how different internal configurations can affect the performance or energy consumption of these stores; they mostly evaluated the **KVS** with different workloads while varying basic client concurrency, not providing enough detailed data about the functioning of these stores.

On the other hand, there were studies exploring and delving deeper into different internal configurations of SQL database systems [42, 29]. Nevertheless, SQL database systems differ from how **LSM-Tree KVS** operate internally. This type of **KVS** performs regular background operations that can have major energy implications and should be analysed in a fine-grain and comprehensive detail, which, to our best knowledge, has not been done.

The existing monitoring tools vary in their granularity and energy modelling methods. The majority of monitoring tools can only monitor resource's energy consumption at the component level [7, 43, 17, 37, 13, 46, 51]. These tools lack the granularity needed to identify energy patterns in the internal operations of **KVS**, such as periodic compaction and flush events; in fact, even application-level and process-level energy monitoring tools [3, 25, 34, 61, 32, 53, 74] are not capable of this deep level of analysis. To accurately capture the energy usage of these events, it is necessary to utilise thread-level measurements. The only tool capable of this type of measurement is EnergAt [35]. By calculating thread usage based on **CPU** time, EnergAt is oblivious to changes in core frequencies and only produces correct measurements whenever all cores are set to the same frequency; furthermore, EnergAt is incapable of identifying and providing energy measurements of application events.

Table 1 provides an overview of various monitoring tools, detailing the type of energy model employed, the granularity of measurements, the targeted resources, and whether the tool is **NUMA**-aware. It also indicates if the tool can accurately measure energy consumption while core frequencies change and whether it can correlate energy usage with specific event data.

Powmon is the only monitoring tool that provides thread-level granularity, is NUMA-aware, and can adapt to frequency changes while tracking event energy usage.

Table 1: Related work summary - Energy monitoring tools

Tool	Energy Model	Granularity	Resources Targeted	NUMA Aware	Frequency Aware	Event Capturing
Carbontracker [7]	DA	C,A	C,D,G			
Lacoste et al. [43]	CRP	A	N/A			
CodeCarbon [17]	DA	C	C,D,G			
PAPI [37]	DA	C	C,D,G			
PMT [13]	DA	C	C,D,G			
LIKWID [46]	DA	C	C,D,G			
perf [51]	DA	C	C,D,G			
Kepler [3]	TM,EA,DA	C,VM,P	C			
SmartWatts [25]	TM	VM,Cnt,P	C,D			
Scaphandre [34]	TM,EA,DA	C,VM,P	C			
Wattmeter [61]	EA	P	C,D			
Henderson et al. [32]	EA	P	C,D,G			
Powerjoular [53]	EA	C,P	C,G*			
HaPPy [74]	EA	A	C		X	
EnergAt [35]	EA	C,A,P,T	C,D	X		
Powmon	EA	C,A,P,T	C,D*,G*	X	X	X

Energy Model(s):

TM Trained Model

EA Energy Attribution

DA Direct Attribution

CRP Cross Reference Prediction

Granularity:

C Component

VM Virtual machine

Cnt Container

A Application

P Process

T Thread

Resources(s) supported:

C CPU

D DRAM

G GPU

R* Resource at Component-level

Remaining:

X Supported

N/A Non Applicable

Chapter 3

PowMon - A fine-grained energy attribution monitoring tool with event correlation

Existing systems lack the capability to deliver an in-depth energy analysis of **LSM-Tree**-based **KVS**. These systems operate on a multi-threaded architecture, requiring energy measurements at a highly granular, thread-level scale. Additionally, **KVS** continuously performs background events whose energy consumption can be significantly influenced by contextual factors—such as the compaction level and the number of involved files. Although prior work has achieved fine-grained thread-level energy measurements, it struggles when dealing with non-uniform core frequencies, and no unified framework has yet been established to correlate the energy cost of these events with their surrounding context.

To address these challenges, we introduce PowMon, a novel energy attribution tool that enables system designers and administrators to accurately measure the energy consumption of concurrent applications, including **LSM-Tree**-based **KVS**. PowMon supports energy readings with fine granularity, including system-level, application-level and thread-level. To correlate energy patterns with application-specific events triggered by different threads, PowMon provides a minimally invasive instrumentation library that allows recording specific events. PowMon's design is built under four core principles:

Fine-grained energy monitoring. PowMon enables fine-grained energy measurements of **CPU** resources and provides system, application and thread level energy metrics without requiring changes to the application code.

General applicability. PowMon is general purpose, being able to collect the energy consumption of different types of applications. In the context of this thesis, we focus on characterising the energy consumption patterns of RocksDB, an **LSM-Tree KVS**.

Flexibility. PowMon is flexible and extensible and can add support for collecting energy consumption from other resources and vendors.

Low intrusiveness. To capture and correlate energy consumption about specific events triggered within the application, PowMon allows for logging event-specific information with minimal modifications to the codebase.

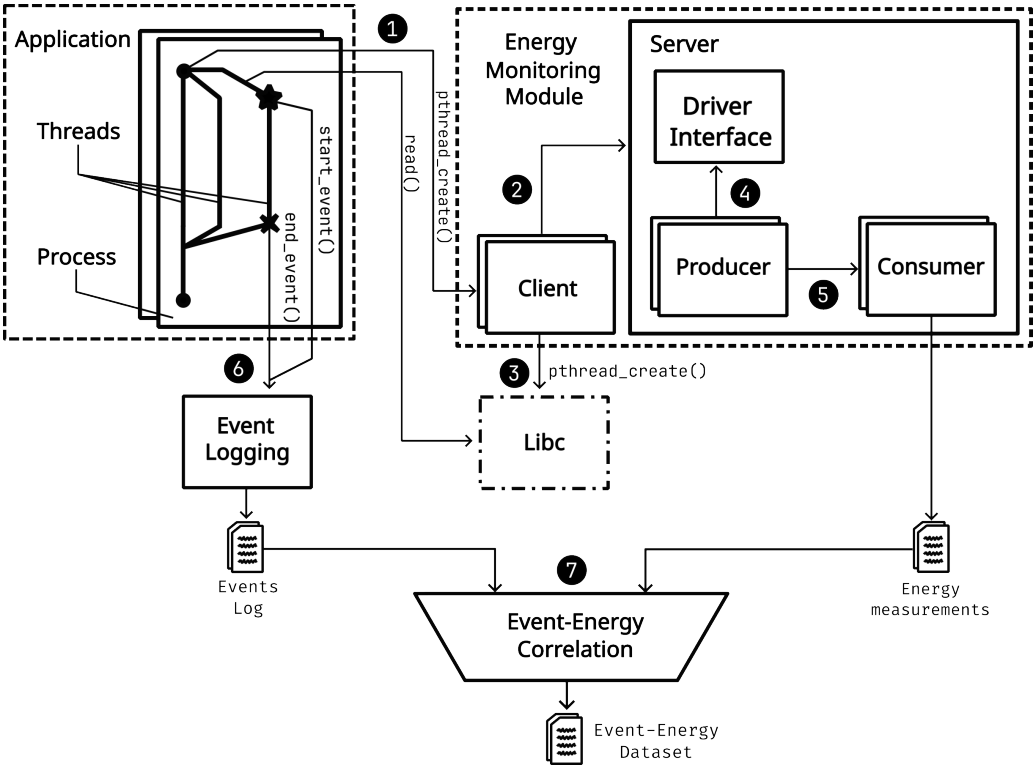


Figure 5: PowMon architecture

3.1 Design Overview

Figure 5 provides a high-level view of PowMon's architecture. The system has three major components: an energy monitoring module, an event logging library, and an event-energy correlation module. The energy monitoring module follows a Client-Server design where each process of the application acts as a client that sends requests to the server entity. This is achieved by leveraging LD_PRELOAD to intercept specific **Standard library C (libc)** and sending relevant information to the server. The server is responsible for receiving state updates and monitoring the application's energy usage and has three main components: the Drivers, the Producer and the Consumer. The Driver interface is used to extract the required metrics while abstracting the complexity of actually retrieving them. The Producer is responsible for computing the energy metrics of unique process threads (i.e., there is one Producer for each process of the monitored

application). Lastly, the Consumer is responsible for formatting and logging the energy metrics in an output file.

The event logging library is an interface that is instrumented in the monitored application to monitor events and associate additional information regarding the event information. This library outputs a comprehensive log file with the events captured.

Finally, the event-energy correlation module is responsible for merging the information from both log files to generate a unified dataset containing the energy consumption of each event alongside additional event information.

Operation workflow. The PowMon workflow comprises two separate concurrent operational branches, one for event logging and one for energy monitoring.

When monitoring energy, an application process can create new child processes and threads or end its execution; the functions `__libc_start_main`, `fork`, `pthread_create`, `pthread_join`, `exit`, `exec` that are performed to the **libc** library are intercepted, as in ❶, by the client component. The client component is responsible for relaying new state information to the server so it can know which processes and threads to monitor, as can be seen in step ❷. As shown in ❸, the client—serving as a wrapper for these functions—invokes the original **libc** library. In ❹, with the current state of processes and threads to monitor each Producer, will gather the necessary metrics to compute the energy consumption of its threads from the Drivers, and in ❺ will transfer the responsibility of logging to the Consumer. Parallel to the workflow described, in ❻, the monitored application interacts with the event logging library to record its events alongside important information. Lastly, the outputs generated from both components are merged in ❼ to achieve a detailed trace log of the events performed.

3.2 Energy Monitoring Module

The energy monitoring module tracks consumption at various levels—from thread and process for applications to system, package, and core for hardware. This section, divided into client and server components, lays the foundation for the monitoring loop: the client extracts and sends application state changes, while the server receives this data, enforces the loop, and logs the results.

3.2.1 Monitor Client

The monitor client intercepts specific events that alter the server state—namely, when a new application spawns, a subprocess is created, a thread is managed, or an application terminates. Using LD_PRELOAD to capture functions such as `__libc_start_main`, `fork`, `pthread_create`, `pthread_join`, `exit`, and `exec` allows us to achieve this without modifying the source code.

Instrumenting these functions is essential to maintaining accurate monitoring of processes and threads throughout an application's lifecycle. The `__libc_start_main` function is intercepted as it serves as a proxy for application startup, allowing the client to establish a connection with the server and notify it of a new process to monitor. Similarly, intercepting `fork` ensures that newly created processes are tracked by the server while maintaining their association with the parent, with a new client instance set up for the subprocess. Monitoring `pthread_create` and `pthread_join` enables the server to register new threads or stop tracking those that are no longer active. The `exit` call is captured to signal the server that the process should no longer be monitored, while the `exec` family of functions is intercepted primarily for cleanup and reinitialise the client in case of execution failures.

The messages sent to the server based on the event captured are:

- **Application/Subprocess start:** Send a request monitor with process and parent process *id*.
- **Thread creation:** Send a request monitor thread message with process and thread *id*.
- **Application finish:** Send a stop monitor request with process *id*.
- **Thread join:** Send a stop monitor thread request with process and thread *id*.

3.2.2 Monitor Server

The monitor server comprises three primary components: the Driver Interface, the Producer, and the Consumer. The Driver Interface consists of a hierarchy of drivers, each categorised by the type of metrics they collect—whether energy or utilization—and by the specific resource they monitor (e.g., **CPU**, **DRAM**, **GPU**). This abstraction hides the complexities of interacting with various interfaces and vendor-specific software, and it allows for seamless integration of new metric sources through the addition of appropriate drivers.

Each monitoring activity within the server is managed by a dedicated pair of Producer and Consumer entities. For example, system monitoring—which spans the server's entire lifetime—employs a Producer and Consumer to track overall system consumption as well as usage at the component (i.e., **CPU**, **GPU**,

and **DRAM**), and core levels. Similarly, for every monitored process, a separate Producer and Consumer are created to capture the energy consumption of the process and its threads.

To accurately capture application-level energy consumption, the monitor server must also account for the energy usage of its subprocesses. To achieve this, a tree-like structure is maintained to represent the application's dependencies, mapping each process *id* to its current subprocess *ids*. Producers linked to these subprocesses are synchronised, ensuring the metrics are retrieved in close temporal proximity for an accurate view of consumption at the application level.

At the core of the monitor server is a structured monitoring loop that methodically cycles through several stages—state updating, synchronization, metric collection, energy computation, and logging—with certain steps executed concurrently.

1) State Update

Whenever an **Application/Subprocess start** or **Application finish** message is received, a Producer and a Consumer are created or deleted, respectively, and the tree-like structure is updated, as ① in pseudo-code algorithm 1. When a **Thread creation** or **Thread join** message is received, the list of threads associated with the process Producer is updated, described algorithmically at ② in pseudo-code algorithm 1.

During each monitoring cycle, it is possible to query **OS API**s to obtain an up-to-date view of the application's dependencies and the active threads. This would be a feasible solution if multi-process applications were not supported. However, per the design goal, the monitoring tool should be **General Purpose**. The challenge in monitoring multi-process applications is that, without keeping state, every monitoring cycle requires identifying all the child processes of the monitored applications. Currently, without delving into extensive implementation details, finding a process's children requires traversing all the system's processes to find their respective parent processes. This process is extremely expensive for an operation that needs to happen on every monitor cycle.

2) Producer Synchronisation

In algorithm 3 at ⑤, before Producers collect metrics, they are synchronised. This synchronisation ensures that newly created Producers collect metrics in close time proximity to existing Producers. This is a crucial step to support multi-process energy monitoring. Since Producers collect information independently, synchronisation makes the collected metrics comparable. The parent process Producer can more reliably aggregate the energy consumption of its children, as their values were gathered in close temporal

proximity. It allows for the comparison of application metrics against system metrics.

3) Metric Collection

Since energy consumption is not a metric that can be measured directly at a fine grain, such as at the thread level, other metrics have to be used to attribute their energy cost. The metric selected to attribute a consumption quota to the thread level was the number of busy **CPU** cycles. The main reason for choosing this metric is its close correlation with the **CPU** energy collected from **Intel-RAPL**.

For application energy monitoring, the metrics required are the **CPU** busy cycles of all the threads involved and the finest-grain measurement of energy consumption. As is common in cloud environments, services are pinned to specific cores. If the provided cores belong to a single package, only the energy from that package can be extracted. This approach generates more precise results as a finer-grained energy metric can be used. If no information is available regarding which cores the application runs on, then the system energy—the energy sum of the available packages—is collected. The same situation arises when the application's cores are distributed across multiple packages. In this case, the total energy consumption is calculated solely based on the energy use of the specific packages that house those cores.

For system energy monitoring, the metrics required are the **CPU** busy cycles of all cores and the energy consumption of the available packages. The remaining metrics can be derived as a sum of these atomic metrics.

The pseudo-code in algorithm 3 in marker ⑥ shows a simplified view of collecting the metrics after the synchronisation step.

4) Energy Computation

Calculating the energy usage of monitored applications requires assigning an energy consumption value at the thread level. Energy attribution at the thread level consists of two steps, demonstrated in ⑦ in pseudo-code algorithm 3. The first step is calculating the **CPU** usage of the thread. This is achieved by calculating the ratio of the thread's **CPU** busy cycles to the total busy cycles of the packages where the application operates. To calculate the package's busy cycles, the sum of the cores' busy cycles is performed.

$$thread\ usage = \frac{thread\ busy\ cycles}{\sum package\ busy\ cycles} \tag{3.1}$$

The second step calculates the thread energy by multiplying the thread **CPU** usage with the total energy consumption of the packages where the application operates.

$$thread\ energy = thread\ usage * \sum package\ energy \tag{3.2}$$

The process energy calculation is shown in ⑧ at algorithm 3 and is the sum of the energy of its threads.

$$process\ energy = \sum thread\ energy \tag{3.3}$$

The energy of a given application is then defined as the sum of the energy consumed by all of its processes, including the main process and corresponding children. The Consumer performs this action, shown in marker ③ at pseudo-code algorithm 2.

$$application\ energy = \sum process\ energy \tag{3.4}$$

Monitoring system components is more direct, requiring a coarser grain of energy consumption. The package energy can directly be extracted from **Intel-RAPL**, shown in ⑨ from algorithm 3, and the system energy can be extracted by the sum of the packages **Intel-RAPL** energy, as in ⑪ from algorithm 3. Energy attribution at the core level, on the other hand, follows a similar reasoning to energy attribution at the thread level. However, at the core level, only one package contributes to the core's energy consumption, which is described by marker ⑩ in algorithm 3.

The first step is calculating the core **CPU** usage. This is done by dividing the number of busy cycles of the core by the total number of busy cycles of the package where the core is situated.

$$core\ usage = \frac{core\ busy\ cycles}{package\ busy\ cycles} \tag{3.5}$$

The next step is multiplying the core's usage by the energy consumption of the package in which it is located.

$$core\ energy = core\ usage * package\ energy \tag{3.6}$$

5) Logging

When the Producer has produced the energy metrics, it passes them to the Consumer entity. The **Consumer** entity organises this information and records it in a log format, as shown in ④ in Consumer algorithm 2.

In the case of multi-process applications, the process Consumer is also responsible for contacting the children's Consumers for their energy values to log the application's energy consumption.

The application Consumer output includes the date-time, application energy, process energy, and the threads' energy. The system Consumer records the date-time information, system energy, packages' energy and cores' energy.

Pseudo-code

Algorithm 1 Server Thread

```
1:  while true do
2:      Receive client request
3:      if Process Start or Process Finish then
4:          dep_tree ← update_dep(pid, ppid, CREATE/BREAK)
5:          Producers.update(pid, CREATE/REMOVE)                    } ①
6:          Consumers.update(pid, CREATE/REMOVE)
7:      else if Thread Start orThread Finish then
8:          Producers.update_tid(pid, tid, ADD/REMOVE)    ②
9:      end if
10: end while
```

Algorithm 2 Consumer Threads

```
1:  while true do
2:      energy ← wait_for_available_energy()
3:      if ApplicationConsumer then
4:          deps ← get_children_deps()
5:          children_energy ← {}
6:          for all dep ∈ deps do                                   } ③
7:              energy ← dep.get_energy()
8:              children_energy.add(energy)
9:          end for
10:         application_energy ← ∑ children_energy + energy
11:         output ← prepare_output(energy, application_energy)
12:     else                                                        } ④
13:         output ← prepare_output(energy)
14:     end if
15:     Log(output)
16: end while
```

Algorithm 3 Producer Threads

```
1:  Join sync group
2:  while true do
3:      Sync()    ⑤
4:      metrics ← Driver.collect_metrics()    ⑥
5:      if ApplicationProducer then
6:          thread_energy ← {}
7:          for all thread ∈ process.threads do
8:              packages ← get_application_packages()
9:              thread_usage ← metrics.get_cycles(thread) / metrics.get_cycles(packages)
10:             energy ← thread_usage × metrics.get_energy(packages)        ⑦
11:             thread_energy.add(energy)
12:         end for                                                          ⑧
13:         process_energy ← ∑ thread_energy
14:         consumer.push_metrics(thread_energy, process_energy, timestamp, ...)
15:     else
16:         core_energy ← {}
17:         package_energy ← {}
18:         for all package ∈ system.packages do
19:             package_energy.add(metrics.get_energy(package))    ⑨
20:             for all core ∈ package do
21:                 core_usage ← metrics.get_cycles(core) / metrics.get_cycles(package)
22:                 energy ← core_usage × metrics.get_energy(package)        ⑩
23:                 core_energy.add(energy)
24:             end for
25:         end for
26:         system_energy ← ∑ package_energy    ⑪
27:         consumer.push_metrics(system_energy, package_energy, core_energy, timestamp, ...)
28:     end if
29:     Sleep(custom period)
30: end while
```

3.3 Event Logging

Providing context to energy metrics at the thread level deepens our understanding of the underlying factors influencing energy consumption. While raw metrics—such as **CPU** usage, energy usage, and disk I/O—offer a broad overview, they do not reveal why consumption patterns change. By integrating detailed context, including the event type, duration, and characteristics of the objects the event interacts with, we can uncover the specific drivers of energy use. For instance, in compaction events, this context might encompass the number of files involved, their total size, and the compaction output level.

To facilitate this enhanced analysis, the module instruments the monitored application to identify and catalogue individual events. In a key-value store, for example, it can distinguish between event types like foreground tasks, compaction tasks, and flush tasks. Rather than merely marking the beginning and end of an event, the module attaches detailed contextual information that is later processed by the framework to correlate each event with overall energy consumption. This approach not only pinpoints the types of events occurring but also enables us to quantify the percentage of total energy they consume, offering deeper insights into energy distribution and supporting more effective optimisation strategies.

It is important to note that this module is optional. While it is essential for a detailed analysis of energy patterns, PowMon can still function without it, making it valuable in scenarios where instrumenting the source code of the monitored application is not feasible, for example, binaries or industrial systems.

3.4 Event-Energy Correlation

The event-energy correlation module is critical in the monitoring pipeline by consolidating data from two primary sources: energy logs and **KVS** events logs. The integration of these datasets enables a detailed analysis of how energy consumption correlates with specific properties of these events. Accurate merging of these logs is essential to uncover meaningful insights. Without this module, analysing these datasets independently would fail to easily identify patterns in the correlation between the event's context and energy consumption. By bridging this gap, the merger module ensures a unified dataset that forms the foundation for subsequent analysis and visualisation.

The event-energy correlation module is created to accommodate both the energy log format and the JSON format used by the event logging library. It is also designed to accurately align the discrete energy data with the date-time interval of the event.

3.4.1 Event Energy Assignment

The energy monitoring module organises energy information by thread *id*, allowing for conversion into a time series for each thread. Each data point includes metrics for the thread's energy consumption. Each observation in the time series represents the cumulative metric usage that occurs exclusively within the time interval defined by the timestamps of the preceding and current data points.

On the other hand, an event entry allows us to retrieve the *id* of the thread that executed the event. By using this information, it is possible to extract several data points that are candidates to have contributed to the event's energy consumption. However, it is important to choose which measurements will contribute to the event's energy usage.

To accurately correlate the events, we only consider the measurements that are unique to the energy consumption of the targeted event. The date-time associated with an energy measurement is related to the energy consumed between the last monitoring period. For this reason, the first measurement after the event's start time is discarded and does not contribute to the energy usage. For example, given a monitoring period of 100 milliseconds. If $t_{start} = 1.000s$ and $t_{measurement} = 1.050s$, then the measurement is discarded because the energy consumption contains energy data that is not from the event. At the other extreme, the last measurement is chosen as the latest measurement within the interval of event date-time. Following the same monitoring period as above, if $t_{end} = 1.400s$ and the latest measurement before the t_{end} is $t_{last} = 1.380s$, then we are sure that the measurement contains energy relative only to this event.

