UNIVERSITY OF CALIFORNIA, SAN DIEGO

Solid State Drives for Big Data and Little Clients

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Jing Li

Committee in charge:

Professor Yannis Papakonstantinou, Co-Chair
Professor Steven Swanson, Co-Chair
Professor Alin Deutsch
Professor George Porter
Professor Paul Siegel

2018

The Dissertation of Jing Li is approved and is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____
Co-Chair

_____
Co-Chair

University of California, San Diego

2018

DEDICATION

To my parents and mentors, I could not have done it without you.

EPIGRAPH

Never lose a holy curiosity.

Albert Einstein

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to acknowledge Professor Steven Swanson and Professor Yannis Papakonstantinou for their supports as the co-chairs of my committee. Through multiple drafts and many long nights, their support and guidance have helped me overcome obstacles and achieve the goals throughout my PhD career. I would like to extend my gratitude to Professor Alin Deutsch, Professor George Porter and Professor Paul Siegel for their valuable comments and feedback as my committee members. I would also like to take this chance to express my gratitude to my mentors during the summer internships.

I would also like to acknowledge the help of other members of the Non-volatile Systems Laboratory and the Big Data Lab at UCSD. Especially, I want to acknowledge Hung-Wei Tseng, Yanqin Jin, Yang Liu and Chunbin Lin for their assistance in solving various research obstacles. I would like to thank my officemates, Yanqin Jin and Yuliang Li for those useful discussions and insights.

I sincerely appreciate the help and support from my family. My father has always been the role model for me, who motivates me to pursue higher education and develop a life-long passion for knowledge. My mother, on the other hand, teaches me to be a good person, shares me with her love and optimism, which encourages me a lot during those hard times.

Chapter 1 and Chapter 4 contain materials from "On the Energy Overhead of Mobile Storage Systems", by Jing Li, Anirudh Badam, Ranveer Chandra, Steven Swanson, Bruce Worthington and Qi Zhang, which appears in the 12th USENIX Conference on File and Storage Technologies. The dissertation author was the primary investigator and first author of this paper. The materials are copyright ©2014 by the USENIX Association.

Chapter 1 and Chapter 2 contain materials from "HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics", by Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou and Steven Swanson, which appears in Proceedings of the VLDB Endowment, Vol. 9, No. 14. The dissertation author was the primary investigator and first author of this paper. The materials are copyright ©2016 VLDB Endowment.

Chapter 1 and Chapter 3 contain materials from "SoftFlash", by Jing Li, Jae Young Do, Sudipta Sengupta and Steven Swanson, which will be submitted for publication. The dissertation author is the primary investigator and first author of this paper.

VITA

2012	Bachelor of Science, Peking University, Beijing
2014	Master of Science, University of California, San Diego
2018	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

"HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics" Proceedings of the VLDB Endowment Volume 9 Issue 14, October 2016, Pages 1647-1658 [58]
"On the Energy Overhead of Mobile Storage Systems" Proceeding FAST'14 Proceedings of the 12th USENIX conference on File and Storage Technologies Pages 105-118 [57]
"Summarizer: trading communication with computing near storage". In Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17). ACM, New York, NY, USA, 219-231 [54]
"Gullfoss: Accelerating and simplifying data movement among heterogeneous computing and storage resources"Tech Report, Department of Computer Science and Engineering, University of California, San Diego [91]
"Hippogriff: Efficiently moving data in heterogeneous computing systems" 2016 IEEE 34th International Conference on Computer Design (ICCD), Pages 376-379 [59]
"HAT: an efficient buffer management method for flash-based hybrid storage systems" Frontiers of Computer Science, Volume 8, Issue 3, Pages 440-455 [61]
"Hotness-aware buffer management for flash-based hybrid storage systems" Proceedings of the 22nd ACM international conference on Conference on information and knowledge management, Pages 1631-1636 [60]
"Challenging the long tail recommendation" Proceedings of the VLDB Endowment, Volume 5, Issue 9, Pages 896-907 [101]
"Log-compact R-tree: an efficient spatial index for SSD" International Conference on Database Systems for Advanced Applications 2011, Pages 202-213 [62]

ABSTRACT OF THE DISSERTATION

Solid State Drives for Big Data and Little Clients

by

Jing Li

Doctor of Philosophy in Computer Science

University of California, San Diego, 2018

Professor Yannis Papakonstantinou, Co-Chair
Professor Steven Swanson, Co-Chair

Big data analytics open challenges for efficiently processing, moving and storing data. Existing research works focus on algorithm design or applying hardware accelerators. However, in current systems, data transfer (from secondary storage or remote nodes) becomes an increasingly important but less-optimized performance bottleneck.

This thesis first presents HippogriffDB, a datawarehouse system that delivers efficient, scalable analytical performance with GPU and SSD. HippogriffDB achieves high efficiency by reconciling the bandwidth mismatch between GPU and IO with the improved data transfer mechanism and data compression strategies. Experiment results demonstrate that HippogriffDB

outperforms state-of-the-art GPU-based databases by up to 10×.

The thesis then presents SoftFlash. SoftFlash offloads some essential database operations from modern data warehouse applications into storage devices. By using hardware accelerator, on-chip processors, as well as software approaches, SoftFlash manages to reduce data traffic over interconnect and to improve the query execution time.

This thesis also covers experimental studies on the energy prospective of flash memory in the context of mobile computing. While the flash hardware is well known to be energy and power efficient, the software stack consumes significantly higher amount of power and energy.

Chapter 1

Introduction

The era of big data opens multiple challenges for applications to efficiently store and process data. Due to the high performance, low power consumption and reasonable cost, flash-based storage solutions act as the primary choice for data persistence in a variety of applications, ranging from big data platforms, such as data centers, to little clients, such as smartphones and tablets.

However, there are three challenges for designing computer systems powered by flash-based storage.

The first challenge is the bandwidth mismatch between different components inside computing systems. For a system accelerated with heterogenous approaches, flash memory may fail to feed in data as fast as required by the hardware accelerators, making the system heavily IO- bottlenecked. For example, running queries in the Star Schema Benchmark[71] on an NVidia Telsa K20 GPU can reach the throughout of up to 60 GB/sec; however, a high-end NVMe SSD can only deliver 2 GB/sec read bandwidth, more than one order of magnitude lower than the internal throughput inside hardware accelerators.

Second, there are multiple computation resources in the system that are under-utilized. Modern flash-based SSDs usually equip with certain amount of computation power inside it. However, application or firmware only use a small portion of the total computation resources, making those in-device computing power severely under-utilized. For systems accelerated with

GPUs or ASICs, the relatively slow storage prevents those accelerators from running in full speed, resulting in large number of idle cycles inside those accelerators.

Last but not least, while flash hardware is rather energy efficient, the entire storage stack is energy intensive. Our research shows that the software stack can take up to 200× more energy than the hardware components.

This thesis presents three works that tackle the three issues above.

1.1 HippogriffDB

As power scaling trends prevent CPUs from providing scalable performance [43, 36, 28], database designers are looking to alternate computing devices for large scale data analytics, as opposed to conventional CPU-centric approaches. Among them, Graphics Processing Units (GPUs) attract the most discussion for the massive parallelism, commercial availability, and full-blown programmability. Previous work [103, 94, 24, 47] proved the feasibility of accelerating databases using GPUs. Experiments show that GPUs can accelerate analytical queries by up to 27× [103, 42, 44].

However, existing GPU-accelerated database systems suffer from size limitations: they require the working set to fit in GPU's device memory. With this limitation, existing GPUs cannot handle terabyte-scale databases that are becoming common [15, 49].

Scaling up GPU-accelerated database systems to accommodate data sets larger than GPU memory capacity is challenging:

1. **The low bandwidth of data transfer in heterogenous system counteracts the benefit GPU accelerators provide.** While main memory has always been fast (up to 8 GB/sec when transferring to a K20 GPU) and new storage devices like solid state drives (SSDs) are improving performance (up to 2.4 GB/sec [1]), the bandwidth demand of GPU database operators is still higher than the interconnect bandwidth and storage bandwidth. As shown in Table 4.1, typical database operators and queries can run $29-82\times$ faster than

the SSD read bandwidth. Without careful design, the slow storage would under-utilize high-performance GPU accelerators.

2. **Moving data between storage devices and multiple computing devices adds overhead.** The data transfer mechanism in existing systems can be both slow and costly. It single-threadedly moves data from the data source to the GPU via CPU and the main memory, failing to utilize the internal papalism inside modern SSDs. This also adds indirections and consumes precious CPU and memory resources, which the system could use for more useful tasks. Recent work [94] shows that this detour can take over 80% of the execution time in typical analytical workloads and can cause the transfer bandwidth less than 40% of the theoretical peak.

3. **Current execution models of GPU-databases do not suit the architecture of GPUs and cause scalability and performance issues.** The query execution models in existing GPU databases [99, 47] are neither efficient nor scalable. They require that the working set fit in the small GPU device memory (usually less than 20 GB [2]). Besides, the intermediate results that the current models produce put pressure on the already scarce GPU memory.

To address the above challenges, we propose HippogriffDB, an efficient, scalable heterogeneous data analytics. The primary issue HippogriffDB tackles is the low performance caused by the bandwidth mismatch between fast computation and slow I/O. HippogriffDB fixes it with compression and optimized data transfer mechanisms. The stream-based execution model it adopts makes HippogriffDB the first GPU-based database system that supports big data queries. The model uses operator fusion mechanism to aggressively eliminate the intermediate results since the penalty of cache misses is costly on the GPU.

HippogriffDB stores the database in a compressed format and decompresses them on the GPU, trading GPU computation cycles for improved bandwidth. It utilizes the massive computation power of the GPU to decompress data, turning the bandwidth gap into improved bandwidth.

HippogriffDB tailors the compression methods to fit better into the GPU architecture. It supports combination of multiple compression methods to boost the effective bandwidth for data transfers. HippogriffDB employs a decision model to select the appropriate compression combination that balances the GPU kernel throughput and I/O bandwidth. We prove the *compression selection* to be an NP-hard problem and propose a 2-approximation greedy algorithm for it. For storage with massive capacity, HippogriffDB adopts adaptive compression: it maintains multiple compressed versions and then choose the best compression schema dynamically so that a wide range of queries can benefit from the compression.

HippogriffDB tackles the low data transfer bandwidth by using mulithreaded, peer-to-peer communication mechanism (DirectNVMe) between the data source (e.g. SSD and NIC) and the GPU. There are two problems in the I/O mechanism of the existing GPU-based analytical system [94]: (1) data transfer relies on CPU/memory to forward the input (2) the single-threaded model under-utilizes the multiple data transfer hardware inside the SSD. To address these two problems, HippogriffDB reengineers the software stack so that the data transfer can be direct from the SSD to the GPU. It also adopts multi-threaded data fetching to take the advantage of the massive parallelism inside modern SSDs.

HippogriffDB achieves high scalability, high kernel efficiency and low memory footprint by adopting a new execution model in GPU processing, called *query-over-block*. It contains two parts. First, HippogriffDB streams input in small blocks, leading to high scalability and low memory footprint. HippogriffDB adopts double buffering to support asynchronous data transfer. Second, the query-over-block model reduces the intermediate results using an *operator fusion* mechanism: it fuses multiple operators into one, turning the intermediate results passing into the local variables passing inside each GPU thread.

We implemented HippogriffDB on a heterogeneous computer system with an NVIDIA K20 and a high-speed NVMe SSD. As an initial look of HippogriffDB design, we focus on star schema queries. We compare it with a state-of-the-art CPU-based analytical database (MonetDB [23]) and a state-of-the-art GPU-based analytical database (YDB [103]), using two popular

benchmarks (the Star Schema Benchmark [71] and the Berkeley Big Data Benchmark [76]). HippogriffDB outperforms MonetDB by up to 147× and YDB by up to 10×. Results also show that HippogriffDB can scale up to support terabyte-scale databases and our optimizations can help achieve up to 8× performance improvement overall.

HippogriffDB makes the following contributions:

1. HippogriffDB improves the performance of GPU-based data analytics by fixing the bandwidth mismatch between fast GPU and slow I/O, using adaptive compression.
2. HippogriffDB improves the data transfer bandwidth and resource utilization by implementing a peer-to-peer datapath that eliminates redundant data movements in heterogeneous computing systems.
3. We identify the compression choosing problem to be an NP-hard problem and provide a 2-approximation greedy algorithm for it.
4. HippogriffDB uses the operator fusion mechanism to avoid intermediate results and to improve kernel efficiency.
5. HippogriffDB uses query-over-block, a streaming query execution model, to provide native support for big data analytics.
6. HippogriffDB outperforms state-of-the-art data analytics systems by 1-2 orders of magnitude, and experiment results demonstrate HippogriffDB's scalability.

1.2 SoftFlash

In recent years, software-defined control and management have become an inevitable trend that is already touching many parts of the data center infrastructure. As the most recent example of the software-defined paradigm, in the networking space, a new type of agility that redefines and manages networking for enhanced functionality like more granular traffic management, security, and deep network telemetry to enable network performance optimization is

available by making network switches and server network interface cards (NICs) programmable. We have also seen a similar trend of programmability evolution with graphic cards. Graphics processing units (GPUs) were initially born out of the needs of multimedia applications, and thereby they shipped with hard-coded multimedia functionality. But today they became fully programable, general-purpose GPUs being currently leveraged by new generations of applications like deep learning. Like this, without needing physical, hardware changes, rapidly-changing requirements in the cloud environment can be supported on-the-fly once infrastructures become dynamically programmable.

Unfortunately, the lack of such programmable capabilities in storage results in a major disconnect in terms of the speed of innovation between application/operating system (OS) infrastructure and storage infrastructure. In cloud, application/OS software is patched with new/improved functionality every few weeks at cloud speed while storage devices are off limits for such sustained innovation during their hardware life cycle of 3-5 years in the data center. Since the software inside the storage device is written by storage vendors as proprietary firmware that is not open for general application developers to modify, we are stuck with a device whose functionality and capabilities are frozen in time, even when many of these are modifiable in software. A period of 5 years is almost eternal in the cloud computing industry, where new features, platforms, and application program interfaces (APIs) are evolving in order of months, and application-demanded requirements from the storage system grow quickly over time. This huge lag in the adaptability and velocity of movement of the storage infrastructure may ultimately impact the ability to innovate in the whole cloud world.

In this project, we make the argument that a fully programmable storage substrate gives opportunities to better bridge the gap between application/OS needs and storage capabilities/limitations, while allowing us to innovate in-house at cloud speed.

Moving compute close to data, the need to analyze and glean intelligence from big data is imposing a shift from the traditional compute-centric model to data-centric model. In many big data scenarios, application performance and responsiveness (demanded by interactive

usage) is dominated not by the execution of arithmetic and logic instructions but instead by the requirement to handle huge volumes of data and the cost of moving this data to the location(s) where compute is performed. When this is the case, moving the compute closer to the data can reap huge benefits in terms of increased throughput, lower latency, and reduced energy usage.

Big data analytics, for example, running inside the SSD can have access to the stored data with tens of GB/sec bandwidth (rivaling DRAM bandwidth), and with latency close to that of accessing raw non-volatile memory. In addition, large energy savings can be achieved because 1) processors inside the SSD is more energy efficient compared to the host server CPU like Intel Xeon, and 2) data does not need to be hauled over large distances from storage all the way up to the host via network, which is more energy expensive than processing it.

Processors inside the SSD are clearly not as compute capable as expensive host processors, but together with in-storage hardware offload engines, a broad range of data processing tasks can be competitively performed inside the SSD. As an example, consider how data analytic queries are processed in general: When an analytic query is given, compressed data that is required to answer the query is first loaded to host and then needs to be decompressed by using host resources. Such fundamental data analytics primitive can be processed inside the SSD by accessing data with high internal bandwidth and by offloading decompression to the dedicated engine. Subsequently stages of the query processing pipeline (such as filtering unnecessary data and doing the aggregation) can also be executed inside the SSD, resulting in greatly reduced network traffics and in saved host CPU/memory resources for other important jobs. This is an opportunity to rethink the data center architecture in the data-centric model with the efficient use of heterogeneous, energy-efficient hardware, which is the way forward for higher performance at lower power.

1.3 Storage energy

NAND-Flash in the form of secure digital cards (SD cards) [83] and embedded multi-media cards (eMMC) [33] is the choice of storage hardware for almost all mobile phones and tablets. SD cards and eMMC consume less energy and provide significantly less performance when compared to solid state disks (SSD). Such a trade-off is acceptable for battery powered hand-held devices like phones and tablets, which run at most one user-facing app at a time and therefore do not require SSD-level performance.

SD cards and eMMC technologies deliver adequate performance while consuming low energy. For example, eMMC 4.5 [81] prototypes promise to deliver up to 3500 random read, and 1500 random write IOPS. Additionally, they promise to deliver up to 150 MBps sequential read and 60 MBps sequential write bandwidth. While the sequential bandwidth is comparable to that of a single platter 5400 RPM magnetic disk, random IOPS performance is an order of magnitude higher than a 15000 RPM magnetic disk. To deliver this performance, eMMC consumes less than 250 milliwatts (Section 4.1) of peak power.

Storage software on mobile platforms, unfortunately, is not well equipped to exploit these low-energy characteristics of mobile-storage hardware. In this paper, we shed light on the energy cost of storage software on popular mobile platforms. The storage software stack consumes as much as 20 times more energy when compared to storage hardware for popular mobile platforms like Android and Windows RT.

We believe that most developers design their applications under the assumption that storage systems on mobile platforms are not energy-intensive. However, experimental results demonstrate the contrary. To help developers, we built a model for energy consumption of storage systems on mobile platforms. Developers can leverage such a model to optimize the energy consumption of storage-intensive mobile apps'.

In addition, we present a detailed breakdown of the energy consumption of various software and hardware components inside the storage subsystem by analyzing data from fine-grained

performance and energy profilers. In summary, this paper makes the following contributions:

1. We analyze the energy consumption of storage systems on Android and Windows RT. We are not aware of any prior work that presents a full view of the energy consumed by storage systems on mobile platforms – including hardware and software.

2. We present a model by which developers can estimate the amount of energy consumed by storage systems used by their applications. The model can help reduce the energy consumed by applications, especially the storage-intensive ones that execute when the screen is off.

3. We propose optimizations to mobile storage software to reduce their energy consumption.

Chapter 2

HippogriffDB

2.1 System overview

HippogriffDB uses data compression and optimized data movement, combined with a stream-based query execution model, to deliver an efficient, scalable system. To prevent slow storage devices from counteracting the speedup that GPUs provide, HippogriffDB redesigns the data transfer path and stores the database in a compressed way, trading GPU cycles for improved I/O bandwidth. It uses a stream-based query processing model to support one-pass big data analytics.

This section provides an overview of the system design, optimized data store and transfer mechanism, and the query-over-block model to support scalable, efficient big data analytic.

2.1.1 System architecture

HippogriffDB targets large databases and stores the database tables in main memory or secondary storage device in the current implementation. It contains three major components.

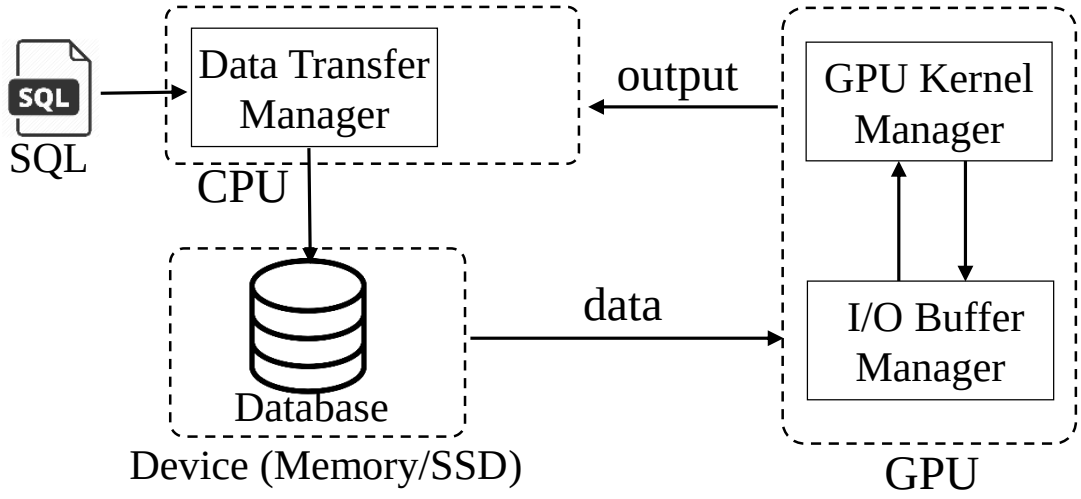


Figure 2.1. System architecture of HippogriffDB.

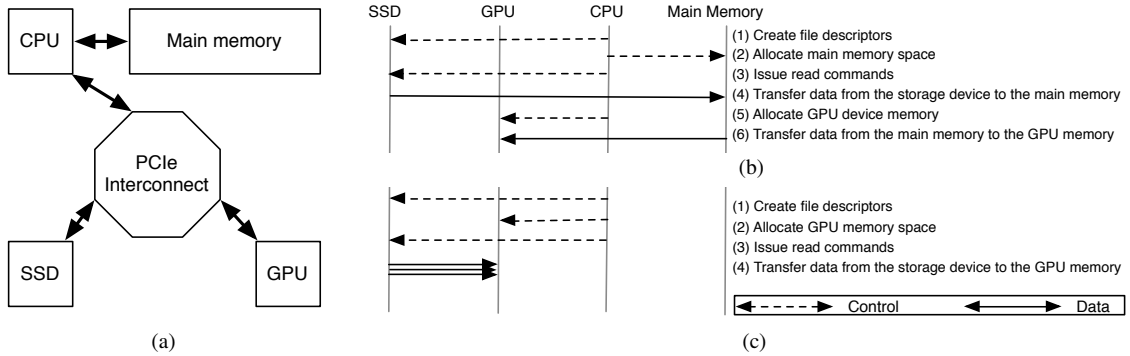


Figure 2.2. (a) The conventional heterogenous platform, (b) the process of moving data between the GPU and the SSD in existing systems, and (c) direct data access in HippogriffDB.

Data Transfer Manager. The data transfer manager moves requested data from the main memory/SSD to the GPU kernel. To support unlimited table input, HippogriffDB transfers and processes data in chunks. Upon receiving a request, it first finds out the position of the requested data and then send data to the GPU input buffer. It adopts a multi-threaded, peer-to-peer communication mechanism between the GPU and the SSD to further improve data transfer bandwidth. In the current implementation of HippogriffDB, database tables reside in the main memory or an NVM Express (NVMe) SSD. To save bandwidth, HippogriffDB stores the database in a compressed, column-store format.

I/O Buffer Manager. HippogriffDB maintains a circular input buffer in the GPU

memory. It works with the *Data Transfer Manager* to overlap data transfer and query processing: an I/O thread and the GPU kernel act as the producer and consumer respectively to copy data from storage to the GPU. HippogriffDB also maintains a result buffer in GPU's memory. If the result can fit in the GPU memory, HippogriffDB will buffer the intermediate results in the GPU until the whole process finishes. Otherwise, HippogriffDB will keep a similar circular buffer for the output.

GPU Kernel Manager. The GPU kernel receives data from the input buffer and evaluates queries on the received data. HippogriffDB supports queries that consist of selection, join, aggregation, and sort operators. The kernel manager implements fused operators that cover the functionality of multiple individual operators to support the query-over-block model HippogriffDB adopts.

Figure 2.2 shows how HippogriffDB processes a query. The *Data Transfer Manager* prepares the relevant columns for a given query by retrieving data from either the main memory or secondary storage devices like SSDs. It then works with the *I/O Buffer Manager* to send relevant columns from the main memory or the SSD to the input buffer in the GPU memory. The *GPU Kernel Manager* then evaluates the query. When query evaluation finishes, the *GPU Kernel Manager* will send the result back to the output buffer.

There is a huge bandwidth mismatch between the GPU kernel and I/O. Without careful design, the slow I/O transfer will counteract the speedup that hardware accelerators provide. HippogriffDB alleviates the mismatch by storing data in a compressed form and using GPU cycles for decompression. To further improve physical I/O bandwidth, HippogriffDB removes the redundant data transfers by implementing a peer-to-peer communication from storage to the GPU. We provide an overview for them in Section 2.1.2.

We also notice the inefficiency and the scalability limitation of current query execution models. HippogriffDB fixes it by introducing a new query model in GPU processing. We provide an overview of it in Section 2.1.3.

2.1.2 Optimizing data movement

The primary obstacle for building an efficient, GPU-based big data analytics is the enormous bandwidth gap between data transfer and the GPU kernel, which can counteract the speedup GPUs provide. HippogriffDB addresses the imbalance that arises between the GPU kernel and I/O by optimizing the data path and trading GPU computation cycles for improved bandwidth.

With modern system setups, the throughput of the GPU kernel (the amount of data the GPU can process per unit time) is $6-12\times$ higher than the I/O bandwidth (the amount of data transferred from storage per unit time) from the main memory to the GPU. The discrepancy between SSD I/O bandwidth and GPU kernel throughput is ever larger. To close the gap between GPU computation throughput and data transfer bandwidth, HippogriffDB applies two optimizations to improve the effective bandwidth of data transfer:

1. It adopts a multi-threaded, peer-to-peer communication mechanism (DirectNVMe) over the PCIe interconnect to remove the data forwarding, improving the bandwidth when moving data from the SSD to the GPU.
2. It compresses tables and uses the GPU to decompress them, effectively converting GPU compute capabilities into improved transfer bandwidth.

DirectNVMe

In conventional heterogenous systems where system components communicate through PCIe (as shown in Figure 2.2(a)), moving data between storage and computation accelerators is complex and inefficient. Figure 2.2(b) illustrates the data transfer path in existing GPU-accelerated systems: they first move data to the main memory (step 1-4) and then copy them to the GPU (step 5-6). This adds significant overhead for applications: applications can spend more than 80% of time simply in moving and copying data from the main memory to the GPU memory [93].

The data transfer mechanism existing heterogenous systems adopt fails to take advantage of the massive internal parallelism inside modern SSDs. Modern SSDs contains multiple data transfer manager inside it to boost the I/O bandwidth and each data transfer manager corresponds to an NVMe queue on the host. Hence, using single-threaded programming interface prevents the SSD from supplying as much data as it could be to accommodate the requirement from high-performance processing units. Our test shows that the bandwidth in YDB [93] only achieve 30% of the peak bandwidth.

To mitigate the cost of moving data, HippogriffDB implements a multi-threaded, peer-to-peer communication mechanism (DirectNVMe) between the SSD and the GPU. Figure 2.2(c) illustrates how HippogriffDB avoids unnecessary copies to and from main memory. After the HippogriffDB API initiates a file transfer and obtains a file descriptor from the operating system (Step (1)), it requests a region in the GPU memory (Step (2)). In Step (3), HippogriffDB issues NVMe read commands that include the GPU memory addresses to the SSD. Finally in Step (4), the SSD pushes data directly from the SSD to the GPU using DMA, without any CPU involvement.

Using a peer-to-peer communication mechanism between two devices brings several benefits to the system. First, it improves the I/O bandwidth as it reduces the traffic on the system interconnects and CPU-memory bus. Second, it frees up the CPU and memory resources which can serve other useful applications. Last, the emancipation of CPU and memory from heavy I/O demands saves power and energy. To further improve data transfer bandwidth, HippogriffDB issues multiple peer-to-peer transfers in parallel to boost performance.

