Accelerated Storage Systems

Aleksandr S. Khasymski

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in the partial fulfillment of the requirement for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Ali R. Butt, Chair
Dimitrios S. Nikolopoulos
Sudharshan S. Vazhkudai
Kirk W. Cameron
Cal J. Ribbens

February 13, 2015
Blacksburg, Virginia, USA

Keywords: Key-value Store, Flash, Software RAID, Accelerator-based Computing

Copyright © 2015, Aleksandr S. Khasymski

Accelerated Storage Systems

Aleksandr S. Khasymski

ABSTRACT

Today's large-scale, high-performance, data-intensive applications put a tremendous stress on data centers to store, index, and retrieve large amounts of data. Exemplified by technologies such as social media, photo and video sharing, and e-commerce, the rise of the real-time web demands data stores support minimal latencies, always-on availability and ever-growing capacity. These requirements have fostered the development of a large number of high-performance storage systems, arguably the most important of which are Key-Value (KV) stores. An emerging trend for achieving low latency and high throughput in this space is a solution, which utilizes both DRAM and flash by storing an efficient index for the data in memory and minimizing accesses to flash, where both keys and values are stored. Many proposals have examined how to improve KV store performance in this area. However, these systems have shortcomings, including expensive sorting and excessive read and write amplification, which is detrimental to the life of the flash.

Another trend in recent years equips large scale deployments with energy-efficient, high-performance co-processors, such as Graphics Processing Units (GPUs). Recent work has explored using GPUs to accelerate compute-intensive I/O workloads, including RAID parity generation, encryption, and compression. While this research has proven the viability of GPUs to accelerate these workloads, we argue that there are significant benefits to be had by developing methods and data structures for deep integration of GPUs inside the storage stack, in order to achieve better performance, scalability, and reliability.

In this dissertation, we propose comprehensive frameworks that leverage emerging technologies, such as GPUs and flash-based SSDs, to accelerate modern storage systems. For our accelerator-based solution, we focus on developing a system that features deep integration of the GPU in a distributed parallel file system. We utilize a framework that builds on the resources available in the file system and coordinates the workload in such a way that minimizes data movement across the PCIe bus, while exposing data parallelism to maximize the potential for acceleration on the GPU. Our research aims to improve the overall reliability of a PFS by developing a distributed per-file parity generation that provides end-to-end

data integrity and unprecedented flexibility. Finally, we design a high-performance KV store utilizing a novel data structure tailored to specific flash requirements; it arranges data on flash in such a way as to minimize write amplification, which is detrimental to the flash cells. The system delivers outstanding read amplification through the use of a trie index and false positive filter.

Dedication

Dedicated to my parents, and wife,
for their endless love, encouragement and support ...

Acknowledgments

I would like to express my deepest gratitude to my advisor, Ali R. Butt, for his continuous guidance, valuable advice, and encouragement during my graduate studies at Virginia Tech. I'm most By welcoming into his research lab, Dr. Butt gave me the opportunity to continue to grow as a graduate student under his mentorship. Without his support, this work would not have been a success. I would also like to express special thanks to my first advisor, Dimitrios Nikolopoulos, for guiding me through my early years at Virginia Tech. Dr. Nikolopoulos posed challenging research problems and helped me establish the initial direction for my graduate research. Additionally, I would like to thank my advisory committee members, Sudharshan S. Vazhkudai, Kirk W. Cameron, and Cal Ribbens for their invaluable feedback and advice throughout my time at Virginia Tech.

I also owe a great deal of thanks to Douglas Santry, for recruiting me as a summer intern at NetApp ATG. This opportunity helped me grow as a researcher and laid the foundation for my future success. His tremendous knowledge and hands-on expertise inspired me and influenced the direction of my research. I would also like to thank Michael Condict, Sandip Shete, Ardalan Kangarlou, and Maxim Smith for their valuable discussions and feedback, as well as as all the fun we had at the NetApp Beer Bashes. I will always cherish the three months I spent interning in North Carolina.

I would also like to thank my colleagues at Distributed Systems and Storage Laboratory (DSSL), especially Mustafa Rafique, Hyogi Sim, Guanying Wang, Min Li, and Krish K.R., for their support and collaboration on research. We spent many sleepless nights at the lab, but their companionship made even the hardest challenges more enjoyable. Many thanks as well to my friends at Virginia Tech, especially Daniela Rojas, Don Conry, Alex Cioaca, and Matthew O'Rourke, for making my time at Virginia Tech a memorable one.