However, this approach may result in the reported energy being lower than the actual energy consumed by the event. To mitigate this issue, the energy sampling periods are kept short, increasing the number of data points captured and minimising the error caused by discarding the two data points at the extremities. At the same time, the sampling interval is carefully balanced, since reducing it excessively would impose a significant **CPU** load and overall system overhead.

3.5 Implementation

This section outlines the implementation of the developed system, focusing on the technical aspects of its key components. This includes the algorithms used, key optimisations applied to improve performance and accuracy, and the challenges encountered during implementation, along with the strategies employed to address them.

3.5.1 Energy Monitoring

The monitoring tool was implemented with 10K **Lines of Code (LoC)** in C++. The client and server communicate using ZeroMQ [73]. The decision to use ZeroMQ was based on its ability to simplify the connection workflow and provide built-in patterns. Unlike traditional socket programming, which often requires a greater effort to manage the connection and perform message framing and data serialisation, ZeroMQ abstracts these steps and offers a simplified **API**, enabling rapid development and reliable communication. We used a push-pull communication protocol, which allows for the establishment of a unidirectional communication channel between client and server (i.e., in our system, the client does not need to receive any data from the server).

Monitor Client

The client is supplied as a shared library that transparently intercepts **Portable Operating System Interface (POSIX)** system calls from the application through dynamic library loading—using LD_PRELOAD. The **libc** is responsible for wrapping a subset of system calls as a C library. When compiling a C program, the **libc** is, by default, dynamically linked to the binary. If, for example, a program intends to use the `fork()` system call, rather than directly invoking the system call, most programs call the **libc** `fork()` function. However, the LD_PRELOAD variable, as its name suggests, enables shared objects to be loaded prior to any other shared object. A symbol is searched for within a list of shared libraries and returned when found. The LD_PRELOAD libraries will be positioned at the beginning of that list. By creating a shared library with the same symbols as the **libc** shared object, the LD_PRELOAD library's objects take precedence, meaning the client's library functions will be invoked instead of the **libc**.

The following functions are intercepted: `__libc_start_main()`, `fork()`, `exit()`, `pthread_create()`, `pthread_join()`, and the `exec` function family.

When `__libc_start_main()` and `fork()` are intercepted, a worker thread is deployed. This thread connects to the server's socket and serves as the endpoint for communication with the server. Its role is to dispatch requests to the server without slowing down the client. In this manner, the client thread does not communicate directly with the server but does so asynchronously via the worker thread.

When these two functions are intercepted, the worker thread sends a request to monitor the process—whether it is the new main program or a new subprocess—using the following format: "pidmonitor:pid:ppid". Conversely, when the main function completes its execution or when `exit()` is intercepted, the worker thread sends a message to the server to cease monitoring the associated process, using the format: "pidsotp:pid".

In contrast, when the `exec` functions are intercepted, no message is sent. This is because if the `exec` function succeeds, the new application will subsequently invoke `__libc_start_main()`. The sole purpose of intercepting these functions is to perform the necessary cleanup before calling the function, and if the `exec` function fails, the worker thread is initialised, and the connection to the server is re-established.

The server must monitor the threads at a system level, as it extracts metrics directly from the **OS**. When the `pthread_create()` function is recorded, the client only has access to the pthread thread *id*. Consequently, the client wraps the thread's executable function and calls an **OS**-specific function to obtain the system's thread *id* before invoking the provided executable function with the appropriate argument. Thereafter, the client issues a message in the format "tidmonitor:pid:system_tid" to the server. On the client side, a mapping between the pthread *id* and the system thread *id* is maintained so that when the `pthread_join()` function is recorded, the client can send the message "tidstop:pid:system_tid" to the server.

Monitor server

The server is executed as a dedicated process in the **OS**. At start-up, it spawns three threads: one for handling client communication and two threads for managing the system's Producer and Consumer. For each new process monitored new threads are created for its Producer and Consumer. Communication between the Producer and Consumer is carried out through a completion queue. Here, the Consumer remains in a waiting state until it is interrupted by newly generated metrics. Furthermore, by leveraging the synchronisation inherent in Producers, an epoch is assigned to each synchronisation event. This approach allows the Producer to transmit the epoch information to the Consumer. Consequently, if the Consumer is a parent process, it can precisely determine which epoch measurements to request from its subprocess Consumers.

As previously mentioned in 3.2.2, the system introduces a Driver abstraction that allows system designers and administrators to implement the logic for supporting various resources—across different **CPU** architectures, **GPU**s, and other components—along with different energy sources. The driver hierarchy was implemented by creating a **DriverCPU** that aggregates all the relevant metrics and information about the **CPU**, including energy through an implemented **DriverCPUEnergy** and **CPU** usage through the implementation of **DriverCPUUsage**. These two latter drivers are then implemented based on system and hardware constraints. This architecture makes it easy to later add support for other **CPU** vendors and **OS**. Currently, however, the **DriverCPUEnergy** was implemented for machines with Intel **CPU**s. The

DriverCPUUsage driver was implemented for machines operating on Linux.

The interfaces used to collect information on the implemented drivers are now discussed. To access the number of cycles busy, the `perf_event` **API** was used. This kernel **API** provides an interface for monitoring hardware and software performance events, such as **CPU** busy cycles and cache misses, among others. The `perf_event` works by opening events for each metric we want to monitor. For each core in the system and each thread being monitored, a new event needs to be opened. The `perf_event_open` system call is used for opening events. An alternative would be using the `/proc/stat` kernel **API** that can provide a **CPU** busy time metric. Although easier to implement, this implementation presented one major limitation. It resorted to time units and not the number of cycles. All the cores must be set at the same fixed frequency to achieve correct energy measurements. A task scheduled at a higher frequency core consumes more energy than another task scheduled in a lower frequency core for the same amount of **CPU** time. If we were to attribute energy based on time, they would both be assigned the same amount of energy in this specific case.

To access energy values, the `/sys/class/powercap/intel-rapl` kernel **API** is used. However, a crucial step is preceded by the **DriverCPUEnergy**, ensuring that it correctly identifies all **RAPL** domains and subdomains available. The `intel-rapl` directory is structured as follows.

```
/sys/class/powercap
|-- intel-rapl
|   |-- intel-rapl:0
|   |   |-- ...
|   |   |-- energy_uj
|   |   |-- max_energy_range_uj
|   |   |-- name
|   |   |-- intel-rapl:0:0
|   |       |-- ...
|   |       |-- energy_uj
|   |       |-- max_energy_range_uj
|   |       |-- name
|   |-- intel-rapl:1
|       |-- ...
|       |-- energy_uj
|       |-- max_energy_range_uj
|       |-- name
|       |-- intel-rapl:1:0
|           |-- ...
|           |-- energy_uj
|           |-- max_energy_range_uj
|           |-- name
```

The `intel-rapl:0` and `intel-rapl:1` contain the packages' energy values. Inside each package

directory, directories with the format of `intel-rapl:number:number` can be found. These directories have a similar structure as the packages directory; however, they are relative to subdomains. The `name` file contains the name of the component being accessed in that directory. For example, the `intel-rapl:0` can contain the name "Package-1", and the `intel-rapl:0:0` directory might have the name "dram". This means that the `intel-rapl:0:0` provides access to the energy values of the **DRAM** subdomain in Package 1. To get the energy value of a component in micro-joules, the file `energy_uj` is inspected. Furthermore, the `max_energy_range_uj` file can be accessed to determine the max energy value reported by the **API**. This value is useful to limit the impact of the following limitation.

The **Intel-RAPL MSRs** registers are known to easily overflow. Under heavy load, the register overflow can occur after approximately 60 seconds [62]. This problem is due to the fact that the **Intel-RAPL** energy register is an unsigned 32-bit integer. To solve this, the driver keeps an overflow counter and returns an unsigned 64-bit integer. This accumulated value will not overflow—at a rate of 200W, it would take more than a millennium for it to do so. The following equation gives the corrected energy returned from the driver:

$$energy\ returned = (max\ value * overflow\ counter) + energy\ uj \tag{3.7}$$

3.5.2 Event Logging

The event logging library was implemented with approximately 2000 **LoC** of C++. The library was developed in C++ because there is a big percentage of **KVS** also developed in this language. In this particular case, RocksDB is also developed in C++.

To minimise intrusiveness to applications, the library is accessed through a single, global object, which prevents changing function calls internal to the application. As a direct result, the amount of lines introduced to RocksDB was low. Only 47 lines of code were introduced to RocksDB to capture compaction events and 26 lines to capture flush events.

As mentioned before in 3.3, a variety of information can be added to an event. This is achieved by implementing a context interface class. The user can add all the attributes it needs to describe the context and only needs to implement a `to_json()` function based on context attributes. To handle JSON data efficiently and easily, the **nlohmann** header-only library was used [48]. The following C++ code demonstrates an example of a header file that implements the context interface.

```cpp
#include <erm/generic_context.hpp>
#include <map>

class CompactionContext : public erm::IContext {
 public:
  CompactionContext(int from, int to, int job_id,
     std::map<int, int> input_quantity,
     std::vector<std::map<uint64_t, uint64_t>> input_info,
     bool trivial_move) :
   from_level(from),
   to_level(to),
   job_id_(job_id),
   input_quantity_(input_quantity),
   input_info_(input_info),
   trivial_move_(trivial_move)
  {}

  CompactionContext(int from, int to, int job_id,
     std::map<int, int> input_quantity,
     std::vector<std::map<uint64_t, uint64_t>> input_info) :
   CompactionContext(from, to, job_id, input_quantity, input_info, false)
  {}
  CompactionContext(int from, int to, int job_id) :
   CompactionContext(from, to, job_id, std::map<int, int>(), std::vector<
       std::map<uint64_t, uint64_t>>(), true)
  {}

  nlohmann::json to_json() const;
  int from_level;
  int to_level;
  int job_id_;
  std::map<int, int> input_quantity_;
  std::vector<std::map<uint64_t, uint64_t>> input_info_;
  bool trivial_move_;
};
```

Since this library can be included in the project without additional linking, it can be used in a **KVS** source code for implementing the `to_json()` function without worrying about dependencies.

3.5.3 Event-Energy Correlation

The merger module is built as an Ansible module. Ansible was used to set up a consistent environment for all test runs in the study. The module was created to integrate easily with the Ansible ecosystem. It has around 500 **LoC** and is developed in Python.

This module's final output is a JSON representing a unified dataset containing the energy usage of each recorded event and its context. The following JSON outlines the structure and information present:

```
{
"events": [
   {
    "thread_id_system": 322166,
    "energy": 71.39638200000002,
    "cpu_cycles": 4169933673,
    "event_type": "compaction",
    "start": "2024-12-17 13:00:09.797000",
    "end": "2024-12-17 13:00:22.151000",
    "job_id": 4,
    "file_num": {"1": 2,"2": 1},
    "inputs_size_info": [
          {"3644": 269714135,"3652": 112927377},
          {"2997": 124256824}
    ]
   },
  ...
],
"thread_info" : {
  "322166": {
    "2024-12-17 13:00:09.763000":{"energy":0.0,"cpu_cycles":0},
    "2024-12-17 13:00:09.865000":{"energy":0.693067,"cpu_cycles":12951010},
    "2024-12-17 13:00:09.966000":{"energy":0.457486,"cpu_cycles":21509567},
    ...
  },
     ...
}
}
```

The JSON contains a list of events in chronological order, with information about each event and its energy usage. Following the list of events, a time series with energy consumption information for each thread is displayed.

With this information, it is possible to analyse a thread's events and evaluate how its energy fluctuates with the start and end of different events at different levels. It is also possible to generate detailed statistics for each compaction output level. This unified dataset can always be upgraded in the future with a broader event context, capturing even more information.

Chapter 4

Energy Consumption in LSM KVS

In this section, we conduct a comprehensive study of the energy impact of **KVS**. Leveraging the developed monitoring tool, we study RocksDB, a popular **LSM-Tree KVS** developed by Meta, with the goal of understanding its energy impact under different configurations, workloads and test environments. Specifically, we seek to answer the following questions:

- Does workload and data distribution impact the energy efficiency of LSM KVS (Section 4.2)?
- How do different event types (namely, compactions, flushes) affect the energy consumption of LSM KVS (Section 4.2)?
- What is the energy efficiency impact of LSM KVS under different storage devices (Section 4.3)?
- How do different memory configurations—specifically block cache size, Memtable size, and maximum number of Memtables—affect the energy efficiency (Section 4.4)?
- How do variations in concurrency settings—including the number of foreground, compaction, and flush threads—affect the overall energy efficiency (Section 4.5)?
- What effect do compaction style and durability configurations have on energy efficiency (Section 4.6)?

4.1 Experimental Setup

The experimental setup forms the foundation of any experimental study. The setup is designed to ensure reliability, reproducibility, and alignment with the research objectives. This section outlines the critical components of the setup, including an overview of the experimental design and the controlled environment. Finally, it will thoroughly walk through the base RocksDB configuration list.

4.1.1 Experimental Environment

The experiments were conducted in a controlled environment to ensure consistency and minimize external factors.

Hardware and OS configurations. The experiment was run on a dual-socket server, each socket being a 24-core Intel Xeon Gold 6342 2.8GHz processor with hyper-threading disabled. The system has 192 GB of DDR4 3200MHz **RAM**, and the **OS** is mounted to a 480 GB Dell **SATA** SSD, running Ubuntu Server 20.04 LTS with kernel version 5.4.0. A 1.6 TB Dell **NVMe** SSD was used to store the run's database files on most runs. Cgroup v2 was set up to restrict maximum memory allocation and guarantee correct **CPU** task pinning.

Software Stack. Ansible was used to deploy a consistent experimental pipeline. Cgroup was handled using systemd as an intermediary—running with version 245. RocksDB benchmark tool, db_bench, was used to generate different runs with different workloads and configurations. RocksDB was used with version 8.7fb, as this was the latest production version available at the start of the study. Multiple metrics were gathered to observe the benchmark tool. These include detailed energy consumption measurements, system-wide I/O, **CPU** and memory statistics, as well as I/O metrics for each db_bench thread. The energy metrics were extracted from the PowMon system with a period of 100ms between measurements. The system-wide metrics were extracted with the dstat tool. Temperature metrics were extracted from the lm-sensors application. Finally, the fine-grained I/O metrics were extracted from the */proc/tid/task/tid/io* kernel interface.

Controlled Environment. To achieve reliable and reproducible results, some precautions were taken to provide a consistent environment for all runs. Before each run, a backup is loaded to ensure each experiment starts with the same database state. The benchmark tool was isolated to one **NUMA** core. This way, memory latency has less variability, and the rest of the software stack runs on the remaining **NUMA** core. Furthermore, db_bench's memory allowance was limited to 16GiB, with the exception of the memory area tests. Memory was limited to ensure that the majority of requests were submitted to persistent storage, minimizing caching effects. The page cache memory utilised by db_bench also counts towards the memory allowance. Furthermore, the page cache and swap are reset before each test run to ensure that every test starts with a cold cache. Finally, before each run, a cool-down period is taken to ensure that the machine's temperature drops to a consistent initial level across all tests.

4.1.2 Experimental Design

The experimental design was structured to evaluate the energy efficiency of the RocksDB **KVS** under varying **KVS** configurations, workloads and types of disk devices.

Setups. The evaluation considered several test categories to understand different factors affecting energy consumption:

- **Workload Patterns:** Examining how varying patterns of read and write requests (e.g., read-intensive vs. write-intensive, or hot key-space distributions) influence energy usage. (Section 4.2)
- **Device Characteristics:** Assessing the impact of different disk devices and their bottlenecks on the key-value store's energy profile. (Section 4.3)
- **Memory Configuration:** Investigating memory-related settings, such as cache size and memtable size. (Section 4.4)
- **Concurrency:** Analysing configurations related to the number of foreground and background tasks. (Section 4.5)
- **Compaction Schemes and Durability Configurations:** Evaluating internal mechanisms like WAL syncing and merge policies. (Section 4.6)

Workloads. A diverse set of workloads was employed to comprehensively evaluate RocksDB's energy performance. Specifically, we used YCSB workloads A through F, under the Workloads and Devices test categories. For the remaining categories, three workload types were tested: a read-only, a mixed read-write and a write-only workload. Furthermore, every batch was executed with two distinct distributions: a uniform distribution (spreading requests evenly across the key space) and a Zipfian distribution (where most requests target a small subset of keys). This design effectively doubled the number of delta runs, allowing us to analyse how each configuration change interacts with different workload characteristics and bottlenecks.

Methodology. Each test run lasted 12 minutes, using an identical database state generated through a fill run containing a key-space of 200 million keys. This fill run triggers several compaction events; although some compactions remain incomplete due to the run's end, they form a backlog processed in subsequent runs. This duration provides insight into how current writes affect the internal mechanisms of

the key-value store since it allows for the backlog to get processed. To ensure the reliability of our results, every batch of tests was repeated three times.

4.1.3 RocksDB Baseline Configuration

Defining a base RocksDB configuration is essential for establishing a consistent reference when evaluating energy efficiency. Our base configuration was refined using insights from previous studies and the RocksDB tuning guide, ensuring it closely resembles production performance.

RocksDB offers many tuning parameters, and advanced optimisations—such as enabling bloom filters to quickly determine if a key is absent in an **SSTable**—can add complexity. Therefore, we focus on the default behaviour and treat such optimisations as separate delta configurations.

The goal is to maintain a configuration that adheres to the basic **LSM-Tree** structure while extracting maximum performance. The base configuration is organised into categories, with each representing a set of parameters affecting a particular aspect of the **KVS**.

Table 2: RocksDB baseline configuration - Benchmarking and Statistics

Configuration	Value	Brief Description
key_size	16	Size of keys in bytes
value_size	1024	Size of values in bytes
duration	720	Duration for tests to run in seconds
num	200 million	Key-space range
stats_interval_seconds	1	Interval in seconds to collect statistics
stats_dump_period_sec	300	Period in seconds to dump stats
statistics	True	Enable/disable capturing statistics
stats_level	kExceptDetailedTimers	Level of stats to collect
histogram	True	Enable/disable histogram collection

Benchmarking and Statistics

The benchmarking and statistics are front-end options—they do not directly affect the internal organisation of the **LSM-Tree**. The benchmarking options indirectly influence how the **LSM-Tree** behaves. For

example, a big key and value size will result in fewer entries being stored in the block cache or memory; however, they do not shape the internal organization of the **LSM-Tree**. This change might cause a bottleneck in the disk, changing how the **KVS** behaves. The statistics configurations help to monitor the internal state of the **KVS**. They are crucial to better understanding the culprit of an energy efficiency change.

Similar to previous studies in **LSM-Tree KVS** [45, 63], the key and value sizes were selected to be 16B and 1KiB, respectively. A key space of 200 million keys was selected, meaning the database could not fit totally in memory. The statistics options were selected to avoid an excessive overhead and still provide important information.

Table 3: RocksDB baseline configuration - Memory Management

Configuration	Value	Brief Description
cache_size	1 GiB	Block cache size
block_size	4 KiB	Block cache block's size
cache_index_and_filter_blocks	False	Index/filter blocks in block cache
pin_l0_filter_and_index_blocks_in_cache	False	Pin index/filter blocks of L0 files in block cache
write_buffer_size	128 MiB	Size of a single memtable
max_write_buffer_size	2	Number of memtables, both immutable and active
min_write_buffer_number_to_merge	1	Only flush after merging minimum number of memtables

Memory Management

The memory and cache management category encompasses configurations related to memory components. It determines how the **MemTable** and the block cache are set up. The block_size is 4KiB, making it easier for blocks to be inserted into the block cache straight from the page cache, as on most systems, the page size is the same. The size of the **MemTable** was chosen based on similar configurations in other benchmarks [63, 8, 45]. The maximum number of **MemTable**s was set to two to permit incoming writes when a flush occurs.

Table 4: RocksDB baseline configuration - LSM Structure

Configuration	Value	Brief Description
compaction_style	Levelled	Compaction style
level0_file_num_compaction_trigger	4	Files in Level-0 to trigger compaction
level0_slowdown_writes_trigger	20	Files in Level-0 that will slow down writes
level0_stop_writes_trigger	36	Files in Level-0 that will stop writes
target_file_size_base	128 MiB	Size of SST files in Level 1
target_file_size_multiplier	1	Size of SST file in Level N = multiplier * STT size Level N-1
max_bytes_for_level_base	256 MiB	Max size of Level 1
max_bytes_for_level_multiplier	10	Max level N size = multiplier * level N-1 size
num_levels	7	Total number of levels
max_background_compactions	4	Max number of background compaction threads
max_background_flushes	1	Max number of background flush threads
compaction_pri	kMinOverLappingRatio	Compaction priority policy

LSM Structure

The compaction and level configurations define the core structure of the **LSM-Tree**, shaping key aspects such as the number and size of **LSM-Tree** levels, **SSTable** size, and the parallel background jobs (e.g., compaction, flush). They also determine the size of Level 0, which is typically equal to or larger than Level 1 to handle write bursts. Level 0's size is estimated by multiplying the **MemTable** size by both the maximum number of **SSTable** files in Level 0 and the number of **MemTable** flushed to it. In this base setup, Level 1 is about half the size of Level 0.

Table 5: RocksDB baseline configuration - Durability and File Management

Configuration	Value	Brief Description
disable_wal	False	If True disable WAL
wal_compression	none	WAL compression algorithm
sync	False	Synchronise all WAL writes to disk
open_files	-1	Number of files the DB can keep open. If -1 then all files will be kept open
format_version	6	Format version of SST files

Durability and File Management

The durability configurations are set to represent a typical scenario in production servers. In case of a system crash, it is important for data in the **MemTable** to be persisted. Of course, a compromise is achieved because we do not ensure the data is persisted on every write; instead, we let the **OS** flush to disk when it deems necessary.

Workload	Description
A	*Write-intensive*: 50% writes, 50% reads
B	*Read-intensive*: 5% read-modify-write, 95% reads
C	*Read-only*: 100% reads
D	*Read-latest*: 5% write, 95% reads
E	*Scan-intensive*: 5% updates, 95% scans
F	50% read-modify-write, 50% reads

Table 6: YCSB core workloads description.

4.2 Energy Efficiency Under Different Workloads

This section focuses on evaluating how different workloads, including operation proportion (read, write and scan) and data distributions (zipf99 and uniform), impact the energy efficiency of RocksDB.