Column-based, compressed tables

To surpass the physical bandwidth limit that the interconnect and the devices set, HippogriffDB stores database tables using a column-based, compressed format and trades GPU decompression cycles for improved effective bandwidth.

HippogriffDB follows the modern column store designs by using implicit virtual-ids [13],

as opposed to explicit record-ids, to avoid bloating the size of data storage. Column-based format provides more opportunity for compression, which further improves the I/O bandwidth [13, 23].

HippogriffDB stores tables in a compressed format and uses GPU idle cycles for decompression to further improve the effective I/O bandwidth. HippogriffDB allows both light-weighted compression and heavy-weighted compression. It also supports compression methods for columns with variable-length.

There is a tradeoff between compression aggressivity and GPU efficiency. Aggressive compressions like Huffman [13] can help reach better compression ratio but also brings more cost to the GPU decompression stage and slows down the GPU accelerators. HippogriffDB adopts a cost-benefit model to evaluate the trade-off and to choose appropriate strategies, using the system characteristics. We identify the compression selection problem to be an NP-hard problem and propose a 2-approximation greedy algorithm. Based on that, we introduce the data-centric compression strategy, which minimize (nearly) the overall space cost.

HippogriffDB observes the limitation of maintaining only one compression plan and hence allows multiple, adaptive compressions when possible. One compression plan may benefit certain kinds of queries but works poorly on others. HippogriffDB fixes this issue by keeping multiple compression schema and choosing the optimal one among all available versions dynamically, trading space for better system throughput.

2.1.3 Query-Over-Block overview

The query-over-block execution model enables the system to efficiently scale beyond the confines of the memory capacity. It contains two aspects: first, it processes inputs as streams and uses double buffering to support asynchronous execution (block-oriented execution); second, it packs multiple operators into one and sends intermediate results via thread-local variables (operator fusion mechanism).

Example to demonstrate existing models and query-over-block. Query 1 (Figure 2.3(a)) compares the revenue for some products that certain manufacturer makes and whose quantity

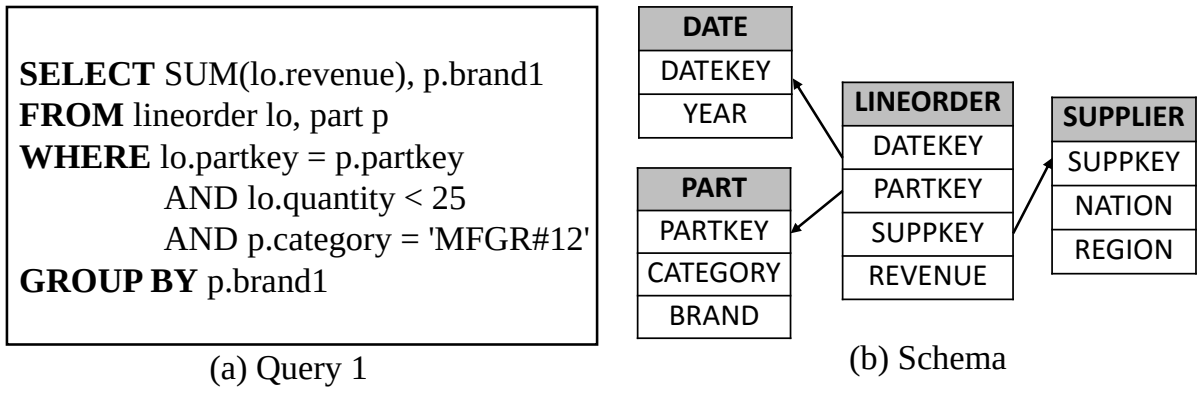


Figure 2.3. Query 1 and its corresponding schema.

is less than 25, grouped by the product brands. Figure 2.3(b) show the corresponding database schema. The fact table `lineorder` refers to three dimension tables `date`, `part` and `supplier`. We show the query execution plan `YDB` [103] generates in Figure 2.4. The existing model that YDB uses , "operator at a time and bulk execution"[23], evaluates each operation (e.g. selection of `quantity` $<$ 25) to completion over its entire input (e.g. `lo_quantity`) and sends the whole intermediate results to the upcoming operator (e.g. `lineorder` $\bowtie$ `part`).

The query-over-block model generates three operators for Query 1, as shown in the dashed, rectangular box in Figure 2.4. Two of them are the new selection operators whose functionalities include traditional selection (e.g. `quantity` $<$ 25), projection. The other one is the new join operator which covers the functionalities of the join in the original plan (e.g. `lineorder` $\bowtie$ `part`) and the aggregation operation (e.g. $\gamma_{\texttt{p.brand1,sum(lo.revenue)}}$).

Comparison of the existing model and query-over-block. YDB's query plan is neither scalable nor efficient for two reasons. First, this plan evaluates each operation to completion over its entire input and sends the results of the previous operator (i.e. `filter_vector`) to the next operator [23]. In this way, the plan generates large intermediate results, that cost precious memory resources and adds latency, since accessing global memory on GPU is slow. Second, it requires that all relevant database columns must fit into GPU memory [103], which limits the system scalability.

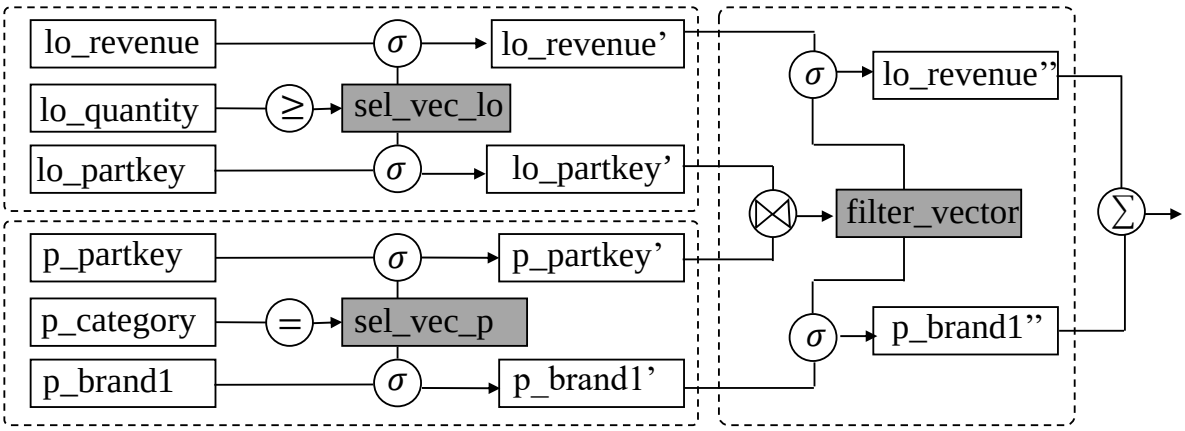


Figure 2.4. Schema of Query 1 generated by YDB. It creates large intermediate results (grey boxes). It also requires all relevant columns in GPU memory, limiting the scalability of the system.

The query-over-block model addresses the limitations by fusing multiple operations into one and providing stream-based operators. Combining multiple operators allows the system to avoid materialization of intermediate results and pass the data more efficiently using fast local memory. The query-over-block model can process input blocks in a streaming approach. In this way, this model supports data sets larger than the GPU memory capacity and hence improves system scalability.

Compared with the existing query processing model used on GPU, this model has several advantages:

1. It enables query processing on partial input, as opposed to the whole table, which has great significance on increase the scalability of query processing.
2. It eliminates redundant and unnecessary intermediate results and mollify the memory pressure on GPU.

2.2 Data compression

HippogriffDB alleviates the bandwidth mismatch by using data compression and trading GPU cycles for improved bandwidth. The massive parallelism inside GPU can produce result

in a throughput that is one order of magnitude higher than the data transfer bandwidth memory or SSD can deliver, creating an imbalance among different components inside a GPU-based analytics system. HippogriffDB closes the gap using compression: it compresses database tables and uses the GPU to decompress them, effectively converting GPU compute capabilities into improved transfer bandwidth. In this way, HippogriffDB improves the I/O bandwidth efficiency and increases the system throughput.

In this section, we first introduce the compression methods that HippogriffDB adopts and then analyze the compression ratio of them. After that, we discuss how to efficiently combine different compression methods to generate a compression plan.

2.2.1 GPU-based compression methods

HippogriffDB supports several compression methods to maximize the effective bandwidth between the GPU and the storage device. In this subsection, we introduce those compression methods and analyze the compression ratio they can achieve.

HippogriffDB carefully chooses compression methods based on two criteria: (1) the decompression algorithm fits with the GPU's vectorization nature. (2) compression is compatible with the execution model. We also make a few changes to the existing compression methods so that GPU can effectively decompress them.

The compression methods that HippogriffDB uses are:

- RLE (run-length encoding): Run-length encoding represents *runs* of data (consecutive numbers of the same value) as a single data value and count [89]. Similar to [87], HippogriffDB can evaluate the columns encoded by RLE directly without decompression cost.
- DICT: DICT (dictionary encoding) replaces the data with its corresponding representation contained in a mapping table. HippogriffDB decompresses data by searching the mapping or calculating the translation function.

- Huffman [48]: HippogriffDB decompresses data on the GPU using a method similar to [73], but HippogriffDB uses a hashtable to replace the Huffman tree on GPU to avoid pointer swizzling. We add padding characters when encoding the data to avoid random access when decompressing.
- String encoding: HippogriffDB supports compression on string columns where the records may have different lengths. HippogriffDB stores those elements consecutively and keeps another array recording the length of each element. HippogriffDB decodes it by calculating the prefix sum of the length array and then getting the start point and the end point (that is, the start point of the next element minus one) of each record.
- DELTA: Delta encoding stores data using differences (deltas) between sequential data, instead of complete files. HippogriffDB uses similar ways introduced in [37] to decompress data.

When we decide the compression schemes, we follow two criteria: (1) it works with the query model that we use to provide scalable analytical system (2) it suits the architecture of GPU environment. To work with the streaming-based model, we only allow one column to be sorted, and for it, we apply RLE as it leads to very high compression ratio. As we are not able to introduce a second primary sort key, we decide to choose a secondary sort key. For the column chosen as the secondary sort key, there is not much opportunity for the data clustering. Hence, RLE would not help a lot here. We take advantage of the orderliness and use the delta between two consecutive elements, as opposed to the original value, to encode the column. For other columns, we evaluate the domain size and the distribution of the column and then choose adequate compression schemes for them.

To sum up, HippogriffDB compresses a table in a heuristic strategy: it first sorts the table using two sort keys (one primary and one secondary sort key). It then applies *RLE* on the primary sort key column and *DELTA* on the secondary sort key column. For other columns, it applies *DICT* or *Huffman* if possible.

QUERY 2

```
SELECT SUM(lo.revenue), d.year, p.brand1
FROM lineorder lo, date d, part p, supplier s
WHERE lo.orderdate = d.datekey AND lo.partkey = p.partkey
        AND lo.suppkey = s.suppkey  AND lo.quantity < 25
        AND p.category = 'MFGR#12'  AND s.region = 'AMERICA'
GROUP BY d.year, p.brand1
```

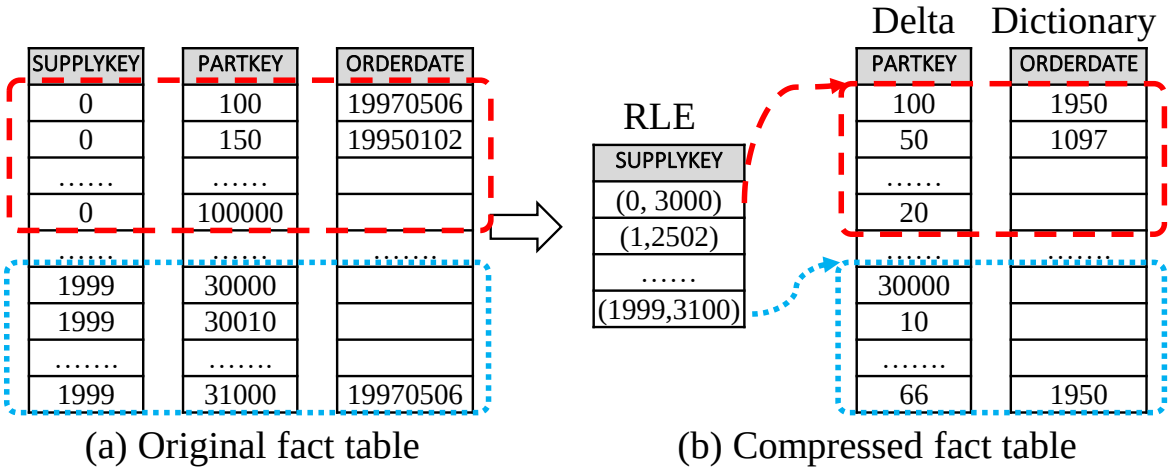


Figure 2.5. Example of a compression plan. *RLE* applies to the primary sort key column (`SUPPLEYKEY`) and *DELTA* applies to secondary sort key column (`PARTKEY`). The `ORDERDATE` column uses *DICT* due to its limited number of distinct values.

Figure 2.5 illustrates how HippogriffDB compresses columns using the method discussed above. We use Query 2 as an example here (Figure 2.3(b) shows the corresponding schema). Query 2 is an extension for Query 1: it compares revenue for product classes, for suppliers in a certain region, grouped by the products brand and year. In the example, `SUPPLYKEY` and `PARTKEY` work as the primary and secondary sort key respectively. We encode them using *RLE* and *DELTA*. HippogriffDB will evaluate the other columns and use *DICT* or *Huffman* on them if possible (`ORDERDATE` in the example).

To compare the quality of different compression methods, we provide the compression

Table 2.1. Compression ratio of different methods.

Method	compression ratio
RLE	$2 \times a/m$
DICT	$\log_2 d/4$
Huffman	$[E_n, E_n + 1]$
String	$(avg_l + 4)/ml$
DELTA	$\sum_i i \cdot P(\texttt{Beta}(1, \frac{m}{a}) < b * 2^{7i})$

ratio of different methods in Table 2.1 [1].

2.2.2 Data-centric compression plan

HippogriffDB compresses tables using a data-centric optimal plan to minimize (nearly) the overall space cost and reduce the amount of data transfer from storage to the GPU. HippogriffDB can apply different compression methods to different columns and we call each combination a compression plan. Different plans result in different compression ratios, and some of them are not suitable for the query-over-block model or are too complicated for the GPU to decompress. In this section, we first define balanced compression plans and then discuss the algorithm to generate data-centric optimal compression plans.

Balanced compression plan

HippogriffDB requires that compression plans be compatible with the query-over-block model and that do not overburden the GPU. We call a plan with such properties a *balanced compression plan*. In this subsection, we study the conditions for a plan to be balanced and introduce a way to convert an imbalanced compression plan to a balanced one.

As HippogriffDB adopts implicit row-id, it can only sort one column at most. The streaming query model requires all columns to keep the same order so that HippogriffDB can successfully reconstruct a record. As a result, a compatible compression plan can have at most one sort key. It is possible to have a secondary sort key besides the primary one.

[1] In the following analysis, we assume that columns are independent and they have uniform distribution.

HippogriffDB faces tradeoff between better compression ratio and the cost of slowing down GPU kernels due to decompression. The aggressivity of compression methods that HippogriffDB should use depends on the discrepancy between computation kernel and data transfer path. We formulate the cost-benefit analysis below. The cost here is GPU decompression cycles and the benefit is reducing the amount of data transfer.

Let T_G denote the GPU kernel time, r_x denote the compression ratio of compression method x (the corresponding column size is C_x), D_x denote the decompression bandwidth of it. Suppose the data transfer rate is B_{IO}. Say we use n compression methods and suppose $\forall i < j, C_i * r_i/D_i <= C_j * r_j/D_j$. The time to transfer compressed data is:

$$(\sum_{i=1}^{n} C_i * r_i)/B_{IO}$$

The time to decompress and run the query is:

$$T_G + \sum_{i=1}^{n} (C_i * r_i)/D_i$$

To prevent GPU from running slower than I/O, we require that:

$$(\sum_{i=1}^{n} C_i * r_i)/B_{IO} >= T_G + \sum_{i=1}^{n} (C_i * r_i)/D_i$$

for a compression plan to be a balanced one.

The problem of selecting the optimal compression combination is an **NP-hard** problem, which can be reduced from the 0-1 Knapsack problem [63]. Similar to the greedy algorithm for the 0-1 Knapsack problem, we propose a **2-approximation** greedy algorithm, which has two steps. First, we sort columns in non-decreasing order of $(C_i * r_i)/D_i$. Second, greedily pick columns in above order. Due to the space limitation, we omit the proof for the NP-hardness and approximation.

Data-centric compression

HippogriffDB explores all balanced compression combinations and chooses the one with the minimal (nearly) space cost to reduce the data transfer amount. In this subsection, we introduce an algorithm to pick up a balanced plan that comes with the minimal (nearly) space cost, called *data-centric compression*.

There are multiple ways to compress a database table and Algorithm 1 provides a strategy to find the data-centric compression. Choosing different primary and secondary sort key results different compression strategies. HippogriffDB explores all possible combinations and calculates the compression ratio each strategy can reach (Line 1 - 11). Not all compression combinations are favorable, as some compressions overburden the GPU in the decompression stage. We use the cost-benefit model (Section 4.4) to adjust the compression strategies (Line 13). If we are able to find a better compression plan, we save this plan as the up-to-date best plan and update the best compression ratio (Line 14 - 16).

2.3 SSD-specific Optimizations

HippogriffDB improves the data transfer from the relative slow SSD by allowing direct data transfer from the SSD to the GPU and using a query-adaptive compression to further improve compression ratio, trading storage space for improved system throughput. While new storage devices are improving performance, the read bandwidth still falls behind the GPU kernel throughput by at least one order of magnitude, creating a big imbalance among different components inside the system. We present two mechanisms to mitigate the bandwidth mismatch. One is to provide a more efficient data transfer mechanism between storage and the GPU. The other is using adaptive compression to improve the effective transfer bandwidth over PCIe.

In the following sections, we first introduce the new data path HippogriffDB adopts. We then illustrate the inefficiency of the fixed compression strategy and based on such observation, introduce an “adaptive” compression strategy to extend the benefit of compression for a wide-

Input: A table with several columns
Output: The data-centric optimal compression plan
$d_i \leftarrow$ compression ratio of *DICT* on column i ;
$S \leftarrow 0$;
for $i \leftarrow 1$ ***to*** n **do**
  $S \leftarrow S + d_i$;
end
$minratio \leftarrow 1$;
for $i \leftarrow 1$ ***to*** n **do**
  // **for** $j \leftarrow i+1$ ***to*** n **do**
    //column i as the primary sort key and column j as secondary sort key ;
    $rle_i \leftarrow$compression ratio of *RLE* ;
    $delta_{ij} \leftarrow$compression ratio of *DELTA* ;
    $cr \leftarrow rle_i + delta_{ij} + S - d_i - d_j$;
    // calculate the current compression ratio
    cost-benefit verification to adjust the plan to a balanced plan and update cr ;
    if $minratio > cr$ **then**
      $minratio \leftarrow \min\{minratio, cr\}$;
      save this plan
    end
  end
end

Algorithm 1: Compress a database table.

range of queries.

2.3.1 DirectNVMe

HippogriffDB relies on three components (shown in grey in Figure 2.6) to provide the multi-threaded, peer-to-peer data transfer:

DirectNVMe API: HippogriffDB provides an API for programmers to specify the sources and destinations for data transfers.

DirectNVMe runtime system: The runtime system maintains the runtime information from all processes using NVMEDirect.

DirectNVMe: HippogriffDB uses DirectNVME to perform peer-to-peer transfers between the SSD and GPU.

Compared with existing systems, HippogriffDB improves the I/O bandwidth in two aspects:

HippogriffDB implements a peer-to-peer data transfer path between the SSD and the GPU by re-engineering the software stack of NVMe SSDs as in [9, 91]. When the storage system receives a request with a GPU device address as the source or destination, the NVMe software stack leverages NVIDIA's GPUDirect [3] to make the source or destination GPU device memory address visible for other PCIe devices (by programming the PCIe base address registers). Upon the success of exposing device memory to PCIe interconnect, our NVMe software stack issues NVMe read to the SSD using these GPU addresses as the DMA addresses instead of main memory addresses. The SSD then directly pulls or pushes data from or to the GPU device memory, without further interventions from the CPU and the main memory.

HippogriffDB uses multi-threaded data transfer. It invokes multiple threads (4 threads in the current design) to read data from SSD. To provide fair sharing among processors, the NVMe SSD periodically polls the software-maintained NVMe command queue for each processor. As a result, the SSD can under-utilize both internal access and outgoing bandwidth if only one or two processes are issuing commands to the SSD. HippogriffDB fixes this problem by querying

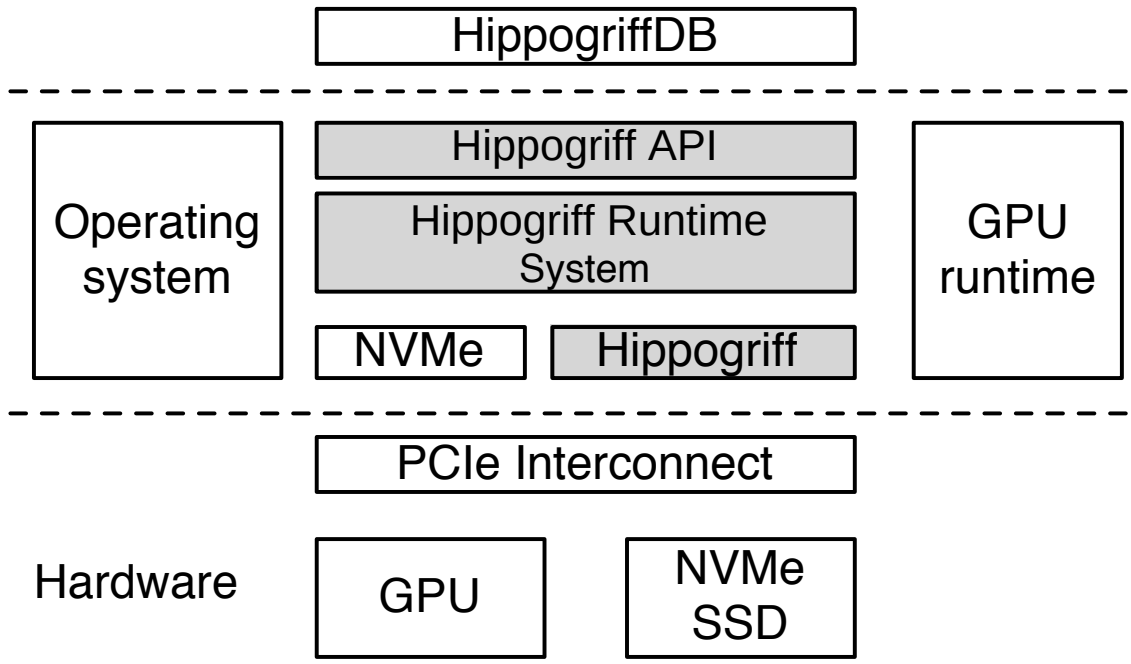


Figure 2.6. Data path in HippogriffDB. HippogriffDB implements a multi-threaded, peer-to-peer data path to boost the I/O bandwidth.

the occupancy of the SSDs NVMe command queues. If the queues are nearly empty, it boosts performance by running multiple peer-to-peer transfers in parallel to improve bandwidth.

2.3.2 Adaptive compression

HippogriffDB uses adaptive compressions to further improve compression ratio for databases on secondary storage. The Algorithm 1 in Section 2.2 minimizes the space cost, however, it could work poorly on some queries. In this section, we first show the inefficiency of the fixed compression and then demonstrate how HippogriffDB fixes the problem with the adaptive approach.

The best compression plan for one query can work poorly on other queries. For Query 2, the best compression plan is as follow: *RLE* on `lo_partkey`, *DELTA* on `lo_supplykey`, *DICT* for `lo_orderdate` and `lo_revenue`. However, this plan works poorly on the query below, as one column in it is not compressed and compression ratio of other columns is not as good as it could be.

```
SELECT c.nation, s.nation, d.year, SUM(lo.revenue) as revenue
FROM customer c, lineorder lo, supplier s, date d
WHERE lo.custkey = c.custkey
        AND lo.suppkey = s.suppkey
        AND lo.orderdate = d.datekey
        AND c.region = 'ASIA'
        AND s.region = 'ASIA'
        AND d.year >= 1992 and d.year <= 1997
GROUP BY c.nation, s.nation, d.year
```

To be adaptive to different queries, HippogriffDB proposes the query-adaptive compression mechanism by allowing each table to keep multiple compressed version. For the example above, instead of having a version that applies *RLE* on `lo_partkey`, *DELTA* on `lo_supplykey`, *DICT* for `lo_orderdate` and `revenue`, we also have another version which applies *RLE* on `lo_supplykey`, *DELTA* on `lo_custkey`, *DICT* for `lo_orderdate` and `lo_revenue`. Hippo-

griffDB may maintain other compression versions as well.

During the query processing, HippogriffDB examines all available compression plans, checks the metadata of all compressed versions and then calculates the overall compression ratio for each of them. HippogriffDB will then use the one with the best compression ratio and send it to the GPU memory.

Suppose the number of foreign-key columns (on which *Huffman* and *DICT* work poorly) is n, the adaptive strategy will produce at most $n(n-1)$ different compression plan. Each compression plan will create a compression schema, which can be a big space overhead. The following theorem reduces the space cost by half, without significant performance degradation.

Theorem 1. *Given two compression plan A and B, where the only difference between them is that they switch the primary and secondary sort key, the compression ratio difference is asymptotically 0 (Assume that $P = o(N)$, where P is the cardinality of the primary sort key and N is the number of rows in the fact table).*

Proof.
1. The compression ratio for columns other than the primary/secondary sort key column is the same.
2. The compression ratio on the primary sort key column is $2 * P/N$. It is asymptotically 0 and the difference from the primary sort key column is asymptotically 0.
3. The distributions of interval on secondary sort key column are the same between compression plan A and B (asymptotically). Hence, the compression effect should be the same asymptotically as well.

□

For databases with large number of foreign key columns, we trade slight compression degradation for big space efficiency improvement. Instead of enumerating the primary-secondary sort key combination, we only enumerate the primary sort key column. In this way, we reduce the space complexity from $O(n^2)$ to $O(n)$ with a minor cost (we can still use other compression methods for the column that is originally encoded using *DELTA*).

$$
\begin{aligned}
NA &\Rightarrow \gamma_{G;\, f_1(.)\mapsto N_1,\ldots,f_n(.)\mapsto N n_\gamma} Join \\
&\mid\ Join \\
Join &\Rightarrow F \bowtie D_1 \bowtie D_2 \ldots \bowtie D_n \\
&\mid\ F \\
F &\Rightarrow \pi_{att}\, \sigma_c\, Fact \\
&\mid\ Fact \\
Di &\Rightarrow \pi_{att}\, \sigma_c\, Dimension \\
&\mid\ Dimension
\end{aligned}
$$

Figure 2.7. An attribute grammar of SSQs

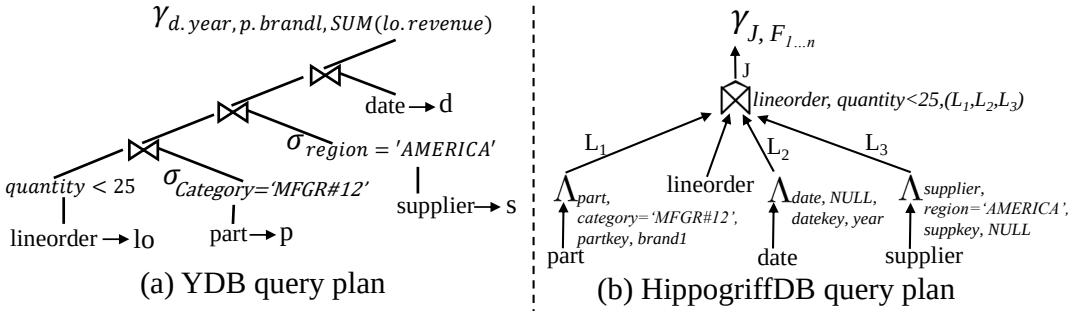


Figure 2.8. The query plan for Query 2 by YDB and HippogriffDB. Query plan that HippogriffDB generates can avoid intermediate results and support dataset larger than GPU memory.