Finally, a very special thank you to my family, for their endless support and love. I will be forever grateful to my mother and father, Mariyana and Sergey Hasamski for a lifetime of encouragement, especially in the pursuit of my education in computer science. Many thanks are also due to Karen and Muhito for their love and patience through each day of my "two more years". This undertaking would have been impossible without the unwavering support and love of my dedicated family.

Table of Contents

ABSTRACT ii

Dedication iv

Acknowledgments v

List of Tables viii

List of Figures ix

Chapter 1 Introduction 1

1.1 Integrating GPUs in the Storage Stack 2

1.2 Optimizing Key-Value Stores for Flash 4

1.3 Research Contributions 5

1.4 Dissertation Organization 6

Chapter 2 Related Work 7

2.1 GPU-Accelerated Storage Systems 7

2.2 Key-Value Stores 13

2.3 Out-of-core I/O Processing Systems 16

Chapter 3 Flash-Accelerated Systems 18

3.1 Design 19

3.2 Evaluation 25

3.3 Chapter Summary 47

Chapter 4 GPU-Accelerated Systems 48

4.1 GPU-Accelerated Cost-Effective Distributed RAID 48

4.2 Offloading Flash Address Translation to a GPU 62

4.3 Chapter Summary 65

Chapter 5 Conclusion 66

5.1 Future Research 66

Bibliography 68

List of Tables

3.1 Yahoo Cloud Serving Benchmark. 26
3.2 Trie index average memory footprint per key. 31
3.3 Average memory footprint per key for optimal trie index configuration. . . . 32

4.1 RAID reconstruction time and normalized speedup with respect to 1 Node. . 61

List of Figures

3.1 Cache Oblivious Lookahead Array (COLA). 20
3.2 The Cache Oblivious No-lookahead Array (CONA). 21
3.3 Merge process absorbing writes into the CONA 24
3.4 CONA insertion performance. 27
3.5 CONA insertion performance. 28
3.6 CONA Insertion performance under *Workload A* (50/50 read/write). 29
3.7 Insertion performance under *Workload B* (95/5 read/write). 30
3.8 Lookup performance under *Workload C* (100/0 read/write). 30
3.9 Read distribution among the levels of the CONA. 31
3.10 Lookup throughput of YCSB workloads under varying trie block sizes. 32
3.11 Threaded merging performance. 33
3.12 Percentage merge time hidden by threading. 33
3.13 CONA read performance under YCSB workloads. 34
3.14 CONA write performance under YCSB workloads. 34
3.15 *Workload A* write performance. 35
3.16 *Workload A* merge performance. 35
3.17 *Workload A* read performance. 36
3.18 *Workload F* write performance. 37
3.19 *Workload F* merge performance. 38
3.20 *Workload F* read performance. 38
3.21 *Workload B* write performance. 39

3.22 *Workload B* merge performance. 40
3.23 *Workload D* write performance. 40
3.24 *Workload A* merge performance. 41
3.25 *Workload D* read performance. 41
3.26 *Workload C* read performance. 42
3.27 *Workload B* read performance. 42
3.28 SILT write performance under *Workload A*. 43
3.29 SILT write performance under *Workload B*. 43
3.30 SILT write performance under *Workload D*. 43
3.31 SILT write performance under *Workload F*. 43
3.32 SILT read performance under *Workload A*. 44
3.33 SILT read performance under *Workload B*. 44
3.34 SILT read performance under *Workload C*. 44
3.35 SILT read performance under *Workload D*. 44
3.36 SILT read performance under *Workload F*. 44
3.37 leveldb write performance under *Workload A*. 45
3.38 lleveldb write performance under *Workload B*. 45
3.39 leveldb write performance under *Workload D*. 45
3.40 leveldb write performance under *Workload F*. 45
3.41 leveldb read performance under *Workload A*. 46
3.42 leveldb read performance under *Workload B*. 46
3.43 leveldb read performance under *Workload C*. 46
3.44 leveldb read performance under *Workload D*. 46
3.45 leveldb read performance under *Workload F*. 46
3.46 KV store read performance comparison. 47
3.47 KV store write performance comparison. 47