Experimental setting. The batch of tests consisted of running the same base configuration and changing the workload applied to each test. All workloads mentioned in table 6 were used.

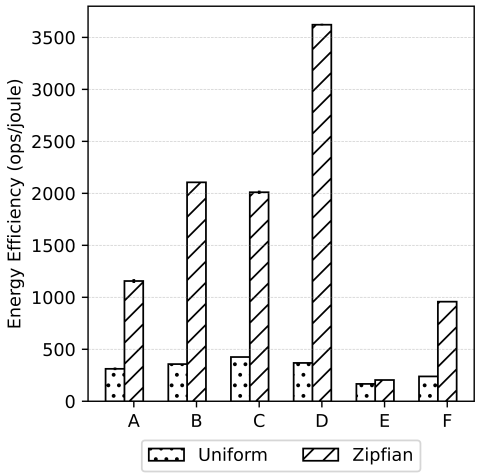


Figure 6: Average energy efficiency YCSB workloads with default configuration.

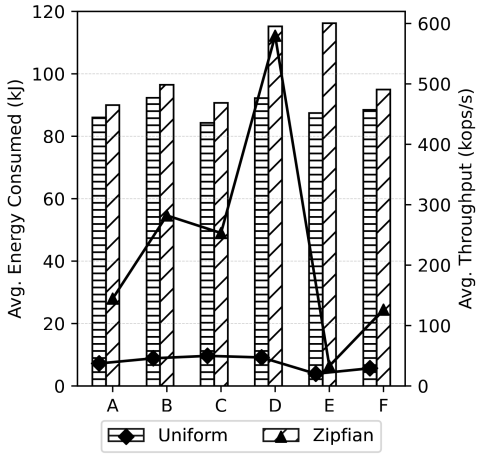


Figure 7: Average energy consumption and throughput of YCSB workloads with default configuration.

Figure 6 presents a histogram representing the average energy efficiency for the different **YCSB** workloads under two distinct data distributions: Uniform and Zipfian. Figure 7 provides a complementary analysis by presenting the average energy consumption and throughput for the same **YCSB** workloads. The histogram represents energy consumption, measured in kilojoules, while throughput, measured in thousand operations per second (kops/s), is depicted using the line plot. The energy from these results was extracted from the package (**NUMA** node) where RocksDB was running in isolation. The throughput was calculated based on the total operations reported by the RocksDB benchmark tool.

Results. The impact of data distribution on energy efficiency is evident across different workloads. Under a uniform data distribution, the read-only workload C demonstrates the highest energy efficiency, whereas the scan-intensive workload E is the least efficient. When comparing read-intensive workloads (B, C, and D) to write-intensive workloads (A and F), workload C—the most energy-efficient among the read-intensive workloads—exceeds the efficiency of the most energy-efficient write-intensive workload, A, by 31%.

Under a Zipfian data distribution, workload D emerges as the most energy-efficient, while the scan-intensive workload E remains the least efficient. Additionally, the most efficient read-intensive workload, D, surpasses the most efficient write-intensive workload, A, by a percentage difference of 103%. Across all workloads, the Zipfian distribution consistently exhibits greater energy efficiency than the uniform distri-

bution. The smallest efficiency difference is observed in the scan-intensive workload E, where the Zipfian distribution is 20% more efficient. For write-intensive workloads, the energy efficiency difference between the two distributions ranges from 115% in workload A to 120% in workload F. Similarly, for read-intensive workloads, the difference is more pronounced, varying from 130% in workload C to 163% in workload D.

Regarding energy consumption, the workload with the lowest energy usage varies between distributions. Under the uniform distribution, workload C consumes the least energy at 84.30 kJ, whereas under the Zipfian distribution, workload A is the most energy-efficient, consuming 90.67 kJ. Conversely, the highest energy consumption is observed in workload E for the Zipfian distribution, reaching 116.18 kJ, while in the Uniform distribution, workload B registers the highest consumption at 92.36 kJ.

When analysing throughput, workload E exhibits the lowest performance in both distributions, achieving 20.21 kops/s under the Uniform distribution and 33.04 kops/s under the Zipfian distribution. On the other hand, the most performant workload differs between distributions. In the uniform distribution, workload C reaches the highest throughput at 49.93 kops/s, while workload D achieves a significantly higher 579.72 kops/s in the Zipfian distribution.

Discussion. The Zipfian distribution is more efficient than a uniform data distribution. In the Zipfian model, 99% of requests are directed toward a small group of keys, creating a high likelihood of these keys being present in the **MemTable** or the first persisted level, L0.

For write-intensive workloads, write amplification decreases when compared to a uniform distribution. When writing data, there is a greater chance that an outdated key will be found in the **MemTable** and updated in place. Even if the key is not located in the **MemTable**, there is a high probability that outdated keys will be discarded during compactions in lower-numbered levels. The read path is generally shorter for read-intensive workloads, which reduces read amplification. The read request can be fulfilled immediately if the key is present in the **MemTable**. If the key is stored in the persistent levels, it will likely be found in the lower-numbered levels. These shorter read and write paths lead to less **CPU** energy being expended for each type of operation, resulting in better efficiency.

The read-intensive workloads are more energy efficient than the write-intensive workloads. This is attributed to the fact that the write path is less efficient than the read path. The write path is deferred to background tasks, which, when overloaded, cause delays and limit the amount of write operations performed. Essentially, the effects of overloading the write path are seen after a period of excessive write rate, which at a constant rate will become the rate at which the **KVS** can absorb writes. The background compactions rewrite the same key-value entry several times, which increases the write-amplification. In contrast, this process of organising the disk components and readjusting the **LSM-Tree** structure causes

a more efficient lookup process, improving the efficiency of the read path.

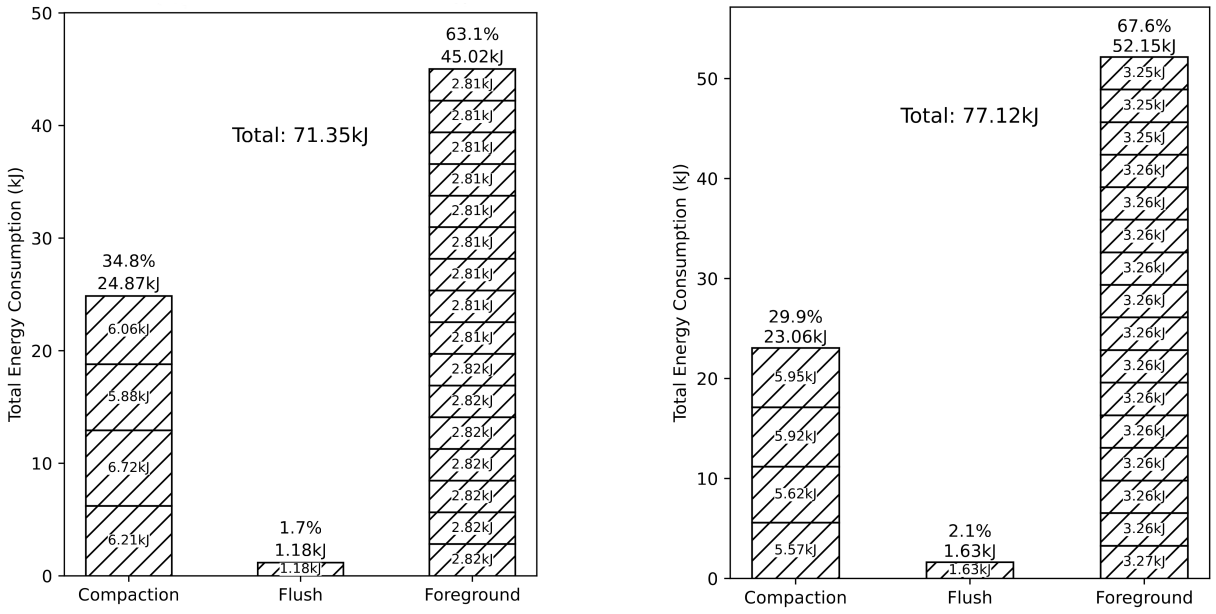
The scan-intensive workload E was the least energy-efficient compared to the other **YCSB** workloads. While the uniform and Zipfian distributions show high overall energy consumption, workload E consistently performs poorly. This underperformance is mainly due to the nature of scan operations, which involve retrieving multiple sequential keys and require maintaining a working set that spans all disk and memory components. As a result, scan operations lead to high disk utilisation because more data blocks need to be loaded into memory. For example, once a key is found in a point query lookup at a lower-numbered level, the remaining components are not searched. However, a scan operation must search all components to find the next potential key. Additionally, **CPU** utilisation increases during scan operations because each sequential key iteration requires multiple comparisons to identify the next key and update the cursors associated with each component. It is important to note that while a range query returns a sequence of key-value pairs, our results report this operation as a single query. Although a seek operation is less efficient than a lookup query, executing several successive lookup queries would likely be less efficient than performing a single range query—especially once all disk components are already loaded the next key can be determined by simply comparing the current set of cursors.

Takeaways

We draw two main takeaways from these experiments.

1) Zipfian data distributions are more efficient than uniform. Zipfian data distributions help reduce read and write amplification. This improvement stems from better temporal locality, resulting in more frequent reads from memory components and lower-numbered levels, as well as more in-place writing on the **MemTable** and the removal of outdated entries in the lower-numbered levels.

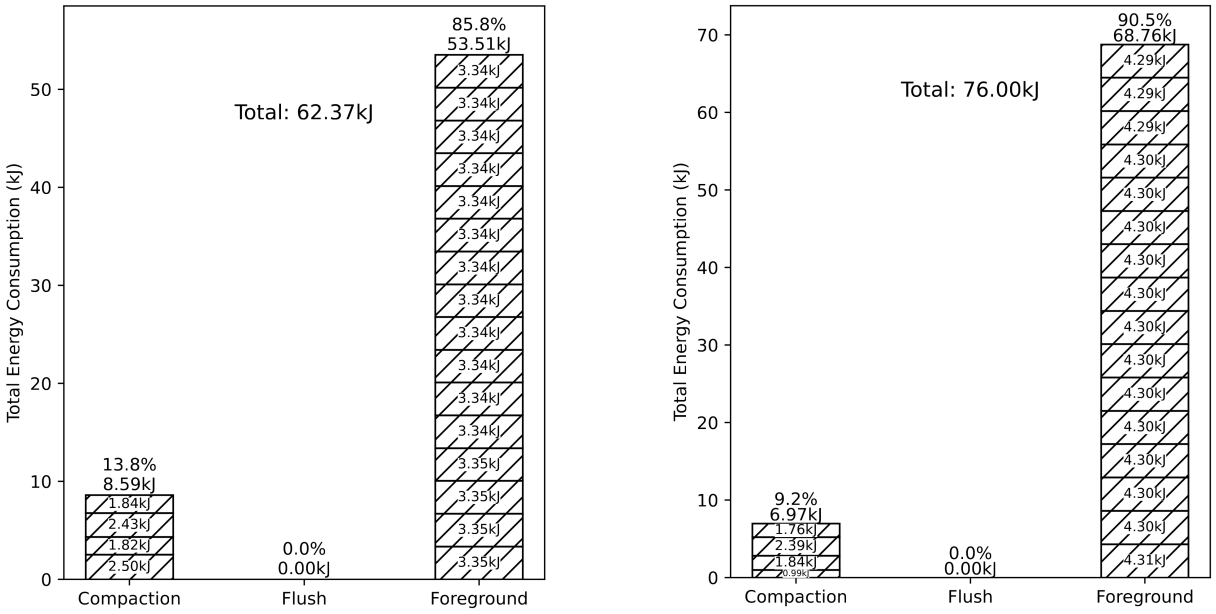
2) Read-intensive workloads are more energy-efficient. Although the read-path is longer than the write-path, read operations do not cause state changes that trigger expensive background operations (flush and compactions), directly impacting the client and involving increased processing load due to key-value entry sorting (compactions). Furthermore, workloads with a higher rate of writes experience more trashing of the block cache, resulting in worse efficiency.



(a) Uniform data distribution

(b) Zipfian data distribution

Figure 8: Breakdown of workload A total energy consumption across compaction, flush and foreground threads, under the uniform and Zipfian data distributions.



(a) Uniform data distribution

(b) Zipfian data distribution

Figure 9: Breakdown of workload C total energy consumption across compaction, flush and foreground threads, under the uniform and Zipfian data distributions.

4.2.1 Energy consumption analysis of different types of events

Figures 8 and 9 illustrate the distribution of total energy consumption across different threads and event types, specifically compaction, flush, and foreground. Each bar in the figures represents an event type

and is further divided into stacked segments corresponding to individual threads' contributions. The total energy consumption for each event type, measured in kilojoules, is displayed at the top of each bar, along with its percentage of the overall energy usage. Figure 8 details the total energy consumption for workload A, with subfigure 8a focusing on the uniform data distribution and subfigure 8b concentrating on the Zipfian distribution. In contrast, figure 9 presents the total energy usage for workload C, with subfigures 9a and 9b addressing the uniform and Zipfian distributions, respectively. The energy in these graphs is sourced from the application energy consumption retrieved from the PowMon monitoring system.

Results. When analysing the energy usage of workload A, the event type that consumes the most is the foreground, while the event type that consumes the least is the flush in both distributions. In the uniform data distribution, the foreground events are responsible for a total usage of 45.02 kJ, while in the Zipfian distribution, it consumes 52.15 kJ. The flush event type consumes 1.18 kJ in the uniform data distribution and 1.63 kJ in the Zipfian.

When analysing the energy usage under workload C, the total energy consumption from foreground events is greater than that from other events in both data distributions. Conversely, flush events do not contribute to the total application energy usage. In the uniform distribution, the foreground events are responsible for the total usage of 53.51 kJ, while in the Zipfian, they are responsible for 68.76 kJ. When focusing on energy consumed by compaction events, they consumed 8.59 kJ in the uniform distribution; in the Zipfian distribution, the total compaction consumption was 6.97 kJ.

A comparison of the two workloads reveals that the energy attributed to compaction events is lower on workload C across both distributions—registering a difference of 97% in the uniform distribution and 107% in the Zipfian. In both distributions, the energy consumption for flush events is zero on workload C, while the usage in workload A represents 2% of the total workload consumption. In contrast, foreground events show higher energy usage on workload C, with differences of 17% in the uniform distribution and 28% in the Zipfian.

Foreground and flush events consume more energy cumulatively when performed on a Zipfian data distribution, which is due to the higher number of operations performed compared to a uniform distribution. Conversely, the cumulative energy from compaction events is higher in the uniform distribution—not because of a larger total number of compactions, since the Zipfian distribution actually has more compaction events, but it could be a direct cause of the Zipfian distribution compactions having more duplicated entries, resulting in fewer data blocks to be processed, and consequently, less energy consumption.

Discussion. When comparing the two workloads, the write-intensive workload A, in total, consumes more energy on compaction events (in both uniform and Zipfian distributions), which is attributed to a higher number of compactions performed, due to the write-intensive nature of workload A. Conversely, workload C consumes more energy on foreground events because it performs more foreground operations, and without write operations, the energy for flush events remains zero.

Takeaways

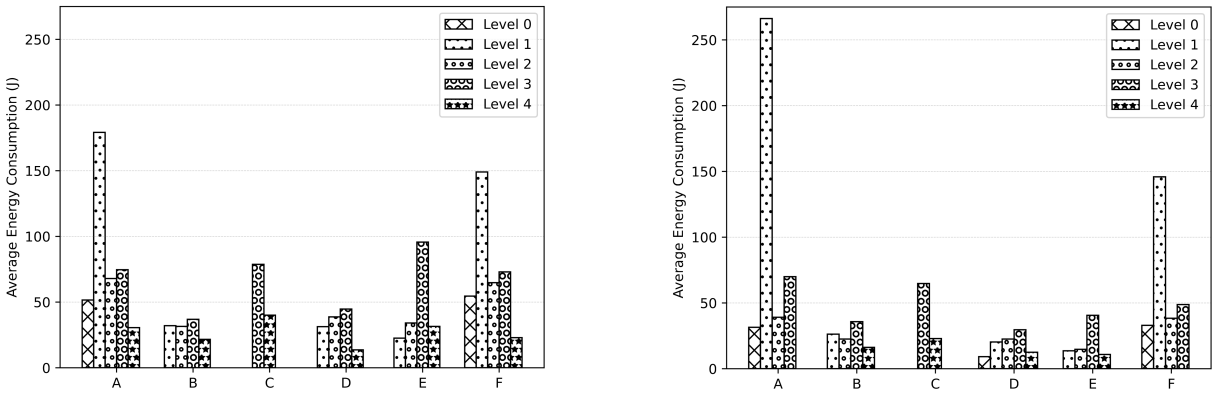
We draw two main takeaways from this subsection.

1) Energy consumption pattern of events changes with data distribution. Foreground and flush events incur a higher cumulative energy on a Zipfian data distribution due to a faster rate of operations. In contrast, although a Zipfian distribution triggers a larger number of compactions, uniform compactions consume more energy cumulatively. This is likely due to Zipfian having more duplicate entries and having to process fewer unique data blocks, resulting in lower energy consumption per compaction.

2) Workload type affects the energy distribution across the different event types. Write-intensive workloads consume more overall energy on compaction events because they have a larger number of compactions performed. Conversely, read-intensive workloads expend more energy on foreground events because they perform more foreground read operations, resulting in a small number of compaction jobs.

4.2.2 Energy consumption analysis of compaction events

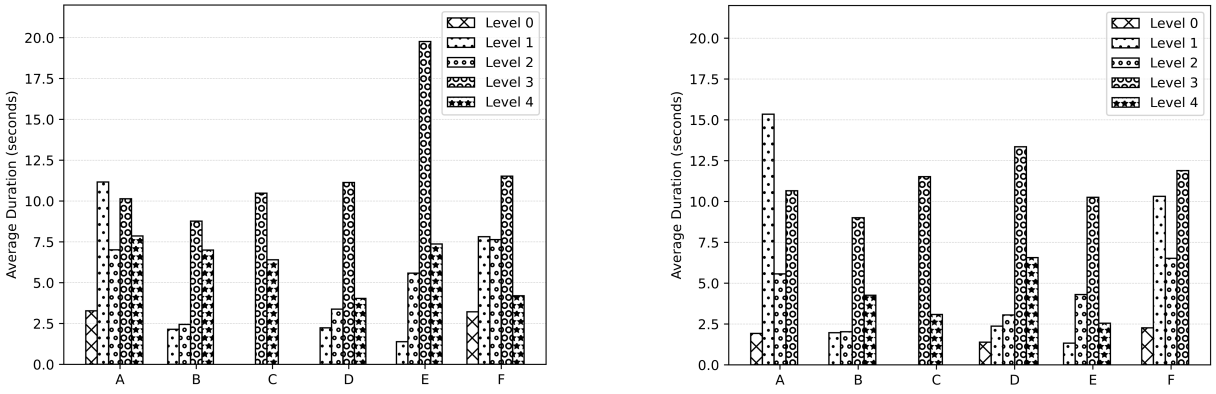
Figure 10 shows the average energy consumption across different compaction output levels for the different **YCSB** workloads, with subfigure 10a focusing on the uniform distribution and subfigure 10a in the Zipfian. Each bar represents a compaction's average energy consumption to a specific output level on a specific **YCSB** workload. As figure 10, figure 11 shows the average duration a compaction event takes across different output levels and **YCSB** workloads, with subfigure 11a and 11b focusing on the uniform and Zipfian respectively. For cumulative energy consumption metrics across the different compaction outputs, refer to figures 34 and 35 in the appendix. The energy and duration values from figures 10 and 11 are sourced from the PowMon monitoring system. The average energy values are presented in Joules, and the average duration is shown in seconds.



(a) Uniform data distribution

(b) Zipfian data distribution

Figure 10: Average compaction energy across the YCSB workloads, under the uniform and Zipfian data distributions after a warmup period of 300 seconds.



(a) Uniform data distribution

(b) Zipfian data distribution

Figure 11: Average compactions duration across the YCSB workloads, under the uniform and Zipfian data distributions after a warmup period of 300 seconds.

Results. After the warmup phase, figure 10 shows that write-intensive workloads (workloads A and F) consume, on average, more energy on compactions happening to output level one in both uniform and Zipfian distributions. On the other hand, read-intensive workloads—such as workloads B, C and D—have a higher average energy on compactions to output level three. In the uniform distribution, the highest average consumption in compaction events to output level one was achieved in the write-intensive workloads, with workload A consuming an average of 179 J while workload F consumed a lower relative average of 149 J. When concentrating on the average consumption for compactions to level one, workload F's average energy usage differs by 129% from the highest average achieved by read-intensive workloads—workload B. When focusing on the average energy of compaction events to level three, the read-intensive workload C achieved the highest average consumption with 79 J out of the write-intensive and read-intensive work-

loads. However, on average, the write-intensive workloads consume more energy in compaction events to output level three than the remaining read-intensive workloads. In the Zipfian distribution, the same pattern of level one compaction events consuming more, on average, than any other compaction output level arises, with workload A consuming an average of 266 J and workload F 146 J. Relative to other workload averages, workload F differs by a minimum of 139% from the remaining read-intensive workloads in events to level one. Compactions to level three are the most energy-consuming on average when focusing on read-intensive workloads. However, the write-intensive workload A consumes the highest average energy for compactions to this level, with 70 J, compared to the average consumption of 65 J of read-intensive workload C.

Regarding the average duration of compaction events, in read-intensive workloads, compaction events targeting higher output levels generally exhibited longer average durations—except for compactions to output level four—across both uniform and Zipfian distributions. In contrast, under write-intensive workloads, the average duration of compaction events at lower output levels increased and became comparable to those at higher output levels. For instance, in workload A, the average duration for compactions to output level one exceeded that for output level three. Overall, compactions to output level three tended to have the longest average durations across both distributions.

Under the uniform distribution, all workloads—except for write-intensive workload A—demonstrated that compaction events to output level three had the longest average duration. Furthermore, in read-intensive workloads, a substantial difference was observed between the average durations for compactions to output levels one and two compared to those to level three, with workload B exhibiting a 113% difference and workload D a 107% difference between the averages for levels two and three. This divide was less pronounced in write-intensive workloads, with the maximum difference between the averages across output levels being 41%.