In practice, database users may have additional knowledge about the database and could reduce the space overhead even further.

Remark In this section, we discuss the optimizations for the secondary storage. We use SSD as an example to discuss the peer-to-peer data transfer and query-adaptive compression. Those optimizations also work for other storages as well. For example, the adaptive compression is applicable in a distributed topology, as the space is abundant in such architecture. [4] also allows a direct data transfer from NICs to the GPU, bypassing the CPU/memory overhead for the distributed databases.

2.4 Query-over-block model

The query-over-block model handles data sets larger than the GPU memory. It provides high scalability by processing input in a streaming manner, and removes intermediate results and

Table 2.2. Categorization of operations in Query 2.

Category	Operations
Category 1	p.category = MFGR#12 s.region = AMERICA'
Category 2	lo.orderdate = d.datekey lo.partkey = p.partkey lo.quantity $<$ 25
Category 3	SUM(lo.revenue) GROUP BY d.year, p.brand1

improves GPU kernel efficiency using the operator fusion mechanism.

In this section, we first introduce the data schemas that we focus on, then formally (re)define the queries running on these schemas. Based on the new definition, we introduce three physical operators and show how HippogriffDB uses them to optimize the query plan.

2.4.1 Schema

HippogriffDB targets OLAP and data warehouse applications. Data warehouses typically organize data into *multidimensional cubes* (or *hypercubes*). and map hypercubes into the relational database using a star schema or a snowflake schema. The star schema maintains tables in the star topology: a central table contains fact data and multiple tables radiate out from it. The central fact table and the rest of the tables connect through the primary and foreign keys of the database. Existing comparison results show that star schema is prevalent in data warehouses [53, 13].

HippogriffDB focuses on star schema queries (SSQ). For other queries and schemas, HippogriffDB can work as an accelerator on SSQ subexpressions and leave the rest to classic methods.

2.4.2 Query definition

Our query processing system focuses on SSQs and we optimize them by redesigning the physical query plan generator. In this subsection, we first describe the characteristics of these

queries and formally (re)define the attribute grammar of SSQs.

The operators inside an SSQ fall into three categories. The first category is unary operations, such as *selection* and *projection*, on dimension tables. The second category includes unary operations on the fact table and *natural join* between the fact and the dimension tables. The last category is *aggregation* and *group by* on the join results. As an example, we categorize the operations in Query 2 into these three categories, as shown in Table 2.2.

Figure 2.7 provides the *Normalized Algebra (NA) expression* for SSQs. This attribute grammar has the ability to describe all queries running on star schemas. The root of the *NA* is either an aggregation of a join or just a join. A join here is either a series of joins over $F, D_1, \ldots D_n$ or F itself, where F is a (unary operations of) fact table and D_i is a (unary operations of) dimension table.

2.4.3 Query-over-block execution model

HippogriffDB supports scalable and efficient processing for queries on star schemas by introducing a stream-based query model, called query-over-block. Query-over-block model improves the scalability and efficiency for SSQs using an "operator fusion" mechanism and a stream-based approach. In this subsection, we first introduce three new physical operators to produce physical query plans and then demonstrate how query-over-block model generates query plans using these physical operators.

Operator fusion

We implement three physical operators using operator fusion mechanism to improve kernel efficiency and to eliminate intermediate results. Below, we first define these physical operators and then discuss their implementations.

Physical operator definition

Based on the attribute grammar in Section 2.4.2, HippogriffDB introduces three corresponding physical operators: $\Lambda_{R,c,K,V}$, $\boxtimes_{R,c,L_1,\ldots,L_n}$, and $\Gamma_{J,f_{1,\ldots,m}}$. We define the three operators logically

as follows:

1. $\Lambda_{R,c,K,V}$ outputs $\pi_{K,V}\sigma_c(R)$, a hashmap L with $\pi_K\sigma_c(R)$ as the keys and $\pi_V\sigma_c(R)$ as the values.
2. $\bowtie_{R,c,L_1,\ldots,L_n}$ outputs $\sigma_c(R) \bowtie L_1 \bowtie \ldots \bowtie L_n$.
3. $\gamma_{J,A,f_{1,\ldots,m}}$ outputs the results of $\gamma_{A,f_{1,\ldots,m}}(J)$.

Implementation We implement the physical operators as follows:

1. $\Lambda_{R,c,K,V}$. We implement the hashtable using Cuckoo Hashing [72]. Each GPU thread evaluates the selection condition on a certain input and, if the condition is met, inserts the input into the hash index. We use atomic instructions that CUDA provides to avoid conflicts in the parallel program.
2. $\bowtie_{R,c,L_1,\ldots,L_n}$. We assign each GPU thread a row in the fact table. The GPU thread first evaluate selections on the relation R and then probes the hash indices of $L_1, \ldots, L_n$. We assume the hash indices can fit in the GPU memory. We discuss the memory requirement at the end of this section.
3. $\gamma_{J,A,f_{1,\ldots,m}}$. We use hashmap or array (if we know the domain size in advance) for the aggregation. We use atomic instructions to resolve conflicts between different threads.

Operator fusion mechanism

HippogriffDB mollifies the memory contention that the intermediate results cause using the operator fusion mechanism when implementing the physical operators. This mechanism combines multiple operators into a single GPU kernel and passes the result of the previous operator via local variables inside each GPU thread.

HippogriffDB assigns each row in the fact table a GPU thread and this thread performs the functionality of multiple operators. HippogriffDB evaluates all operators inside one thread body by passing the intermediate results using local variables. In this way, we compact multiple database operators into a single GPU kernel program and turn intermediate result passing into local variable passing inside each GPU thread.

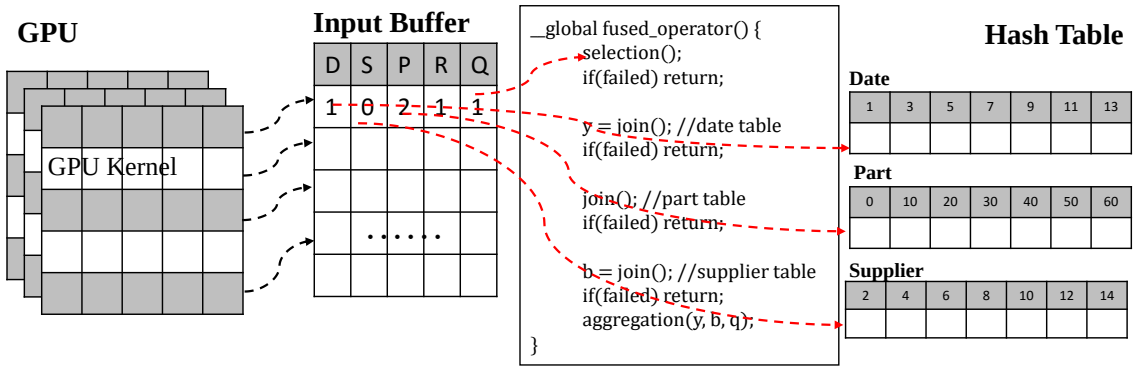


Figure 2.9. HippogriffDB fuses multiple operators (one selection, three joins and one aggregation) into one to eliminate the intermediate results and improve kernel efficiency.

For example, in the physical operator $\boxtimes_{R,c,L_1,\ldots,L_n}$, we fuse all joins (hash joins) and selections on the fact table into one GPU kernel. The kernel first evaluates the selection operation on a given row. If the row survives the selection, it will proceed to join with other dimension tables. The GPU kernel passes intermediate results using local variables inside the thread. The implementation of physical operator $\Lambda_{R,c,K,V}$), $\gamma_{J,f_{1,\ldots,m}}$ follows a similar approach.

Figure 2.9 shows an example of the operator fusion approach using Query 2. HippogriffDB query model fuses one selection, three joins and one aggregation into one GPU kernel and it passes intermediate results, such as *year* (denoted as *y* in the figure) and *brand*1 (denoted as *b*), using local variable.

We use the same example to compare our query plan with the query plan that existing models generate. Figure 2.8 (a) presents the corresponding query plan that most existing GPU-based databases adopt. After joining *lineorder* and *part*, the system sends the intermediate results to join another table, *supplier*, and then generates another set of intermediate results, and so forth. The existing query plan sends large intermediate results several times during the query execution, which is costly as accessing GPU global memory is slow. In addition, storing those intermediate results on a memory-scarce device hurts the scalability of the system. Figure 2.8(b) shows our approach. It packs the selections on the dimension tables and hash index building into the Λ operator. It also fuses the three natural joins, the selection on table `lineorder`, and the

aggregation into one GPU kernel. In this way, HippogriffDB avoids all intermediate results.

Discussion [14] uses invisible joins to reduce redundant data transfer. It first evaluates the invisible join and then, based on the join results, reads the other rows on demand. HippogriffDB doesn't adopt this approach, as the second step would involve large amount of random reads to the SSD, which is slow for a flash-based SSD (and also hard disk).

Input: A dimension table R, a list of selection conditions $\mathscr{C}$=$\{\mathscr{C}_1,...,\mathscr{C}_m\}$ and the index format (key_col, val_col)
Output: A hash index for the selected tuples, $\mathscr{H}$
Func build_index($R,\mathscr{C},key\_col,val\_col$) ;
$\mathscr{H} \leftarrow \varnothing$;
cuda_build_index<<<...>>> ($\mathscr{H}$, $R,\mathscr{C},key\_col,val\_col$) ;
return $\mathscr{H}$;
Func cuda_build_index($\mathscr{H}$, $R,\mathscr{C},key\_col,val\_col$) ;
r $\leftarrow R[thread\_id]$;
for $c \in \mathscr{C}$ **do**
  if $eval(c,r) == false$ **then**
    return false ;
  end
end
cuckooInsert($\mathscr{H}$, P_{r,key_col}, P_{r,val_col});
// $P_{r,k}$ is the projection of r on column k

Algorithm 2: Generate hash index for the dimension tables

Block-oriented execution plan

HippogriffDB uses a block execution plan to solve the scalability problem caused by limited GPU memory. It streams the fact table in small blocks to support datasets larger than GPU memory capacity. HippogriffDB adopts double buffering to support asynchronous data transfer, which allows the overlapping between the kernel execution and data transfer.

HippogriffDB generates physical query plan for a given query in three phases. It first pushes down operators on the dimension table and builds an in-GPU-memory hash indexes for them (physical operator $\Lambda_{R,c,K,V}$) (we assume they can fit in the GPU memory). Algorithm 2 provides the details in this process. In the second phase, it evaluates the second category using the hash index built in the previous stage (physical operator $\bowtie_{R,c,L_1,...,L_n}$). The third stage evaluates

Input: A fact table F, dimension table indices $\mathscr{H}$=$\{\mathscr{H}_1,...,\mathscr{H}_n\}$, a list of selection conditions $\mathscr{C}$=$\{\mathscr{C}_1,...,\mathscr{C}_m\}$, join conditions $\mathscr{J}$=$\{J_1,...,J_l\}$, groupby columns $\mathscr{G}$=$\{G_1,...,G_k\}$, aggregation function $\mathscr{A}$=$\{A_1,...,A_p\}$
Output: Analytical results $\mathscr{R}$
Func fused_kernel($F,\mathscr{H},\mathscr{J},\mathscr{C},\mathscr{G},\mathscr{A}$) ;
$\mathscr{R} \leftarrow \varnothing$;
cuda_fused_kernel<<<...>>>($\mathscr{R},F,\mathscr{H},\mathscr{J},\mathscr{C},\mathscr{G},\mathscr{A}$);
return $\mathscr{R}$;
Func fused_kernel($\mathscr{R},F,\mathscr{H},\mathscr{J},\mathscr{C},\mathscr{G},\mathscr{A}$) ;
r $\leftarrow R[thread\_id]$;
for $c \in \mathscr{C}$ **do**
  if $eval(c,r) == false$ **then**
    return false ;
  end
end
for $j \in \mathscr{J}$ **do**
  if $\mathscr{H}_j.find\ (P_{r,j}) == NULL$ **then**
    return false ;
  end
  evalAggr($\mathscr{R}$, $P_{r,\mathscr{G}}$, $\mathscr{A}$) ;
end

Algorithm 3: Operator fusion algorithm

aggregations on the join results (physical operator $\Gamma_{J,f_{1,\ldots,m}}$). We provides the operator fusion mechanism in Algorithm 3.

HippogriffDB keeps the hash indices of dimension tables in GPU memory and streams the fact table to evaluate queries. HippogriffDB adopts an input buffer to enable asynchronous data transfer for efficient streaming,. The data transfer manager continues to transfer data while the kernel manager evaluates the received input.

Query-over-block provides higher scalability compared with the *bulk execution plan* [23] that most existing column databases adopt. In the bulk execution query model, all related columns must reside in the GPU memory before query processing. Given the limited capacity of GPU memory and lack of virtual memory support, this requirement will severely bound the scalability of the database system. Instead of using bulk execution, HippogriffDB adopts the block execution model. This model allows the GPU kernel to work on a small chunk of the whole input and to process input in a streaming manner. In this way, the query-over-block model can support data sets that are larger than the GPU memory.

Memory requirement In the current implementation, HippogriffDB requires the hash indices of the dimension tables to fit in the GPU memory. We also maintain the input and output buffer in the GPU memory. Hence, the memory requirement is $S_{\texttt{input}} + S_{\texttt{output}} + \sum H_i$, where H_i denotes the size of hash index of dimension table i and $S_{\texttt{input}}$, $S_{\texttt{output}}$ denote the size of the input/output buffer.

Limitation We use the star schema queries as an example to demonstrate the query-over-block execution model. The query-over-block model first pushed down the selection on the dimension tables and then fuses the other operators (selection, join and aggregations on the fact table) into one GPU kernel. This approach can work on other schemas as well, such as snowflake schemas without any difficulty, as both steps in the query-over-block can be applied to snowflake schemas as well. The query-over-block has two limitations. First, it assumes that the indices for the dimension table can fit into the GPU device memory. We look forward to using multiple

GPUs or enlarged GPU device memory in the future generations to solve this issue. Second, it requires that the query execution can be streamlined. As we discussed above, both star-schema and snowflake-schema queries can be streamlined. However, there are also other queries that are hard to be processed in a streaming manner, e.g., many-to-many joins. In this case, we may need to adopt the CPU and host memory to help process these queries.

2.5 EXPERIMENTAL METHODOLOGY

We built HippogriffDB and a testbed that contains an Intel Xeon processor, an NVIDIA K20 GPU and a PCIe-attached SSD. We evaluate HippogriffDB using two popular data analytic benchmarks and we compare it with two state-of-the-art data analytic systems. This section describes our test bed, benchmark applications, and the two systems that we compare HippogriffDB with.

2.5.1 Experimental platform

We run our experiments on a server with an Intel Xeon E52609V2 processor. The processor contains 4 cores and each processor core runs at 2.5 GHz by default. The server contains 64 GB DDR3-1600 DRAM that we used as the main memory in our experiments. The GPU in our testbed is an NVIDIA Tesla K20 GPU accelerator, which contains 5 GB GDDR5 memory on board. The K20 GPU connects to the rest of the system through 16 lanes of the PCIe interconnect that provides 8 GB/sec I/O bandwidth in each direction. We use a high-end PCIe-attached SSD as the secondary storage device (with 1 TB capacity). The testbed uses a Linux system running the 3.16.3 kernel. We implement the GPU operator library in HippogriffDB based on NVIDIA CUDA Toolkit 6.5.

2.5.2 Benchmarks

To evaluate our system, we use two popular analytical benchmarks. The two benchmarks are the Star Schema Benchmark (SSBM)[71] and the Berkeley Big Data Benchmark (BBDB)[76].

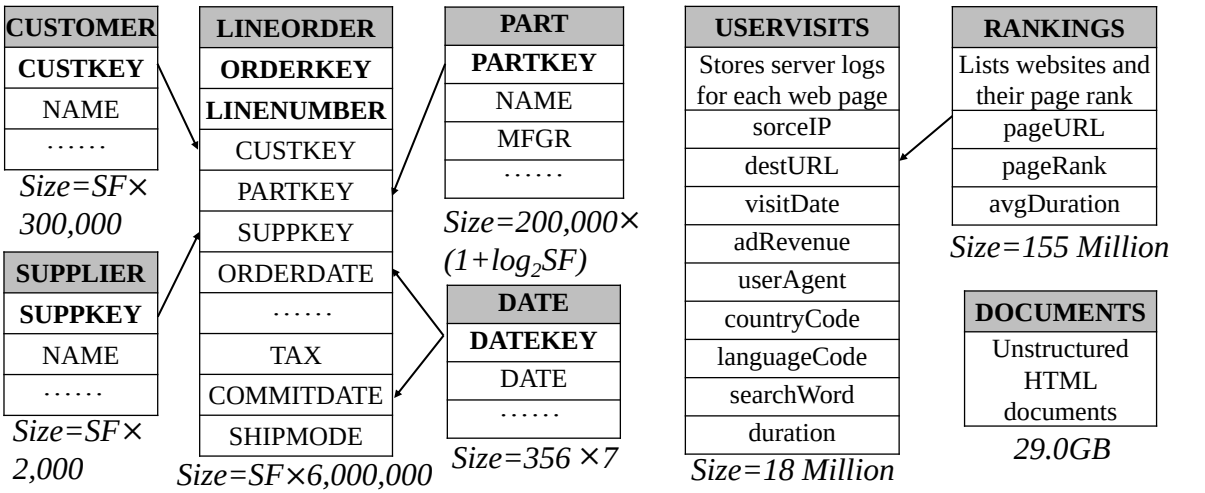


Figure 2.10. Schema of the database in SSBM and BBDB.

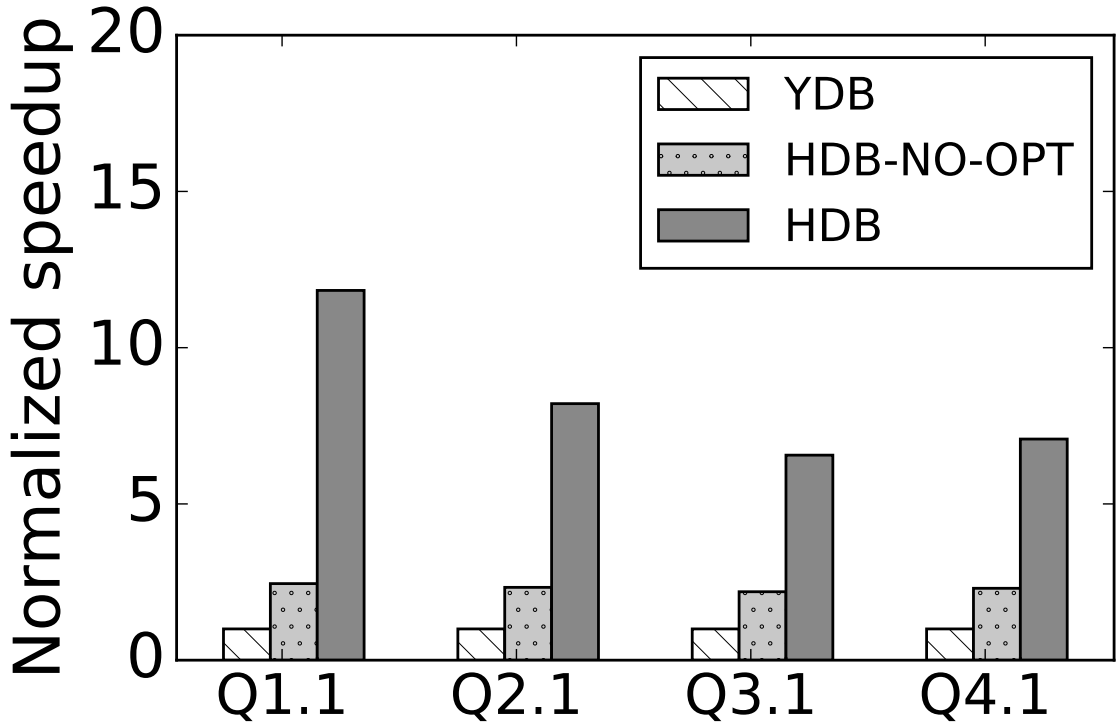
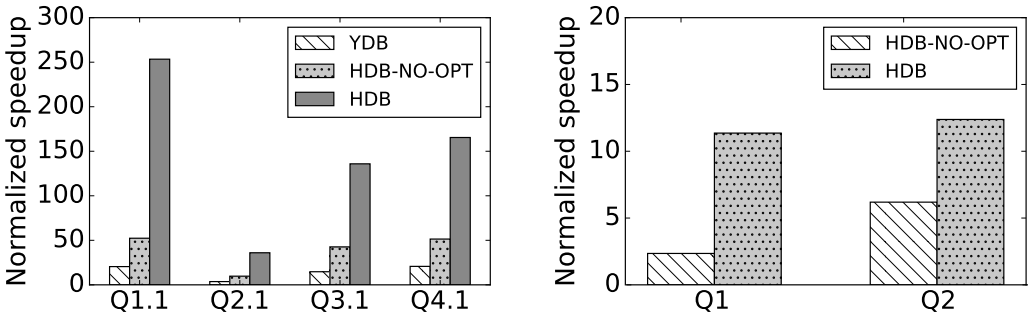


Figure 2.11. Performance of different systems (SF=10) when data are in SSD. HippogriffDB outperforms YDB by up to 12×.



(a) Normalized speedup on SSBM (b) Normalized speedup on BBDB

Figure 2.12. Normalized speedup relative to MonetDB (SF=10) when data are in memory. HippogriffDB outperforms competitors by 1-2 orders of magnitude.

SSBM is a widely used benchmark in database research due to its realistic modeling of data warehousing workloads. In SSBM, the database contains one fact table (`lineorder` table) and four dimension tables (`supplier`, `customer`, `date` and `part` table). The fact table refers to the other four dimension tables, as shown in Figure 2.10(a). SSBM provides 13 queries in 4 flights. HippogriffDB supports all 13 queries. When the scale factor is 1, the total database size is about 0.7 GB. We vary the scale factor from 1 to 1000 in our experiments. The database size is 0.7 TB when the scale factor reaches 1000.

BBDB includes several search engine workloads. The database in BBDB contains three tables, depicting documents, pageranks and user visits information, as shown in Figure 2.10(b). The benchmark contains 4 queries. The third query contains a string join, which current HippogriffDB does not support, and the last one involves an external Python program. Hence, we evaluate our system using Query 1 and Query 2 in this benchmark.

2.5.3 Competitors

We compare HippogriffDB with two analytical database systems, MonetDB[23] and YDB[103]. MonetDB is a state-of-the-art column-store database system that targets analytics over large inputs. YDB is a GPU execution engine for OLAP queries. Experiment results show that YDB runs up to 6.5× faster than its CPU counterpart on workloads that can fit in GPU's

memory.

2.6 Results

In this section, we present the experimental results for HippogriffDB. This section first presents the end-to-end performance compared with the two competitors. After that, we evaluate the effectiveness of the proposed methods in balancing component throughput inside the system. We then evaluate the query-over-block model.

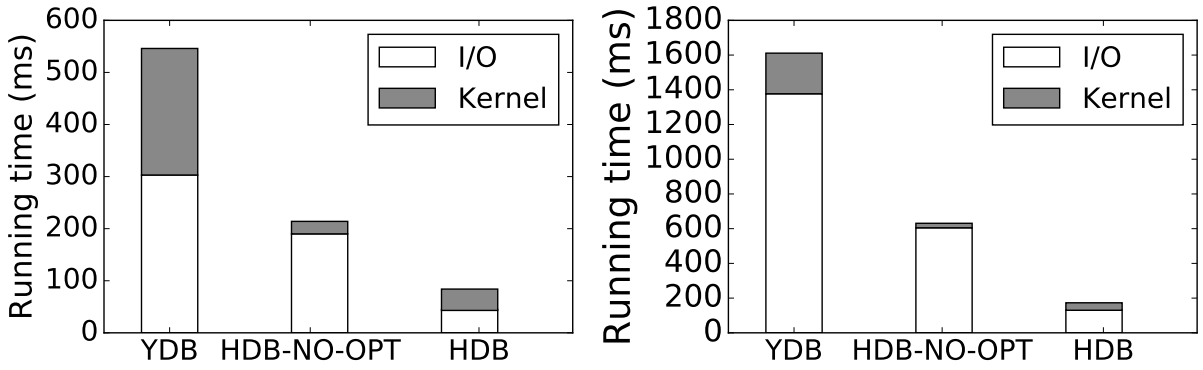
2.6.1 Overall performance

We first evaluate the speedup that HippogriffDB can achieve. We compare out system with two baselines: MonetDB[23] and YDB[103]. We provide two versions of HippogriffDB here: HippogriffDB without compression and pipelining optimizations (HDB-NO-OPT) and the full-fledged version (HDB).

Figure 2.12(a) shows the normalized speedup of different systems (relative to MonetDB) when data are in the GPU memory. We use 10 as the scale factor here. In this case, the working set size is 0.96-1.44 GB for SSBM. For BBDB, we adopt a 1.2 GB input. As shown in Figure 2.12(a), HDB-NO-OPT outperforms MonetDB by 38× and YDB by 2.6× on average for SSBM queries. The full-fledged version (including compression and pipelining) outperforms MonetDB by 147× and YDB by 9.8× on average. For BBDB, HDB-NO-OPT achieves 4.2× speedup compared with MonetDB and with optimizations the speedup rises to 11.8×, as shown in Figure 2.12(b). HDB-NO-OPT produces less speedup for BBDB compared with SSBM, as the queries in BBDB are relatively simple and cannot fully utilize the GPU computation power.

Figure 2.11 compares the execution time when the database resides in the SSD. As shown in the figure, HDB-NO-OPT outperforms YDB by 2.4× on average. With optimizations, HDB outperforms YDB by 8.4× on average.

Figure 2.13(a) breaks down the execution time of Q1.1 into I/O and kernel execution when HippogriffDB (both HDB-NO-OPT and HDB) and YDB store data in the main memory.



(a) In memory (b) On SSD

Figure 2.13. Break down of query Q1.1 execution time. HDB improves both kernel and I/O efficiency compared with other analytical systems.