4.1 Logical overview of a RAID-6 system. 49
4.2 Bottom row of BDM used to compute parity for the Q device for a system with 7 devices and word size of 7. Gray boxes represent a 1, white a 0. 50

4.3 High-level architecture of the GPU-enabled RAID system. 51
4.4 Control flow in our GPU-enabled RAID system. 53
4.5 GPU parity computation kernel. 56
4.6 Write throughput for a file striped over 16 OSTs + 2 parity OSTs. 57
4.7 GPU encoding throughput. 59
4.8 Effect of number of disks on throughput (file size = 256 MB). 60
4.9 Effect of number of disks on throughput (file size = 1024 MB). 60
4.10 Read throughput with end-to-end data integrity. 61
4.11 Performance of NAS DC benchmark. 62
4.12 Average lookup time vs. size of the COLA. 63
4.13 Average lookup time vs. pointer density of the CONA. 64
4.14 Lookup per Second vs. size of the CONA. 64
4.15 PCIe SSD. 65

Chapter 1

Introduction

Today's large-scale, high-performance, data-intensive applications put a tremendous stress on data centers to store, index, and retrieve large amounts of data. While the capacity, performance, and the mean time to failure (MTTF) of a single disk has been improving, large-scale storage systems and parallel file systems (PFSs) can comprise tens of thousands of drives, thus bringing down the overall mean time to data loss (MTTDL) of the entire system to unacceptably low levels. For example, the Lustre-based Spider II PFS of the Titan supercomputer (No. 1 machine on the Top500 [1] list) comprises 20,000+ disks [2]. An exaflop machine in 2018 is projected [3] to host hundreds of thousands of drives to support the desired I/O throughput. Additionally, the rise of the real-time web, exemplified by technologies such as social media, photo and video sharing, and e-commerce demand minimal latencies, always-on availability and ever-growing capacity from their data stores. These requirements have fostered the development of a large number of high-performance parallel file systems [4–7], scalable NoSQL databases, such as Cassandra [8], Dynamo [9], and BigTable [10], data analytics systems like MapReduce and Hadoop [11,12], and recently key-value stores [13–19].

Arguably Key-Value (KV) stores are at the forefront, delivering the unprecedented scalability and performance demanded from the modern storage systems. KV stores are now ubiquitous, having become the default storage platform for many Internet services. They received substantial attention from both industry — powering e-commerce platforms [9], social media sites [20, 21], online multi-player gaming [15], deduplication subsystems [18], caching systems [17, 22] — and academia, e.g., FAWN-KV scalable and energy-efficient key-value store [19]. KV stores trade some functionality, such as the one provided by complex relational databases, for scalability and high throughput. An emerging trend for achieving low latency and high throughput is a solution, which utilizes both DRAM and flash by storing an efficient index for the data in memory and minimizing accesses to flash, where both keys and values are stored. A prime example of this is Facebook's recently announced McDipper [23], which is a flash-backed KV store compatible with the popular `memcached` protocol. Recent proposals are examining how to improve KV store performance in this area as well [13–19, 24]. Many of these system have shortcomings, however, such as expensive sorting and excessive read and write amplification, which is detrimental to the life of the flash.

Another trend in recent years is equipping large scale deployments with energy-efficient, high-performance co-processors, such as Graphics Processing Units (GPUs). On one end of the spectrum, GPUs power supercomputers such as Titan, currently one of the top supercomputer in the TOP500 list. On the other end, GPUs are also being used for data mining [25], computational finance [26], bioinformatics [27], and are being deployed in large scale cloud setups, such as Amazon's EC2 [28]. The appeal of the GPU lies in the high core count on a single chip and the high main memory bandwidth, delivering an order of magnitude higher throughput than DDR3, due to the wider bus and higher clock rate. Recent work has explored using these resources to accelerate compute-intensive I/O workloads, including RAID parity generation, encryption, and compression [29–31]. While this research proved the viability of GPUs to accelerate these workloads, we believe there are benefits gained by developing novel methods and data structures for deep integration of GPUs inside the storage stack, including enhanced performance, scalability, and reliability.