Regarding the Zipfian distribution, all workloads—except for workload A—showed that compaction events to output level three took, on average, the longest time, with the read-intensive workload D reaching an average of 13 s. In write-intensive workloads, workload A recorded the longest average duration for compactions to output level one at 15 s, while workload F had an average of 10 s for compactions to the same level, corresponding to a 14% difference relative to compactions to level three within that workload. In the read-intensive category under the Zipfian distribution, the difference between the average durations for compactions to output levels one and two and those to level three remained significant, with workload B showing a minimum percentage difference of 127% between levels two and three.

Discussion. When comparing Zipfian and uniform data distributions, the uniform distribution consumes, on average, more energy for each compaction output level overall. This could be attributed to the Zipfian distribution's higher number of duplicate entries, which leads to processing fewer unique data blocks. Overall, the average duration of compaction events per output level remains similar or decreases in the Zipfian distribution. In cases where the duration increases in the Zipfian distribution, the increase can be explained by a higher average number of processed files. This situation may result in a higher overhead to maintain a larger number of files opened, but at the same time, the resulting unique data processed is lower, resulting in a longer duration with less power usage.

When comparing the read-intensive and write-intensive workloads, the analysis shows that write-intensive workloads (workloads A and F) tend to consume, on average, more energy at lower-numbered level compactions (levels zero and one). This is because write-intensive workloads have more incoming writes, triggering new compactions that include a large amount of new data in the page cache (a bigger probability of the compaction selected tables still being present in the cache). Although the compactions to these levels typically involve a large number of files, their duration is either smaller or similar to that at higher levels. This results in these lower-numbered level compactions occurring with higher power values and thus increasing the average energy cost. In contrast, read-intensive workloads (workloads B, C and D) spend more energy on average on higher-numbered level compactions. In these workloads, the duration increases markedly at higher output levels because most events load the selected tables from disk, not from faster mediums like the page cache. This is a direct result of high-level output compactions either being too sparsely scheduled or a significant percentage of these high-level output compactions being done as part of the backlog. When compactions are only sparsely scheduled and not in a cascading fashion, such as when lower-level compactions do not trigger successive higher-level ones, the selected tables are less likely to be present in the page cache because lower-numbered compaction output tables, once written to the page cache, will most likely be evicted. At the same time, backlog compactions must be loaded from disk because their selected tables were generated in a previous run. This characteristic leads to the observed disparity between the duration and energy consumption at lower versus higher output levels.

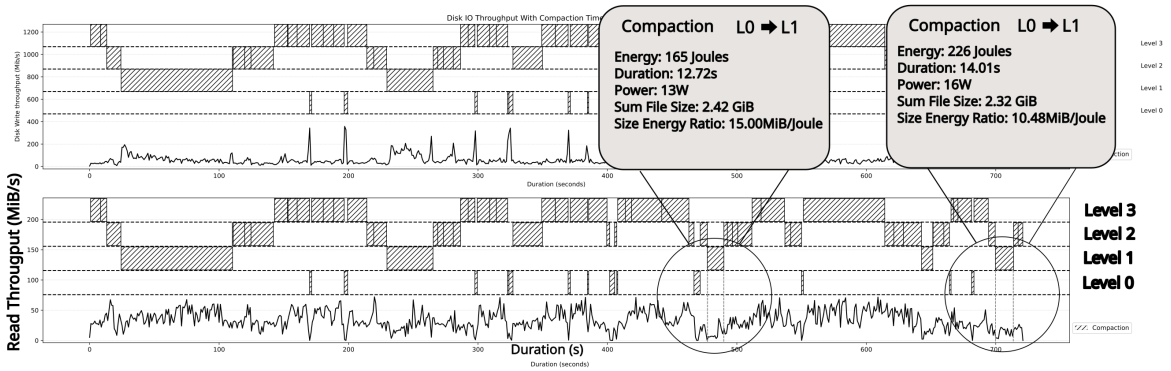


Figure 12: Read I/O and event context for two compactions to output level 1.

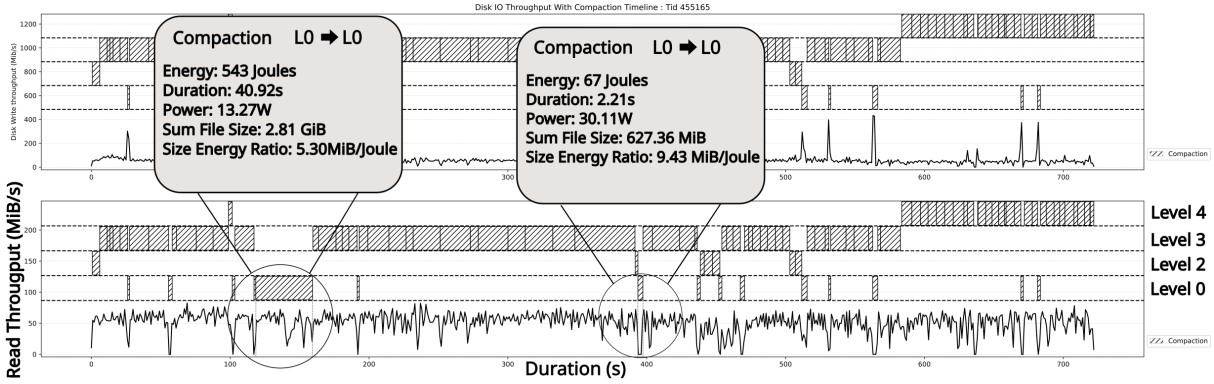


Figure 13: Read I/O and event context for two compactions to output level 0.

In figure 12 and 13, we observe how the compaction thread's read I/O can affect the energy efficiency of a compaction event. In this scenario, energy efficiency was calculated based on the ratio between compaction size and energy. It measures how much data can be compacted through just one Joule of energy.

Results. Figure 12 highlights two compactions performed by the same thread. These compactions are to the same output level 1. Furthermore, they both have a similar compaction size and number of files involved. However, one has more read I/O activity than the other. The first compaction I/O read activity can be described by a low throughput. Only 144MiB were read from the disk out of the total compaction size. On the other hand, the second compaction had more read I/O activity than the first compaction, with 210MiB read from the disk.

Figure 13 shows two compactions performed by the same thread to the same output level 0. Although they do not have similar sizes, the figure highlights two compactions with two distinct I/O patterns. The first compaction is longer and has an I/O pattern with many disk I/O reads. The second compaction,

on the other hand, is a faster compaction with practically no disk reads at all. The first compaction read 2.06GiB from the disk, while the second compaction barely read anything, with just 14MiB read from the disk.

Discussion. In all compaction events highlighted in figures 12 and 13, the I/O read from the disk did not add up to the total compaction size. Compaction threads cannot access the block cache, so they must use the page cache to load the data directly to compensate for the difference in data read from the disk and compaction size. Data read directly from the page cache allows the compaction thread to operate at higher power and finish the compaction faster. When the compaction thread has to read from the disk, the thread has to wait for the data. The threads reading more from the disk will likely be subjected to more operations in kernel space to handle the I/O, increasing the number of non-productive **CPU** cycles and decreasing the energy efficiency of these events.

Takeaways

We draw three main takeaways in this subsection.

1) Compactions on uniform distributions consume more energy than Zipfian. Compactions on uniform data distributions use more energy than on Zipfian, likely because they process more unique data blocks. In the Zipfian, more entries are duplicated and skipped during the merge sort, thus not loaded into memory and included in the output tables.

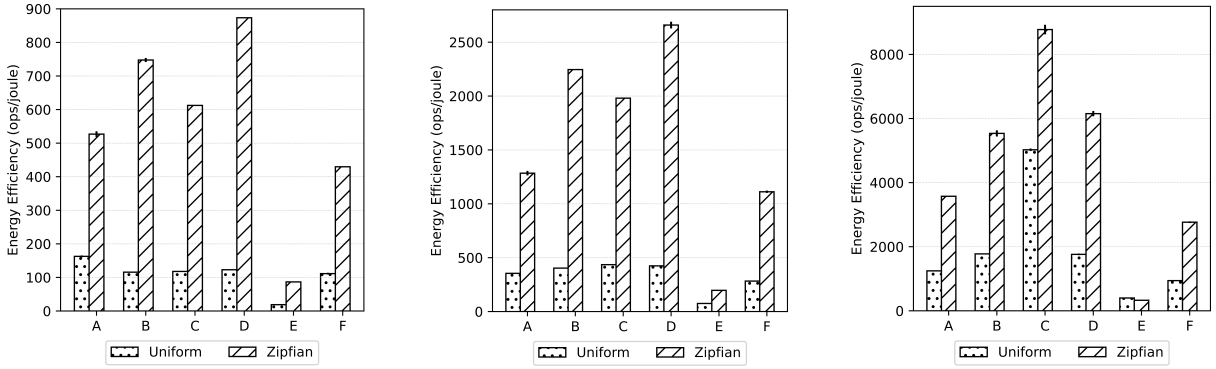
2) Compaction-level energy distribution is mainly dictated by workload disk access pattern. Write-intensive workloads consume more energy at lower-level compactions because frequent writes keep new data in the page cache. In contrast, read-intensive workloads face longer and more energy-consuming compactions at higher numbered levels because of sparser scheduling at these levels, increasing the chance of page cache evictions and more disk loading.

3) Page cache access increases energy efficiency of compactions. Direct page cache reads decrease compaction duration and increase power consumption, while disk reads incur delays and extra kernel overhead, decreasing energy efficiency.

4.3 Energy Efficiency Under Different Storage Devices

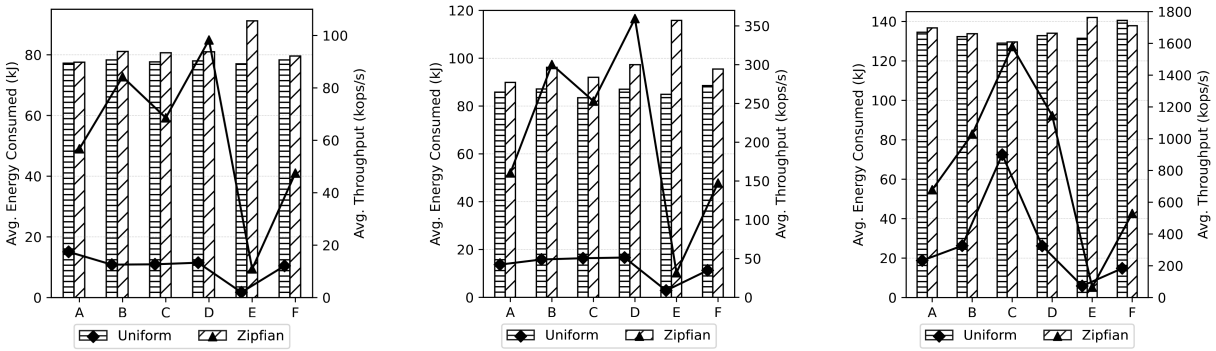
In **KVS** like RocksDB, the underlying storage device is critical in shaping overall performance and energy efficiency. Different types of storage devices vary in terms of throughput and latency. Depending on the workload, these properties can significantly impact the performance of the **KVS**.

Experimental setting. The batch of tests consisted of running the same base configuration and changing the workload applied to each test. All the **YCSB** workloads, in table 6, were used. Three batches of tests were made, each containing a different disk device. The devices used were: **SATA**, **NVMe** and a **RAM** disk. Since the **RAM** disk was limited in space, the original 200M key-space dataset could not be used. A new dataset of 80M entries was created and used.



(a) SATA device (b) NVMe device (c) RAM device

Figure 14: YSCB workloads average energy efficiency in different disk devices.



(a) SATA device (b) NVMe device (c) RAM device

Figure 15: YCSB workloads average energy consumption and throughput in different disk devices.

Figure 14 presents the average energy efficiency for the **YCSB** workloads when running on different disk devices. Energy efficiency, measured in operations per Joule, is evaluated across different **YCSB**

workloads and different disk devices while considering two distinct data distributions: Uniform and Zipfian. The histograms illustrate the differences in energy efficiency across the different configurations. The subfigures examine how energy efficiency changes when different **YCSB** workloads are applied in different devices: **SATA** device 14a, **NVMe** device 14b and **RAM** device 14c.

Figure 15 provides a complementary analysis by presenting the average energy consumption and throughput for the same **YCSB** workloads. The histograms represent energy consumption, measured in kilojoules, while throughput, measured in thousand operations per second (kops/s), is depicted using line plots. Similar to figure 14, subfigure 15a analyses the impact of changing workload on **SATA** device, subfigure 15b on **NVMe** device and subfigure 15c on **RAM** device.

Results. Focusing on the **SATA** results, in the Zipfian distribution, the energy efficiency is greater for read-intensive workloads such as B, C, and D than for write-intensive workloads A and F and scan workload E. In this distribution, among the read-intensive workloads, the most efficient was **YCSB** workload D, with a percentage difference of 35% to workload C. The least energy-efficient workload was the scan workload E, with the maximum difference between different workloads being 164%, in the Zipfian distribution. On the other hand, in the uniform distribution, the most efficient was the write-intensive workload A, while the least efficient was the scan workload E, with a difference of 158%. Regarding energy consumption, there was low variability across all workloads of the same distribution, except for workload E in the Zipfian distribution, which consumed significantly more than the others. In the uniform distribution, the workload that consumed the most was workload F, with 78.35 kJ, while workload E consumed the least energy, with 77.00 kJ, a difference of less than 2%. The throughput presented two distinct patterns. In the Zipfian distribution, the highest performant workload was workload D with 98.28 kops/s; in the uniform, workload A with 17.45 kops/s. However, in both distributions, the least performant workload was the scan-intensive workload E, with 10.95 kops/s in the Zipfian distribution and 2.01 kops/s in the uniform.

Regarding the **NVMe** results, read-intensive workloads—workloads B, C and D—are more energy-efficient when compared with the remaining workloads in both distributions. However, in the Zipfian distribution, workload D was the most efficient, while in the uniform distribution, it was workload C. The least efficient was the scan-intensive workload E in both distributions. Regarding the energy consumption in the Zipfian distribution, the workload with the highest consumption was workload E, with 115.84 kJ, while workload A consumed the least with 85.78 kJ. On the other hand, in the uniform distribution, workload F consumed the most with 88.55 kJ and workload C the least with 83.47 kJ. In both distributions, the highest throughput was observed in read-intensive workloads, with workload D achieving the best performance: 359.45 kops/s for the Zipfian distribution and 51.31 kops/s for the uniform distribution.

The least performant workload in both distributions was workload E, which recorded 31.72 kops/s for the Zipfian distribution and 8.81 kops/s for the uniform distribution.

Regarding the **RAM** device results, the least performant workload was the scan-intensive workload E, while the most performant workload was the read-only workload C in both distributions. Read-intensive workloads were more efficient in both data distributions. Regarding energy consumption, in the Zipfian distribution, the workload that consumed the most was workload E with 142.03 kJ; in the uniform distribution, it was workload F with 140.58 kJ. On the other hand, the workloads that consumed the fewest was, in both distributions, workload C, with 129.61 kJ in the Zipfian and 128.94 kJ in the uniform. In contrast to the energy consumption, workload C exhibited the highest throughput among all workloads in both distributions, reaching 1579.79 kops/s under the Zipfian distribution and 899.89 kops/s under the uniform distribution. Conversely, workload E demonstrated the lowest throughput, measuring 65.02 kops/s in the Zipfian distribution and 72.69 kops/s in the uniform distribution.

Discussion. Device properties, such as bandwidth and latency, impacted the energy efficiency of various workloads. The **YCSB** workloads were categorised based on the dominant operations they execute into three groups: read-intensive (Workloads B, C, and D), write-intensive (Workloads A and F), and scan-intensive (Workload E). The results indicate that the energy efficiency of these workload categories varies significantly with the underlying storage device. Specifically, write-intensive workloads demonstrated enhanced efficiency on **SATA** devices when focusing on uniform data distributions. In contrast, read-intensive workloads performed better on **NVMe** and **RAM** devices. This dichotomy is primarily attributed to the distinct I/O bottlenecks inherent in each device type. The read bandwidth represents a primary limitation in **SATA** devices. This bottleneck restricts the number of foreground read operations for read-dominated workloads, thereby limiting the performance gains that might be achieved by increasing the read load. Conversely, when the workload shifts towards a higher proportion of writes, the overall disk I/O increases. Although this shift reduces foreground reads, the system can absorb more write operations, improving performance despite the device's inherent limitations. Furthermore, the overall energy consumption remains relatively consistent across different workload types. This phenomenon can be explained by the continuous processing of background tasks—such as backlog compactions—in read-intensive workloads, contributing to a steady energy consumption. Similarly, while write-intensive workloads exhibit a marginal increase in energy consumption in the background threads, they exhibit a decrease in foreground activity, helping to maintain an overall balanced energy profile.

In the context of the Zipfian distribution, the findings indicate that on both **SATA** and **NVMe** devices, workload C is primarily constrained by the device latency. The latency experienced during read operations

on these devices is several orders of magnitude higher than that observed on **RAM** devices. For the other read-intensive workloads (B and D) on **SATA** and **NVMe** devices, a small proportion of write operations resulted in the **MemTable** being populated with hotspot keys. This led to an advantage, as a portion of the read operations were served from memory rather than incurring the higher latency associated with disk accesses, thereby enhancing performance. When the latency bottleneck was alleviated, the performance of workload C improved markedly, particularly because its block cache remained unaffected by incoming writes. Moreover, despite the performance gains on the **RAM** device, workload C consumed the least energy. This efficiency is attributed to its energy consumption being largely limited to foreground thread activity, as the compaction backlog was processed very rapidly.

Furthermore, the other workloads exhibited similar levels of energy consumption, albeit at the expense of performing less efficient operations, such as increased block cache housekeeping.

Regarding how the device type impacts efficiency, for the same workload, energy consumption was lower in the **SATA** device, followed by the **NVMe**, with the **RAM** disk device consuming the most energy. As the disk speed increases, the **CPU** spends less time idle, waiting for information. This leads to a busier utilisation from the **CPU**, increasing the power usage. Despite the increase in energy consumption, disk speed increases lead to better energy efficiency. The idle power usage from waiting for data does not advance the progress of a read or write task. With increasing speeds, it is possible to have more efficient busy cycles that improve a task's progress. For this reason, the throughput increases faster than the energy consumption with faster disk speeds.

Takeaways

We draw two main takeaways in this category of tests.

1) Device limitations impact workload energy efficiency in distinct ways. Device properties—especially bandwidth and latency—play a crucial role in determining energy efficiency across different workload types. Read, write, and scan-intensive workloads each leverage the strengths and mitigate the weaknesses of various storage devices. For example, write-intensive workloads perform better on **SATA** devices under uniform data distributions, whereas read-intensive workloads excel on **NVMe** and **RAM** devices due to lower read latencies. Under Zipfian distributions, I/O latency becomes a dominant factor for certain read workloads, highlighting the importance of caching and efficient background processing.

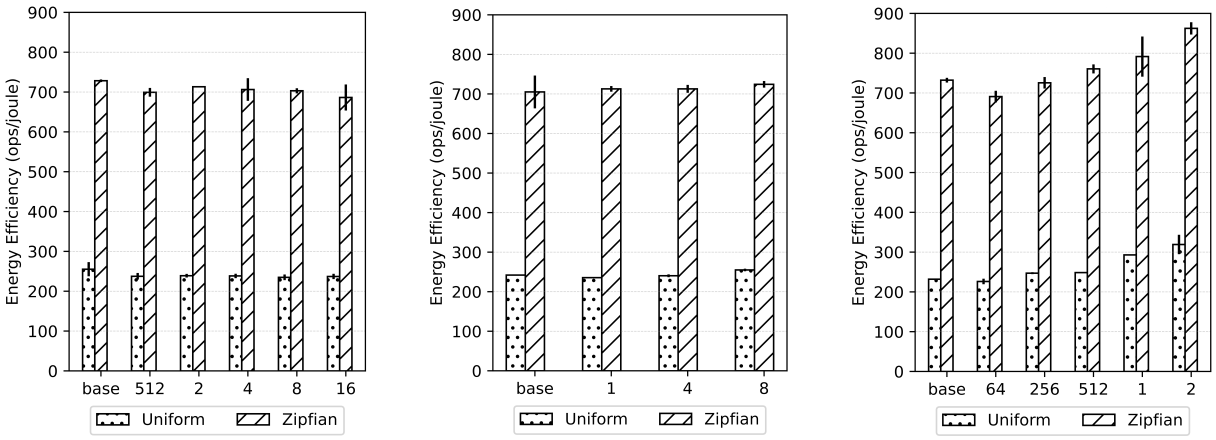
2) Faster disk speeds lead to better efficiency. While the **SATA** device consumes less energy due to more CPU idle time, faster disks (**NVMe** and **RAM**) reduce idle waiting and boost throughput. This

higher CPU utilisation increases power usage but results in more efficient task progress, yielding better overall energy efficiency despite the increased energy consumption.

4.4 Energy Efficiency Under Different Memory Configurations

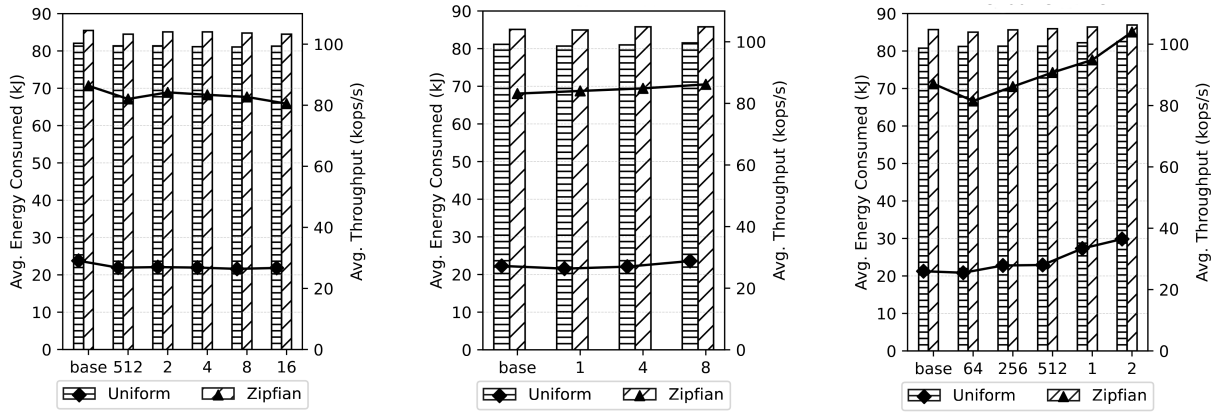
Memory settings can improve the performance of a **KVS** by improving cache performance and reducing I/O pressure. The block cache is used in reads to store uncompressed data blocks. On the other hand, the **MemTable** is a memory component critical for handling bursts of write operations and can serve as a cache for reads.