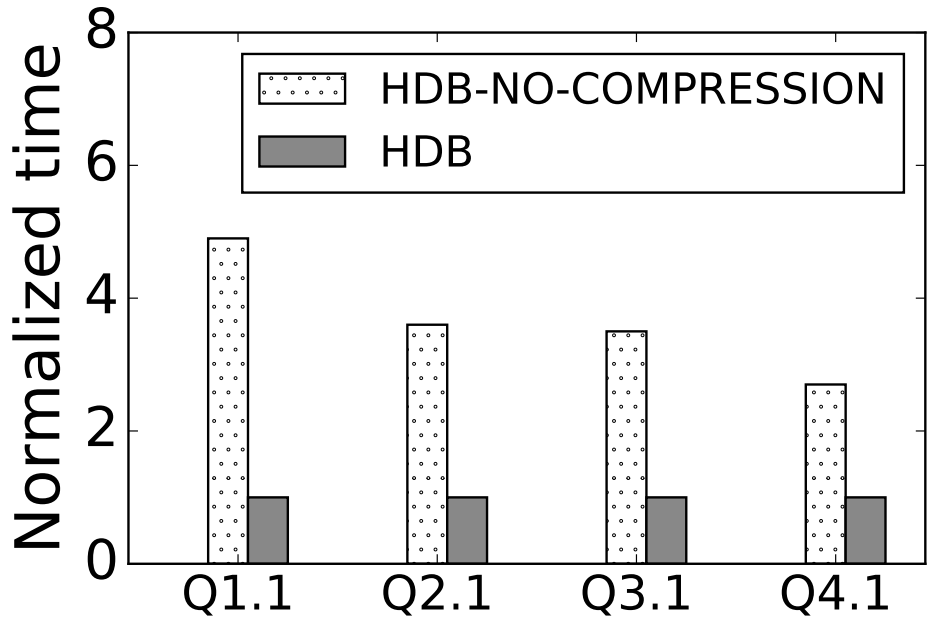


Figure 2.14. HippogriffDB with and without compressions, SF=10. Compression helps improve system throughput by up to 5×.

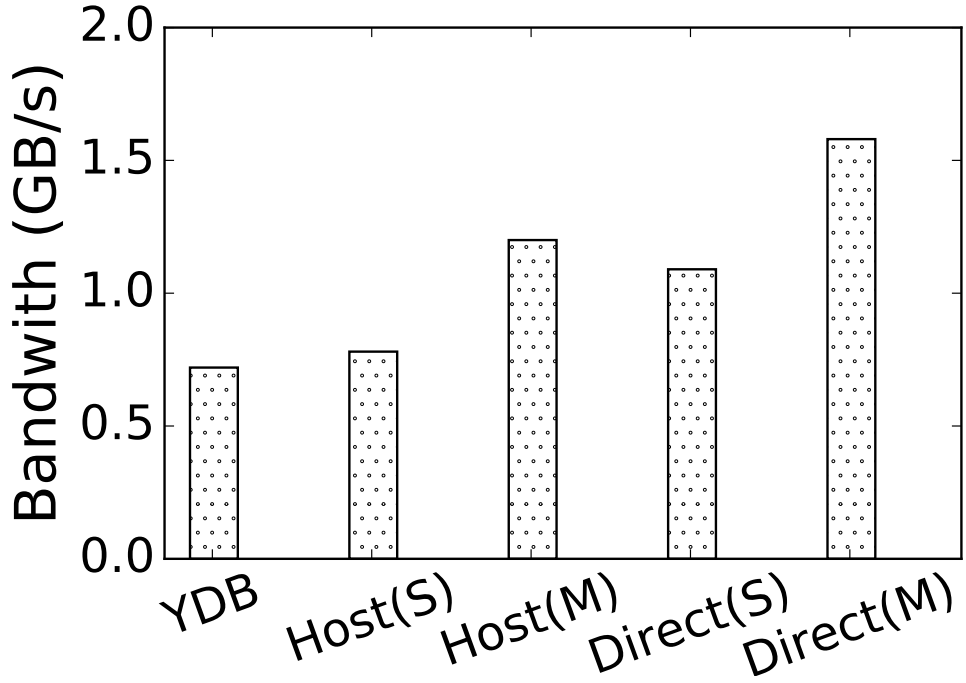


Figure 2.15. The I/O bandwidth of different data paths. Multi-threaded, peer-to-peer data transfer helps improve I/O bandwidth

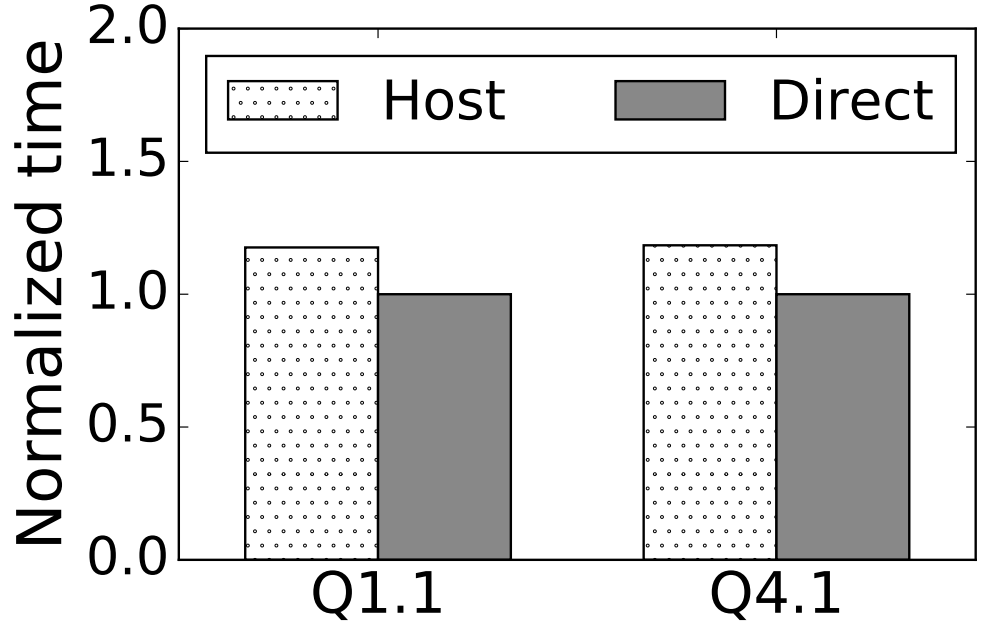


Figure 2.16. Effect of peer-to-peer I/O optimization, SF=1000. Direct datapath and multi-threaded help improve system throughput by 18%.

We do not show MonetDB here, as it is not a GPU-based database and it does not have these two stages. To measure the execution time breakdown, we disable the pipelining mechanism in our systems. The execution time breakdown indicates that the majority of performance boost comes from the GPU kernel. By removing intermediate results and using new physical operators, HDB-NO-OPT runs 9.8× faster than YDB. In addition, the data transfer rate in HDB-NO-OPT is also 36% faster than YDB, due to less software and metadata overhead[2]. Compression reduces the table size by 4.6× and hence reduces the I/O time in HDB. Though the decompression adds additional cost to the GPU processing, because of the significant improvement from the I/O stage, HDB still achieves 2.5× speedup compared with HDB-NO-OPT.

We also show the execution time breakdown for the SSD version in Figure 2.13(b). The performance gain in this case is mainly from the optimized data transfer in HippogriffDB. The inefficiency of I/O in YDB agrees with the results in [94]. The I/O bandwidth HDB can achieve is up to 2.3× larger than its competitor.

2.6.2 Close the GPU-I/O bandwidth gap

HippogriffDB fixes the gap between the fast GPU kernel and the slow data transfer by overcoming the I/O bottleneck in two ways: (1) it compresses databases and trades idle GPU cycles for decompression to achieve better data transfer efficiency. (2) it redesigns the data path to bypass the host CPU and the main memory when transferring data from the SSD to the GPU. In this subsection, we evaluate these approaches.

Effect of compression

HippogriffDB stores data in a compressed format and trades GPU cycles for better I/O performance. In this subsection, we study the effect of data compression in terms of bandwidth improvement.

[2]We use the same method as in[94] to run YDB: warm up memory by executing each query once before the experiments. Reading from a warm cache could be slower compared with reading directly from the main memory due to some operating system overhead.

Table 2.3. Compression ratio of query-adaptive and query-insensitive compression. Query-adaptive compression can keep good compression ratio when the database scales up (x-axis is the scale factor).

	1	**10**	**100**	**1000**
HDB-Q-ADAPTIVE	0.28	0.30	0.31	0.32
HDB-Q-INSENSITIVE	0.42	0.46	0.48	0.49
DICT	0.44	0.48	0.52	0.55

We first compare the execution time with compression (HDB) and without compression (HDB-NO-COMPRESSION) for various queries. We use 10 as the scale factor here. Figure 2.14 shows the comparison results. Compression can achieve $2.8\times - 4.9\times$ improvement in system throughput. As discussed in Section 2.2, compression on the foreign key columns is the most difficult, due to its large cardinality. Compression benefits most in Q1.1, as this query only involves one foreign key. For other queries, HDB can still reach a rather decent compression ratio.

HippogriffDB adopts query-adaptive compression for databases stored in SSD. We compare the compression ratio difference between a query-adaptive compression (HDB-Q-ADAPTIVE) and a fixed approach (HDB-Q-INSENSITIVE) using the example given in Section 2.3.2. We show the compression ratio in Table 2.3. As shown in the table, the query-adaptive compression can maintain decent compression ratio when databases scaling up while the compression efficiency of the fixed approach degrades significantly. It's because the fixed compression fails to apply effective compression methods on critical foreign keys. Previous literature [103] indicates that DICT can achieve a satisfying compression effect on small data sets while our results show that the performance of DICT also degrades rapidly when data sets scale up.

Effect of peer-to-peer data transfer

Observing that data transfer bandwidth is the system bottleneck, we adopt several optimizations to improve the bandwidth. In this subsection, we evaluate the bandwidth improvement using the multi-threaded, peer-to-peer communication mechanism.

We compare the proposed methods with two other baselines. The first one is from YDB,

which uses `mmap` to open files and then `memcpy` data to the GPU pinned buffer. A second baseline is as follows: we use file I/O to open a file, read data to pageable memory and copy them to the GPU memory. We also compare the bandwidth between single-threaded I/O and multi-threaded I/O.

Figure 2.15 compares the I/O bandwidth of YDB, single-threaded host route (Host(S)), multi-threaded host route (Host(M)), single-threaded peer-to-peer route (Direct(S)) and multi-threaded peer-to-peer route (Direct(M)). As shown in the figure, the data transfer mechanism HippogriffDB uses can reach up to 1.6 GB/s, 20% higher than a highly optimized host route and 2.3$\times$ higher than the YDB data path. Multi-threaded data transfer can improve the bandwidth by up to 52%, as it can better saturate the SSD internal read bandwidth. We list the effect of different data transfer paths on system performance in Figure 2.16. We compare the execution time of using an optimized host route and using peer-to-peer data transfer. Experiment shows that the peer-to-peer data transfer helps reduce the end-to-end latency by 19%.

Figure 2.17 compares the throughput of moving data from the SSD to the GPU using DirectNVME (M) against standard NVMe (NVMe), pipelined NVMe (NVMe-pipeline) that overlaps SSD access with GPU memory copy and the single channel, peer-to-peer DirectNVME (DirectNVME (S)). We report the data transfer throughput under different file sizes, excluding the overhead of allocating all necessary resources (e.g. memory buffers) along the data paths. Because the K20 GPU has only 4.8GB device memory available for applications, we examine these route options with file sizes under 4 GB.

DirectNVMe (M) outperforms all other route options. The performance advantage of DirectNVMe (M) becomes more significant as file size increases. When transferring a 4GB file between the SSD and the GPU, a DirectNVMe) that performs file access requests using a single NVMe command queue only achieves bandwidth of 1110 MB/sec, due to the under-utilized NVMe SSD resources. DirectNVMe (M), on the other hand, offers up to 2221 MB/sec bandwidth.

NVMe-pipeline improves the performance of standard NVMe by compensating for laten-

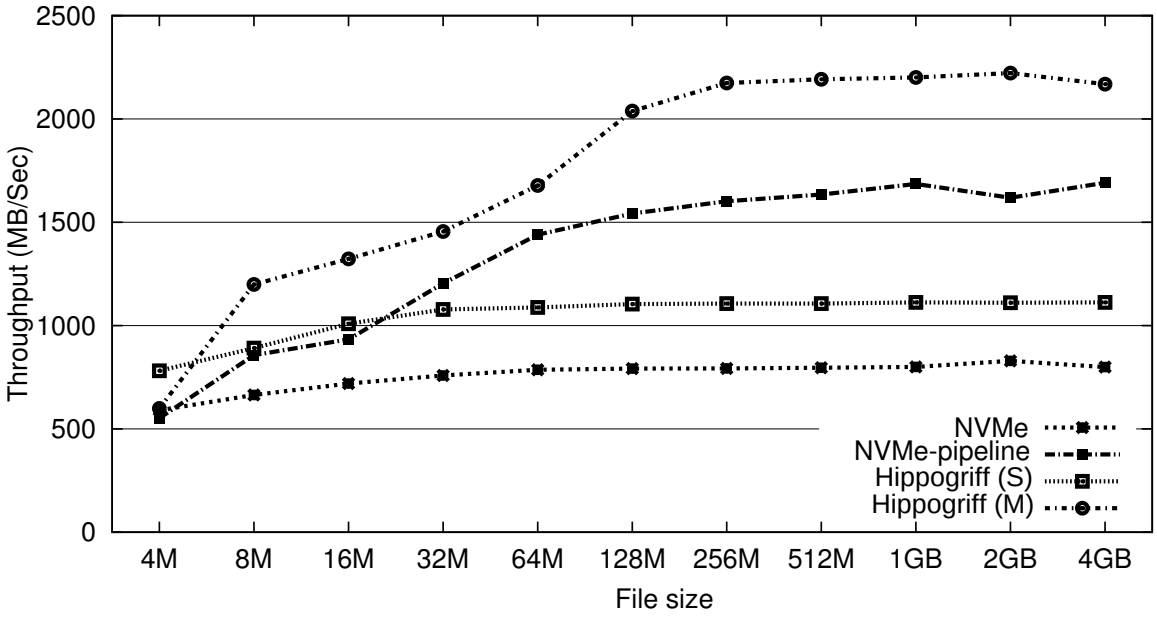


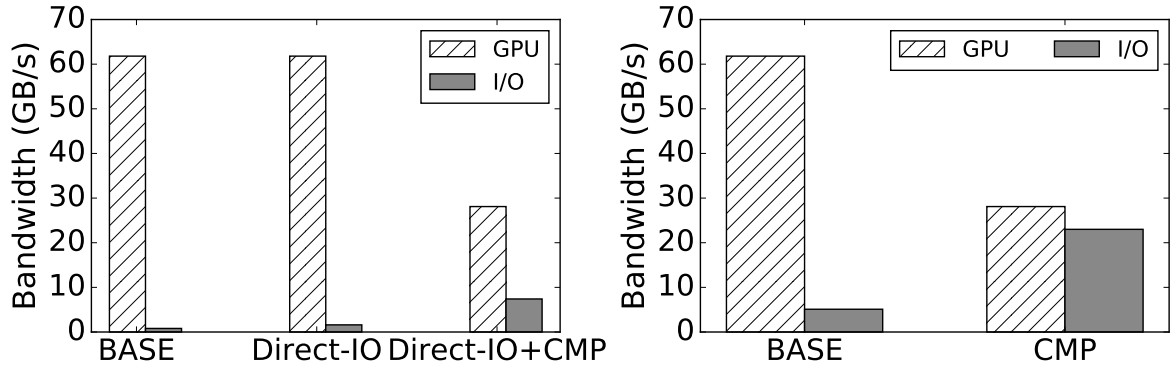
Figure 2.17. The throughput of different data paths in HippogriffDB.

Table 2.4. Normalized scalability performance with increasing SF (from 1 to 1000). Results testify the scalability of HippogriffDB (x-axis is the scale factor).

	1	**10**	**100**	**1000**
YDB	12.27	98.03	N/A	N/A
HippogriffDB	1	10.27	93.60	938.0

cies with multiple data transfers. However, NVMe-pipeline can still only achieve a throughput of 1691 MB/Sec between the SSD and GPU for 4 GB files, 34% slower than DirectNVMe (M), because NVMe-pipeline requires more CPU resources.

As a summary of the effect of the endeavours discussed above, Figure 2.18 shows the effect of narrowing the bandwidth gap between the GPU kernel and I/O. We compare the difference between GPU kernel and data transfer bandwidth using SSBM Q1.1 and show the results for both SSD-based and memory-based HippogriffDB. For the SSD version, the initial gap (BASE) is up to 82×. The direct data transfer (Direct-IO) brings it down to 38× and the compression (Direct-IO+CMP) further brings the gap down to 3.9×. For the in-memory version, compression (CMP) narrows the gap from 12× to 1.2×, very close to achieving the balance.



(a) SSD (b) Memory

Figure 2.18. Effect of closing the GPU-IO bandwidth gap. Proposed approaches narrow the GPU-IO gap by up to 21×.

2.6.3 Query-over-block model evaluation

The query-over-block model makes HippogriffDB the first GPU-based database system that provides native support for big data analytics. The query model uses several optimizations to improve performance, including removing materialization and double buffering. In this subsection, we first evaluate the system scalability and then analyze the effect of the proposed optimizations.

System scalability

We test the scalability of HippogriffDB by varying the scale factor from 1 to 1000 (database size from 0.7 GB - 0.7 TB). We run the SSBM Q1.1 in the experiment without compression. Table 2.4 reports the execution time for queries on YDB and HippogriffDB. The database resides in SSD in this experiment. As shown in the table, YDB cannot support queries when the scale factor is above 10 while HippogriffDB shows its superiority by scaling up to support terabyte-level input.

When scaling up, the throughput of HippogriffDB remains stable (as same as the data transfer bandwidth). This is because the GPU kernel always runs faster than data transfer bandwidth in this case. We show the GPU kernel throughput in Figure 2.19: the speed GPU

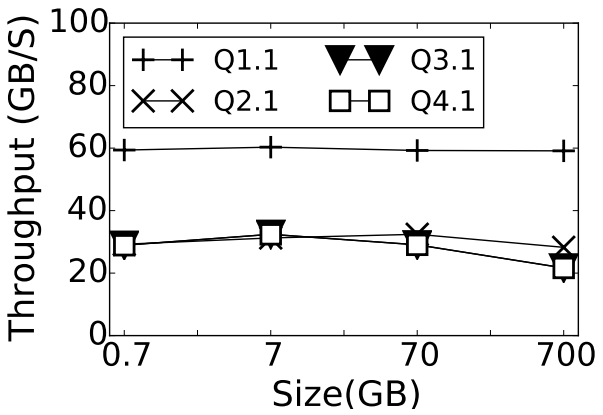


Figure 2.19. Scalability of GPU kernels on SSBM. GPU kernel throughput is consistently higher than I/O bandwidth by over one order of magnitude.

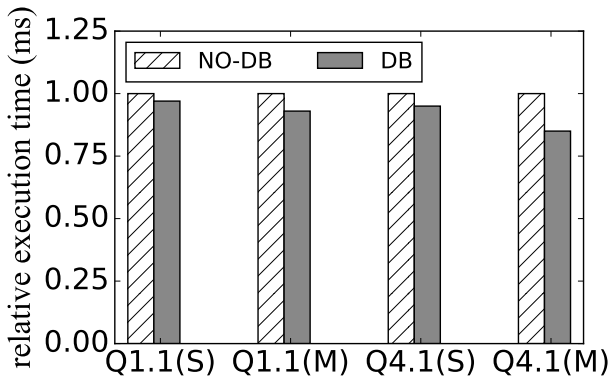


Figure 2.20. Effect of double buffering. Double buffering helps reduce the execution time by up to 15% without compression. It can further improve system performance with other optimizations.

processes database queries (more than 20 GB/s) is at least 12× higher than the I/O bandwidth. This trend sustains when the input scales up to terabyte-level tables. The double buffering always keeps I/O device busy and saturates the I/O bandwidth. As a result, the performance remains the stable when scaling up. The I/O bandwidth without optimization is not satisfiable and that's the reason we propose compression and peer-to-peer data transfer to improve the effective I/O bandwidth.

Effect of double buffering

HippogriffDB uses double buffering to overlap the data transfer and kernel execution, reducing the execution time of query processing. We compare the effect of using double buffering

in Figure 2.20 (SF = 10). The double buffering reduces the execution time for Q1.1 in SSBM by 3% and 7% for SSD-based and memory-based HippogriffDB. For Q4.1, it can help improve the execution time by 5% and 15% respectively. Double buffering works better on complex queries, as the GPU kernel time consumes higher portion in the total execution time for complex queries. Double buffering can further improve the system performance in combination with other optimizations, such as compression. With data compression, the gap between faster part and slower part narrows and hence the overlapping can result in more performance gain.

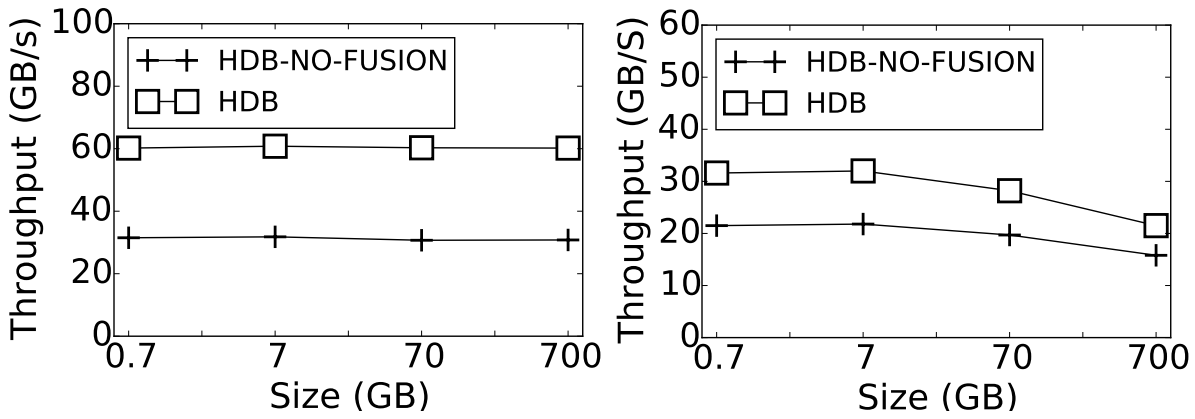
Effect of avoiding intermediate results

Figure 2.21 compares the the benefit of reducing intermediate results using SSBM Q1.1 and Q4.1. We vary SF from 1 to 1000 (database size from 0.7 GB - 0.7 TB). We compare the throughput of HippogriffDB (HDB) and HippogriffDB without operator fusion (HDB-NO-FUSION). As shown in Figure 2.21, the GPU kernel throughput improves by 91% for Q1.1 and 43% for Q4.1. Reducing intermediate results works better on light-weighted queries. For heavy-weighted queries, the computation can take a significant portion of time and operator fusion will not optimize for this part. For query 4.1, the benefits of reducing intermediate results decreases with the growth of the scale factor. It is also because the computation load increases with the growth of the scale factor.

2.6.4 Proposed optimizations on wimpy hardware

In the previous sections, we discuss the proposed optimizations on high-end hardware. In this subsection, we examine the optimizations on wimpy hadware, such as low-end GPUs.

While the DirectNVMe does not work with low-end GPUs, the query-over-block model can still improve the kernel efficiency on the wimpy hardware. By reducing the intermediate results, the query-over-block execution model improves the GPU processing rate by 2.9× for light-weighted query (SSBM Q1.1, SF = 10) and by 2.4× for heavy-weighted query (SSBM Q4.1, SF=10). Although peer-to-peer data transfer is not supported on the low-end GPU or



(a) Q1.1 (b) Q4.1

Figure 2.21. Effect of removing intermediate results. Removing intermediate results can improve query execution time by up to 91%.

non-PCIe SSD, the multi-threaded transfer mode still helps boost the bandwidth. As shown in Figure 2.17, using multi-threaded I/O can achieve 1.6 GB/s on our SSD. Compression still works to narrow the bandwidth mismatch between the SSD and the GPU. For example, the compression can increase the effective I/O bandwidth by 4.6× on SSBM Q1.1 while the GPU can still process at 22.9 GB/s, 3.3× larger than the effective bandwidth.

2.7 Related Work

With the end of Dennard scaling [30] (power density stays constant), it is hard for general purpose CPUs to provide scalable performance in the future due to the power challenges[43, 36]. In recent years, researchers in database community started to think about heterogeneous computing as a way to overcome the scaling problem of CPUs and a way to reach better performance and scalability for databases [45, 46, 20, 96, 85, 79, 31, 44] By offloading computation to multiple components, including GPUs, FPGAs and other hardware accelerators, heterogeneous computing opens new opportunities for database designers to explore parallelization, pipelining and high computation bandwidth and to achieve better scalability and performance for data-intensive workloads.

Among various hardware accelerators, GPU is the one that draws the most attention.

Several full-fledged GPU database query engines[47, 24, 103] came out in the recent years. Ocelot [47] provides a hybrid analytical query engine as an extension to MonetDB. Ocelot designs a set of hardware-oblivious operators, which can run efficiently on both CPUs and GPUs. For the GPU part, database tables need to fit in the GPU memory of Ocelet, which severely limits the scalability of the system. HyPE [24] is a hybrid analytical engine utilizing both the CPU and the GPU for query processing. It analyzes the input query and system resources, and then decides a hybrid query plan in a way that can take the advantages of both the CPU and the GPU. YDB [103] is a GPU-based data warehouse query engine. It adopts column-based storage format and several compression methods to reduce the table size. Though YDB allows database store in the main memory or the SSD, it still assumes that the working set can fit in the main memory. To this extent, YDB does not improve the system scalability. YDB also has a problem of low GPU utilization, as the majority of time is spent on data preparation and transfer. Based on YDB, MultiQx[94] improves the throughput of YDB by allowing concurrent running of multiple queries. HippogriffDB differs from the previous work as HippogriffDB is targeting large scale database systems (TB scale input). HippogriffDB allows database table storage on the secondary storage device. To cope with the limited GPU memory capacity, HippogriffDB introduces streaming database operations which enable data processing on small chunks.

Several CPU-based databases also use block-oriented execution model [23]. In CPU-based solution, the system bottleneck is the limited bandwidth of the memory bus and the large intermediate results of the materialization process. Hence, they use a cache-aware approach so that the intermediate results can fit into the cache and can be reduced by the following highly-selective operators. However, the motivation in CPU-based databased does not hold for a GPU-based system, as the GPUs are advertised to have massive memory bandwidth (100s GB/sec). Our blocks are used to remove the scalability limitation of the existing query models used in existing GPU-based data analytics. When choosing the block size, we adopt the size that is large enough to deliver good I/O bandwidth from the SSD to the GPU, which is much large than the cache size (10s KB on the GPU).

Compression is a popular strategy to reduce the storage space and the amount of data transfer. Several works [37, 73, 70, 103] discussed the usage of compression on GPU-related database system. YDB [103] supports dictionary and run-length encoding compression for database tables. The compression helps reduce the space overhead in YDB, so that YDB can support tables which are lightly larger than the GPU memory capacity. In [37], Fang et al. studied the performance of various compression and decompression algorithms on GPU. [73] studied the performance of variable length encoding method and reached the conclusion that variable length encoding, such as Huffman encoding, is not efficient on GPU compared with on CPU, due to the random access in the decompression stage. HippogriffDB differs from the previous work in that HippogriffDB supports more compression methods and creates multiple compressed versions for each table using different strategies. When a query arrives, HippogriffDB will decide the one with the best compression ratio and send it to the query processing unit. In short, HippogriffDB uses the query-adaptive compression. Wu et al. [99] proposed a primitive fusing strategy to reduce the back-and-forth traffic between GPU and hosts. HippogriffDB adopts a similar technology to reduce the data exchange.