In this work, we propose comprehensive frameworks that leverage emerging technologies, such as GPUs and flash-based SSDs, to accelerate modern storage systems. For our accelerator-based solution, we focus on developing a system that features deep integration of the GPU in a distributed parallel file system. Previous research focused on identifying computationally intensive kernels, for instance hashing or compression, and performing a simple *kernel offload*. Although this approach provides some benefits, the excessive data movement over the PCIe bus easily becomes a bottle-neck, and is a major drawback. What is needed is a framework that build on the resources available in the file system and coordinates the workload in such a way that minimizes data movement across the PCIe bus, while exposing data parallelism to maximize the potential for acceleration on the GPU. To this end, our research aims to improve the overall reliability of a PFS by developing a distributed per-file parity generation that provides, end-to-end data integrity and unprecedented flexibility. To tackle the challenges associated with unlocking the full potential of flash-based SSDs to accelerated I/O workloads, we design a high-performance KV store built around a data structure that is tailored around the specific requirements of flash. It arranges data on flash in such a way as to minimize write amplification, which is detrimental to the flash cells. The system also delivers outstanding read amplification through the use of a trie index and false positive filter.

In this chapter, we provide the background and motivation for the research done in this dissertation.

1.1 Integrating GPUs in the Storage Stack

In recent years, GPUs from NVIDIA and AMD have shifted from closed peripherals used to render graphics images to inexpensive commodity parallel accelerators. They provide general-purpose APIs that can be used to accelerate many types of computations. While GPUs are mainly used in scientific workloads, recent studies are applying them to I/O

workloads [32–36]. These efforts have shown that GPUs can be used effectively for parity computation using Reed-Solomon coding [37] as well as other I/O workloads, such as hashing [38].

Furthermore, large-scale machines are beginning to be provisioned with GPUs. For example, the state-of-the-art Keeneland supercomputer [39], is a combination of Intel Nehalem and NVIDIA Tesla GPUs. Similarly, the No. 1 machine on the Top500 [1] list, Titan [40], is a hybrid architecture, with each node featuring two 16-core AMD Opteron processors and a Tesla X2090 GPU.

GPUs provide a cost-efficient solution compared to general purpose CPUs (GPPs), especially when GPUs are coupled with a few GPPs [41]. As a result, GPUs are quickly being adopted in a myriad of fields, ranging from scientific workload processing [42] to education in the developing world [43]. These architectures present opportunities to explore the utility of GPUs towards improving storage system reliability and performance.

Utilizing GPUs as commodity accelerators for computationally intensive storage primitives has been on the rise [44]. *stdchk* [45] uses hashing to detect content similarity between two successive checkpoint images. Several efforts [46–48] have attempted to improve the performance of hash computations by offloading them to the GPU. Similarly, GPUs have also been used to accelerate parity computation [49] and data encryption [50, 51] for storage systems.

Unlike pure hardware based solutions typical in storage systems, including hardware RAID controllers, GPUs are programmable. The best fault tolerance that a hardware RAID controller typically supports is a Reed-Solomon [52] implementation of RAID-6. In contrast, any number of coding techniques can be used in a software solution, including triple parity RAID or any implementation of RAID-6. As a result the programmability of the GPUs provides a unique opportunity to exploit the advances in parity encoding, such as minimum density coding schemes like Blaum-Roth [53] and Liberation codes [54].

Additionally, a GPU-accelerated software-based solution allows for a much higher implementation flexibility and increased fault tolerance. For example, implementing a per-file RAID scheme allows each file or directory tree to have a desired fault tolerance level. Small files can use a simple RAID-1, while large ones can use the state-of-the-art RAID-6 code. In a block-based RAID, it is difficult or impossible to directly map any lost sectors back to higher-level file system data structures. Furthermore, a client driven per-file RAID system does not impose any spatial limitation on the locality of drives, allowing data to be spread across the system and not just one location. Hardware RAID controllers, on the other hand, typically require all disks in an array to be co-located on the same blade. This can result in data loss because all the drives in the array can fail simultaneously, due to issues like power failure or over-heating.

Finally, an accelerator based solution can also allow each client to generate parity independently, opening the door for end-to-end data integrity checking to increase reliability. Typically data has to pass through several network interconnects and memory and storage hierarchies, all of which can introduce errors, albeit with very small probability. If absolute

data integrity is required, the client can choose to obtain parity as part of a read operation and check consistency of the data on demand.