Experimental setting. The experimental environment remained the same as the RocksDB base configuration, apart from the memory allowance given to the db_bench tool. One of the goals of this area of tests was to see how increasing some components' sizes would affect energy efficiency. Three batches of tests were prepared, each run on a different workload, including write-only, mixed read-write and read-only workloads. The mixed read-write and read-only workloads were run with a different database filled with 200 million unique keys, ensuring a full key space. This was done to ensure we analysed how the block cache affects a read when the key is present in the **KVS** and not when there is a miss. The batch of tests consisted of running the base configurations while changing one memory configuration at a time. The block cache size was changed five times with the following sizes: 512 MiB, 2GiB, 8GiB and 16GiB. The maximum number of **MemTable** was changed thrice with the following numbers: 1, 4, 8. Finally, the size of the **MemTable** was changed with the following sizes: 64MiB, 256MiB, 512MiB, 1GiB and 2GiB. Each configuration was run in a uniform and Zipfian distribution. A test with the baseline configuration for each distribution and configuration group was run to compare the values with the baseline results.



(a) Block cache size (b) Max. Number Memtables (c) Memtable size

Figure 16: Average energy efficiency for different memory configurations in write-only workload.



(a) Block cache size (b) Max. Number Memtables (c) Memtable size

Figure 17: Average energy consumption and throughput for different memory configurations in write-only workload.

Figure 16 presents the average energy efficiency for different memory configurations under the write-only workload. Energy efficiency, measured in operations per joule, is evaluated across varying memory settings while considering two distinct data distributions: Uniform and Zipfian. The histrograms illustrate the differences in energy efficiency across configurations. Subfigure 16a examines the impact of block cache size, showing how different cache sizes influence the efficiency of operations. Subfigure 16b explores the effect of varying the maximum number of **MemTable**s, analysing whether an increase in the number of **MemTable**s contributes to improved energy efficiency. Subfigure 16c evaluates the role of write buffer size in energy efficiency.

Figure 17 provides a complementary analysis by presenting the average energy consumption and throughput for the same memory configurations. The histrograms represent energy consumption, mea-

sured in kilojoules, while throughput, measured in thousand operations per second (kops/s), is depicted using line plots. Similar to figure 16, subfigure 17a analyses the effect of block cache size, subfigure 17b examines the impact of the maximum number of **MemTable**s, and subfigure 17c considers variations in **MemTable** size. The energy from these results was extracted from the package where RocksDB was running in isolation. The throughput was calculated based on the total operations reported by the RocksDB benchmark tool.

Write-only workload results. For different block cache sizes, the energy efficiency remains constant for the uniform distribution; for the Zipfian, its efficiency may decline with bigger block cache sizes. For the different size values, the energy consumption of the **CPU** remains unaltered and with low variability. In the uniform distribution, the percentage difference between the lowest consumption configuration and highest consumption was 1%; in the Zipfian, the difference was 2%. The energy consumption in the baseline configuration was 81.56 kJ in the uniform distribution and 85.17 kJ in the Zipfian. The throughput, however, had more variability, with the maximum percentage difference between different settings in the uniform being 8% and 7% in the Zipfian. However, the throughput had a comparable variability between repetitions of the same cache size. In the uniform distribution, the throughput of the baseline configuration was 29.06 kops/s; in the Zipfian, it was 86.37 kops/s.

Varying the maximum number of **MemTable** did not seem to greatly impact the energy efficiency values. The energy consumption in the uniform distribution runs remained within a 1% difference between configurations, and the variability between repetitions of the same configuration value was lower than 1%; in the Zipfian distribution, the variability was also low, within a difference of 1% between different settings and less than 1% within the same configurations values. The energy consumption of the baseline configuration was 80.99 in the uniform distribution and 84.29 kJ in the Zipfian distribution. The throughput had a higher variability between repetitions of the same configuration value; however, it was only on the Zipfian distribution. In the Zipfian distribution, the configuration with fewer operations per second was the base configuration with a limit of two **MemTable**s, with 83.18 kops/s; the highest was when the limit was at eight with 86.18 kops/s marking a percentage difference of 4% between them. However, the maximum difference between repetitions with the same value was 10%; in the uniform, it was 2%. Between different configuration values, the maximum percentage difference in the uniform was 9%, while comparing the maximum number **MemTable**s to be one and eight; the lowest throughput was reported when the **MemTable** was limited to one, with a value of 26.35 kops/s, and the highest was when the limit was eight with 28.82 kops/s, marking a 9% change increase.

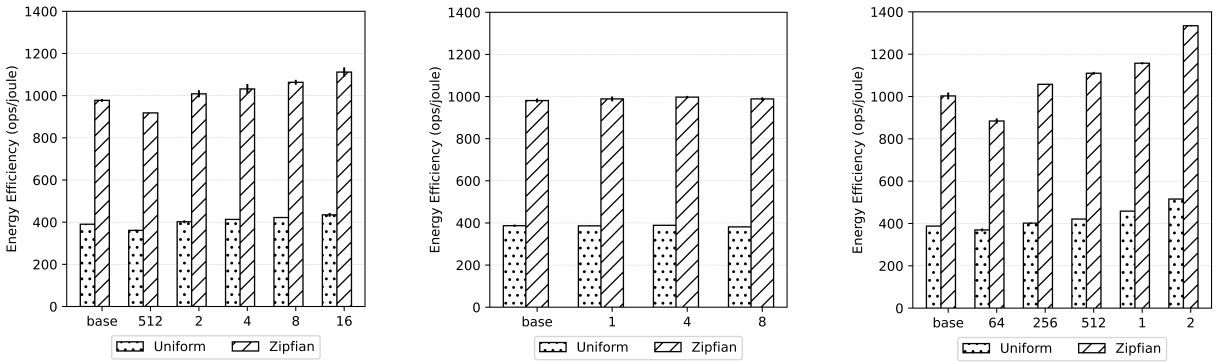
For different **MemTable** sizes, the impact on energy efficiency was significant. The efficiency in-

creases with bigger **MemTable** sizes. The energy efficiency increased with a percentage change of 38% in the uniform distribution and 18% in the zipfian. The energy consumption presented low variability with a maximum percentage difference of 2% between consumptions in both distributions. For the uniform distribution, the energy consumption of the baseline configuration was 80.94 kJ, while in the Zipfian, it was 85.50 kJ. The throughput in the uniform distribution increased from 25.49 kops/s when the size was 64MiB to 36.48 kops/s when the size was 2GiB, marking a 43% change increase; in the Zipfian, the throughput increased from 81.48 kops/s in the smallest size to 103.98 kops/s in the largest, marking a 28% increase.

Write-only workload discussion. Adjusting the block cache size did not have an impact on energy efficiency. The block cache is not used during write-only workloads as the block cache is used only for reads. The apparent trend in efficiency from different block cache sizes seems to result from a high variability observed throughout tests of this workload.

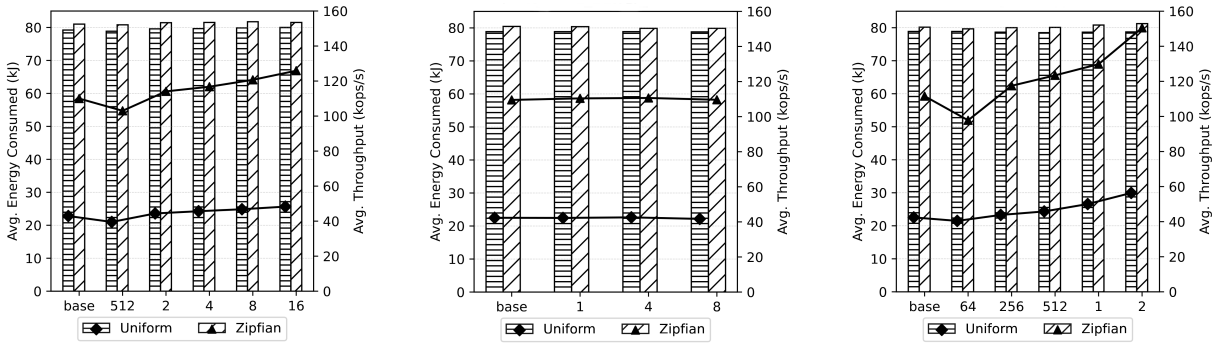
Increasing the maximum number of **MemTable**s could increase energy efficiency. However, in the Zipfian distribution, the variability between repetitions of the same configuration value is too big to make assumptions about the energy efficiency trend. On the other hand, on a uniform data distribution, the throughput maximum percentage difference between repetitions is just 2%, and the maximum difference between configurations is 9%. This could hint at better efficiency when increasing the maximum number of **MemTable**, since the energy remained stable but performance improved. If this is the case, it can be explained by a higher chance of more than one **MemTable** being selected for merge before a flush, resulting in the early deletion of duplicate entries and improving performance at a reduced energy cost.

Increasing the **MemTable** size had a positive impact on the energy efficiency in both Zipfian and uniform data distributions. With a bigger **MemTable** size, there is a higher probability that writes are performed in place in the write buffer, which can be confirmed by a smaller ratio between compactions and operations performed, improving performance. This performance improvement, however, did not come at the expense of higher energy expenditure, as the increased energy from performing more foreground operations was offset by a decreased energy expenditure from background threads because fewer compactions were being performed.



(a) Block cache size (b) Max. Number Memtables (c) Memtable size

Figure 18: Average energy efficiency for different memory configurations in mixed read-write workload.



(a) Block cache size (b) Max. Number Memtables (c) Memtable size

Figure 19: Average energy consumption and throughput for different memory configurations in mixed read-write workload.

Figure 18 presents the average energy efficiency for different memory configurations under the mixed read-write workload. In contrast, figure 19 provides a complementary analysis by presenting the average energy consumption and throughput for the same memory configurations. Both figures follow the same format and structure as figures 16 and 17.

Mixed read-write workload results. When increasing the block cache size, the energy efficiency trends upwards in both data distributions. In the Zipfian distribution, the efficiency increases by 21% between a cache size of 512MiB and 16GiB; in the uniform, the efficiency increases by 20%. In both distributions, the **CPU** energy consumption does not vary too much with different sizes. The variability between energy consumption values of the different cache sizes was low, with the maximum percentage difference being under 2% in both distributions. For the uniform data distribution in the baseline configuration, the energy usage was 79.23 kJ, while in the Zipfian, it was 81.05 kJ. Regarding throughput, it also

shows an upward trend when the block cache size increases, in both distributions. In the uniform distribution, the highest throughput is achieved when the block cache is 16GiB with a value of 48.3 kops/s; the lowest throughput was achieved with the lowest cache size with a value of 39.54 kops/s. In the Zipfian, the highest throughput was also achieved when the cache size was 16GiB with 125.92 kops/s and the lowest when the cache size was also 512MiB with 103.02 kops/s.

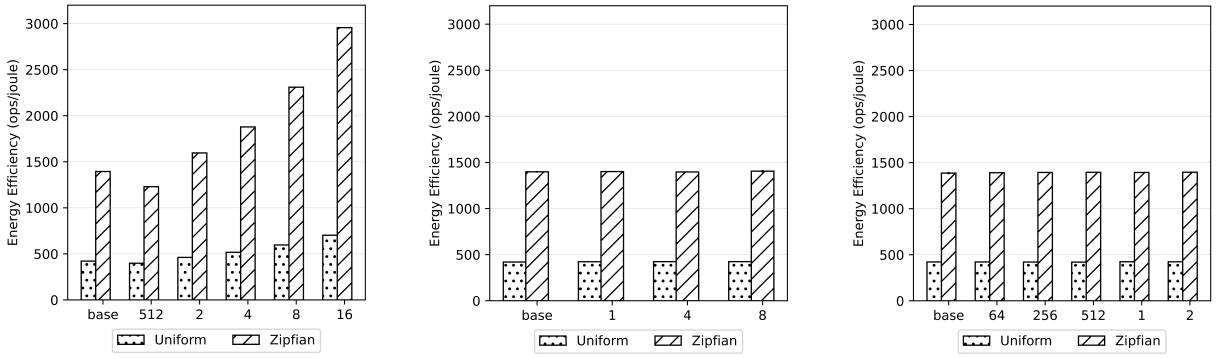
Increasing the maximum number of **MemTable**s in both distributions did not appear to impact the energy efficiency. The energy consumption remained stable through the different limits in the number of **MemTable**s, with the maximum difference between consumption being under 1% in both distributions. The energy consumption in the baseline configuration was 78.85 kJ in the uniform distribution and 80.42 kJ in the Zipfian. The throughput between different configurations remains constant in both distributions, with the maximum percentage difference being 2%, while the maximum difference between repetitions of the same configuration was also under 2%. The throughput in the baseline configuration was 42.34 kops/s in the uniform distribution, while in the Zipfian distribution, it was 109.54 kops/s.

Similar to the write-only workload, the mixed read-write workload efficiency also benefits from having a bigger **MemTable**. An upward trend is visible in both distributions. However, this increase is marginally bigger in the Zipfian with a 54% increase, contrasting a 39% change in the uniform. Regarding energy usage, it remained stable through the different size configurations, with a maximum difference of under 1% in the uniform distribution and short of 2% in the Zipfian. The throughput followed the same trend as the efficiency.

Mixed read-write workload discussion. Increasing the block cache size enhances energy efficiency by improving the block cache hit/miss ratio. With a larger cache, data blocks containing the queried key are more likely to be found in memory, reducing lookup latency and leading to greater efficiency.

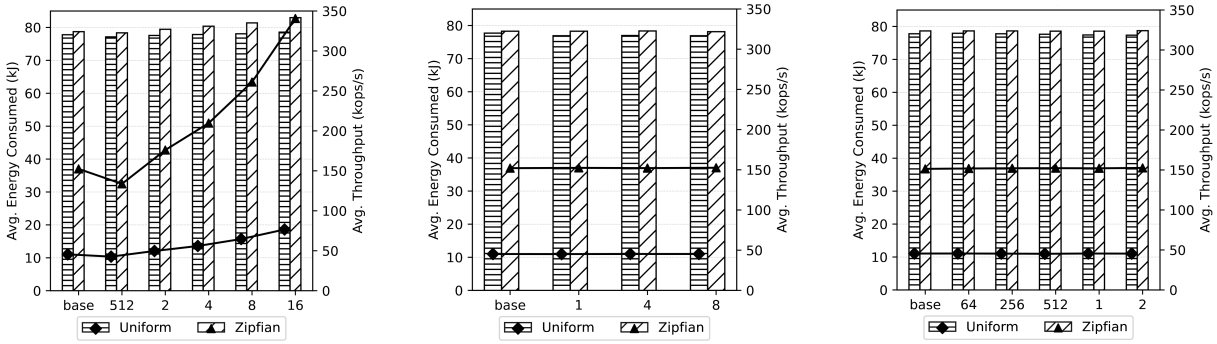
The results from changing the maximum number of **MemTable**s do not show a significant trend to extract meaningful insights.

However, changing the size of the **MemTable** increased the energy efficiency, especially in the Zipfian distribution. Since almost all requests are made to the same key range, most requests will be served by the **MemTable**. Increasing the size of the **MemTable** further increases the probability of keys being found in the write buffer. Furthermore, increasing the write buffer size decreases the number of flushes and compactions since there is a higher probability that keys in **MemTable** are duplicated and merged before a flush, resulting in less data being moved to the persisted levels.



(a) Block cache size (b) Max. Number Memtables (c) Memtable size

Figure 20: Average energy efficiency for different memory configurations in read-only workload.



(a) Block cache size (b) Max. Number Memtables (c) Memtable size

Figure 21: Average energy consumption and throughput for different memory configurations in read-only workload.

Figure 20 presents the average energy efficiency for different memory configurations under the read-only workload. In contrast, figure 21 provides a complementary analysis by presenting the average energy consumption and throughput for the same memory configurations. Both figures follow the same format and structure as figures 16 and 17.

Read-only workload results. Increasing block cache size shows an increase in energy efficiency. This increase in energy efficiency has more expression in the Zipfian distribution. The variability between repetitions remains small in both distributions, with the biggest percentage difference being under 1% in both distributions. The energy consumption increases with bigger cache sizes, with the biggest consumption being when the block cache size is 16GiB for both distributions, with 82.97 kJ in the Zipfian and 78.54 kJ in the uniform. The lowest consumption was reported when the cache size was 512MiB, with 78.73 kJ in the Zipfian and 77.79 kJ in the uniform distribution. The throughput shows the same upward trend as

the energy consumption. For both distributions, the biggest throughput was achieved with a cache size of 16GiB with 340.61 kops/s for the Zipfian and 76.59 kops/s for the uniform distribution. The lowest throughput for the Zipfian distribution was 133.71 kops/s and 42.63 kops/s for the uniform, on a cache size of 512MiB, in both distributions. The percentage change between the lowest and highest throughput configurations was 155% and 80% for Zipfian and uniform distributions, respectively.

Varying the maximum number of **MemTable**s does not impact energy efficiency, consumption, or throughput. All the energy consumption values for the Zipfian distribution remain within a 1% difference of the baseline configuration—two **MemTable** limit—value of 78.27 kJ; the uniform values also are within a 1% difference in comparison to the baseline configuration value of 77.68 kJ. The same pattern is observed in the throughput, where the biggest difference for both distributions, when compared to the baseline, is 1%. The average throughput reported from the baseline configuration value was 152.03 kops/s and 45.25 kops/s, Zipfian and uniform distributions, respectively.

Changing the **MemTable** size also did not impact energy efficiency, energy usage and throughput. The energy consumption of the baseline configuration was 78.64 kJ for the Zipfian distribution and 77.80 kJ for the uniform. The maximum difference observed between energy consumption values was below 1%. The throughput of the baseline configuration was 151.39 kops/s and 45.52 kops/s for the Zipfian and uniform distributions, respectively. The maximum difference between throughput results was short of 1%.

Read-only workload discussion. When increasing the block cache size, the increase in energy efficiency can be attributed to a bigger block cache hit/miss ratio. With a bigger block cache size, there is a higher chance a data block containing the queried key will be present in the cache, leading to less latency in data block lookup and improving efficiency.

Since this is a read-only workload, the energy efficiency was constant through different configurations of the **MemTable**. No writes are being performed, so the **MemTable** remains empty and cannot be used as an extra memory cache for reading values.

Takeaways

We draw three main takeaways from these three experiments.

1) Larger block cache size increases energy efficiency for read-dominated workloads. Increasing the block cache in workloads with reads enhances the likelihood that searched data blocks are present, improving latency and increasing energy efficiency. This increase is more significant in the read-

only workload, as new writes do not trash the block cache.

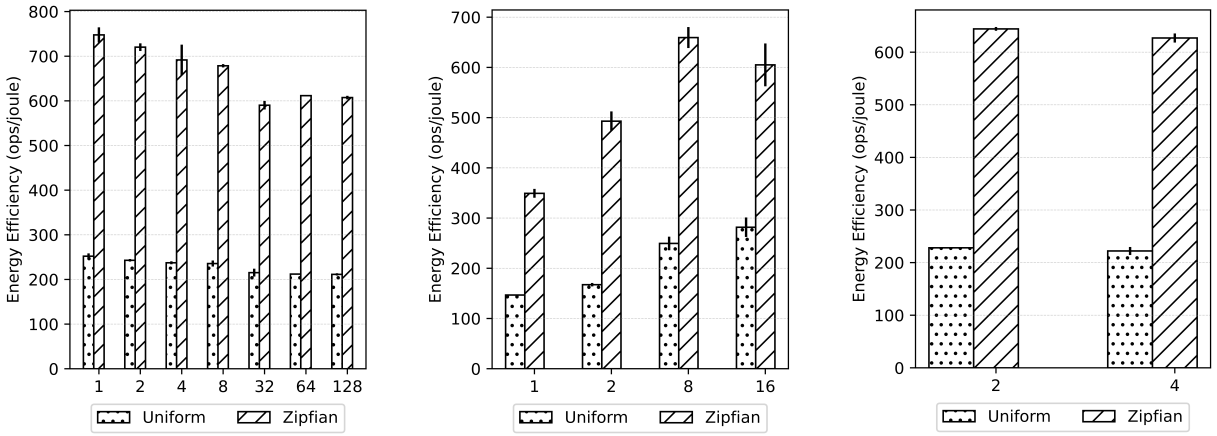
2) The maximum number of memtables might not impact energy efficiency. On read-only workloads, this configuration did not impact energy efficiency. On workloads with writes, the impact on energy efficiency was not evident. The change in energy efficiency was too small and too close to the variability between the same configuration repetitions to confidently assert that this configuration had a negative or positive impact on energy efficiency.

3) Larger memtable size increases energy efficiency for write-dominated workloads. On both workloads with proportions of writes, this configuration positively impacted the energy efficiency of the **KVS**. This efficiency increase resulted from a bigger number of **MemTable** hits in the mixed read-write workload, and a bigger number of in-place writes in the **MemTable** in both workloads.

4.5 Energy Efficiency Under Different Concurrency Configurations

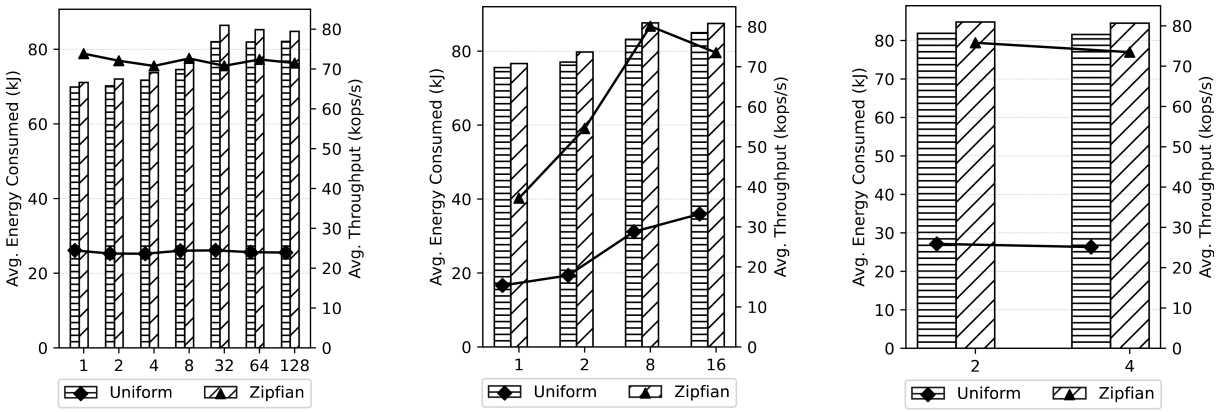
This category of tests focuses on evaluating how different configurations of foreground, compaction and flush threads can affect the energy efficiency of RocksDB.