There are several related projects on the direct communication between two PCIe devices. For example, GPUDirect [3] offers direct communication between two GPUs and [52] offers direct communication between Network Interface Card (NIC) and the GPU. Our work differs from those work in two ways. First, our work demonstrates that low I/O bandwidth from the SSD to the GPU is largely due to the failure to fully utilize the internal parallelism inside the SSD. To address this issue, we adopt multi-threaded I/O to boost the utilization of the multiple data transfer units. Second, our work offers direct communication between a GPU and a PCIe SSD.

Several works [92, 82] discussed the gap between throughput of GPU kernel and off-chip memory bandwidth and proposed compression and decompression to alleviate discrepancy. HippogriffDB differs from these works in two aspects. First HippogriffDB tries to reduce the gap between between GPU kernel and SSD I/O throughput. Second, HippogriffDB achieves

better compression ratio by using aggressive and adaptive compression strategies.

2.8 Conclusion

In this paper, we proposed HippogriffDB, an efficient, scalable heterogenous data analytics system. HippogriffDB is the first GPU-based data analytics that can scale up to support terabyte input. HippogriffDB reaches high performance by fixing the huge imbalance between GPU kernel and I/O using compression and peer-to-peer transfer path. HippogriffDB uses a stream-based query processing model to process data sets larger than the GPU memory. Our comprehensive experiments have demonstrated the superiority of HippogriffDB in terms of both scalability and performance.

Chapter 2 contains materials from "HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics", by Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou and Steven Swanson, which appears in Proceedings of the VLDB Endowment, Vol. 9, No. 14. The dissertation author was the primary investigator and first author of this paper.

Chapter 3

SoftFlash

3.1 Background

This section provides some background knowledge about modern big data analytical system and storage format, as well as the motivation to develop SoftFlash to overcome the limitations in existing system.

3.1.1 Apache Hive

Apache Hive[5] is an OnLine Analytical Processing (OLAP) system that runs on top of Apache Hadoop[6] to answer multi-dimensional queries and analysis. Hive supports a SQL-like query language, called HiveQL, to query database systems that reside on Hadoop's HDFS or other compatible file systems. While traditional SQL queries must be implemented manually with MapReduce APIs to be executed on a distributed platform, Hive provides native support to transparently maps HiveQL queries into computation frameworks or jobs, such as MapReduce or Spark, increasing the scalability of data analytic in a natural manner. A wide range of companies, including Facebook, Amazon and Netflix, are actively developing, maintaining and utilizing Hive[7].

Hive supports multiple table storage formats, including raw text format, RCFile and ORC[8]. RCFile (Record Columnar File) is a data format designed for MapReduce-based data warehouse systems. It improves the storage performance significantly over raw text format by

combining the advantages of both row-store and column-store. By using the combination of row and column formats, it reaches a better balance between fast data reading and analyzing, storage space efficiency, and adaptability to dynamic data access patterns. ORC format, as its name implies, is an optimized RC format that aims to massively speed up query processing and reduce file sizes.

3.1.2 ORC

Introduced in Hive version 0.11, the Optimized Row Columnar (ORC) file format provides a highly efficient way to speed up query processing and to reduce file sizes. ORC format helps improve performance in both analyzing and appending data.

ORC is a self-describing file format that combines the advantage of both row and column stores. It supports efficient large streaming reads, as well as fast random row access as well. ORC files organize data groups into stripes that are around 64 MB by default. The stripes inside each file are independent, inherently supporting query processing in a distributed manner. Within each strip, ORC files adopts column-oriented format and reader can read just the columns it needs. The columnar storage scheme that ORC adopts enable the reader to read, decompress and process exact the data that are necessary for query execution, reducing a great amount of redundant data transferring. The type-awareness of ORC format makes the appending process more compression-friendly as the type knowledge provides additional information to build efficient compression and indexing optimizations on the original data.

The ORC format also facilitates an important database optimization: predicate pushdown. By using the indexes information in the ORC format, the physical operators can decide whether to skip a stripe in a file for a given query. The ORC format provides both coarse and fine grain indexing for different skip granularity.

Many large companies use ORC. Facebook manages to save tens of petabytes in their data warehouse by using ORC and demonstrates the superiority of ORC over RCFile[8]. Yahoo also uses ORC to store their production data and their benchmark results shows significant

performance improvement.

3.1.3 Motivation

While the ORC format provides an efficient support, there are still several challengers for implementing predicate pushdown in cloud environment efficiently. A typical cloud architecture consists of compute nodes and storages nodes. Unfortunately, database storage and query execution happen on different types of machines: storages nodes manage database table storage and the query exsection engine, which runs on the compute nodes, needs to fetch the entire table from the storage nodes. In other words, before the query execution engine preforms the predicate pushdown on the compute node, a significant amount of data transferring already happened over the network.

The waste of network bandwidth motivates us to design SoftFlash, a Hive-based analytical system that optimizes the network traffic and server-side computation intensity by offloading computations closer to storage. Different from existing projects, SoftFlash implements database offloading based on modern (encoded and compressed) analytical formats. It takes advantage of hardware accelerators, as well as computation power inside storage devices, to provide high performance data filtering at the storage level.

3.2 System overview

In this chapter, we present SoftFlash, a Hive-based analytical system. SoftFlash optimizes the network traffic and server-side computation intensity by offloading computations closer to storage. SoftFlash is the first system that implements database offloading on the modern data storage formats. It takes advantage of hardware accelerators, as well as computation power inside storage devices, to provide high performance data filtering at the storage level. It also optimize the computation intensity and network bandwidth via various software approaches.

This section provides an overview of the system design, operator pushdown, in-SSD processing and server-storage communication.

3.2.1 Hive in cloud environment

Conventional compute model has been computation-centric: it moves the entire dataset from storage node to the compute node. Though it benefits a wide range of computation-intensive workloads, it also put pressure on the interconnect as it migrates large amount of data from the storage node to compute node. For typical data analytical queries, such as datawarehouse queries, a large amount of early filtering could be done at very early stage, opening opportunities for reducing interconnect traffics. As shown in Figure 3.1, even with the predicate pushdown turn on, Hive still needs to send the entire database table from the storage nodes to the compute nodes, consuming previous network bandwidth.

In this work, we optimize query processing by migrating computation from compute node to storage node, as opposed to moving data in the opposite direction. We utilize hardware accelerators for heavy decompression task and the local programmability for light-weighted decoding and filtering tasks. In this way, we significantly reduce the amount of data transferred via interconnect, as well as improve the scalability of the data analytical system. Our system deliver high performance data analytics by successfully pipelining the decompression, filtering and query execution stage. Experiments on synthetic and real datasets shows improvement over various baselines.

3.2.2 System architecture

SoftFlash targets modern data analytics and utilizes in-SSD computation power to reduce data traffic over interconnect. Figure 3.3 provides an overview of the system architecture. SoftFlash contains the following major components:

Data analytical engine. SoftFlash aims to reduce the amount of data transferred from storage to compute code for modern data analytical systems. In the current implementation, SoftFlash uses Apache Hive[5] as the data analytical engine on the server side. To offload predicates into the storage layer, SoftFlash takes advantage of the predicate pushdown feature in

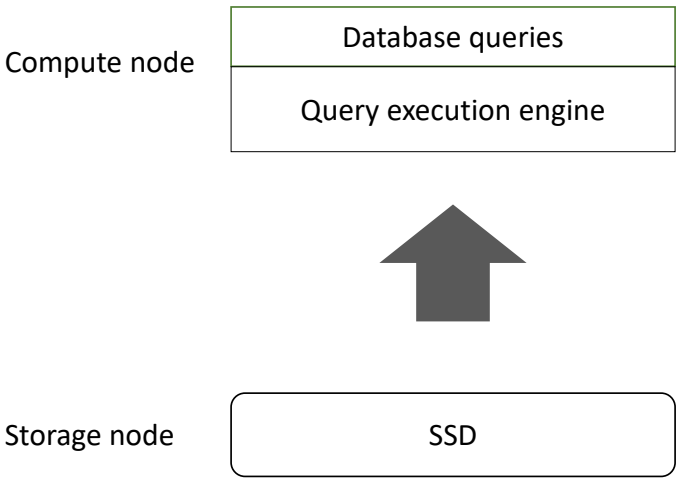


Figure 3.1. Traditional Hive architecture. Traditional computing architecture sends the entire database table from storage nodes to compute nodes.

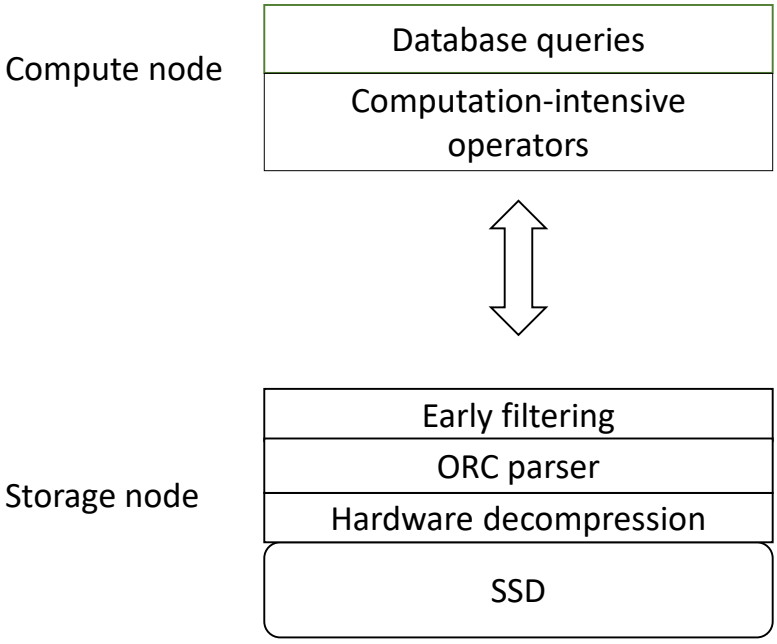


Figure 3.2. Data centric model. Data centric model filters at the storage level, reducing the data traffic between storage and compute nodes.

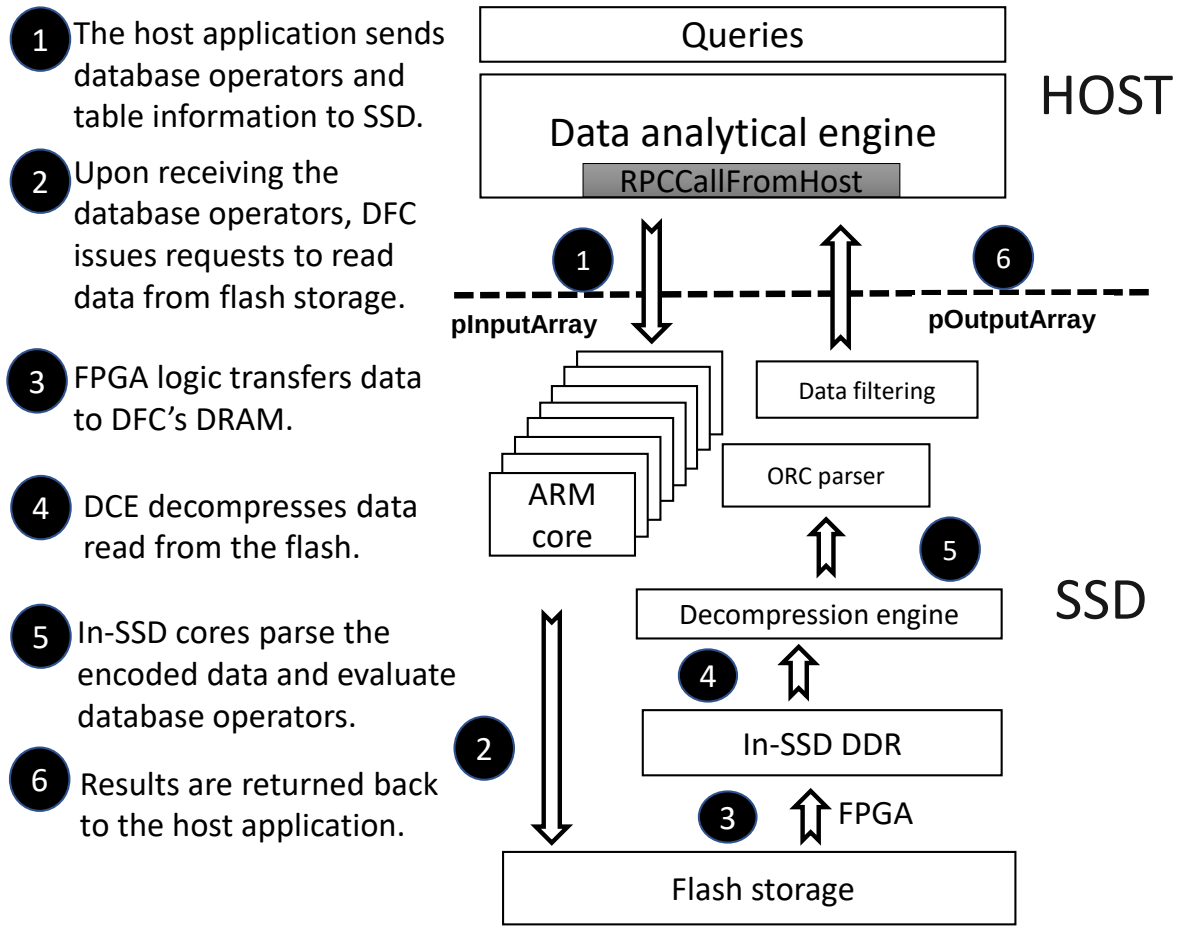


Figure 3.3. System architecture of SoftFlash. SoftFlash offloads data filtering operations to the storage layer, reducing the network overhead significantly.

Hive and extracts the filtering conditions of the a given query. SoftFlash then sends the predicates down to storage over the network.

Network interconnect. SoftFlash offloads predicates of a given query to the storage layer, and the storage layer sends filtered data back to the server side. SoftFlash reduces the network overhead by using efficient serialization/de-serialization formats and amortizing network overhead via multi-batch data transfer.

Hardware accelerators. SoftFlash works on modern data analytical formats, which involves certain kinds of encoding and compression schemes. SoftFlash takes advantage of the hardware accelerators at the storage level to overcome the heavy computation resources requirement for data decompression.

Predicate evaluation. The storage layer in SoftFlash evaluates the predicates and only outputs the filtered data back to the host. SoftFlash uses the in-device computation power for the predicate pushdown. To alleviate the limited computation resource inside the storage device and reach better performance, SoftFlash takes advantage of database indexes and implements skip function to filter out unwanted data in chunks.

3.2.3 Data flow

We show the data flow for a given query in Figure 3.3.

1. Given a query, SoftFlash turns on the predicates pushdown feature in Hive so that all the predicates appears as leaf nodes in the query plan tree. The server side then serializes the predicates and sends the predicates to the storage device via a RPC call.

2. Upon receiving the predicates information, the storage device analyzes the information specified by the predicates and fetches corresponding data from the flash storage.

3. The memory controller transfers data from the flash memory to the on-device DRAM.

4. DCE (Data compression/decompression engine) decompresses the original data format.

5. SoftFlash uses the on-device ARM cores for decoding and predicate evaluation. SoftFlash first probes the database index to perform coarse-grained filtering and skips the unwanted data in chunks. It preforms record-level filtering after that.

6. SoftFlash sends the filtered results back to the server side. SoftFlash maintains a buffer on the on-device memory and sends multiple batches of data in a single network communication to amortize the cost.

3.3 Operation pushdown

For high-selective data analytics, pushing filtering close to storage can reduce a large amount of data traffic over the interconnect. As modern data analytical engine usually store data in compressed and encoded formats, SoftFlash first utilizes hardware accelerators and on-board programmability to decompress and decode data before applying any filtering operations.

In this section, we first introduce how we use hardware accelerators and the cores inside SSD to prepare data for further predicate evaluation. After that, we discuss the filtering push down in SoftFlash.

3.3.1 Hardware data decompression

Modern data analytical systems store data in compressed formats to save space and reduce data transfer overhead. Hive supports two compression schemes: SNAPPY and ZLIB. In SoftFlash, we focus on the ZLIB compression format and we utilize the hardware decompression engine to provide high-performance decompression operator.

3.3.2 Data decoding

Modern data analytical systems also use data encoding to reduce the space cost. ORC is a widely-used format for Hive applications.

SoftFlash supports ORC decoding at the storage level. Based on an open-source ORC decoder, SoftFlash implements the functionalities to decode ORC format for filtering operations at the later stage. As SoftFlash adopts a daughter SSD as the storage system, it defines the interfaces to access the storage in the decoding software.

To better affiliate fast filleting and to avoid additional decoding cost, SoftFlash implements skip and seek functions. If SoftFlash decides that a certain block does not contain any records that meet the filtering conditions, it can skip the block to avoid reading, decompression and decoding unnecessary blocks. SoftFlash also supports efficient seek to go over multiple unneeded blocks.

3.3.3 Filtering pushdown

SoftFlash pushes the predicate evaluation down to storage node so that the query execution engine on the compute node process only the values that the current query needs.

SoftFlash applied two layers of filtering: batch-level filtering and record-level filtering. In the ORC format, there is a RowIndex and a Bloomfilter index for each group of 10,000 rows. For range queries, SoftFlash uses the RowIndex to affiliate filtering. The RowIndex contains metadata, such as min/max of a given group. The intersection of the query range and data range can decide if there are any possible rows that meet the criteria. For point queries, SoftFlash first uses RowIndex to perform a coarse-grained evaluation and then uses Bloomfilter index to perform fine-grained filtering.

SoftFlash can also perform row level filtering by evaluation the predicate on a row by row basis. To improve the performance of row level filtering, it uses SIMD instructions to accelerate the evaluation process. SoftFlash also conducts some aggressive low-level data structure and algorithm optimization for this part.

SoftFlash uses the seek() and skip() functions to jump over those blocks that do not meet the predicate. In this way, SoftFlash saves significant amount to time in reading, decompression and decoding efforts.

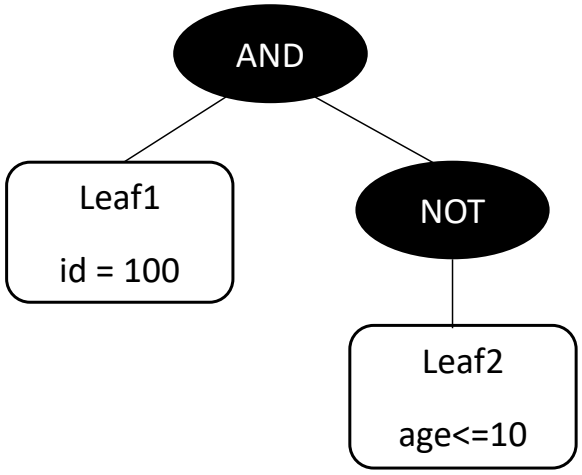


Figure 3.4. Tree structure of a Sarg object. This example denotes "id = 100 AND age $\leq$ 10"

3.4 Hive

SoftFlash uses Apache Hive as the server-side analytical engine, which pushes down the filtering operators in the query plan tree. SoftFlash then offloads the filter conditions to the storage and collects filtered data from it. This section first present the filtering operator formats and the serialization/deserialization of the predicates. It then discusses the implementation of the RPC communication between the server and the storage side.

3.4.1 Predicate pushdown

SoftFlash takes advantage of the query plan generator in Hive to generate a pan where predicate operators are all located at the leaf nodes. Hive internally uses a search argument class to preresnt the predicates. In Hive, a Sarg object is represented in a tree-struct. For example, "id = 100 AND age $\leq$ 10" is represented as in Figure 3.4:

To serialize this tree structure, we turn it into a linear representation using the following rules:

1. Turn the tree structure into a linear representation using Polish notation. For example, the tree above will re-write as AND Leaf1 NOT Leaf2. Using the encoding table below, it will

be translating into “1 16 2 16” in the serialization process (we do not distinguish leaves here).

AND	OR	NOT	Leaf
1	0	2	16

2. The contents of each leaf will follow in the formats specified below: list of (op_code, literal), list of column_id. In the example, leaves will be translated into “EQUAL 100 LESS_THAN_EQUAL 10 Col_id Col_age”. Using the encoding table below, it will be further translated into “0 100 2 10 1 2” (assuming column id and age are the first two columns)

EQUAL	*LESS_THAN*	*LESS_THAN_EQUAL*
0	1	2

At the storage side, when receiving the serialized format, the device uses the on-device cores to de-serialize the byte array to reconstruct the tree structure. The de-serialization function is called at the beginning of a ORC reader program and will be called only once.

3.4.2 RPC communication

To reduce the compute pressure to a minimum for the storages nodes and to save the computation resources, SoftFlash chooses to manage the data traffic between compute nodes and storage nodes itself, as opposed to running HDFS on the storage node. SoftFlash uses RPC over network to send predicate to and receive candidate values from storage nodes.

SoftFlash setups an RPC server/client on storage/compute nodes respectively and defines two RPC functions. The first one is to send predicate from Hive to storage node. When the storage node receives the predicate, it will evaluate the predicates and prepare candidate data that meet the selection criterion. The application running on the storage node keeps the candidate data in a circular buffer. The second RPC will get those selected values from the circular buffer

```
service {
   rpc GetSargArray(SargByteArray)
         returns (null) {};
   rpc GetBatchArray(null)
         returns (RowBatchByteArray) {};
}
```

Figure 3.5. Hive and SoftFlash communicates with the two RPC interfaces.

and the query execution on the compute node will proceed with received data. To support better concurrency between multiple query executions, SoftFlash uses asynchronous interface for both RPC functions.

3.5 Results

3.5.1 Experimental methodology

We build SoftFlash and a testbed that contains a server equipped with Intel Xeon processor and two PCIe-attached programmable SSD. We evaluate SoftFlash using both macrobenchmark and microbenchmark and compare SoftFlash with original Hive. This subsection describes our test bed, benchmark applications, and the measurement metrics we use to evaluate SoftFlash.

We currently run our experiments on a server with an Intel Xeon E52609V2 processor. The processor contains 4 cores and each processor core runs at 2.5 GHz by default. The server contains 64 GB DDR3-1600 DRAM that we used as the main memory in our experiments. The testbed uses a Linux system running the 3.16.3 kernel. The programmable SSD we currently use equips with eight cores in it, each of which can operates at up to 1.8 GHz. The SSD contains 16 GB memory space on board. A modified Ubuntu operating system is running inside the SSD. We connect a M.2 daughterboard as storage in our system, which can deliver up to 2 GB/sec read bandwidth.

We use both macro- and micro- benchmark to test the performance of SoftFlash. We use both the LINEITEM table defined in TPC-H benchmark and synthetic tables. We modified

the LINEITEM table slightly by turning date and decimal numbers into integer numbers. For synthetic tables, we populate tables of multiple columns with randomly generated integers. We test queries from TPC-H and SSBM, as well as synthetic queries. We vary the data distribution and query selectivity to test the SoftFlash performance.

Our experiments include some simulation results based on the SSD I/O bandwidth assumption.

3.5.2 Network traffic reduction

The primary goal of SoftFlash is to reduce the data traffic over interconnect. SoftFlash achieves this by offloading filtering to the storage side. This subsection evaluates the network traffic reduction with micro- and macro- benchmarks.

Figure 3.6 shows the effect of traffic reduction with various queries. SoftFlash can reduce over 90% data transferring between the storage nodes and the compute nodes. SoftFlash achieves it by implementing filtering locally. The effect of traffic reduction improves with the increment of selectivity, as SoftFlash only sends the data trunks that are selected while the baseline sends the entire table. SoftFlash performs well on the synthetic dataset, as the data range in this workload is wider and the benefit of compression and encoding is less with regards to the space cost.

3.5.3 Query execution time

SoftFlash can also improve query execution time. First, it adopts high performance hardware accelerators to decompress data, which is around five times faster than the Intel processor we use. Second, it reduces the data traffic, which saves query execution time as well.

Figure 3.7 reflects the improvement of query execution. SoftFlash mangers to improve the query execution time by up to 2.5×. SoftFlash performs better on the microbenchmark, for two reasons. First for this query, the traffic reduction effect is better than the other two queries. Second, the filtering condition for this query is simpler and hence the processors on the SSD can evaluate the predicates more efficiently.

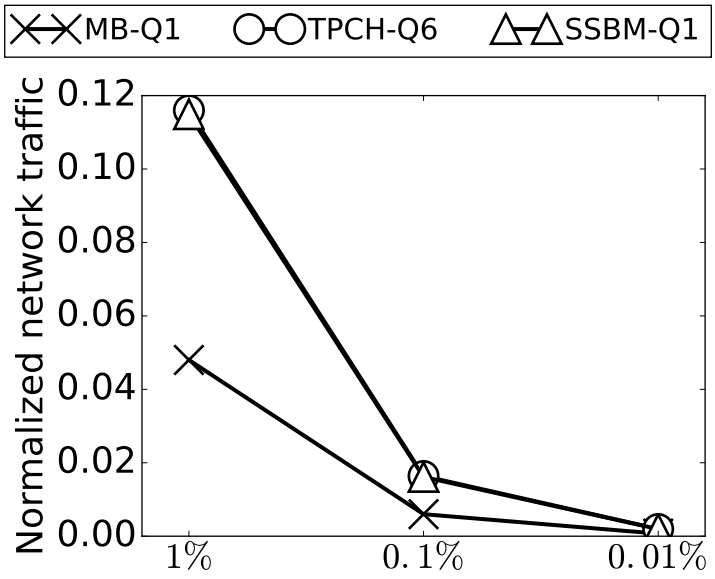


Figure 3.6. SoftFlash reduces the data traffic over interconnect by over 90%.

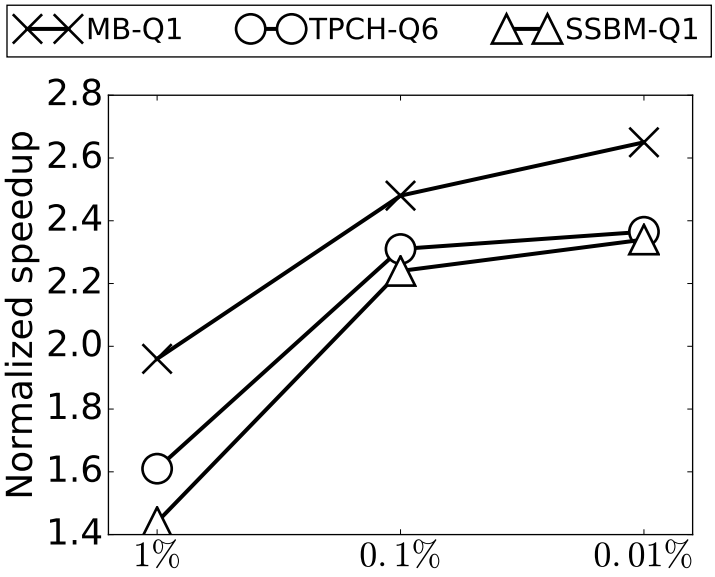


Figure 3.7. SoftFlash improves query processing ability by up to 2.5×.

3.6 Related work

Recently, high-end SSDs begin to embed controllers with multiple programmable cores. Programmability integrates into more and more storage devices, opening new opportunities to offload the computation to components other than GPU. Equipping the storage hardware with computability was purposed more than a decade ago, however, with the relatively slow processor unit and low internal bandwidth, such idea does not raise enough attention at that time. However, with the emergence of fast storage and technical advances, cur- rent storage device can equip powerful processor units and provide higher internal bandwidth, making more sense for offloading operations down and processing dataflow in the storage device.