1.2 Optimizing Key-Value Stores for Flash

KV stores serve variety of application that demand high-throughput and low-latency data access. The performance, capacity, and power consumption characteristics of current NAND-flash based SSDs make them an attractive medium for KV store [13–19, 24]. However, to extract the maximum performance out of the flash-based storage, it is necessary to use flash-aware data structures and algorithms.

SSDs differ significantly in the way they store data from regular HDDs. A major drawback of the flash memory is that it does not allow overwrites. Updates to an existing page are accomplished by a read/modify/write cycle, which reads the old page, applies the updates, and writes it to a new location on flash.The original page is marked as stale and is eventually recovered to the free page pool by the garbage collection process. As a direct consequence, metadata or any form of small update must be buffered. The same goes for any small write in general. The smallest size of data that can be written to an SSD is a NAND-flash page, typically 4 KB or 16 KB depending on the SSD model. Random subpage sized writes need to be avoided at all costs or risk a dramatic increase in write amplification. As a result, when stored on SSDs, data structures need to cache their metadata and write it to the drive as infrequently as possible. Ideally, the file format for the metadata as well as the data should never be updated in-place.

Moreover, SSDs have several levels of internal parallelism. A “clustered block” is a set of NAND-flash memory modules that can be accessed concurrently, typically 16 MB or 32 MB in size. If writes are performed in the size of at least one clustered block, they use all the available parallelism, and can reach the same performance as sequential writes [18].

We designed the proposed data structures around these requirements. Our proposed KV store writes data in large contiguous levels, which takes full advantage of the internal parallelism available in SSDs. Additionally, data is never updated in place. Crucially, index metadata stored per level is also immutable, which allows it to be persisted on flash without incurring large write amplification. Finally we optimize the design of our flash-based system to performs best under a read-heavy workload, which is a typical deployment of a KV store. Once warmed up, caches service many more GET than SET requests. For example, a recent analysis of workloads running on deployed KV stores show a GET to SET ratio of 30 to 1, which is significantly higher than what most synthetic workloads typically assume [55].

1.3 Research Contributions

The objective of this research is to leverage emerging technologies, including GPUs and flash-based SSDs, to accelerate modern storage systems. In the process, we develop frameworks and data structures that realize the full potential of these technologies to improve reliability and increase overall system performance. In the following, we highlight research contributions that we make in this dissertation.

1. We design, develop and analyze a system to utilize low-cost GPUs in conjunction with a PFS to provide fault tolerance and end-to-end data integrity. We capitalize the resources provided by the PFS, striping individual files over multiple disks, with the computational power of a GPU to provide flexible and fast parity computation for encoding and rebuilding of degraded RAID arrays. We attain end-to-end data integrity by performing encoding and decoding at the compute node, where data is produced and consumed. We implement our client-driven, per-file RAID in the widely used Lustre PFS [56], which will facilitate wider adoption of our system. Specifically, the contribution of this work are the following:

    - Parallelized and accelerated two state-of-the-art minimum density RAID-6 coding schemes – Blaum-Roth [53] and Liberation codes [54] – on GPUs using CUDA.
    - Analyzed the coding process and extracted fine and coarse-grain parallelism. We leveraged both types of parallelism to maximize the acceleration at GPUs.
    - Desiged a system featuring client-driven, per-file parity computation accelerated by a GPU. Unlike traditional hardware-based approaches it is flexible, as it can transparently switch between a per-file RAID-1 for small files to a desired sized RAID-6 as the file grows, and provides end-to-end data integrity guarantees.
    - Showed through a prototype implementation that client-side parity generation using a GPU imposes an acceptable overhead on the overall client performance and achieves a coding throughput of over 3.0 GB/s on each client, thus saturating the network.

2. We design and implement a comprehensive solution that uses novel data structures to deliver low read and write amplification on flash using fast, yet memory-efficient indexing and lookups. Our proposed Cache Oblivious No-Lookahead Array (CONA) structure addresses shortcomings of state-of-the-art KV stores, including expensive sorting and excessive read and write amplification. Writes to the CONA's log-like structure are never in-place and the efficient in-memory index only requires a single flash read per lookup, complementing current flash technology.

    Additionally, CONA offers a range of customization options that can be used to tune read and write amplifications, as well as tailor insertion versus lookup performance to a given application's requirement.