Experimental setting. The experimental environment remained the same as the RocksDB baseline configuration. Three batches of tests were prepared, each running with a different workload, including the write-only, mixed read-write and read-only workloads. Each batch ran the baseline configurations while only changing one thread setting at a time. The foreground threads were adjusted to 1, 2, 4, 8, 32, 64, and 128, while compaction threads were configured to 1, 2, 8, and 16. Additionally, flush threads were set to 2 and 4 to examine parameter variations.



(a) Foreground threads (b) Compaction threads (c) Flush threads

Figure 22: Average energy efficiency for different concurrency configurations in the write-only workload.



(a) Foreground threads (b) Compaction threads (c) Flush threads

Figure 23: Average energy consumption and throughput for different concurrency configurations in the write-only workload.

Figure 22 presents the average energy efficiency for different thread settings under the write-only workload. Energy efficiency was measured in operations per Joule and evaluated across different concurrency configurations, considering two distinct data distributions: Uniform and Zipfian. The bar graphs illustrate the differences in energy efficiency across configurations. The subfigure 22a shows how different numbers of foreground threads affect the energy efficiency. Subfigure 22b explores the effect of varying the number of compaction threads. Subfigure 22c evaluates how different numbers of flush threads impact energy efficiency.

Figure 23 complements the previous figure by presenting the average energy consumption and throughput for the same concurrency configurations. The bar graphs represent energy consumption, measured in kilojoules, while throughput, measured in thousand operations per second (kops/s), is shown using

line plots. Similar to figure 22, subfigure 23 analyses the impact of changing the number of foreground threads, subfigure 23b examines the effect of varying compaction threads number, and subfigure 22c considers different number of flush threads. The energy depicted in figure 22 and 23 was extracted from the package where RocksDB ran in isolation; the throughput was calculated based on the total operations reported by db_bench.

Write-only workload results. Increases in the number of foreground threads cause a drop in energy efficiency. On a Zipfian data distribution, the energy efficiency changes by 21% between one foreground thread and 32 threads; beyond this point, the efficiency stabilizes with no significant variations. On the other hand, on a uniform distribution, the energy efficiency gradually declines, exhibiting a 16% drop between one foreground thread and 128 threads. Energy usage trends indicate that more foreground threads increase energy consumption. However, the energy consumption stabilises when the number of foreground threads is increased by more than 32. As for throughput, it remained consistent through the different configuration values. The single-thread configuration on a Zipfian data distribution achieved an average of 73.85 kops/s, with other setting values within a 4% maximum difference. Similarly, the uniform data distribution on a single-thread configuration achieved 24.46 kops/s with other thread values also being under 4% difference.

On the other hand, increasing the number of compaction threads, in general, positively impacted energy efficiency. The energy efficiency peaked with 8 compaction threads on the Zipfian data distribution, increasing 89% compared to the single-thread setting and then decreasing 8% when doubling the compaction threads to 16. On the uniform distribution, the efficiency gradually increased, achieving an increase of 92%. With respect to energy consumption, as the number of compaction threads increases from one to 16, energy consumption rises for both distributions.

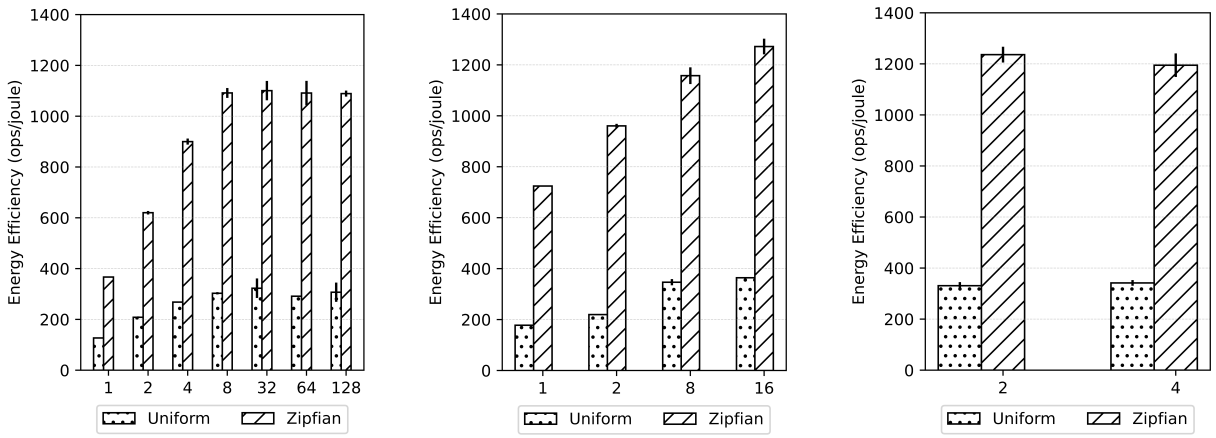
Finally, varying the number of flush threads causes a slight change in energy efficiency. The marginal decrease is negligible, reflecting a 3% decrease between the different thread values in both distributions. For comparison, between repetitions of the same thread number on the uniform data distribution experiments, the maximum difference was 5%. A look at energy consumption reveals that it remains constant with different thread settings. The single-thread flush setting consumed 81.84 kJ and 84.81 kJ on the uniform and Zipfian distributions, respectively. The maximum percentage difference between energy usage was under 1% in both distributions. Regarding throughput, there was a visible decrease in the Zipfian distribution experiments, decreasing from 75.78 kops/s on the single-thread setting to 72.66 kops/s at four flush threads, representing a 4% decrease. The maximum difference observed between the four flush thread setting repetitions was 2%. In the uniform distribution, the single-thread setting achieved an aver-

age of 25.89 kops/s with a maximum difference between thread values of 3% and a maximum difference of 5% between repetitions when there were four flush threads.

Write-only workload discussion. The drop in energy efficiency when increasing the number of foreground threads is explained by a bottleneck from the compaction threads' inability to cope with incoming writes, resulting in a constant throughput. However, the energy consumption increases with the number of threads until the number of threads in the application amounts to the maximum number of cores in the processor socket. This increase is explained by more threads actively trying to write and being sent to a sleep period due to write stalls demanded by compaction threads. This form of semi-active waiting from foreground threads increases energy consumption without improving performance. For this reason, the energy efficiency drops with an increased number of foreground threads. It then stabilises because the number of threads actively performing a **CPU** task is limited by the number of cores in the socket, and thus both energy and throughput stabilise.

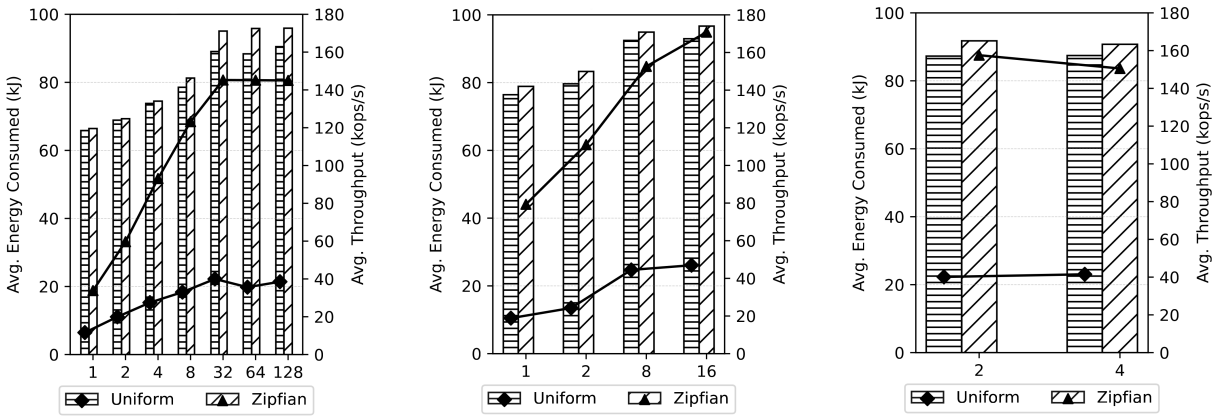
On the other hand, increasing the number of compaction threads positively impacted energy efficiency. Increasing the number of compaction threads makes it possible to parallelise the compaction work, freeing the low-numbered levels faster to accept more writes from the foreground threads. Finally, the energy consumption increases because more threads are doing parallel compactions, although the throughput increases faster than energy usage. However, on a Zipfian distribution, when the number of compaction threads increased from 8 to 16, there was a decrease in throughput while the energy consumption remained constant. This resulted from low-level compactions taking longer than usual, resulting in accumulation of **SSTable**s in level 0 and subsequent saturation of the **MemTable**, resulting in stalls and degradation of client performance. The low-level compactions usually have a higher chance of loading data directly from the page cache, resulting in faster low-level compaction times. However, increasing the number of parallel compactions increases the chance of multiple compactions reading and writing to the page cache, causing trashing and forcing the low-level compactions to load more from disk.

Lastly, the results from varying the number of flush threads were insignificant to make assumptions on their impact on energy efficiency.



(a) Foreground threads (b) Compaction threads (c) Flush threads

Figure 24: Average energy efficiency for different concurrency configurations in the mixed read-write workload.



(a) Foreground threads (b) Compaction threads (c) Flush threads

Figure 25: Average energy consumption and throughput for different concurrency configurations in the mixed read-write workload.

Figure 24 presents the average energy efficiency for different concurrency configurations under the mixed read-write workload. In contrast, figure 25 provides a complementary analysis by presenting the average energy consumption and throughput for the same concurrency configurations. Both figures follow the same format and structure as figures 22 and 23.

Mixed read-write workload results. Energy efficiency improves significantly as foreground threads increase, though the stabilisation point differs across distributions. In the Zipfian distribution, efficiency rises rapidly, reaching a 197% increase by 8 threads, when compared to the single-thread efficiency, after

which it stabilizes, fluctuating within a 1% difference. Similarly, the uniform distribution continues improving until 32 threads, achieving a 154% increase before levelling off within a maximum 10% difference. Regarding energy consumption, both distributions follow the same pattern of increasing energy usage, followed by a levelling usage period. In the uniform distribution, consumption initially rises from 65.87 kJ with a single foreground thread to 89.07 kJ at 32 threads, reflecting a 35% increase; it then stabilizes, fluctuating within a 2% difference. In the Zipfian distribution, energy usage increases from 66.40 kJ to 95.06 kJ between one thread and 32 threads, a 43% increase; afterwards, the consumption levels off within a 1% difference. Observing throughput performance, both distributions follow the same trend of performance improvements followed by a plateau when increasing the number of foreground threads. The throughput increases from 11.62 kops/s to 39.97 kops/s in the uniform distribution, while it improves from 33.84 kops/s to 145.23 kops/s in the Zipfian, an improvement of 244% and 329%, respectively. For thread values bigger than 32, variability between repetitions increases, with the maximum difference being 11% for the uniform and 8% for the Zipfian.

Scaling up the number of compaction threads results in an improvement in energy efficiency in both distributions. The energy efficiency improved by 105% in the uniform distribution, while it increased by 76% in the Zipfian. As the number of compaction threads increases, energy consumption rises for both the uniform and Zipfian distributions. Under the uniform distribution, energy consumption increases from 76.43 kJ with one thread to 92.94 kJ with sixteen threads, a 22% increase. In the Zipfian distribution, energy consumption increases from 78.89 kJ at one thread to 96.67 kJ at sixteen threads, a 23% increase. Throughput increases as the number of compaction threads increases, with distinct trends noted between the uniform and Zipfian distributions. For the uniform distribution, throughput rises from 18.87 kops/s with one thread to 46.97 kops/s with sixteen threads, reflecting a 149% increase. In contrast, the Zipfian distribution shows a more pronounced gain, with throughput increasing from 79.33 kops/s at one thread to 170.80 kops/s at sixteen threads, resulting in a 115% increase. These results indicate that the Zipfian distribution exhibits higher throughput across all thread counts, but the uniform distribution shows a more substantial relative improvement as the number of threads increases.

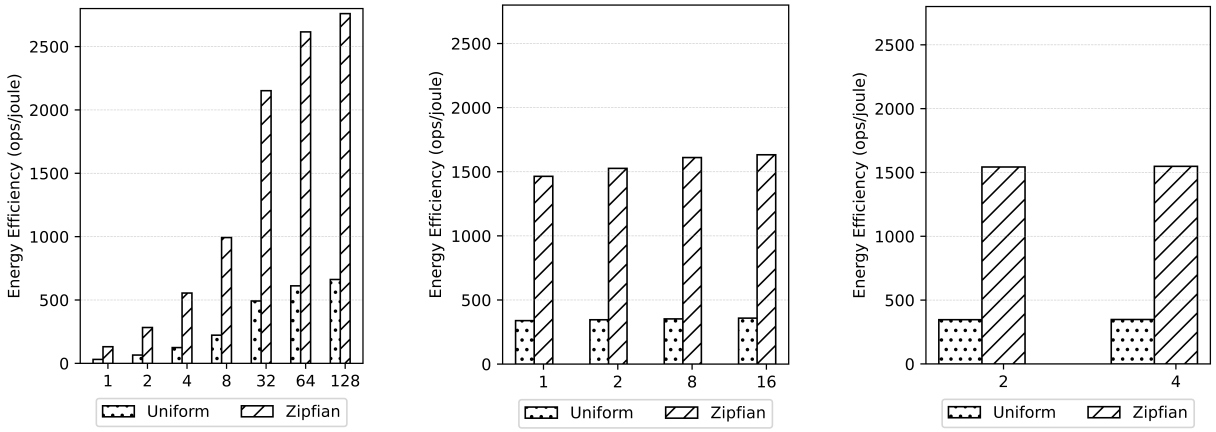
Regarding flush threads, there was a slight but not significant impact on energy efficiency. The maximum difference between different flush settings was 3% for both distributions, while the maximum difference between repetitions was 5% in the Zipfian and 6% in the uniform. The energy consumption remained constant for both distributions. When the number of flush threads was two, the energy usage was 91.80 kJ in the Zipfian distribution and 87.30 kJ in the uniform. Regarding the throughput, there was an apparent decline when the number of flushes increased to four in the Zipfian distribution. However, this

difference in performance between two and four flush threads was smaller than the maximum difference between repetitions; the difference between the average performance was 5%, while the maximum difference between repetitions was 7%. On a uniform distribution, the maximum difference between different flush settings was 3%, and the same maximum difference was observed between repetitions. The average throughput of the configuration with two flush threads was 157.57 kops/s in the Zipfian and 40.11 kops/s in the uniform distribution.

Mixed read-write workload discussion. When the number of foreground threads increases, the energy efficiency increases. The increase in throughput can be attributed to reduced pressure in compaction threads by the decrease in write proportion. The throughput stabilises when compaction threads stop being able to absorb incoming writes when foreground threads can generate a large enough number of writes while maintaining the 50-50 read-write proportion. The energy consumption increases with the number of threads until the point where foreground threads spend most of their time having throttled writes, at which point the consumption plateaus.

Similar to what was observed on the write-only workload, the energy efficiency increases with a bigger number of compaction threads. In this scenario, performance does not degrade when we reach 16 compaction threads because the number of compactions performed is lower, resulting in fewer cases of the page cache being trashed due to big compaction activity.

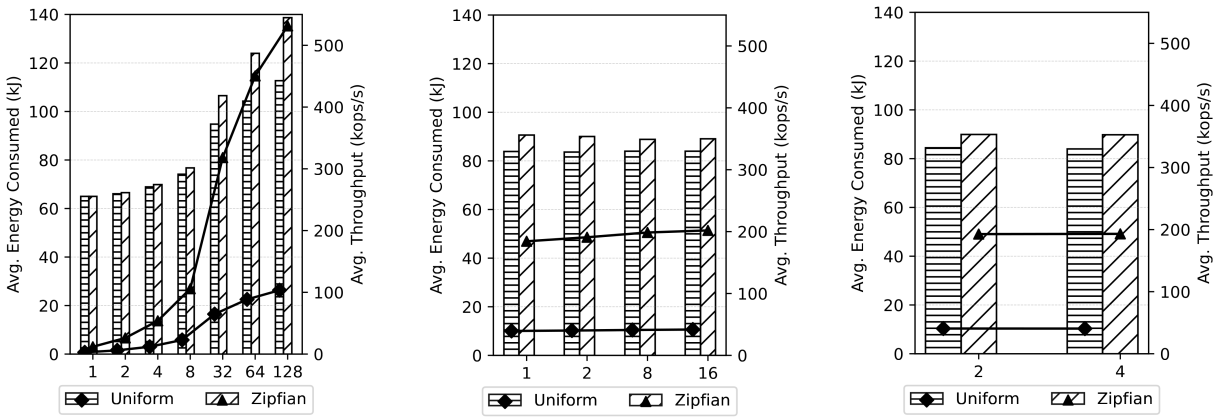
Finally, the outcomes from changing the number of flush threads were too minor to draw conclusions about their effect on energy efficiency.



(a) Foreground threads (b) Compaction threads (c) Flush threads

Figure 26: Average energy efficiency for different concurrency configurations in read-only workload.

Figure 26 presents the average energy efficiency for different concurrency configurations under the



(a) Foreground threads (b) Compaction threads (c) Flush threads

Figure 27: Average energy consumption and throughput for different concurrency configurations in read-only workload.

read-only workload. In contrast, figure 27 provides a complementary analysis by presenting the average energy consumption and throughput for the same concurrency configurations. Both figures follow the same format and structure as figures 22 and 23.

Read-only workload results. Increasing the number of foreground threads positively impacted energy efficiency. There was a notable increase in energy efficiency, with an improvement of 2002% between the single-thread configuration and the configuration using 128 threads for the uniform distribution and a 2025% increase for the Zipfian distribution. Regarding energy consumption, there is also an upward trend in usage. The lowest consumption values are associated with the single-thread configuration, which recorded 64.96 kJ in a uniform distribution and 65.04 kJ in a Zipfian distribution. The highest energy usage was recorded when the number of threads was 128, with 112.65 kJ for the uniform distribution and 138.66 kJ for the Zipfian. Increasing the number of threads from one to 128 on a uniform distribution resulted in a 73% increase, while on a Zipfian distribution, it led to a 113% increase. As for throughput, there was a stark increase when the number of foreground threads increased. On both distributions, the worst performance was recorded in the single-thread setting, while the highest throughput was achieved when the number of threads was set to 128. The performance on the uniform distribution increased from 2.83 kops/s to 103.43 kops/s, a substantial 3555% increase. Regarding the Zipfian distribution, throughput increased from 11.84 kops/s to 531.37 kops/s, representing a significant 4388% growth.

On the other hand, increasing the number of compaction threads did not significantly impact the energy efficiency. There was a gradual increase in efficiency by increases in the number of compaction threads used. However, the increase between the single-thread setting and the configuration with 16

compaction threads was small, being 6% in the uniform distribution and 10% in the Zipfian. The energy consumption remained stable through all thread configurations. On the uniform distribution, the single-thread configuration consumed 83.80 kJ and the maximum percentage difference between configurations was under 1%; on the Zipfian, the single-thread setting consumed 90.70 kJ, with all other settings' usages being under a 3% difference. The results show a slight increase in throughput when the number of threads is increased. The single-thread configuration achieved 39.46 kops/s and 184.40 kops/s for the uniform and Zipfian data distributions, respectively. The highest performance was recorded when the number of compaction threads was 16: 41.77 kops/s for the uniform and 201.94 kops/s for the Zipfian distribution. On the uniform, there was a small increase of 6% in performance, while on the Zipfian, there was an increase of 10%; changes similar to the increases in efficiency.

Lastly, changes in the number of flushes did not impact energy efficiency. On both distributions, the maximum difference between different configurations was under 1%. On the other hand, the maximum difference between repetitions was close to 1%. Similarly, the variability in energy usage from running different thread settings was low, under 1% in both distributions. The energy consumption of the two-flush thread pool setting was 84.48 kJ on the uniform distribution and 89.96 kJ on the Zipfian distribution. The variability in throughput is also very low, with the maximum difference between different settings of the same distribution being well under 1%. The throughput with two threads was 40.63 kops/s on the uniform distribution and 192.74 kops/s on the Zipfian distribution.

Read-only workload discussion. As the number of foreground threads rises, the energy efficiency increases. Since compaction threads do not bottleneck the disk, foreground reads can scale with more foreground threads, as the disk resources are not saturated—disk read bandwidth still increases from 64 to 128 foregrounds. However, when the number of foreground threads is higher than 32, the number of threads is higher than the physical cores available on the socket, causing diminishing returns on throughput. Past 32 threads, a backlog of threads is formed waiting to be scheduled to **CPU** when other threads wait for I/O. The unscheduled threads and the overhead of context switching lead to inefficiencies, causing a slower increase in energy efficiency.

A higher number of compaction threads led to a small boost in efficiency. The backlog of compactions from the previous fill run was being scheduled. With increasing compaction threads, the total number of compactions performed increased. With no incoming writes, performing a bigger total amount of compactions faster allows the **LSM-Tree** to clear level 0 **SSTable**s faster, reducing the total amount of inefficient reads to that level. The number of **SSTable**s read increases significantly when level 0 is not empty, as the key ranges of level 0 files overlap, and all tables must be read to determine whether the key

is present. Eventually, this led to a better throughput, while the energy consumption remained practically unchanged.

Finally, changes in the number of flushes did not affect the efficiency. This makes sense because flush threads were not utilized in all flush configurations, as there were no incoming writes.

Takeaways

We draw three main takeaways from these experiments.

1) Improvements in efficiency by increasing foreground threads are dependent on workloads. On write-intensive workloads, compaction threads might be unable to keep up with incoming writes, causing the foreground threads' writing to stall. In this scenario, increasing the number of foreground threads will increase energy consumption as the number of cores idle in the socket decreases. At the same time, the compaction threads continue issuing delays and stop requests to the foregrounds, causing performance to plateau and efficiency to drop. However, efficiency increases when the proportion of writes drops to a point where compaction threads do not pose a bottleneck. The threads can be scheduled when others wait for I/O, increasing the time cores spend doing productive work.