Active Disk[79] is one of the first attempts to carried on the idea of on-disk processing. By associating each disk with a processor, these local processors can process data on disk without transferring to the CPU. Active disk requires operating system support on both host and disk side. The host operating system is responsible for installation of disk programs called disklets and management of host resident streams, meanwhile on the disks there should be a disk OS which provides three services including disk memory management, stream communication with host and other disk processors and scheduling of disklets. According to the simulation, SQL selection and simple aggregations can benefit great from ActiveDisk.

Smart SSDs[31] are flash storage devices that integrate memory and computation ability (ARM core) inside it. Such design makes it possible to run programs inside a storage device. Jaeyoung et al.[31] explored the possibilities of associating the Smart SSD with relational analytic query processing. They implemented a very basic prototype of database system running on a Smart SSD. The operations they pushed down to Smart SSD include selection and aggregation. Their results shows that pushing-operation-down technic can significant increase the performance (by up to 2×) and reduce power consumption for the query processing (up to 60% reduction). Even though they didnt discuss about the internal mechanism inside the hardware, the fact that they only push down some simple operations implies the hardness to program using SmartSSD.

The database built on Smart SSD by Jaeyoung et al. is a very basic and preliminary step. First, the operations they discussed is very limited (only aggregation and selection), omitting some very important database operations like join. Besides, the aggregation they discussed is also not the general aggregation(withoutGROUP BY).

Willow[85]. The internal storage of Smart SSD is NAND-flash and the connection between storage and host is via SATA. Its more interesting to see the similar mechanism but with more advanced storage technology like PCM and data bus like PCIe. Willow[85] is a programmable PCM-based SSD device, designed by NVSL group at UC San Diego. Like Smart SSD, Willow offers computation capability (MIPS core) inside the storage device to process data without sending it to the host. Figure 6 shows the architecture of Willow. Each controller is connected to a 32- bit MIPS-based CPU called processing unit (PU) with 16 KB local on-chip SRAM. Multiple PUs can operate on the data in parallel before the intermediate results are sent on to the ring. Experiment shows by offloading the database recovery part to storage device, the system throughput can improve by 50%. Pushing operation down to the storage side seems to be a very good idea at the first glimpse: it significantly save the bandwidth as large portion of unnecessary data transfer is avoided.

However, in the PCM and PCIe scenario, considering its no longer extremely expensive to read and transfer data from storage side, the benefit from saving bandwidth could be easily overacted by the divergence of Host CPU and SSD-PU. My experiments shows even for some simple streaming operation (like aggregation), theres no performance gain, not to mention those computation-intensive workloads such as join and sort. The main reason for that is that the SSD-PU is significantly slower than CPU.

However, such result doesnt indicate the programmable storage makes no sense, rather it implies the computation ability for the storage associated PU is very important when facing the new-generation storage device. We are looking forward to some storage device with more powerful computation units. Another direction is combination of GPU and computable SSD and in this case the data transforming between the SSD and GPU is an issue to deal with.

The three works discussed above show a few similarities. Apart from the same essential idea, the cores embedded on the storage device are all rather slow cores, in terms of frequency. The difference comes from the underlying storage device and the throughput of bus connecting host(CPU)side and storage side. The more discrepancy of internal and external bandwidth, the greater of the benefit of ondisk processing. With the rapid improvement of external bandwidth and relatively slow on the internal side, operations that show highly selectivity would benefit most from such idea.

Chapter 3 contains materials from “SoftFlash”, by Jing Li, Jae Young Do, Sudipta Sengupta and Steven Swanson, which will be submitted for publication. The dissertation author is the primary investigator and first author of this paper.

Chapter 4

On the Energy Overhead of Mobile Storage Systems

4.1 The Case for Storage Energy

Past studies have shown that storage is a performance bottleneck for many applications [51]. In particular, background applications such as email, offline messaging, file synchronization, updates for the OS and applications, and certain operating system services like logging and bookkeeping, can be storage-intensive. This section devises estimates for the proportion of energy that these applications spend on each storage subsystem component. Understanding the energy consumption of storage-intensive background applications can help improve the standby times of mobile devices.

We use hardware power monitors to profile the energy requirement of real and synthetic workloads. Traces, logs and stackdumps were analyzed to understand where the energy is being spent.

4.1.1 Setup to Measure Energy

An Android phone and a Windows RT tablet were selected for the storage component energy consumption experiments. While these platforms provide some OS and hardware diversity for purposes of analysis and initial conclusions, additional platforms would need to be tested in order to create truly robust power models.

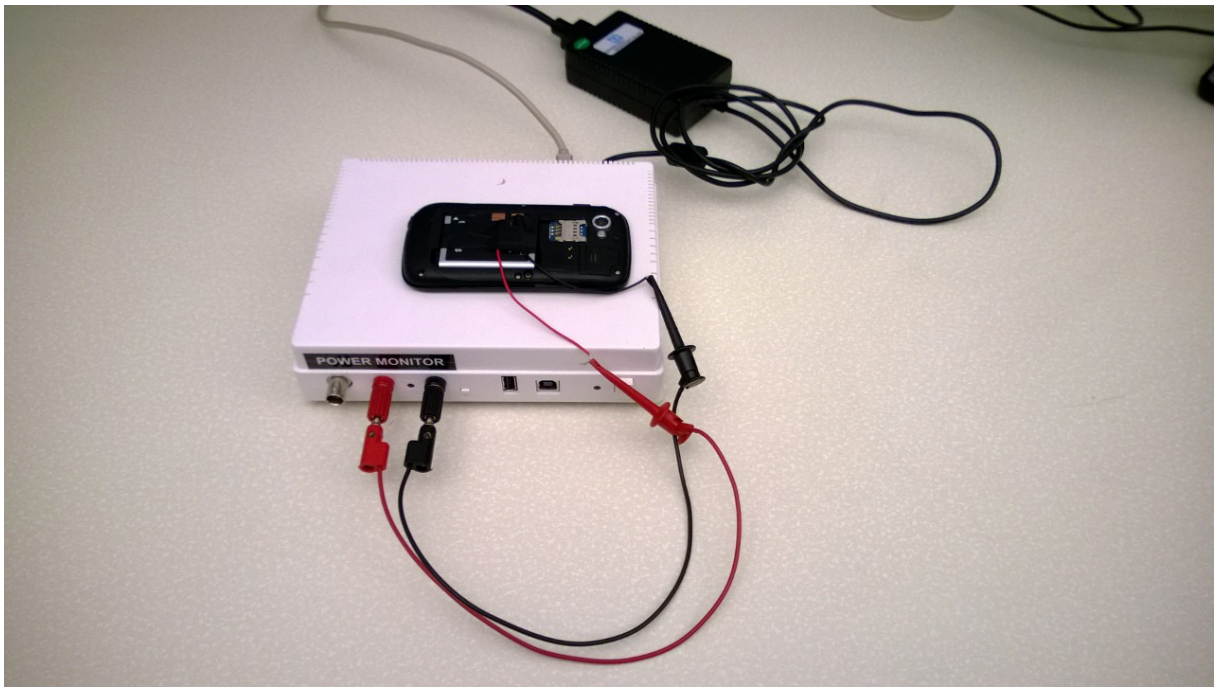


Figure 4.1. Android 4.2 power profiling setup: The battery leads on a Samsung Galaxy Nexus S phone were instrumented and connected to a Monsoon power monitor. The power draw of the phone was monitored using Monsoon software.

Android Setup

The battery of a Samsung Galaxy Nexus S phone running Android version 4.2 was instrumented and connected to a Monsoon Power Monitor [66] (see Figure 4.1). In combination with Monsoon software, this meter can sample the current drawn from the battery 10's of times per second. Traces of application activity on the Android phone were captured using developer tools available for that platform [16, 17].

Windows RT Setup

Two Microsoft Surface RT systems were instrumented for power analysis. The first platform uses a National Instruments Digital Acquisition System (NI9206) [67] to monitor the current drawn by the CPU, GPU, display, DRAM, eMMC storage, and other components (see Figure 4.2). This DAQ captures 10's to 1000's of samples per second.

Figure 4.3 shows the second Surface RT setup, which uses a simpler DAQ chip that

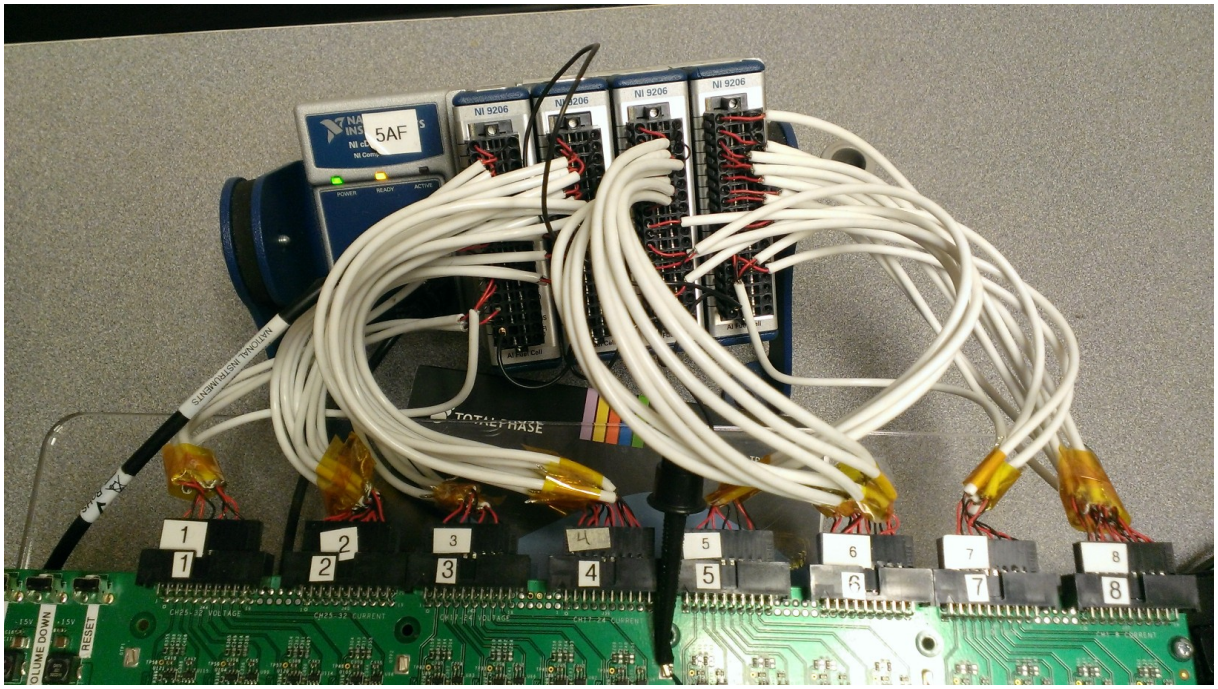


Figure 4.2. Windows RT 8.1 power profiling setup #1: Individual power rails were appropriately wired for monitoring by a National Instruments DAQ that captured power draws for the CPU, GPU, display, DRAM, eMMC, and other components.

Figure 4.3. Windows RT 8.1 power profiling setup #2: Pre-instrumented to gather fine-grained power numbers for a smaller set of power rails including the CPU, GPU, Screen, Radio, eMMC, and DRAM.

captures the current drawn from the CPU, memory, and other subsystems 10's of times per second. This hardware instrumentation is used in combination with the Windows Performance Toolkit [97] to concurrently profile software activity.

Software

Storage benchmarking tools for Android and Windows RT were built using the recommended APIs available for app-store application developers on these platforms [18, 98]. These micro-benchmarks were varied using the parameters specified in Table 4.1. The cache is warmed by reading the entire contents of a file small enough to fit in DRAM at least once before the actual benchmark. The cache is made cold by rebooting the device. The write-back operation is benchmarked using a small file that is cached in DRAM such that it does not trigger a background flush to secondary storage. Such a setting enables us to estimate the energy required for writes to blocks that are cached. Each microbenchmark was run for one minute, and large ranges of sectors were accessed so as to minimize read hits in any in-memory block or file caches for "cold" experiments.

To reduce noise, most of the applications from the systems were uninstalled, and unneeded hardware components were disabled whenever possible (e.g., by putting the platform into airplane mode and turning off the backlight).

4.1.2 Experimental Results

The energy overhead of the storage system was determined via micro-benchmark and real application experiments. The micro-benchmarks enable tightly controlled experiments, while the real application experiments provide realistic IO traces that can be replayed.

Microbenchmarks

To test the energy consumption of eMMC on Windows RT (setup #1), we use Window .Net API to develop a program that can generate the synthetic workload listed in Table 1.

Table 4.1. Storage workload parameters varied between each 1-minute energy measurement.

Parameter	Value Range
Block Size (KB)	4, 8, 16 or 32
Read Cache Config	Warm or Cold
Write Policy	Write-through or Write-back
Access Pattern	Sequential or Random
IO Performed	Read or Write
Benchmark Language	Managed Language or Native C
Full-disk Encryption	Turned on and off

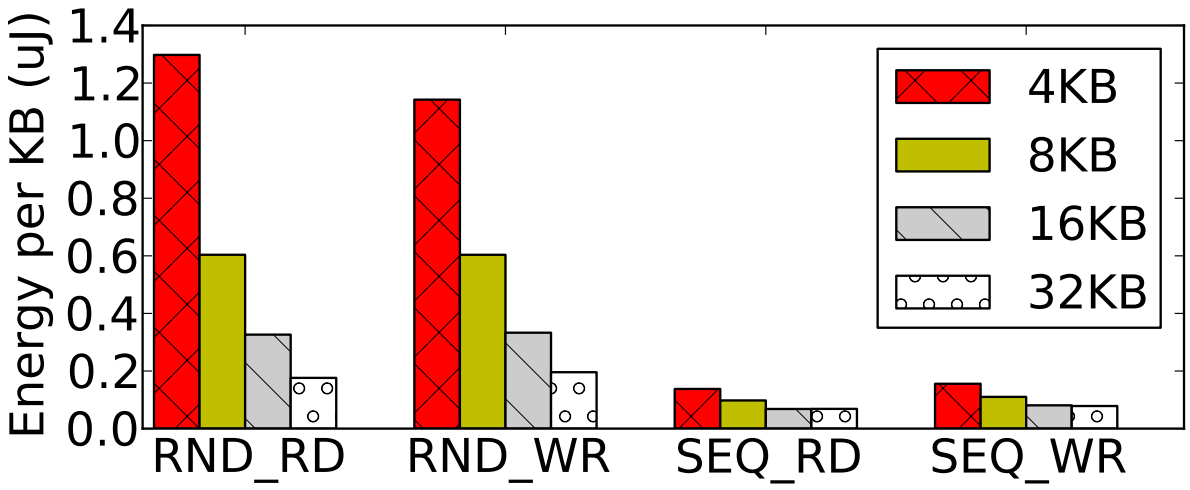


Figure 4.4. Storage energy per KB on Surface RT: Smaller IOs consume more energy per KB because of the per-IO cost at eMMC controller.

Figure 4.4 shows the amount of energy per KB consumed by the eMMC storage for various block sizes and access patterns on the Microsoft Surface RT.

- The eMMC device requires 0.1–1.3 μJ/KB for its operations. Sequential operations are the most energy efficient from the point of view of the device.
- Random accesses of 32 KB are similar energy-efficiency as sequential accesses. Smaller

random accesses are more expensive – requiring more than 1 μJ/KB.

From a performance perspective, the results were as expected; for a given block size, read performance is higher than write performance and sequential IO has higher performance than random IO.

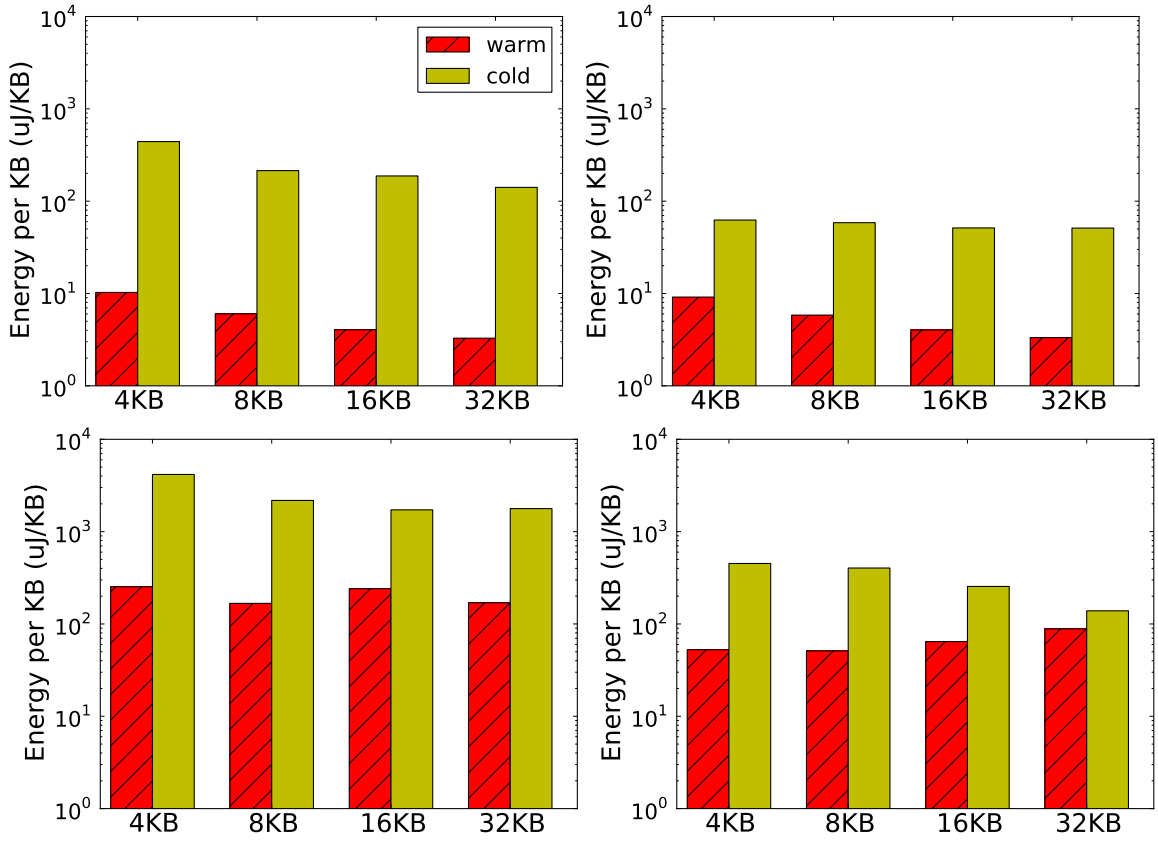


Figure 4.5. System energy per KB on Android: The slower eMMC device on this platform results in more CPU and DRAM energy consumption, especially for writes. "Warm" file operations (from DRAM) are 10x more energy efficient.

Experiment of synthetic workloads is also conducted on Android platform with the configuration described before. We developed benchmark programs written in Java that can generate various kind of IO requests.

Figure 4.5 shows that the energy per KB required by the entire Android platform is two to four orders of magnitude higher than the energy consumption by the eMMC on Surface RT that is the same generation as the eMMC used in the Android platform.

- Sequential reads are the most energy-efficient at the system level, requiring only 1/3 of the energy of random reads.
- Cold sequential reads require up to 45% more system energy.
- Writes are one to two orders of magnitude less efficient than reads due to the additional CPU and DRAM time waiting for the writes to complete. Random writes are particularly expensive, requiring as much as 4200 μJ/KB.

The impact of low-end SD cards on Android platform performance has been well studied by Kim *et al.* [51]. Low write performance, unfortunately, translates directly into high energy consumption for IO-intensive applications. We hypothesize that the idle energy consumption of CPU and DRAM (because of tail power states) contribute to this high energy.

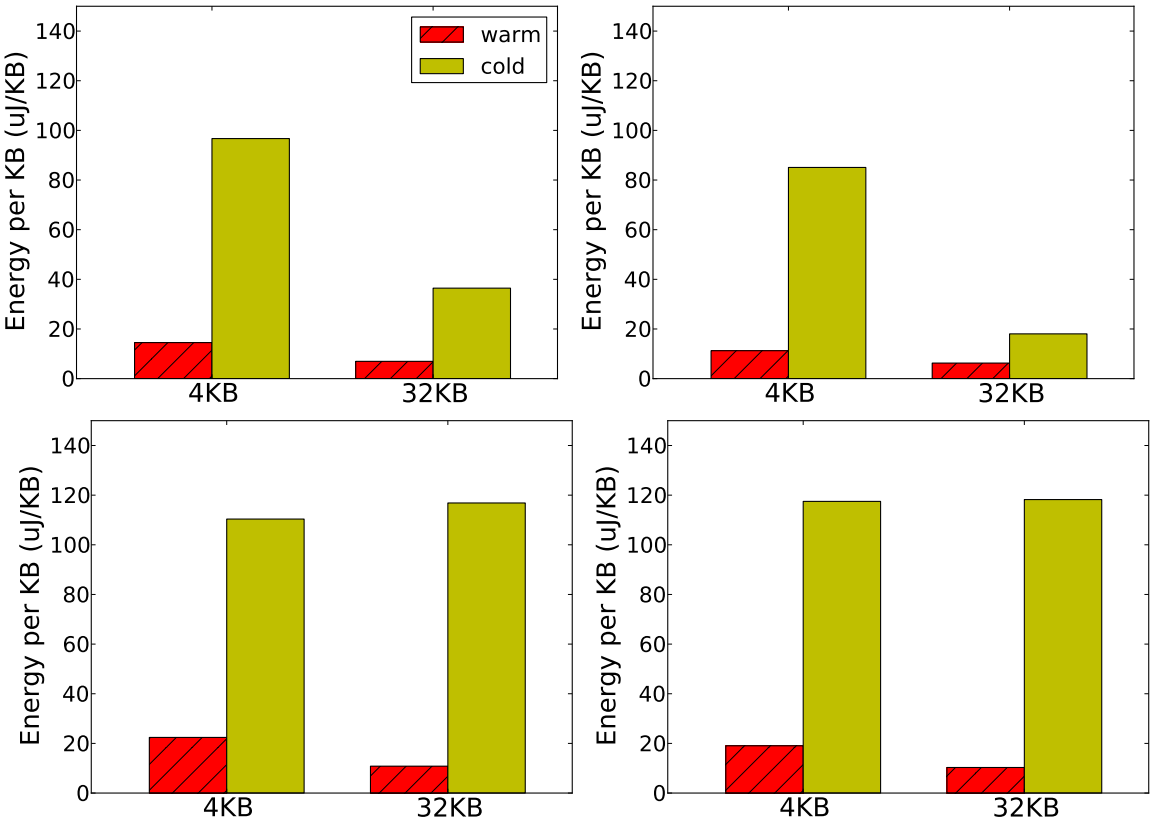


Figure 4.6. System energy per KB on Windows RT: The faster eMMC 4.5 card on this platform reduced the amount of idle CPU and DRAM time. "Warm" file operations (from DRAM) are 5x more energy efficient.

Table 4.2. Storage-intensive background applications profiled to estimate storage software energy consumption.

Email	Synchronize a mailbox with 500 emails totaling 50 MB.
File upload	Upload 100 photos totaling 80 MB to cloud storage.
File download	Download 100 photos totaling 8 0MB from cloud storage.
Music	Play local MP3 music files.
Offline messaging	Receive 100 offline messages.

We also developed a benchmark application to perform similar experiments on Windows RT (setup #2). Figure 4.6 presents the energy per KB needed for for the entire Windows RT platform. All "warm" IO requires less than 20 μJ/KB, whereas writes to the storage device require up to 120 μJ/KB. While some of this is the energy cost at the device, most of it is due to actual execution of the storage driver stack, as discussed later in this section.

Application Benchmarks

Disk IO logs from several storage-intensive applications on Android and Windows RT were replayed to profile their energy requirements. During the replay, OS traces were captured for attributing power consumption to specific pieces of software, as well as intervals where the CPU or DRAM were idle.

This paper focuses primarily on storage-intensive background applications that run while the screen is turned off, such as email, cloud storage uploads and downloads, local music streaming, application and OS updates, and instant messaging clients. Kim *et al.* [51] have reported that poor storage performance is often an application performance bottleneck for mobile platforms. Better understanding and optimization of the energy consumed by such applications would help increase platform standby time.

Table 4.2 presents the list of application scenarios profiled. All IO traces were taken on

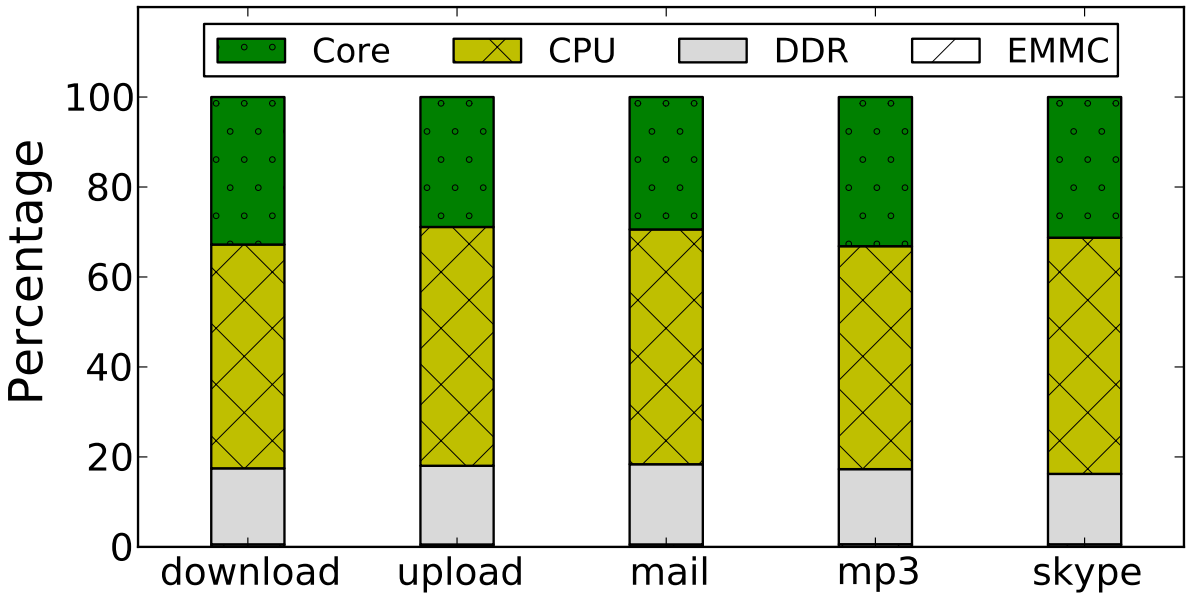


Figure 4.7. Breakdown of Windows RT energy consumption per hardware component. Software consumes more than 20x more energy than the eMMC device for storage-intensive background applications.

battery power with the screen turned off.

We run those applications on Windows RT (setup #2) and collect the trace during the meantime. After that we replay storage-related trace and power readings for individual hardware components are captured. Figure 4.7 plots the energy breakdown for eMMC, DRAM, CPU and Core. The "Core" power rail on the Surface RT SoC supplies the majority of the non-CPU components (GPU 2D and 3D engines, IO controllers, etc.).

The storage software consumes between 5x and 200x more energy than the storage IO

Table 4.3. Breakdown of functionalities with respect to CPU usage for a storage benchmark run on Windows RT. Other APIs include encryption. Overhead from managed language environment (CLR) and encryption is significant – comparable to the actual filesystem API itself.