    The main innovation of this work is in developing data structures which feature:

- *Memory efficient index.* We utilize several (de)compression techniques, including entropy encoded tries [13], to minimize the memory footprint of the key lookup index.
- *Low read amplification.* Random read throughput of flash is a bottleneck in a flash-based system such as ours, so minimizing the number of flash reads is essential for good performance. We design the system such that it will perform at most a single flash read per lookup.
- *Controllable write amplification.* The data structures that power CONA provide predictable flash write amplification. We demonstrate how to bound this amplification, even as the key-value store grows very large.
- *Efficient use of flash.* We chose a log-like structure to store KV pairs in flash. The structure performs its writes in bulk and never in place, thus making it favorable for use with current flash technology.
- *Outperforming State-of-the-art KV stores.* On average, CONA outperforms SILT [13] and leveldb [24] by 2.5x and 2.8x in write throughput, respectively. CONA also simultaneously outperforms these KV stores in read throughput by 2.9x and 1.14x.

1.4 Dissertation Organization

The rest of the dissertation is organized as follows. In Chapter 2, we discuss the related work and background technologies that lay the foundation of the research conducted in this dissertation. In Chapter 3, we detail the design and implementation of the Cache Oblivious No-lookahead Array (CONA) key-value store. We present an evaluation of the performance characteristics of CONA using the YCSB benchmark suite and compare CONA to two state-of-the-art key-value stores. In Chapter 4, we present our work in accelerating storage primitives using a GPU. First, we detail a flexible, fault-tolerant, and high-performance RAID-6 solution. Our system utilizes low-cost, strategically placed GPUs to accelerate parity computation. We evaluate our system on a medium-scale cluster and demonstrate that approach and can provide an efficient alternative to specialized-hardware-based solutions. Next, we explore how the data structure proposed in Chapter 3 can be adapted for use with a GPU. We apply the resulting GPU-accelerated system to perform virtual to physical address translation as part of a Flash Translation Layer (FTL). We conclude the dissertation in Chapter 5 including a discussion of future opportunities for acceleration in the storage stack.

Chapter 2

Related Work

This dissertation focuses on opportunities to accelerate storage primitives. Specifically, we explore accelerating RAID encoding using a GPU, flash-acceleration inside a storage application, i.e. a KV store, as well as integrating a GPU at the storage device driver level. This section summarizes the prior work in GPU acceleration, as well as the state-of-the-art in flash-backed KV stores and out-of-core data analysis.

2.1 GPU-Accelerated Storage Systems

In this section we provide an overview of prior work in GPU acceleration of I/O primitives. We focus on extendable frameworks as well as GPU integration for a specific I/O intensive application.

2.1.1 CUDA Programming Environment

The NVIDIA CUDA or Compute Unified Device Architecture [57] is a programming framework that provides a C-like environment to program GPUs. CUDA defines C functions called *kernels* that can be executes by the multiple threads of the GPU. A CUDA-enabled GPU supports SPMT (single program, multi thread) applications, where a single kernel is executed by multiple processing threads on the streaming multiprocessors of the GPU. GPUs achieve the best performance when all concurrent threads have non-diverging control flows and an aligned data access pattern which allows memory accesses to be coalesced thus extracting the full bandwidth to the main memory of the GPU. CUDA uses page-locked memory to optimize the data transfer between the host (CPU) and the device (GPU). Due to the closed-source nature of the NVIDIA GPU drivers, most of the frameworks described below rely on CUDA to interact with the GPU. Recent effort, such a Gdev [58], are aimed at developing open source drivers for GPUs, which allow virtualization and time sharing of the GPU, as well as launching applications on the GPU directly from kernel space.

2.1.2 StoreGPU

StoreGPU [46] is a library that accelerates hashing primitives using a GPU. StoreGPU is one of the first attempts to utilize a GPU to accelerate storage related computationally intensive primitives. The work focuses on creating direct, as well as sliding window hashing modules. The experimental results provide a proof of concept that the GPU can indeed be used for accelerating the storage stack. The work highlights that the key to achieve the full potential for acceleration of the GPU lies in careful optimization of memory access patterns.