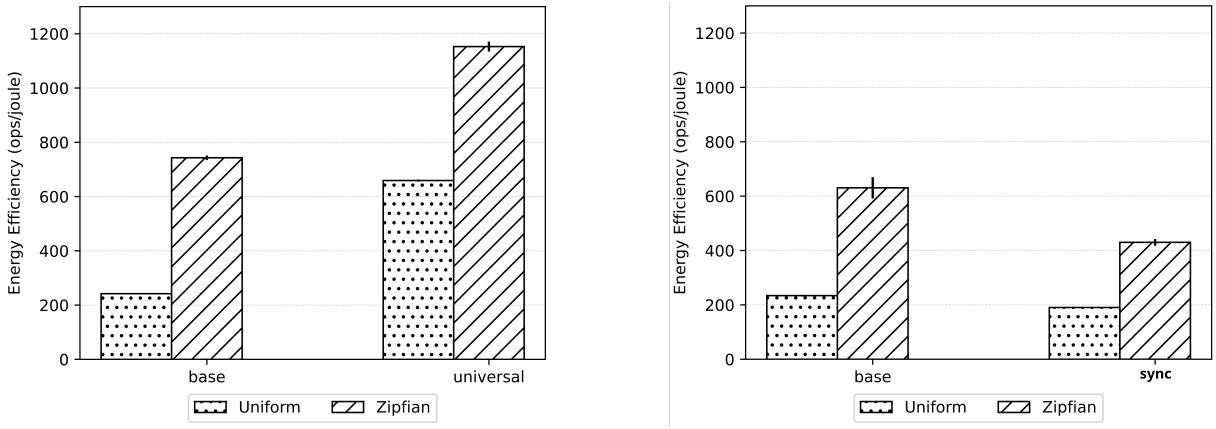
2) Increasing compaction threads generally improves efficiency. On workloads with writes, increasing the number of compaction threads allows low-numbered levels to be compacted faster, enabling a faster rate of flushes. This comes at the expense of higher energy consumption; however, the performance improves faster than energy usage, improving efficiency. There is a point, though, where increasing the number of compaction threads can decrease efficiency, when the compaction activity becomes to big and trashes the page cache, leading to bigger low-level compaction times and client stalls. In the read-only workload, the faster compaction rate associated with more threads clears the tables in level 0 faster, resulting in fewer total searches performed.

3) Flush configurations might not affect efficiency. In the context of these experiments, no significant change was observed to assess the impact of varying flush configurations on efficiency confidently.

4.6 Energy Efficiency in Compaction Schemes and Durability Configurations

The configurations discussed in this section are related to internal components of the **KVS**. This section will describe and discuss the impact of compaction style and **WAL** durability guarantees on energy efficiency on three different workloads.

Experimental setting. The baseline RocksDB configuration was used as a starting ground for the experiments. Three batches of tests were prepared, each running a different workload with different read-write proportions; these included the write-only workload, the mixed read-write workload and the read-only workload. For each test run in the batch, only one configuration was changed at a time. The compaction style was changed from the levelled, the default baseline configuration, to tiered. Moreover, **WAL** syncing was switched from off, also the baseline setting, to on.



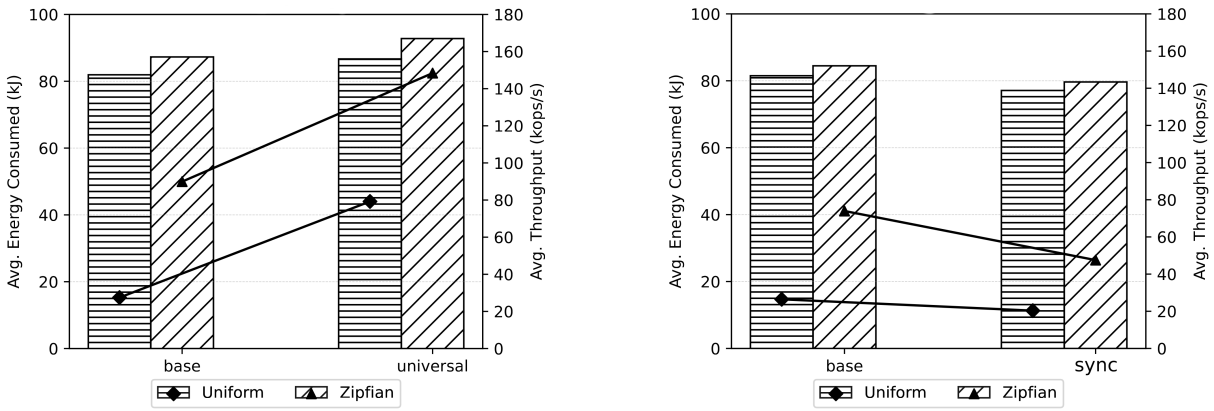
(a) Compaction style

(b) WAL synchronisation

Figure 28: Average energy efficiency for different compaction schemes and durability configurations in write-only workload.

Figure 28 presents the average energy efficiency for different compaction schemes and durability configurations under a write-only workload. Energy efficiency, measured in operations per Joule, is evaluated across varying settings while considering two different data distributions: uniform and Zipfian. The bar graphs illustrate the differences in energy efficiency across configurations. Subfigure 28a examines the impact on the energy efficiency of using the baseline levelled configuration or the universal tiered compaction style. Subfigure 28b explores how **WAL** syncing impacts energy efficiency.

Figure 29 provides a complementary analysis by presenting the average energy consumption and



(a) Compaction style

(b) WAL synchronisation

Figure 29: Average energy consumption and throughput for different compaction schemes and durability configurations in write-only workload.

throughput for the same configurations. Energy consumption, measured in kilojoules, is represented by the bar graphs, while throughput, measured in thousand operations per second (kpos/s), is depicted using line plots. Similar to figure 28a, subfigure 29a analyses the effect of compaction style, and subfigure 29b examines the impact of **WAL** syncing. The energy depicted in figures 28 and 29 was extracted from the package where RocksDB ran in isolation; the throughput was calculated based on the total operations reported by the RocksDB benchmark tool.

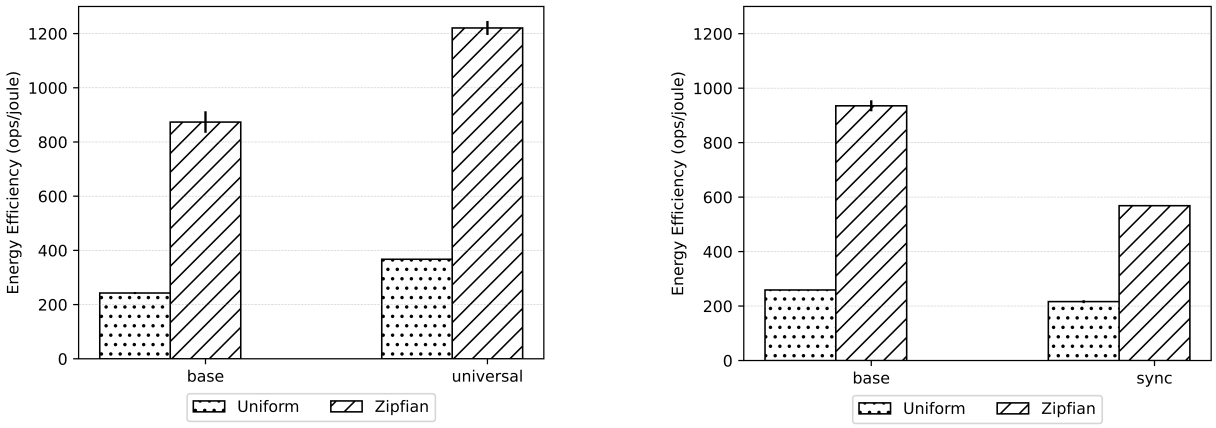
Write-only workload results. When modifying the compaction style from the levelled style to the universal tiered style, there was an increase in energy efficiency. The increase in efficiency from the levelled to the universal was 172% in the uniform distribution and 55% in the Zipfian. The energy consumption of the levelled compaction style for the uniform data distribution was 81.86 kJ, while on the tiered, it was 86.56 kJ. For the Zipfian distribution, the levelled consumed 87.33 kJ and the tiered 93.05 kJ. The increase in energy consumption from the levelled to the tiered was 6% in the uniform distribution and 7% in the Zipfian. Regarding throughput, the uniform distribution increased 188% from 27.48 kops/s in the levelled to 79.23 kops/s in the universal tiered. On the Zipfian, there was a positive change between the levelled and the tiered of 65% from 89.93 kops/s to 148.39 kops/s, respectively.

Regarding the impact of **WAL** syncing, when it was turned on, the energy efficiency decreased 23% in the uniform distribution and 36% in the Zipfian. On a uniform data distribution, the energy dropped from 81.36 kJ, when the **OS** decided when to synchronise the **WAL** writes, to 76.90 kJ when the **WAL** was synchronised every write, marking a 6% decrease. On the Zipfian data distribution, the consumption decreased from 83.73 kJ when **WAL** syncing was disabled to 79.86 kJ when syncing was enabled, a 5%

decrease. Regarding throughput, there was a clear decrease in both data distributions when **WAL** syncing was activated. In the uniform distribution, the throughput went from 26.44 kops/s to 20.33 kops/s, a 23% decrease; in the Zipfian, the throughput dropped 36% went from 73.94 kops/s to 47.53 kops/s. The changes in throughput matched the drop in efficiency for both distributions.

Write-only workload discussion. Switching from a levelled compaction style to a universal tiered style allows the **KVS** to absorb more writes. The increase in throughput comes from the fewer stalls imposed on foreground threads, enabling a faster rate of writes to be submitted by foregrounds. The main reason compaction threads can accept more write requests is that due to the merge policy of the tired strategy, there is less write amplification, resulting in an optimised strategy for writes. Although there is a slight increase in energy consumption because more write requests are being made, the increase in throughput is more significant, leading to increased efficiency.

On the other hand, doing a synchronisation for each **WAL** write decreases the energy consumption and throughput compared to the baseline configuration. The decrease in throughput is directly related to the waiting period of the synchronisation operation; essentially, RocksDB is bottlenecked by the disk latency in each write. In the waiting period, the **CPU** is idle, resulting in energy spent on cycles that are not advancing the number of operations performed. The energy decreases because more cores have a higher proportion of idle cycles residency, but these cycles are not productive, resulting in a higher decline in throughput.

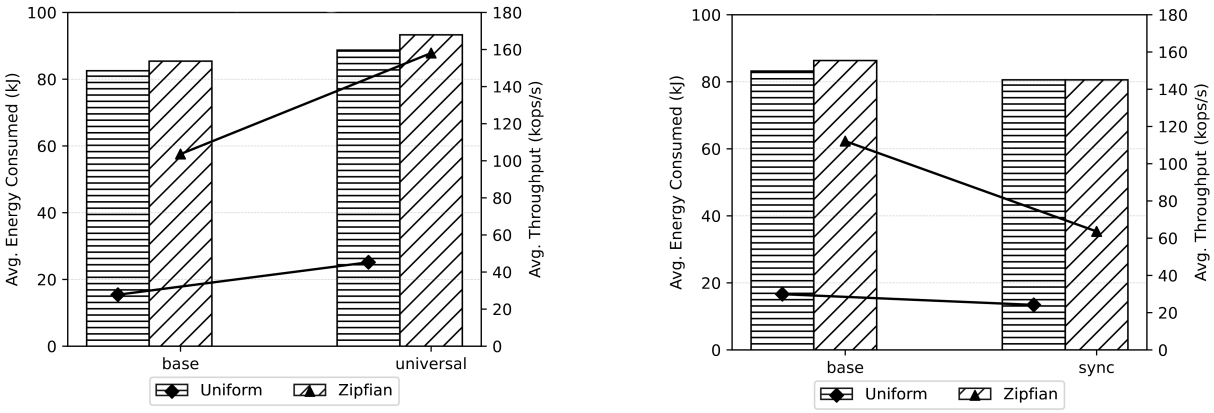


(a) Compaction style

(b) WAL synchronisation

Figure 30: Average energy efficiency for different compaction schemes and durability configurations in mixed read-write workload.

Figure 30 presents the average energy efficiency for different compaction strategies and durability configurations under the mixed read-write workload. In contrast, figure 31 provides a complementary



(a) Compaction style

(b) WAL synchronisation

Figure 31: Average energy consumption and throughput for different compaction schemes and durability configurations in mixed read-write workload.

analysis by presenting the average energy consumption and throughput for the same compaction strategies and durability configurations. Both figures follow the same format and structure as figures 28 and 29.

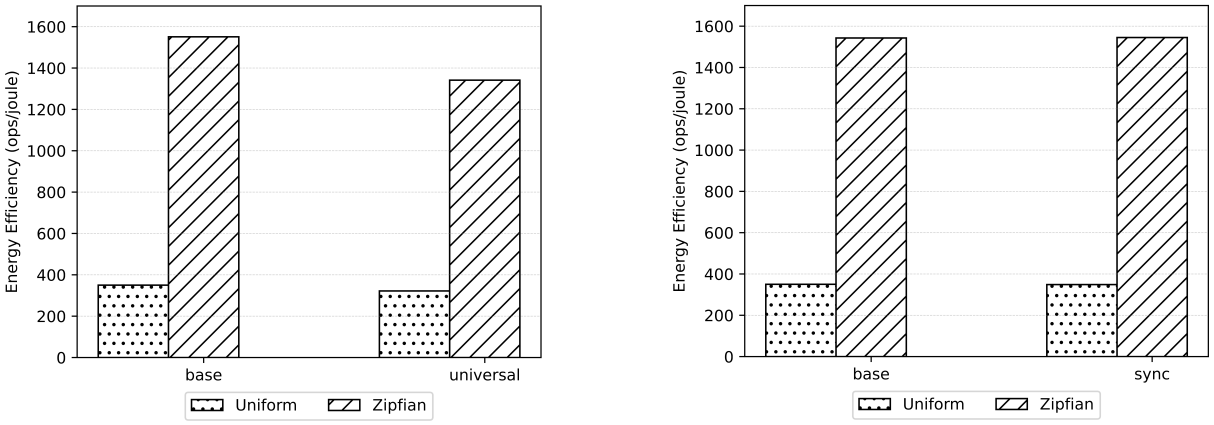
Mixed read-write workload results. A positive impact on energy efficiency is observed when comparing the energy efficiency between the levelled compaction style and the tiered. The efficiency increases by 52% in the uniform distribution and 40% in the Zipfian. For both distributions, the energy usage is the lowest in the levelled compaction style, with 82.50 kJ in the uniform and 85.429 in the Zipfian; in the tiered compaction style, the consumption was 88.75 kJ and 93.26 kJ, in the uniform and Zipfian, respectively. The increase in energy consumption was 8% for the uniform distribution and 9% for the Zipfian. For the levelled compaction strategy, the throughput under the uniform data distribution was 27.77 kops/s, while under the Zipfian distribution, it was 103.62 kops/s. For the universal tiered compaction, the throughput on the uniform distribution was 45.26 kops/s and 158.10 kops/s in the Zipfian. Comparing the throughput of the two compaction styles, the tiered strategy achieved a 63% higher throughput than the levelled under the uniform distribution and a 53% higher throughput in the Zipfian distribution.

Doing a **WAL** synchronisation every time a write is performed worsens energy efficiency. Letting the **OS** independently flush the **WAL** pages and instead ensuring the write is persisted on disk decreased efficiency by 17% on the uniform distribution and 39% in the Zipfian. The energy usage dropped from 83.18 kJ to 80.60 kJ in the uniform distribution, a decline of 3%; in the Zipfian distribution, the consumption reduced from 86.32 kJ to 80.58 kJ, a reduction of 7%. In both distributions, the performance also fell when performing synchronisation on every write. In the baseline configuration, the recorded throughput was 29.92 kops/s in the uniform distribution and 112.12 kops/s in the Zipfian distribution. When syncing,

the throughput was 24.18 kops/s in the uniform distribution and 63.62 kops/s in the Zipfian.

Mixed read-write workload discussion. Similar to what was observed in the write-only workload experiment, the tiered compaction strategy enables a faster write absorption rate. The levelled strategy on this mixed read-write workload is still being bottlenecked by the compaction threads' inability to compact faster, and so the rate of writes is being slowed down by write stalls. On the other hand, although there is a higher read amplification associated with the internal structure of the universal style, this drawback is offset by the ability to absorb more write operations, increasing the total number of operations performed.

Lastly, when forcing a synchronisation for each **WAL** write the energy efficiency declines. Since the read-write proportion is fixed and disk latency becomes the primary bottleneck for write operations, this synchronization process can result in a notable decrease in both read and write throughput. As in the write-only workload, it is expected that the energy consumption will decrease as there is a bigger percentage of idle cycles—due to the waiting period of the synchronisation function—being performed, requiring less energy usage.



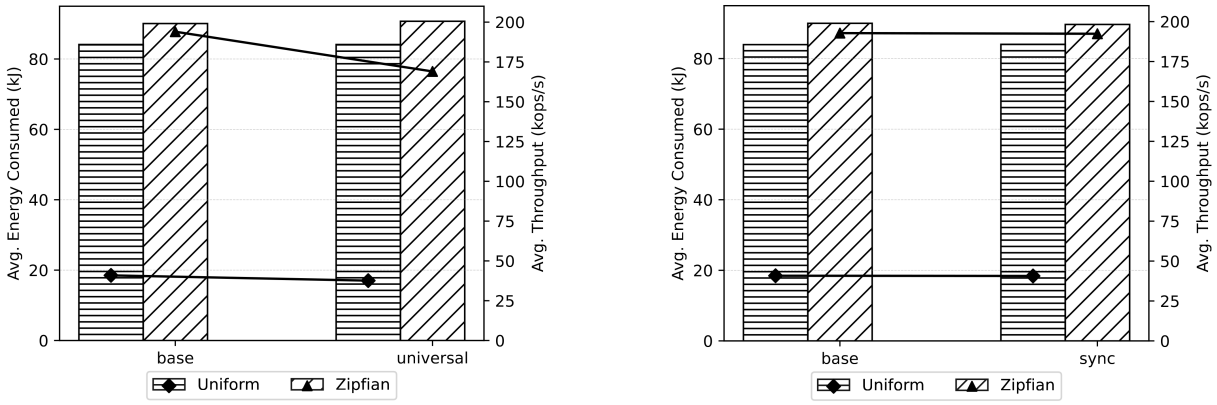
(a) Compaction style

(b) WAL synchronisation

Figure 32: Average energy efficiency for different compaction schemes and durability configurations in the read-only workload.

Figure 32 presents the average energy efficiency for different compaction strategies and durability configurations under the read-only workload. In contrast, figure 33 provides a complementary analysis by presenting the average energy consumption and throughput for the same compaction strategies and durability configurations. Both figures follow the same format and structure as figures 28 and 29.

Read-only workload results. When comparing the compaction style between the baseline configuration levelled strategy and the universal tiered, the energy efficiency in the universal tiered style decreases



(a) Compaction style

(b) WAL synchronisation

Figure 33: Average energy consumption and throughput for different compaction schemes and durability configurations in the read-only workload.

by 8% in the uniform distribution and 14% in the Zipfian data distribution. However, the energy consumption of both compaction styles remains stable. In the uniform distribution, the levelled strategy consumed 84.15 kJ and the universal strategy 84.13 kJ, with virtually no difference between the usages of the two compaction settings; in the Zipfian distribution, the levelled compaction consumed 90.06 kJ and the universal compaction 90.74 kJ. On the other hand, the throughput depicted a decrease in performance when switching to a universal compaction style. In the uniform distribution, the levelled strategy averaged 40.91 kops/s, while the universal averaged 37.61 kops/s, an 8% decrease on par with the drop in efficiency; in the Zipfian, the levelled style averaged 193.97 kops/s, and the tiered averaged 169.02 kops/s, a 13% decline.

The **WAL** synchronisation setting did not have a visible impact on energy efficiency. The percentage difference between settings in both distributions was under 1%. Energy usage also remains constant when activating the **WAL** setting; in both distributions, the difference was well under 1%. The energy consumption reported in the uniform distribution was 83.95 kJ when periodic syncing was not activated and 84.00 kJ when syncing was on; in the Zipfian distribution, the consumption was 89.97 kJ when syncing was disabled and 89.66 kJ when enabled. Similarly, the difference between the throughput reported in both settings and in both distributions was under 1%. When the syncing was disabled, the throughput under the uniform distribution was 40.81 kops/s, while under the Zipfian distribution, it was 192.80 kops/s; when syncing was enabled, the throughput was 40.67 kops/s and 192.39 kops/s, in the uniform and Zipfian distributions, respectively.

Read-only workload discussion. On the read-only workload, the levelled compaction style ensures an internal structure that allows for reduced read amplification, resulting in more efficient reads when compared to the universal style. The throughput gain compared to the universal style is due to the reduced read amplification from the levelled compaction style.

The **WAL** synchronisation setting did not impact energy efficiency in this workload. Because it is a read-only workload, no writes are performed, so the synchronisation function is not performed.

Takeaways

We take two major takeaways from these experiments.

1) Compaction style energy efficiency is dependent on workload. On write-intensive workloads, such as write-only and mixed read-write workloads, the bottleneck is the compaction threads' inability to finish compaction fast enough to absorb new writes to disk. In this scenario, the reduced write implication of the universal style makes compactions more efficient. However, the levelled strategy becomes more efficient on read-dominant workloads due to the lower read amplification. In fact, it is possible to see a gradient of decreasing efficiency while comparing the levelled to the universal style, whereby increasing the read proportion causes an inferior increase in efficiency or, in the case of the read-only, a decrease in efficiency.

2) WAL syncing decreases the energy efficiency of writes. When syncing every time a write operation is performed on the **WAL** file, the idle time of the synchronisation function increases the proportion of idle cycles, decreasing the power consumption; however, while the foreground threads are in this reduced power state, they are not advancing in the number of operations performed, reducing throughput and efficiency. This setting does not impact energy efficiency during read-only workloads since there are no write operations.

4.7 Guidelines for Energy-Efficient KVS Design

The energy efficiency of the evaluated **LSM-Tree KVS** proved to be highly sensitive to its internal configurations, as well as the type of workload and data distributions it was subject to. With this in mind, we condense our findings into a set of guidelines that **KVS** design should follow to improve energy efficiency.

Highly configurable KVS. What improves energy efficiency in one workload setting may worsen it in another, as seen when adhering to the same compaction strategy across different workloads. On the other hand, when there are no memory constraints, increasing the block cache size improves efficiency in read-intensive workloads, and increasing the **MemTable** size improves efficiency in write-intensive workloads. Creating a **KVS** that enables this level of customisation at run-time, adjusting configurations based on workload characteristics, ensures the **KVS** is set up to achieve better efficiency at all times.