Library Name	CPU Time Spent (%)
Filesystem APIs	49.7%
CLR APIs	25.8%
Other APIs	24.5 %

itself, depending on how the DRAM power is attributed. The fact that storage software is the primary energy consumer for storage-intensive applications is consistent with our hypothesis from the micro-benchmark data presented in the previous section.

Table **??** provides an overview of the stack traces collected on the Windows RT device using the Windows Performance Toolkit [97]. Overall, the CPU was 73.8% utilized. The majority of the CPU activity resulted from native filesystem (50%) and Common Language Runtime (26%) APIs. The CLR is the virtual machine on which all the applications on Windows RT run. While there was a tail of other APIs contributing to CPU utilization, the largest group was associated with encryption.

Energy overhead of native filesystem APIs has been studied recently [26]. However, the overhead from disk encryption (security requirement) and the managed language environment (privacy and isolation requirement) are not well understood. Security, privacy, and isolation mechanisms are of a great importance for mobile applications. Such mechanisms not only protect sensitive user information (e.g., current location) from malicious applications, but they also ensure that private data cannot be retrieved from a stolen device. The following section further examines the impact of disk encryption and managed language environments on storage systems for Windows RT and Android.

4.2 The Cost of Encryption

Full-disk encryption is used to protect user data from attackers who have physical access to a device. Many current portable electronics have an option for turning on full-disk encryption to help users protect their privacy and secure their data. BitLocker [22] on Windows and similar features on Android allow users to encrypt their data. While enterprise-ready devices like Windows RT and Windows 8 tablets ship with BitLocker enabled, most Android devices ship with encryption turned off. However, most corporate Exchange and email services require full-disk encryption when they are accessed on mobile devices.

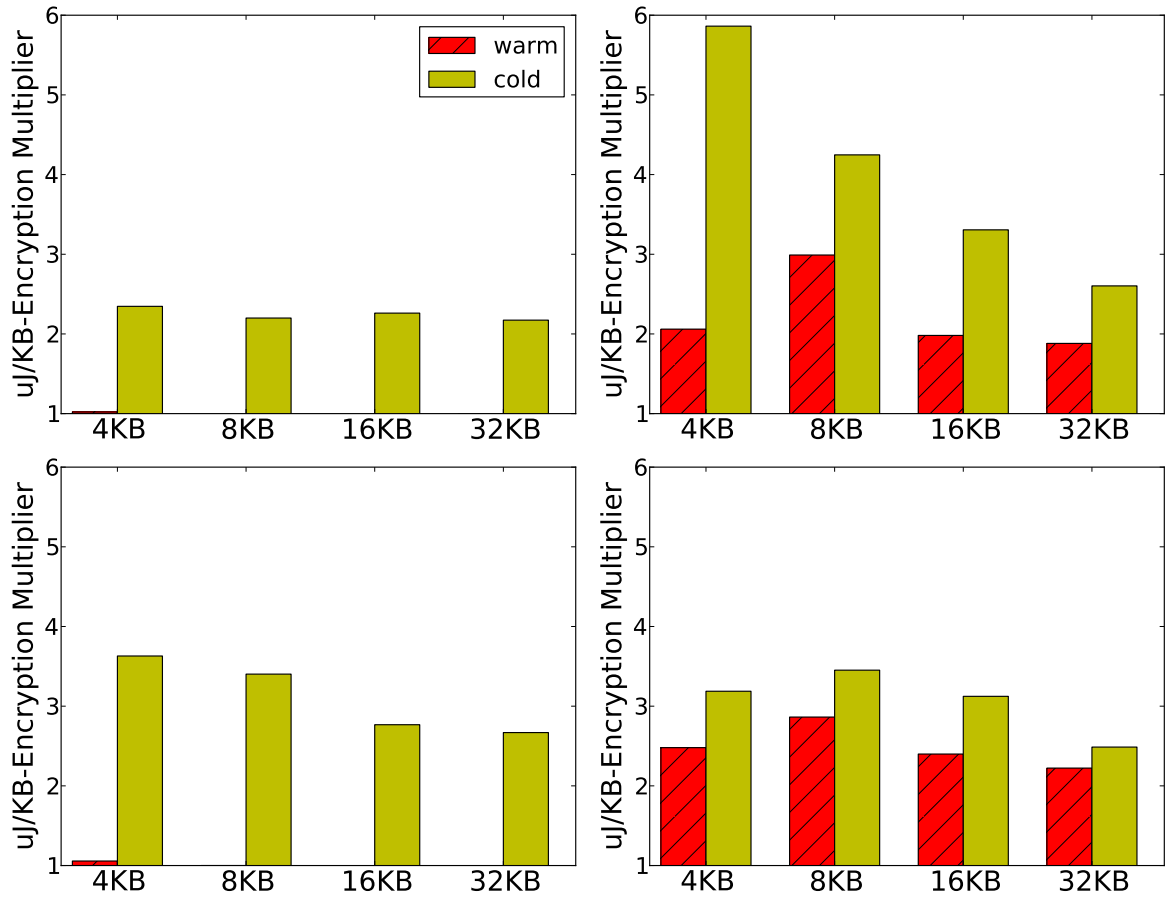


Figure 4.8. Impact of enabling encryption on Android phone: 2.6–5.9x more energy per KB.

Encryption increases the energy required for all storage operations, but cost has not been well quantified. This section presents analysis of various unencrypted and encrypted storage-intensive operations on Windows RT and Android.

Energy measurements were taken for workloads with variations of the first set of parameters shown in Table 4.1 as well as with encryption enabled and disabled while using the managed language APIs. The results are shown in Figures 4.8, and 4.9 for Android and Windows RT, respectively. Each bar represents the multiplication factor by which energy consumption per KB increases when storage encryption is enabled.

"Warm" and "Cold" variations are shown. As before, "Warm" represents a best-case scenario where all requests are satisfied out of DRAM. "Cold" represents a worst-case scenario where all requests require storage hardware access. In all cases except Android writes , "Warm"

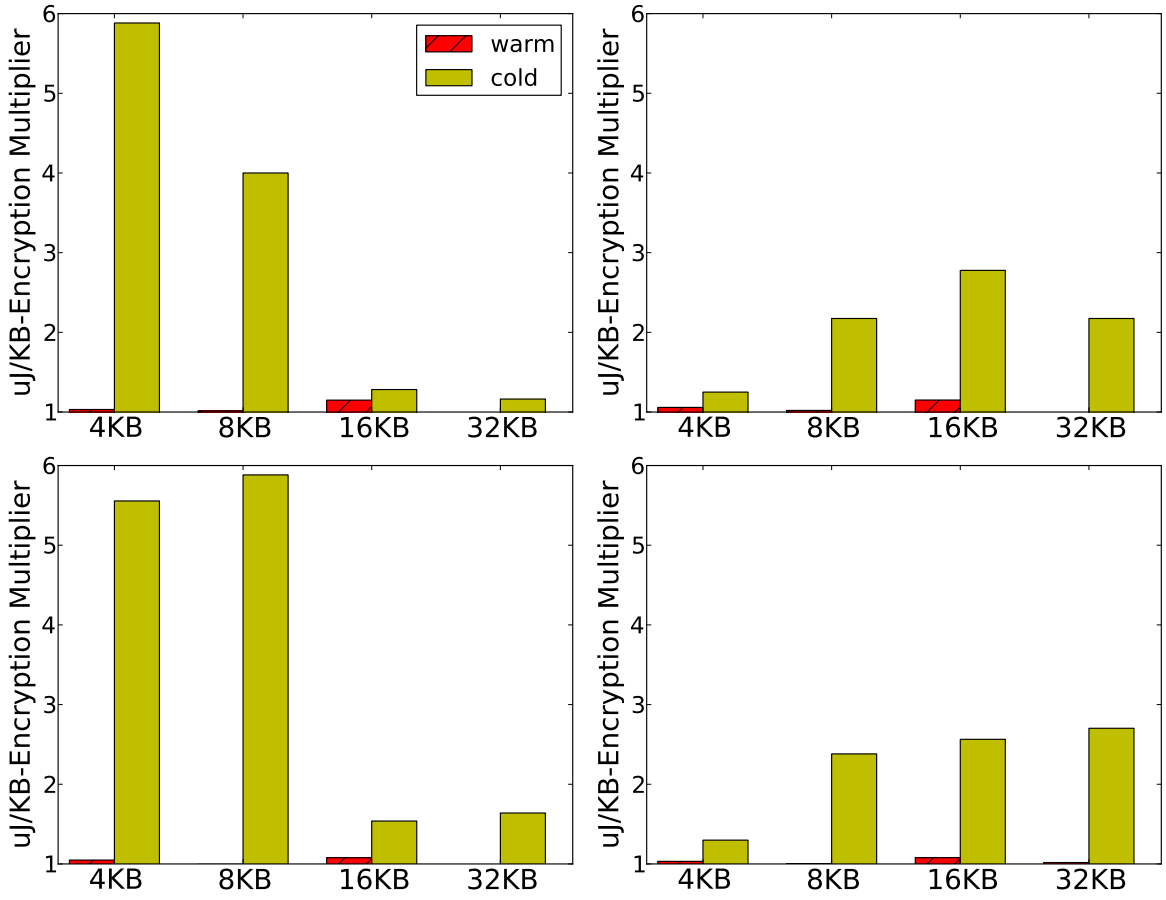


Figure 4.9. Impact of enabling encryption on Windows RT tablet: 1.1–5.8x more energy per KB.

runs have lower energy requirements per KB.

The cost of encryption, however, still needs to be paid when cached blocks are flushed to the storage device. Section 4.4 presents a model to analyze the energy consumption for a given storage workload for cached and uncached IO.

We illustrate the overhead of encryption with the following experiment. We compare the energy required when we turn on/off the encryption mechanism on Android device. Synthetic traces generated with Java code are adopted here and we vary granularity, cache setting, write policy and access pattern during the experiment. Figure 4.8 presents the encryption energy multiplier for the Android platform:

- The energy overhead of enabling encryption ranges from 2.6x for random reads to 5.9x for random writes.
- Encryption costs per KB are almost always reduced as IO size increases, perhaps due to the amortization of fixed encryption start-up costs.
- Android appears to flush dirty data to the eMMC device aggressively. Even for small files that can fit entirely in memory and for experiments as short as 5 seconds, dirty data is flushed, thereby incurring at least part of the energy overhead from encryption. Therefore, Android's caching algorithms do not delay the encryption overhead as much as expected. They may also not provide as much opportunity for "over-writes" to reduce the total amount of data written, or for small sequential writes to be concatenated into more efficient large IOs.

Similar experiment is reproduced on Windows RT platform (setup #2) using Windows API. Figure 4.9 presents the energy multiplier for enabling BitLocker on the Windows RT platform:

- The energy overhead of encryption ranges from 1.1x for reads to 5.8x for writes.
- The energy consumption correlation with request size is less obvious for the Windows platform. While increasing read size generally reduces energy costs, as was the case for the

Android platform, write sizes appear to have the opposite trend. All of the tested request sizes are fairly small, so we would expect that this trend would reverse as request sizes increased beyond 32 KB.

- DRAM caching does delay the energy cost of encryption for both reads and writes, even for experiments as long as 60 seconds. This could provide opportunity to reduce energy because of over-writes, and also due to concatenation of writes to form larger writes.

4.3 The Runtime Cost

Applications on mobile platforms are typically built using managed languages and run in secure containers. Mobile applications have access to sensitive user data such as location, passwords, intellectual property, and financial information. Therefore, running them in isolation from the rest of the system using managed languages like Java or the Common Language Runtime (CLR) is advisable. While this eases development and makes the platform more secure, it affects both performance and energy consumption.

Any extra IO activity generated as a result of the use of managed code can significantly increase the average storage-related power, especially since mobile storage has such a low idle power envelope. This section explores the performance and energy impact of using managed code.

We run experiments on both Windows RT device (setup #2) and Android device. The first set of parameters from Table 4.1 (granularity and access pattern)are again varied during a set of benchmarking runs using native and managed code APIs for Windows and Linux with encryption disabled.

The pre-instrumented Windows RT tablet is specially configured (via Microsoft-internal functionality) to allow the development and running of applications natively. The native version of the benchmarking application uses the `OpenFile`, `ReadFile`, and `WriteFile` APIs on Windows. The Android version uses the Java Native Interface [50] to call the native `C fopen`,

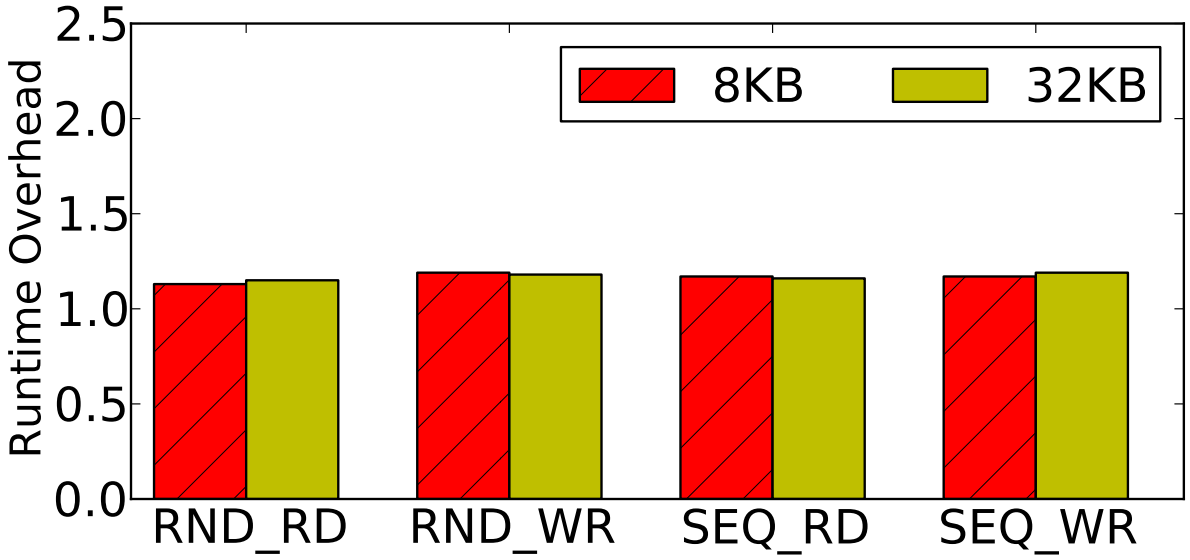


Figure 4.10. Impact of managed programming languages on Windows RT tablet: 12.6–18.3% more energy per KB for using the CLR.

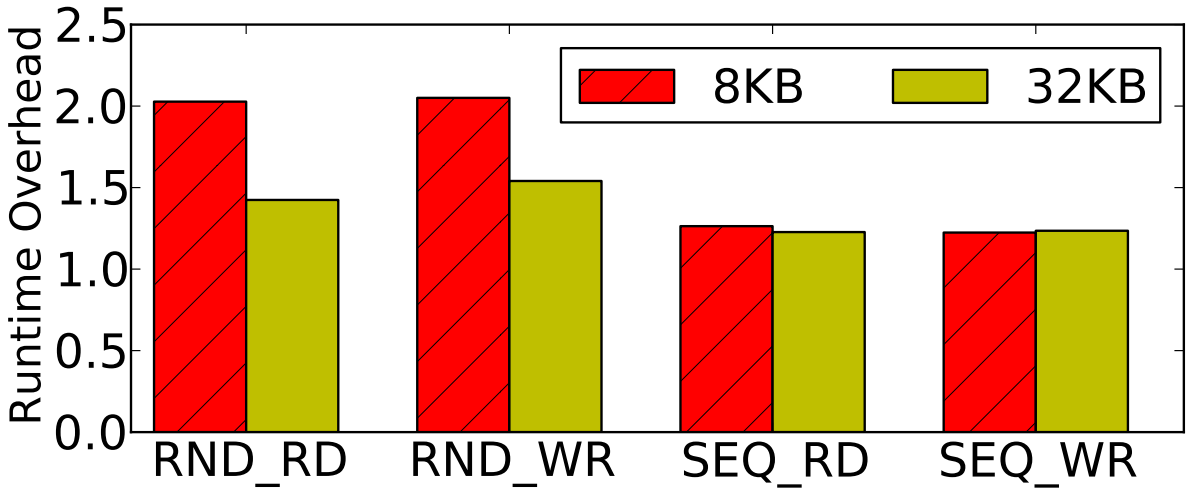


Figure 4.11. Impact of managed programming language on Android phone: 24.3–102.1% more energy per KB for using the Dalvik runtime.

`fread`, `fseek`, and `fwrite` APIs.

The measured energy consumptions for the Windows and Android platforms are shown in Figures 4.10, and 4.11, respectively. Each bar represents the multiplication factor by which energy consumption per KB increases when using managed cost rather than native.

- On Windows RT, the energy overhead on storage systems from running applications in a managed environment is between 12.6–18.3%.
- The overhead on Android is between 24.3–102.1%. We believe that the higher energy overhead for smaller IOs (some not shown) is due to a larger prefetching granularity used by filesystem APIs of the managed environment. For larger IO sizes (some not shown), the overhead was always lower than 25%.

Security and privacy requirements of applications on mobile platforms clearly add an energy overhead as demonstrated in this section and the previous one. If developers of storage-intensive applications take these overheads into account, more energy-efficient applications could be built.

4.4 Energy Model for Storage

As shown in the previous sections, encryption and the use of managed code add a significant amount of overhead to the storage APIs – not only in terms of performance, but also in terms of energy. Therefore, we believe that it is necessary to empower developers with tools to understand and optimize the energy consumed by their applications with regard to storage APIs.

This section first attempts to formalize the energy consumption characteristics of the storage subsystem. It then presents EMOS (Energy Modeling for Storage), a simulation tool that an application or an OS developer can use to estimate the amount of energy needed for its storage activity. Such a tool can be used standalone or as part of a complete energy modeling system such as WattsOn [65].

4.4.1 Modeling Storage Energy

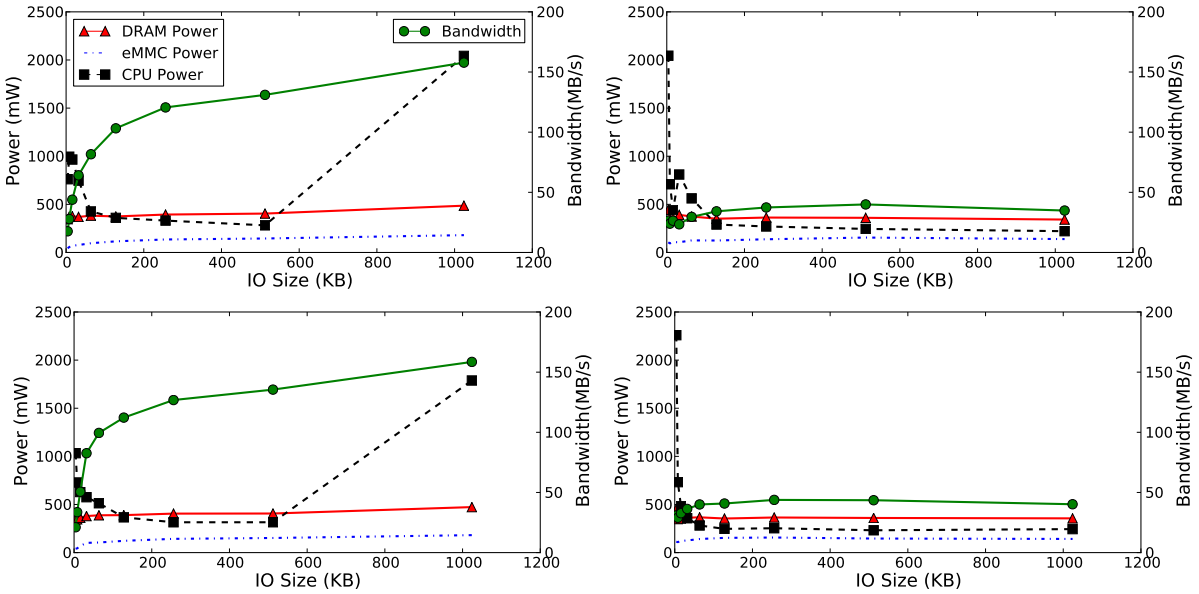


Figure 4.12. Power draw by DRAM, eMMC, and CPU for different IO sizes on Windows RT with encryption disabled. CPU power draw generally decreases as the IO rate drops. However, large (e.g., 1 MB) IOs incur more CPU path (and power) because they trigger working set trimming activity more frequently.

The energy cost of a given IO size and type can be broken down into its power and time components. If the total power of read and write operations are P_r and P_w, respectively, and the corresponding read and write throughputs are T_r and T_w KB/s, then the energy consumed by the storage device per KB for reads (E_r) and writes (E_w) is:

$$E_r = P_r/T_r, E_w = P_w/T_w$$

The hardware "energy" cost of accessing a storage page depends on whether it is a read or a write operation, a file cache hit or miss, sequential or random, encrypted or not, and other considerations not covered by this analysis, such as request interarrival time, interleaving of different IO sizes and types, and the effects of storage hardware caches or hybrid FLASH devices (e.g., SLC/MLC).

In this model, P is comprised of CPU(P_{CPU}), memory (P_{DRAM}), and storage hardware(P_{EMMC}) power. Figure 4.12 shows the variation of each of these power components for random and sequential, uncached, unencrypted, reads and writes.

P_{DRAM} can be modeled as follows:

- For writes, the DRAM consumes 450 mW when the IO size is less than 8 KB. When the IO size is greater than or equal to 8 KB, this power is closer to 360 mW. The increase in power at lower IO sizes is probably due to a block size of 4 KB used by NTFS on Windows RT.

- For reads, DRAM power increases linearly with request size from 350 mW for 4 KB reads to 475 mW for 1 MB reads. Write throughput rates are low enough that DRAM power variation for different write sizes is low.

Storage unit power (P_{EMMC}) can be modeled as follows:

- For writes, the eMMC power variation due to sequentiality and request size is fairly low – from 105 mW for 4 KB IOs to 140 mW for 1 MB IOs.

- For random and sequential reads, the eMMC power varies from 40 mW for 4 KB IOs to 180 mW for 1 MB IOs, with most of the variation coming from IO sizes less than 4 KB.

The graphs show that P_{CPU} follows an exponential curve with respect to the IO size. However, the CPU power actually tracks the storage API IOps curve, which is T/IO_size. Since IOps actually follows an exponential curve when plotted against IO size, a linear correlation exists between P_{CPU} and IOps (see Figure 4.13). The two scatter plot outliers that consume high CPU power at low IOps are the 1 MB sequential and random read operations. The bandwidth of these workloads (160 MB/s) was large enough and the experiments was long enough for the OS to start trimming working sets. If the other request size experiments were run for long enough, they would also incur some additional power cost when trimming finally kicked in.

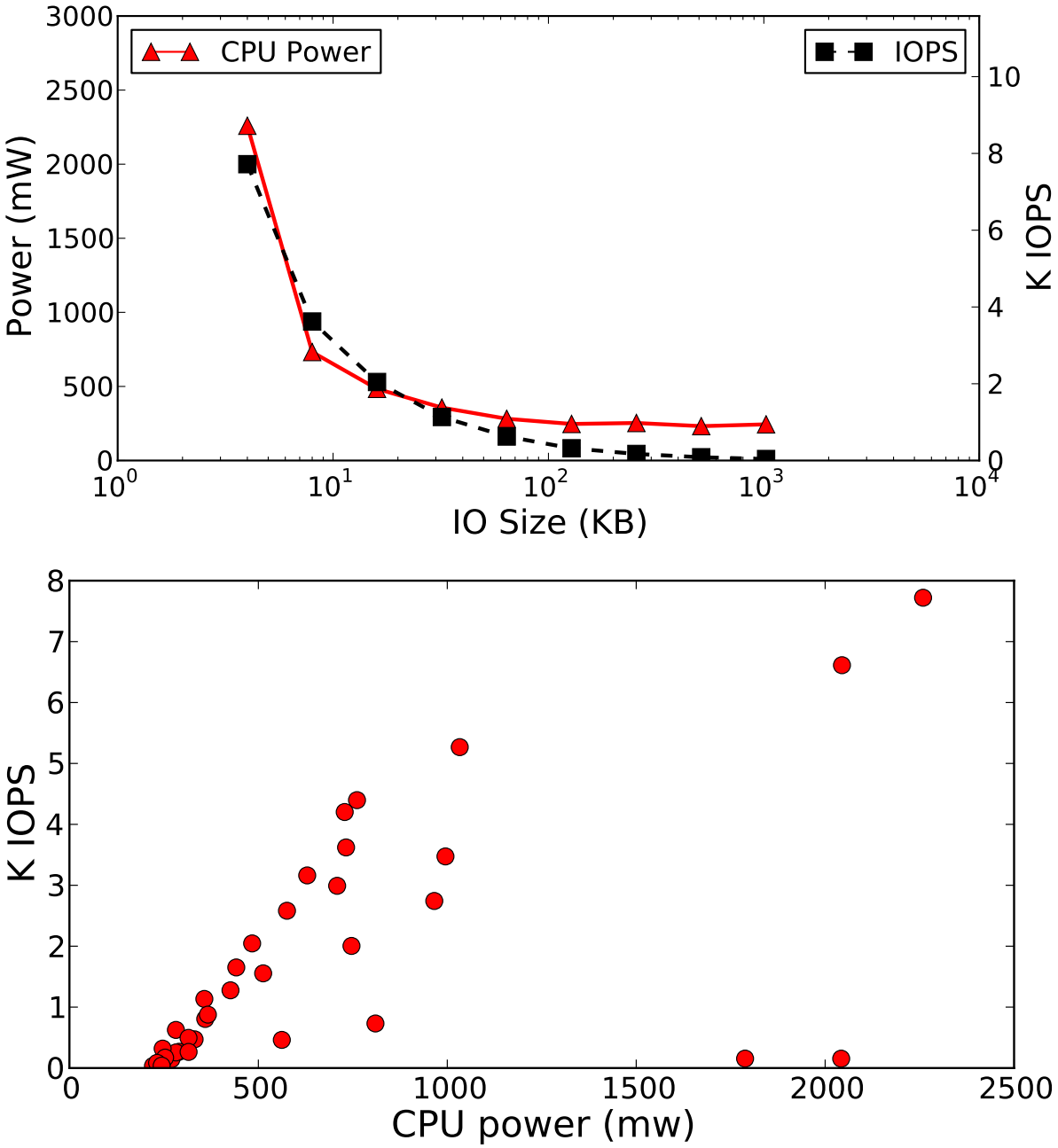


Figure 4.13. CPU power & IOps for different sizes of random and sequential reads. Both metrics follow an exponential curve and show good linear correlation. The two outliers in the scatter plot towards the bottom right are caused by high read throughput triggering the CPU-intensive working set trimming process in Windows.

With Encryption: If similar graphs were plotted for the experiments with encryption enabled, the following would be seen:

- All component power values generally increase with IO size.
- P_{DRAM} is higher for reads than writes, staying fairly constant at 515 mW. For writes, the power increases linearly with IO size, varying from 370 mW for 4 KB IOs to 540 mW for 1 MB IOs. This variation is mostly because of the extra memory needed for encryption to complete.
- P_{EMMC} values for reads and writes are similar to their unencrypted counterparts. Given that encryption (and decryption) in current mobile devices is handled using on-SoC hardware, this is to be expected.
- P_{CPU} is fairly linear with IOPS for reads, but the power characteristics for writes are more complex due to dynamic encryption algorithms. Windows systems use a combination of software (general purpose CPUs) and/or hardware (on-SoC offload engines) for encryption based on the request size and type.

4.4.2 The Energy MOdeling for Storage Simulator

The EMOS simulator takes as input a sequence of timestamped disk requests and the total size of the filesystem cache. It emulates the caching mechanism of the operation system to identify hits and misses. Each IO is broken into small primitive operations, each of which has been empirically measured for its energy consumption.

Energy of primitive operations: Ideally, component power numbers (P_{CPU}, P_{DRAM}, and P_{EMMC}) would be generated for every platform. It would not be feasible for a single company to cake on this task, but the possibility exists for scaling out the data capture to a broader set of manufacturers and recombining their component data into a single per-platform model. For the purposes of this paper, the EMOS simulator is tuned and tested on the Windows table and Android phone platforms utilized in the previous sections' experiments.

Table 4.4. Energy (uJ) per KB for different IO requests. Such tables can be built for a specific platform and subsequently incorporated into power modeling software usable by developers for optimizing their storage API calls.

Platform	Caching	IO Size	RND_RD	RND_WR	SEQ_RD	SEQ_WR
Windows RT	Hit	8KB	14.2	22.4	11.2	19.0
		32KB	11.4	18.2	8.6	18.2
	Miss	8KB	96.7	110.4	85.0	117.5
		32KB	36.4	116.8	18.0	118.2
Android	Hit	4KB	10.3	252.9	9.1	52.6
		8KB	6.0	167.2	5.8	51.0
		16KB	4.0	240.7	4.0	64.4
		32KB	3.3	169.7	3.3	88.5
	Miss	4KB	441.9	2402.7	62.5	451.8
		8KB	214.4	2176.7	58.5	403.5
		16KB	187.6	1720.9	51.3	254.9
		32KB	141.0	1776.0	51.1	138.8

For each platform, the average energy needed for completing a given IO type (read/write, size, cache hit/miss) is measured. The energy values are aggregated from DRAM, CPU, and Code (idle energy values are subtracted).

A table such as Table 4.4 can be built to summarize the measured energy consumption required for each type of storage request. EMOS uses a regression method to calculate energy values for request sizes not found in the table.

Simulation of cache behavior: Cache hits and misses have different storage request energy consumptions. Since many factors affect the actual cache hit or miss behavior (e.g., replacement policy, cache size, prefetch algorithm, etc.), a subset of the range of possible cache characteristics was selected for EMOS. For example, only the LRU (Least Recently Used) cache replacement policy is simulated, but the cache size and prefetch policy are configurable.

EMOS was validated using the 4 KB random IO micro-benchmarks on the Android platform without any changes to the default cache size, or prefetch policy. The measured versus calculated energy consumption of the system were compared for workloads of 100% reads, 100%

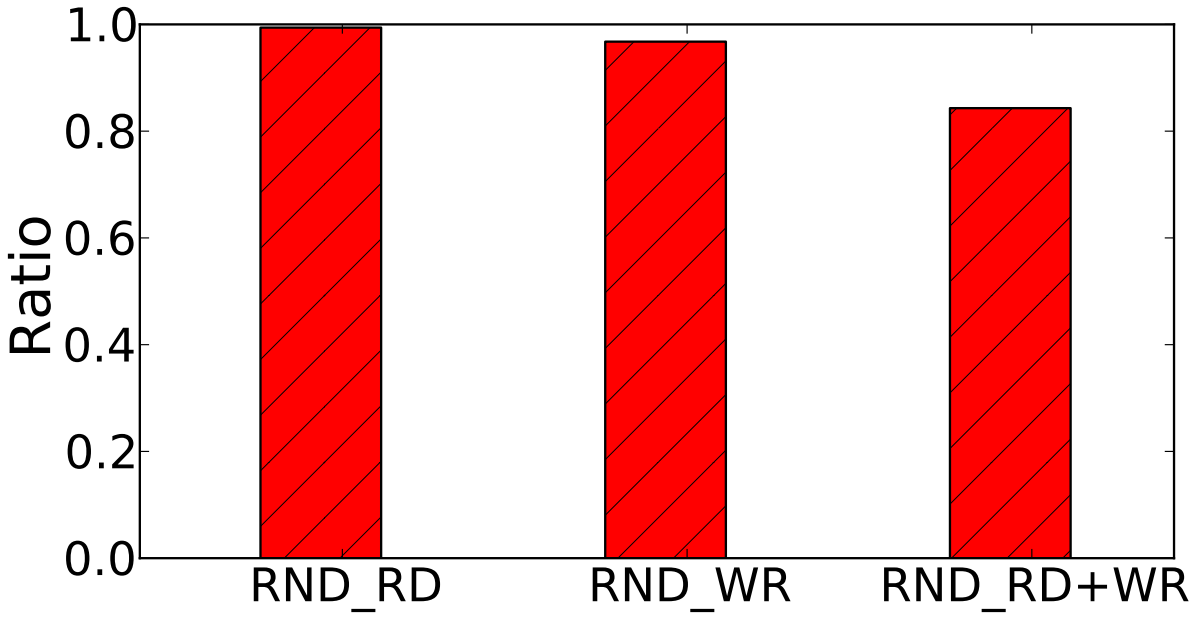


Figure 4.14. Experimental validation of EMOS on Android shows greater than 80% accuracy for predicting 4 KB random IO micro-benchmark energy consumption.

writes, and a 50%/50% mix. Figure 4.14 shows that while the model is very accurate for pure read and write workloads, it is only 80% accurate for a mixed workload. We attribute this to the IO scheduler and the file cache software behaving differently when there is a mix of reads and writes. Future investigations are planned to account for such software behavior.

4.5 Discussion: Reducing Mobile Storage Energy

We suggest several ways to reduce the energy consumption of the storage software stack through hardware and software modifications. These suggestions should reduce encryption energy costs as well as the energy gap between native IO and managed language IO.

4.5.1 Partially-Encrypted File systems

While full-disk encryption thwarts the vast majority of physical security attacks, it may be overkill for some scenarios. It puts an unnecessary burden on accessing data that does not require strict privacy and security guarantees. For example, most operating system files, common

application binaries, some caches, and possibly even media purchased online may not need to be encrypted. A naive solution would be to partition the disk into encrypted and unencrypted file systems / partitions. However, if free space cannot be dynamically shifted between the partitions, this solution may result in wasted disk space. More importantly, some entity has to make decisions about which files to store in which file systems, and the user would need to explicitly make some of these decisions in order to achieve optimal and appropriate partitioning. For example, a user may or may not wish his or her media files to be visible if a mobile device is stolen.

Partially-encrypted filesystems that allow some data to be encrypted while other data is unencrypted represent a better solution for mobile storage systems. This removes the concern over lost disk space, but some or all of the difficulties associated with the encrypt-or-not decision remain. Nevertheless, opens the option for individual applications to make some decisions about the privacy and security of files they own, perhaps splitting some files in two in order to encrypt only a portion of the data contained within. This increases development overhead, but it does provide applications with a knob to tune their energy requirements. GNU Privacy Guard [41] for Linux and Encrypting File Systems [35] on Windows provide such services. However, care must be taken to ensure that unencrypted copies of private data not be left in the filesystem at any point unless the user is cognizant (and accepting) of this vulnerability.

Additional security and privacy systems are needed to fully secure partially-encrypted file systems. Once the data from an encrypted file has been decrypted for usage, it must be actively tracked using taint analysis. Information flow control tools [34, 40, 104] are required to ensure that unencrypted copies of data are not left behind on persistent storage for attackers to exploit.

4.5.2 Storage Hardware Virtualization

Low-cost storage targeted to mobile platforms relies on storage software features. Isolation between applications is provided using managed languages, per-application users and

groups, and virtual machines on Android and Windows RT for applications developed in Java and .NET, respectively. Storage software overhead can be reduced by moving much of this complexity into the storage hardware [26].

Mobile storage can be built in a manner such that each application is provided with the illusion of a private filesystem. In fact, Windows RT already provides such isolation using only software [68]. Moving such isolation mechanisms into hardware can enable managed languages to directly use native APIs for applications to obtain native hardware performance with isolation guarantees.

4.5.3 SoC Offload Engines for Storage Operations

Various components inside mobile platforms have moved their latency- and energy-intensive tasks to SoCs. Audio, video, radio, and location sensors have dedicated SoC engines for frequent, narrowly-focused tasks, such as decompression, echo cancellation, and digital signal processing. This type of optimization may also be appropriate for storage activities. For example, the SoC can help information flow control tools and improve hardware virtualization. Some SoC's already support encryption and decryption of data entirely in hardware, but tuning these algorithms is a nontrivial exercise. Today, we find that most of the storage IO tasks utilize the general purpose CPU cores, which consumes higher energy than dedicated offload engines. Dedicated engines for file system activity could provide metadata databases and file access functionality while ensuring isolation, privacy and security.

4.6 Related Work

To our knowledge, a comprehensive study of storage systems on mobile platforms from the perspective of energy has not been presented to date. Kim et al [51] present a comprehensive analysis of the performance of secondary storage devices, such as SD cards often used on mobile platforms. Past research studies have presented energy analysis of other mobile subsystems, such as networking [19, 39], location sensing [95], the CPU complex [64], graphics [90], and

other system components [21]. Carroll *et al.* [25] present the storage energy consumption of SD cards using native IO. Shye *et al.* [86] implement a logger to help analyze and optimize energy consumption by collecting traces of software activities.

Energy estimation and optimization tools [32, 105, 38, 65, 75, 74, 80, 78, 102] have been devised to estimate how much energy an application consumes during its execution. This paper uses similar techniques to estimate the energy required by various components (hardware and software) inside the storage subsystems of mobile platforms. This analysis is performed from the perspective of the storage stack as opposed to a broader OS perspective or a narrower application perspective.

Energy consumption of storage software has been analyzed in the past for distributed systems [56], servers [77, 84, 88], PCs [69] and embedded systems [27], as opposed to the mobile platforms analyzed in this paper.

Storage systems using new memory technologies like phase-change memory (PCM) focus on analyzing and eliminating the overhead from software [26, 29, 55, 100]. However, existing storage work for new memory technologies focuses only on native IO performance. This paper also includes analysis of managed language environments.

Storage systems on modern mobile platforms face additional overhead due to greater privacy and security requirements. Mobile handheld devices are prone to theft and often contain private information. For this reason, the power overhead of encryption is analyzed in this paper. Protecting private information also requires applications to be isolated from each other, so managed language overheads are also analyzed. The software energy overhead from privacy and security requirements are especially high when compared to the energy consumed by mobile storage hardware.

4.7 Conclusions

Battery life is a key concern for mobile devices such as phones and tablets. Although significant research has gone into improving the energy efficiency of these devices, the impact of storage (and associated APIs) on battery life has not received much attention. In part this is due to the low idle power draw of storage devices such as SD cards and on-board eMMC storage.

This paper takes a principled look at the energy consumed by storage hardware and software on mobile devices. Measurements across a set of storage-intensive micro-benchmarks show that storage software may consume as much as 20x more energy than storage hardware on an Android phone and a Windows RT tablet. The two biggest energy consumers are encryption and managed language environments. Energy consumed by storage APIs increases by up to 6.0x when encryption is enabled. Managed language storage APIs consume 25% more energy compared to their native counterparts.

We build an energy model to help developers understand the energy costs of security and privacy requirements of applications on mobile platforms. The EMOS model can predict the energy required for a mixed read/write micro-benchmark with 80% accuracy. The paper also supplies some observations on how mobile storage energy efficiency can be improved.

Chapter 4 contains materials from "On the Energy Overhead of Mobile Storage Systems", by Jing Li, Anirudh Badam, Ranveer Chandra, Steven Swanson, Bruce Worthington and Qi Zhang, which appears in the 12th USENIX Conference on File and Storage Technologies. The dissertation author was the primary investigator and first author of this paper. The materials are copyright

Bibliography

[1] http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-dc-s3700-spec.pdf.

[2] http://www.nvidia.com/object/tesla-servers.html.

[3] https://developer.nvidia.com/gpudirect.

[4] https://trademarks.justia.com/865/43/nvmedirect-86543720.html.

[5] https://hive.apache.org.

[6] http://hadoop.apache.org.

[7] https://aws.amazon.com/emr/details/hadoop/.

[8] http://orc.apache.org.

[9] Project donard: Peer-to-peer communication with nvm express devices. http://blog.pmcs.com/project-donard-peer-to-peer-communication-with-nvm-express-devices-part-two/. Accessed: 2010-09-30.

[10] Boston, MA, June 2010.

[11] Washington, DC, June 2011.

[12] Salzburg, Austria, April 2011.

[13] Daniel J Abadi. *Query execution in column-oriented database systems*. PhD thesis, Massachusetts Institute of Technology, 2008.

[14] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *SIGMOD*, pages 967–980. ACM, 2008.

[15] Divyakant Agrawal, Philip Bernstein, Elisa Bertino, Susan Davidson, Umeshwas Dayal, Michael Franklin, Johannes Gehrke, Laura Haas, Alon Halevy, Jiawei Han, et al. Challenges and opportunities with big data 2011-1. 2011.

[16] Android Application Tracing. http://developer.android.com/tools/debugging/debugging-tracing.html.

[17] Android Full System Tracing.
http://developer.android.com/tools/debugging/systrace.html.

[18] Android Storage API.
http://developer.android.com/guide/topics/data/data-storage.html.

[19] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications. In *Proc. ACM IMC*, Chicago, IL, November 2009.

[20] Nagender Bandi, Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Fast data stream algorithms using associative memories. In *SIGMOD*, pages 247–256, 2007.

[21] Jeffrey Bickford, H. Andres Lagar-Cavilla, Alexander Varshavsky, Vinod Ganapathy, and Liviu Iftode. Security versus Energy Tradeoffs in Host-Based Mobile Malware Detection, June 2011.

[22] BitLocker Drive Encrytion.
http://windows.microsoft.com/en-us/windows7/products/features/bitlocker.

[23] Peter A Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, volume 5, pages 225–237, 2005.

[24] Sebastian Breß and Gunter Saake. Why it is time for a hype: A hybrid query processing engine for efficient gpu coprocessing in dbms. *VLDB*, 6(12):1398–1403, 2013.

[25] Aaron Carroll and Gernot Heiser. An Analysis of Power Consumption in a Smartphone. In *Proc. USENIX ATC* [10].

[26] Adrian M. Caulfield, Todor I. Mollov, Louis Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing safe, user space access to fast, solid state disks. In *Proc. ACM ASPLOS*, London, United Kingdom, March 2012.

[27] Siddharth Choudhuri and Rbi N. Mahapatra. Energy Characterization of Filesystems for Diskless Embedded Systems. In *Proc. 41st DAC*, San Diego, CA, 2004.

[28] Eric S Chung, Peter A Milder, James C Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In *MICRO*, pages 225–236. IEEE Computer Society, 2010.

[29] Jeremy Condit, Edmund B. Nightingale, Cristopher Frost, Engin Ipek, Doug Burger, Benjamin Lee, and Derrick Coetzee. Better I/O Through Byte-Addressable, Persistent Memory. In *Proc. 22nd ACM SOSP*, Big Sky, MT, October 2009.

[30] Robert H Dennard, VL Rideout, E Bassous, and AR Leblanc. Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, 1974.

[31] Jaeyoung Do, Yang-Suk Kee, Jignesh M Patel, Chanik Park, Kwanghyun Park, and David J DeWitt. Query processing on smart ssds: opportunities and challenges. In *SIGMOD*, pages 1221–1230. ACM, 2013.

[32] Mian Dong and Lin Zhong. Self-Constructive High-Rate System Energy Modeling for Battery-Powered Mobile Systems. In *Proc. 9th ACM MobiSys* [11].

[33] eMMC 4.51, JEDEC Standard.
http://www.jedec.org/standards-documents/results/jesd84-b45.

[34] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy MOnitoring on Smartphones. In *Proc. 9th USENIX OSDI*, Vancouver, Canada, October 2010.

[35] Encrypting File System for Windows.
http://technet.microsoft.com/en-us/library/cc700811.aspx.

[36] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ISCA*, pages 365–376, 2011.

[37] Wenbin Fang, Bingsheng He, and Qiong Luo. Database compression on graphics processors. *VLDB*, 3(1-2):670–680, 2010.

[38] Jason Flinn and Mahadev Satyanarayanan. Energy-Aware Adaptation of Mobile Applications, December 1999.

[39] Rodrigo Fonseca, Prabal Dutta, Philip Levis, and Ion Stoica. Quanto: Tracking Energy in Networked Embedded Systems. In *Proc. 8th USENIX OSDI*, San Diego, CA, December 2008.

[40] Roxana Geambasu, John P. John, Steven D. Gribble, Tadayoshi Kohno, and Henry M. Levy. Keypad: An Auditing File SYstem for Theft-Prone Devices. In *Proc. 6th ACM EUROSYS* [12].

[41] GNU Privacy Guard: Encrypt files on Linux.
http://www.gnupg.org/.

[42] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *SIGMOD*, pages 325–336. ACM, 2006.

[43] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(EPFL-ARTICLE-168285):6–15, 2011.

[44] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *SIGMOD*, pages 511–524, 2008.

[45] Bingsheng He and Jeffrey Xu Yu. High-throughput transaction executions on graphics processors. *VLDB*, 4(5):314–325, 2011.

[46] Jiong He, Mian Lu, and Bingsheng He. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *VLDB*, 6(10):889–900, 2013.

[47] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. Hardware-oblivious parallelism for in-memory column-stores. *VLDB*, 6(9):709–720, 2013.

[48] David A Huffman et al. A method for the construction of minimum redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[49] HV Jagadish, Johannes Gehrke, Alexandros Labrinidis, Yannis Papakonstantinou, Jignesh M Patel, Raghu Ramakrishnan, and Cyrus Shahabi. Big data and its technical challenges. *Communications of the ACM*, 57(7):86–94, 2014.

[50] Java Native Interface. http://developer.android.com/training/articles/perf-jni.html.

[51] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting Storage on Smartphones. 8(4):14:1–14:25, 2012.

[52] Sangman Kim, Seonggu Huh, Yige Hu, Xinya Zhang, Amir Wated, Emmett Witchel, and Mark Silberstein. Gpunet: Networking abstractions for gpu programs. In *Proceedings of the International Conference on Operating Systems Design and Implementation*, pages 6–8, 2014.

[53] Ralph Kimball and Margy Ross. *The data warehouse toolkit: The definitive guide to dimensional modeling*. John Wiley & Sons, 2013.

[54] Gunjae Koo, Kiran Kumar Matam, Te I, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: Trading communication with computing near storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 219–231, New York, NY, USA, 2017. ACM.

[55] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory. In *Proc. 11th USENIX FAST*, San Jose, CA, February 2013.

[56] Jacob Leverich and Christos Kozyrakis. On the Energy (In)efficiency of Hadoop Clusters. *ACM SIGOPS OSR*, 44:61–65, 2010.

[57] Jing Li, Anirudh Badam, Ranveer Chandra, Steven Swanson, Bruce Worthington, and Qi Zhang. On the energy overhead of mobile storage systems. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 105–118, Berkeley, CA, USA, 2014. USENIX Association.

[58] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics. *Proc. VLDB Endow.*, 9(14):1647–1658, October 2016.

[59] Yang Liu, Hung-Wei Tseng, Mark Gahagan, Jing Li, Yanqin Jin, and Steven Swanson. Hippogriff: Efficiently moving data in heterogeneous computing systems. In *Computer Design (ICCD), 2016 IEEE 34th International Conference on*, pages 376–379. IEEE, 2016.

[60] Yanfei Lv, Bin Cui, Xuexuan Chen, and Jing Li. Hotness-aware buffer management for flash-based hybrid storage systems. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, pages 1631–1636. ACM, 2013.

[61] Yanfei Lv, Bin Cui, Xuexuan Chen, and Jing Li. Hat: an efficient buffer management method for flash-based hybrid storage systems. *Frontiers of Computer Science*, 8(3):440–455, 2014.

[62] Yanfei Lv, Jing Li, Bin Cui, and Xuexuan Chen. Log-compact r-tree: an efficient spatial index for ssd. In *International Conference on Database Systems for Advanced Applications*, pages 202–213. Springer, 2011.

[63] Silvano Martello and Paolo Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[64] Antti P. Miettinen and Jukka K. Nurminen. Energy Efficiency of Mobile Clients in Cloud Computing. In *Proc. 2nd USENIX HotCloud*, Boston, MA, June 2010.

[65] Radhika Mittal, Aman Kansal, and Ranveer Chandra. Empowering Developers to Estimate App Energy Consumption. In *Proc. 18th ACM MobiCom*, Istanbul, Turkey, August 2012.

[66] Monsoon Power Monitor. http://www.msoon.com/LabEquipment/PowerMonitor/.

[67] National Instruments 9206 DAQ Toolkit. http://sine.ni.com/nips/cds/view/p/lang/en/nid/209870.

[68] .NET Isolated Storage API. http://msdn.microsoft.com/en-us/library/system.io.isolatedstorage.isolatedstoragefile.aspx.

[69] Edmund B. Nightingale and Jason Flinn. Energy-Efficiency and Storage Flexibility in the Blue File System. In *Proc. 5th USENIX OSDI*, San Francisco, CA, December 2004.

[70] Molly A O'Neil and Martin Burtscher. Floating-point data compression at 75 gb/s on a gpu. In *GPGPU*, page 7. ACM, 2011.

[71] Patrick ONeil, Elizabeth ONeil, Xuedong Chen, and Stephen Revilak. The star schema benchmark and augmented fact table indexing. In *Performance evaluation and benchmarking*, pages 237–252. Springer, 2009.

[72] Rasmus Pagh and Flemming Friche Rodler. *Cuckoo hashing*. Springer, 2001.

[73] Ritesh Patel, Yao Zhang, Jason Mak, Andrew Davidson, John D Owens, et al. *Parallel lossless data compression on the GPU*. IEEE, 2012.

[74] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: Fine Grained Energy Accounting on Smartphones. In *Proc. 7th ACM EUROSYS*, Bern, Switzerland, April 2012.

[75] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. Fine-Grained Power Modeling for Smartphones using System Call Tracing. In *Proc. 6th ACM EUROSYS* [12].

[76] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, pages 165–178. ACM, 2009.

[77] Eduardo Pinheiro and Ricardo Bianchini. Energy Conservation Techniques for Disk Array-Based Servers. In *Proc. 18th ACM ICS*, Saint-Malo, France, June 2004.

[78] Feng Qian, Zhaoguang Wang, Alexandre Gerber, Z. Morley Mao, Subhabrata Sen, and Oliver Spatschek. Profiling Resource Usage for Mobile Applications: a Cross-layer Approach. In *Proc. 9th ACM MobiSys* [11].

[79] Erik Riedel, Christos Faloutsos, and David Nagle. Active disk architecture for databases. Technical report, DTIC Document, 2000.

[80] Arjun Roy, Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazieres, and Nickolai Zeldovich. Energy Management in Mobile Devices with Cinder Operating System. In *Proc. 6th ACM EUROSYS* [12].

[81] Samsung eMMC 4.5 Prototype. http://www.samsung.com/us/business/oem-solutions/pdfs/eMMC_Product%20Overview.pdf.

[82] Vijay Sathish, Michael J Schulte, and Nam Sung Kim. Lossless and lossy memory i/o link compression for improving performance of gpgpu workloads. In *PACT*, pages 325–334. ACM, 2012.

[83] Secure Digital Card Specification. https://www.sdcard.org/downloads/pls/simplified_specs/.

[84] Priya Sehgal, Vasily Tarasov, and Erez Zadok. Evaluating Performance and Energy in File System Server Workloads. In *Proc. USENIX ATC* [10].

[85] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable ssd. In *OSDI*, pages 67–80, Broomfield, CO, October 2014. USENIX Association.

[86] Alex Shye, Benjamin Scholbrock, and Gokhan Memik. Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures. In *Proc. 42nd IEEE MICRO*, New York, NY, December 2009.

[87] Brian Smith. A survey of compressed domain processing techniques. *Cornell University*, 1995.

[88] Mark W. Storer, Kevin M. Greenan, Ethan L. Miller, and Kaladhar Voruganti. Pergamum: Replacing Tape with Energy Efficient, Reliable, Disk-Based Archival Storage. In *Proc. 6th USENIX FAST*, San Jose, CA, 2008.

[89] Jukka Teuhola. A compression method for clustered bit-vectors. *Information processing letters*, 7(6):308–311, 1978.

[90] Narendran Thiagarajan, Gaurav Aggarwal, Angela Nicoara, Dan Boneh, and Jatinder Pal Signh. Who Killed My Battery: Analyzing Mobile Browser Energy Consumption. In *Proc. WWW*, Lyon, France, April 2012.

[91] Hung-Wei Tseng, Yang Liu, Mark Gahagan, Jing Li, Yanqin Jin, and Steven Swanson. Gullfoss: Accelerating and simplifying data movement among heterogeneous computing and storage resources. Technical report.

[92] Nandita Vijaykumar, Gennady Pekhimenko, Adwait Jog, Abhishek Bhowmick, Rachata Ausavarungnirun, Chita Das, Mahmut Kandemir, Todd C Mowry, and Onur Mutlu. A case for core-assisted bottleneck acceleration in gpus: enabling flexible data compression with assist warps. In *ISCA*, pages 41–53. ACM, 2015.

[93] Kaibo Wang, Yin Huai, Rubao Lee, Fusheng Wang, Xiaodong Zhang, and Joel H Saltz. Accelerating pathology image data cross-comparison on cpu-gpu hybrid systems. *VLDB*, 5(11):1543–1554, 2012.

[94] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. Concurrent analytical query processing with gpus. *VLDB*, 7(11):1011–1022, July 2014.

[95] Yi Wang, Jialiu Lin, Murali Annavaram, Quinn A. Jacobson, Jason Hong, Bhaskar Krishnamachari, and Norman Sadeh-Koniecpol. A Framework for Energy Efficient Mobile Sensing for Automatic Human State Recognition. In *Proc. 7th ACM Mobisys*, Krakow, Poland, June 2009.

[96] Zeke Wang, Huiyan Cheah, Johns Paul, Bingsheng He, and Wei Zhang. Accelerating database query processing on opencl-based fpgas. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 274–274. ACM, 2016.

[97] Windows Performance Toolkit.
http://msdn.microsoft.com/en-us/performance/cc825801.aspx.

[98] Windows RT Storage API.
http://msdn.microsoft.com/en-us/library/windows/apps/hh758325.aspx.

[99] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *MICRO*, pages 107–118. IEEE Computer Society, 2012.

[100] Xiaojian Wu and A. L. Narasimha Reddy. SCMFS: A File System for Storage Class Memory. In *Proc. IEEE/ACM SC*, Seattle, WA, November 2011.

[101] Hongzhi Yin, Bin Cui, Jing Li, Junjie Yao, and Chen Chen. Challenging the long tail recommendation. *Proceedings of the VLDB Endowment*, 5(9):896–907, 2012.

[102] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. AppScope: Application Energy Metering Framework for Android Smartphones using Kernel Activity Monitoring. In *Proc. USENIX ATC*, Boston, MA, June 2012.

[103] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. The yin and yang of processing data warehousing queries on gpu devices. *VLDB*, 6(10):817–828, 2013.

[104] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazieres. Making Information Flow Explicit in HiStar. In *Proc. 7th USENIX OSDI*, Seattle, WA, December 2006.

[105] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proc. 8th IEEE/ACM/IFIP CODES+ISSS*, Taipei, Taiwan, 2010.