Improve temporal locality across all level compactions. Lower-level compactions exploit temporal locality from recent flushes to improve compaction efficiency. On the other hand, compactions at higher-numbered levels do not benefit from lower-latency reads and must incur high-latency reads directly from disk, thereby decreasing efficiency and increasing idle energy usage. **KVS** should restructure merge policies to improve temporal and overall cache locality across all compaction levels.

Reducing contention. Multithreaded operations often face lock contention, resulting in waiting periods and inefficiencies. An increase in the number of foreground or background clients/workers can worsen congestion in these contention areas. To address this issue, **KVS** should focus on minimising synchronisation between threads by redesigning components critical to both foreground and background operations to reduce thread contention. One practical approach could be to apply a shared-nothing architecture at different levels of the **KVS** system.

Chapter 5

Conclusion

With the escalating energy demands of data centres, driven by the growth in digital data and the widespread adoption of backend storage solutions like **KVS** for storing and processing services in the cloud, there has been a growing interest in using energy more effectively in these infrastructures. However, with the end of the Dennard scaling, reducing hardware's energy consumption without losing performance is no longer feasible; therefore, a software-centric approach to improving the energy efficiency of data centres is expected. As **KVS** play a crucial role as the backend storage of many cloud services, understanding their energy profile is essential; however, to the best of our knowledge, no study has set out to uncover the energy impact of these storage systems.

To address this gap, we presented an in-depth experimental study that systematically characterises the energy consumption of LSM-based **KVS**. Specifically, we focused on RocksDB, a widely used and configurable **LSM-Tree KVS**. Due to the multi-threaded environment of **KVS**, energy monitoring tools should correlate the energy of a singular thread with its event type and contextual information. However, as existing solutions did not present such a feature, we developed PowMon, a fine-grained power monitoring system, capable of capturing thread-level energy measurements and correlating them with event contextual data.

The study reveals that workload characteristics and data distributions profoundly shape energy efficiency in **KVS**. Read-intensive workloads are consistently more efficient than write-heavy ones, as writes trigger disruptive background operations (e.g., compactions) that incur energy costs from block cache thrashing and increased sorting loads. Notably, Zipfian data distributions enhance efficiency by reducing read/write amplification through frequent in-memory operations (**MemTable** and block cache) and minimising compactions. In contrast, uniform distributions incur higher energy costs due to more misses in memory operations and a higher density of compaction jobs. Memory configurations further modulate efficiency: larger block caches enhance read performance by reducing disk access, improving efficiency in these workloads, and bigger **MemTable**s optimise write-heavy workloads through in-place updates. Con-

currency properties—such as number of foreground clients and compaction threads—directly influence throughput and energy use differently in different workload types and data distributions, underscoring the need for balanced thread management. Surprisingly, faster disks (**NVMe**, **RAM**) achieve superior overall energy efficiency compared to **SATA** devices, despite higher power consumption, due to reduced idle waiting. Compaction strategies also reveal critical trade-offs: universal compaction improves efficiency in write-heavy workloads by limiting write amplification, while levelled compaction benefits efficiency in read-heavy workloads through reduced read amplification.

These findings underscore the need for **KVS** to be highly configurable, enabling the alignment of workload, data distribution, and hardware properties with the most efficient configurations. Furthermore, this work also provides an efficiency benchmark for future **KVS** to compare to when exploring different **LSM-Tree** organisations, enabling designers to know if they are making progress towards energy-efficient **KVS**.

5.1 Future work

In this section, we discuss several avenues opened by the work presented in this thesis, and also improvements on this work's limitations.

Explore foreground events energy impact. This study covered the implications of applying different **KVS** configurations, essentially benchmarking how different optimisations and internal structures can impact energy efficiency. Moreover, we analysed how different event types can impact energy differently and further explored the energy profile of compaction events. However, foreground events account for the biggest slice of the total energy expenditure in **KVS** operations, even in write-intensive workloads. Deeply investigating how these foreground events impact the energy consumption of the **KVS** can lead to novel energy-based optimisations.

Expand study to more KVS. Several optimisations have been done on top of the classic **LSM-Tree** structure, such as the development of new **KVS**, such as PebblesDB [63], Monkey [18], and Dostoevsky [19]. The performance improvements achieved by these systems could result in a disproportional increase in energy consumption, decreasing energy efficiency. For this reason, benchmarking and profiling these new systems could help recognise possible energy-intensive trends and determine whether the improvements were made at the expense of energy efficiency. Indeed, the PowMon monitoring system, combined with the experimental methodology developed in this dissertation, helps researchers, system

administrators and system designers to evaluate the energy efficiency of other **KVS**.

Realistic benchmarking. The benchmarks in this study were done following synthetic **YCSB** workloads. Although these benchmarks can provide insights into general practical applications, realistic application traces, such as those from Facebook or X (formerly known as Twitter), would increase the confidence in the results and insights extracted when applied in real-world scenarios.

Bibliography

[1] Rocksdb bloom filter. https://github.com/facebook/rocksdb/wiki/RocksDB-Bloom-Filter. Accessed on 2025-04-09.

[2] Inc. Advanced Micro Devices. Preliminary processor programming reference (ppr) for amd family 17h model 31h, revision b0 processors. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/55803-ppr-family-17h-model-31h-b0-processors.pdf, 2020. Accessed: 2025-02-12.

[3] Marcelo Amaral, Huamin Chen, Tatsuhiro Chiba, Rina Nakazawa, Sunyanan Choochotkaew, Eun Kyung Lee, and Tamar Eilam. Kepler: A framework to calculate the energy consumption of containerized applications. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, pages 69–71, 2023.

[4] Inc. Amazon Web Services. Overview of amazon web services: Aws whitepaper, 2024. PDF; accessed February 2, 2025.

[5] Advanced Micro Devices (AMD). amd_energy: Linux kernel module for amd processor energy counters. https://github.com/amd/amd_energy. Accessed: 2025-02-12.

[6] Thomas Anderson, Adam Belay, Mosharaf Chowdhury, Asaf Cidon, and Irene Zhang. Treehouse: A case for carbon-aware datacenter software. *SIGENERGY Energy Inform. Rev.*, 3(3):64–70, October 2023.

[7] Lasse F. Wolff Anthony, Benjamin Kanding, and Raghavendra Selvan. Carbontracker: Tracking and predicting the carbon footprint of training deep learning models, 2020.

[8] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing latency spikes in Log-Structured merge Key-Value stores. In *2019*

USENIX Annual Technical Conference (USENIX ATC 19), pages 753–766, Renton, WA, July 2019. USENIX Association.

[9] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, page 107–141, New York, NY, USA, 1970. Association for Computing Machinery.

[10] Daniel Bedard, Min Yeol Lim, Robert Fowler, and Allan Porterfield. Powermon: Fine-grained and integrated power monitoring for commodity computer systems. In *Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon)*, pages 479–484, 2010.

[11] Dhruba Borthakur. The history of rocksdb. Accessed: 2025-02-11.

[12] Zhao Cao, Shimin Chen, Feifei Li, Min Wang, and Xiaoyang Sean Wang. Logkv: Exploiting key-value stores for log processing. In *Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org, 2013.

[13] Stefano Corda, Bram Veenboer, and Emma Tolley. Pmt: Power measurement toolkit. In *2022 IEEE/ACM International Workshop on HPC User Support Tools (HUST)*, pages 44–47, 2022.

[14] Intel Corporation. Introduction to intel® architecture: The basics. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-introduction-basics-paper.pdf. Accessed: 2025-01-08.

[15] NVIDIA Corporation. *NVIDIA Management Library (NVML) Developer Guide*. https://docs.nvidia.com/deploy/nvml-api/.

[16] Intel Corporaton. Intel® 64 and ia-32 architectures software developer's manual volume 3b, 2023.

[17] Benoit Courty, Victor Schmidt, Sasha Luccioni, Goyal-Kamal, MarionCoutarel, Boris Feld, Jérémy Lecourt, LiamConnell, Amine Saboni, Inimaz, supatomic, Mathilde Léval, Luis Blanche, Alexis Cruveiller, ouminasara, Franklin Zhao, Aditya Joshi, Alexis Bogroff, Hugues de Lavoreille, Niko Laskaris, Edoardo Abati, Douglas Blank, Ziyao Wang, Armin Catovic, Marc Alencon, Michał Stęchły, Christian Bauer, Lucas Otávio N. de Araújo, JPW, and MinervaBooks. mlco2/codecarbon: v2.4.1, May 2024.

[18] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 79–94, New York, NY, USA, 2017. Association for Computing Machinery.

[19] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 505–520, New York, NY, USA, 2018. Association for Computing Machinery.

[20] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. SOSP '07, page 205–220, New York, NY, USA, 2007. Association for Computing Machinery.

[21] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. A validation of dram rapl power measurements. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, page 455–470, New York, NY, USA, 2016. Association for Computing Machinery.

[22] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Stumm. Optimizing space amplification in RocksDB. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR '17)*, Santa Cruz, CA, USA, January 2017. Online.

[23] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Trans. Storage*, 17(4), oct 2021.

[24] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011.

[25] Guillaume Fieni, Romain Rouvoy, and Lionel Seinturier. Smartwatts: Self-calibrating software-defined power meter for containers. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, May 2020.

[26] The Apache Software Foundation. Apache cassandra. https://cassandra.apache.org/, 2025. Accessed: 2025-02-02.

[27] Sanjay Ghemawat and Jeff Dean. Leveldb. https://github.com/google/leveldb, 2011. Accessed: 2025-02-02.

[28] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003.

[29] Binglei Guo, Jiong Yu, Bin Liao, Dexian Yang, and Liang Lu. A green framework for dbms based on energy-aware query optimization and energy-efficient query processing. *J. Netw. Comput. Appl.*, 84(C):118–130, April 2017.

[30] Bryan Harris and Nihat Altiparmak. Ultra-low latency ssds' impact on overall energy efficiency. In *Proceedings of the 12th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage '20, USA, 2020. USENIX Association.

[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.

[32] Peter Henderson, Jieru Hu, Joshua Romoff, Emma Brunskill, Dan Jurafsky, and Joelle Pineau. Towards the systematic reporting of the energy and carbon footprints of machine learning, 2022.

[33] Paul Horowitz and Winfield Hill. *The Art of Electronics*. Cambridge University Press, USA, 3rd edition, 2015.

[34] Hubblo. Scaphandre: Energy consumption metrology agent. https://github.com/hubblo-org/scaphandre. Accessed: 2025-02-05.

[35] Hongyu Hè, Michal Friedman, and Theodoros Rekatsinas. Energat: Fine-grained energy attribution for multi-tenancy. In *Proceedings of the 2nd Workshop on Sustainable Computer Systems*, HotCarbon '23. ACM, July 2023.

[36] Stratos Idreos and Mark Callaghan. Key-value storage engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2667–2672, New York, NY, USA, 2020. Association for Computing Machinery.

[37] Heike Jagode, Anthony Danalis, Giuseppe Congiu, Daniel Barry, Anthony Castaldo, and Jack Dongarra. Advancements of papi for the exascale generation. *The International Journal of High Performance Computing Applications*, 0(0):10943420241303884, 0.

[38] Avita Katal, Susheela Dahiya, and Tanupriya Choudhury. Energy efficiency in cloud computing data centers: a survey on software technologies. *Cluster Computing*, 26(3):1845–1875, Jun 2023.

[39] Manolis Katsaragakis, Christos Baloukas, Lazaros Papadopoulos, Verena Kantere, Francky Catthoor, and Dimitrios Soudris. Energy consumption evaluation of optane dc persistent memory for indexing

data structures. In *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 75–84, 2022.

[40] Kepler Project. Power model design kepler. https://sustainable-computing.io/design/power_model/, 2025. Accessed: 2025-01-21.

[41] 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), mar 2018.

[42] Dimitrios Koutsoukos, Raghav Bhartia, Michal Friedman, Ana Klimovic, and Gustavo Alonso. Nvm: Is it not very meaningful for databases? 16(10), 2023.

[43] Alexandre Lacoste, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. Quantifying the carbon emissions of machine learning, 2019.

[44] Hemasri Sai Lella, Rajrupa Chattaraj, Sridhar Chimalakonda, and Manasa Kurra. Towards comprehending energy consumption of database management systems - a tool and empirical study. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, EASE '24, page 272–281, New York, NY, USA, 2024. Association for Computing Machinery.

[45] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 447–461, New York, NY, USA, 2019. Association for Computing Machinery.

[46] LIKWID Tools. LIKWID tools website. https://github.com/RRZE-HPC/likwid. Accessed: 2025-02-05.

[47] Weiwei Lin, Fang Shi, Wentai Wu, Keqin Li, Guangxin Wu, and Al-Alas Mohammed. A taxonomy and survey of power models and power modeling for cloud servers. *ACM Comput. Surv.*, 53(5), September 2020.

[48] Niels Lohmann. Json for modern c++. https://github.com/nlohmann/json. Accessed on 2025-02-18.

[49] Chen Luo and Michael J. Carey. Lsm-based storage techniques: a survey. *The VLDB Journal*, 29(1):393–418, Jan 2020.

[50] Stella Mikrou, Anastasios Papagiannis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. Power and performance analysis of persistent key-value stores. *CoRR*, abs/2008.13402, 2020.

[51] Ingo Molnár. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/, 2009. Accessed: 2025-02-05.

[52] MongoDB, Inc. Mongodb online reference. Accessed: February 10, 2025.

[53] Adel Noureddine. PowerJoular and JoularJX: Multi-Platform Software Power Monitoring Tools. In *18th International Conference on Intelligent Environments (IE)*, Jun 2022.

[54] Mike Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *1999 USENIX Annual Technical Conference (USENIX ATC 99)*, Monterey, CA, June 1999. USENIX Association.

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

[56] OpenStack. Openstack swift documentation. https://docs.openstack.org/swift/latest/. Accessed: 2025-02-02.

[57] Oracle Corporation. Mysql reference manual. https://dev.mysql.com/doc/. Accessed: 2025-02-07.

[58] Oracle Corporation. Oracle database documentation. https://docs.oracle.com/en/database/oracle/oracle-database/. Accessed: 2025-02-07.

[59] Anastasios Papagiannis, Giorgos Saloustros, Giorgos Xanthakis, Giorgos Kalaentzis, Pilar Gonzalez-Ferez, and Angelos Bilas. Kreon: An efficient memory-mapped key-value store for flash storage. *ACM Trans. Storage*, 17(1), January 2021.

[60] PostgreSQL Global Development Group. Postgresql documentation. https://www.postgresql.org/docs/. Accessed: 2025-02-07.

[61] Feitong Qiao, Yiming Fang, and Asaf Cidon. Energy-aware process scheduling in linux. In *Proceedings of the 3rd Workshop on Sustainable Computer Systems (HotCarbon'24)*, page 7, New York, NY, USA, 2024. ACM.

[62] Guillaume Raffin and Denis Trystram. Dissecting the software-based measurement of cpu energy consumption: a comparative analysis, 2024.

[63] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 497–514, New York, NY, USA, 2017. Association for Computing Machinery.

[64] Maria Ramos, João Azevedo, Kyle Kingsbury, José Pereira, Tânia Esteves, Ricardo Macedo, and João Paulo. When amnesia strikes: Understanding and reproducing data loss bugs with fault injection. *Proc. VLDB Endow.*, 17(11):3017–3030, July 2024.

[65] David Reinsel, John Gantz, and John Rydning. Data age 2025: The evolution of data to life-critical. Technical report, International Data Corporation (IDC), 2017.

[66] David Reinsel, John Gantz, and John Rydning. The digitization of the world: From edge to core. Technical report, IDC & Seagate, 2018. Accessed: 2025-02-02.

[67] A. Sharma. Dragon: A distributed graph query engine. https://code.fb.com/data-infrastructure/dragon-a-distributed-graph-query-engine/, 2016. Accessed: 2025-02-03.

[68] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, page 1–10, USA, 2010. IEEE Computer Society.

[69] 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.

[70] Vernier Software & Technology. Watts up pro. https://www.vernier.com/files/manuals/wu-pro.pdf. Accessed: 2025-02-12.

[71] 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.

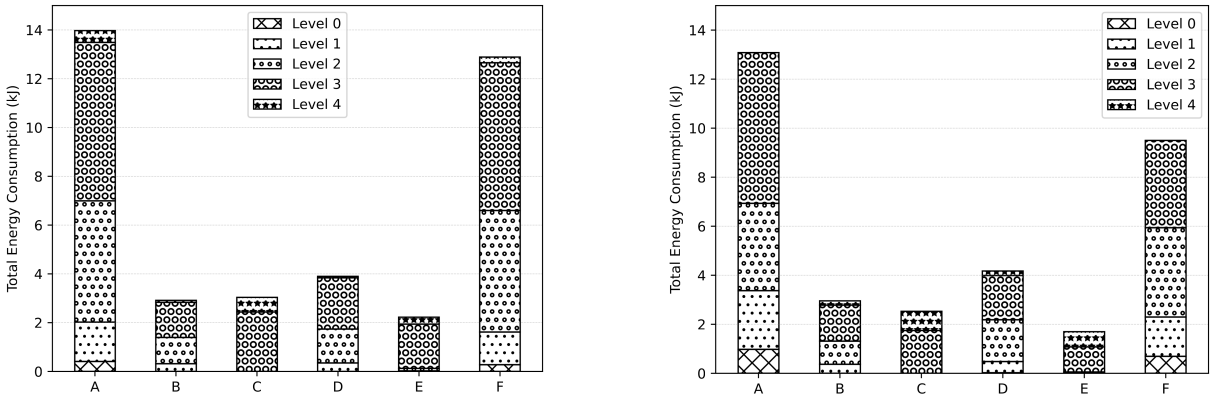
[72] WiredTiger. Wiredtiger. https://github.com/wiredtiger/wiredtiger. Accessed: 2025-02-03.

[73] ZeroMQ Community. Zeromq: Distributed messaging library. https://zeromq.org/. Accessed: 2025-02-18.

[74] Yan Zhai, Xiao Zhang, Stephane Eranian, Lingjia Tang, and Jason Mars. HaPPy: Hyperthread-aware power profiling dynamically. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 211–217, Philadelphia, PA, June 2014. USENIX Association.

[75] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. REMIX: Efficient range query for LSM-trees. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 51–64. USENIX Association, February 2021.

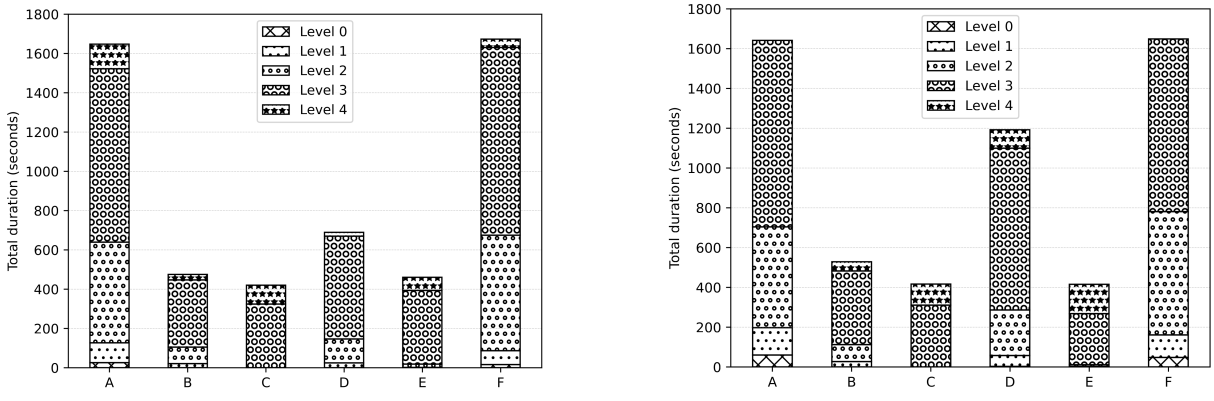
Appendix A
Cumulative Compaction Analysis



(a) Uniform data distribution

(b) Zipfian data distribution

Figure 34: Cumulative compaction energy across the YCSB workloads, under the uniform and Zipfian data distributions after a warmup period of 300 seconds.



(a) Uniform data distribution

(b) Zipfian data distribution

Figure 35: Cumulative compaction duration across the YCSB workloads, under the uniform and Zipfian data distributions after a warmup period of 300 seconds.

Figure 34 presents the cumulative energy consumption across the different **YCSB** workloads, with subfigure 34a focusing on the uniform data distribution and 34b on the Zipfian. Each bar represents the total compaction energy of a **YCSB** workload and is subdivided into stacked segments representing the energy consumption of compaction events made to individual output levels. In the same way as figure

34, figure 35 presents the cumulative duration across different **YCSB** workloads, with subfigures 35a and 35b focusing on the uniform and Zipfian distributions respectively. The energy and duration values from figures 34 and 35 are sourced from the PowMon monitoring system. The energy values are presented in kilojoules, and the duration is shown in seconds.

After the warmup phase, the analysis shows that write-intensive workloads—like A and F—consume more compaction energy than their read-intensive counterparts (B, C, and D) across both data distributions. Under the uniform distribution, scan-intensive workload E registers the lowest compaction energy at 2.2 kJ, while write-intensive workload A tops the chart with 13.97 kJ. Among the read-intensive group, workload D has the highest energy consumption of 3.9 kJ—a difference of 107% relative to workload F. In the Zipfian distribution, workload E again consumes the least at 1.68 kJ, whereas workload A remains the most energy-demanding at 13.08 kJ. Within the read-intensive category in this distribution, workload D again leads with 4.18 kJ, showing a 78% difference compared to workload F.

Regarding the overall duration of compactions, write-intensive workloads consistently register a longer cumulative duration than read-intensive ones across both data distributions. Under the uniform distribution, the read-only workload C recorded the shortest compaction duration, spending 421 seconds in compaction events. Conversely, the write-intensive workload F spent the longest time compacting with 1672 seconds—a small difference of 8% compared to workload A and a difference of 120% relative to the read-only workload. In the Zipfian distribution, write-intensive workloads continue to dominate in compaction time, with workload F again leading at 1649 seconds, while workload A's duration differed by under 1%. Additionally, the scan-intensive workload E and the read-intensive workload C spent the least time compacting, with workload C totalling 416 seconds and differing by under 1% from workload E. Overall, in the Zipfian distribution, the difference between the cumulative compaction durations of workload C and the write-intensive workload F was 119%.