2.1.3 A GPU Accelerated Storage System

As a continuation of StoreGPU, Gharaibeh et al. [47] incorporate the proposed hashing modules in a FUSE-based content addressable storage system called MosaStore [59]. The work provides some insights into the system level challenges connected with the GPU integration, such as hiding data transfer and memory allocation overheads, simplifying the use of multiple GPUs and the overall integration effort. The authors also evaluate the impact of the system on both compute and I/O bound applications, showing that offloading hashing primitives to the GPU does not introduce a bottleneck for competing I/O-intensive applications. Each file in MosaStore is divided into equally sized blocks and a metadata manager maintains a block-map for each file. On a write, the system retrieves the current block-map, computes hashes for every new block to be written using the GPU, and only stores blocks that have no match in the previous version of the file.

2.1.4 CrystalGPU

CrystalGPU [60] is a framework that builds on top of the CUDA toolkit and abstracts some of the complexities associated with doing computation on the GPU. The framework preallocates and reuses memory buffers, which amortizes allocation time that in CUDA can be significant. It can also overlap kernel computation with communication. The framework provides the *job* abstraction, which is a reusable container with its own input and output buffers. A target application requests a free job, populates the input buffer and specifies the kernel function to be executed on the GPU. After submitting the job, the application can either block or provide a callback function to be executed once the output is generated and copied to the host output buffer.

2.1.5 GPUstore

The current state-of-the-art NVIDIA and AMD proprietary drivers do not support accessing the GPU from kernel space, therefore GPU accelerated storage systems rely on a userspace daemon to execute the GPU requests. GPUstore [61] is one such framework used to accelerate the Linux AES cryptographic library on the GPU [62, 63]. Encryption speedup of up to 6× is shown, but only for large page sizes, in the order of megabytes, limiting the practicality of the system. On important feature of GPUstore is that it substantially decreases the latency of a GPU kernel launch by keeping the kernel alive even after it has completed its execution. In contract to standard approaches, GPUstore provides a custom messaging mechanism that can be used to input data and receive results from the GPU without terminating the kernel. GPUstore incurs full latency only when a GPU kernel that provides a different service needs to be loaded.

2.1.6 A Lightweight, GPU-Based Software RAID System

Gibraltar RAID [29] is a GPU-based RAID system that focuses on accelerating Reed-Solomon [52] parity codes to create a block based RAID in user-space. Gibraltar RAID implements the standard C library calls *pread*() and *pwrite*. The system bypasses the Linux buffer cache and instead implements its own stripe cache to mitigate some of the performance degradation associated with an incomplete write to a stripe. The cache is optimistic, such that if a write request does not fill an entire stripe, the system delays the expensive read-modify-write operation in anticipation that the stripe will be written to in the near future. The system is benchmarked against the popular *md* drive included in the Linux operating system, showing comparable results for streaming reads and writes. Performance of degraded reads and writes, on the other hand, is $2-3\times$ faster, due to the higher encoding throughput of the GPU.

2.1.7 Real-time parallel hashing on the GPU

Alcantata et el. [64] present a parallel hashing algorithm that is suitable to build large hash tables in the GPU in real time. The algorithm is a hybrid based on the classical FKS perfect hashing scheme [65] and the recently developed cuckoo hashing [66], with the first stage using FKS to divide items into small buckets, and the second subdividing each bucket in parallel into three cuckoo hash tables. The system is applied to two applications: 3D surface intersection for moving data and image matching using geometric hashing. An interesting approach which is applied to these two uses cases is that hash tables are continuously recreated, thus making both construction time and lookup time important. The algorithm is compared to a state-of-the-art GPU parallel radix sort [67], achieving similar construction times with a 5× speedup in lookup time at the expense of 40% increase is storage space.

2.1.8 Parallel Lossless Data Compression on the GPU

Patel et al. [31] present a parallelization of the popular bzip2 compression pipeline: Burrows-Wheeler transform (BWT), move-to-front transform (MTF), and Huffman coding. The contribution of the work lies in designing parallelizations for the three major stages of the bzip2 pipeline to make it more suited for the GPU. A novel, merge-sort-based BWT is introduced, which minimizes global communication. The strictly serial MTF is replaced by a parallel scan-based version. Finally, the Huffman tree is constructed using a parallel reduction scheme. Despite all these optimization, the resulting algorithm is 2.78× slower than the serial version. The authors identify the string sort in the BWT as the major bottleneck, because it produces high thread divergence, resulting in implicit serialization. The authors conclude that even given the worse performance, the GPU can be used as a compression co-processor, freeing CPU resources at the expense of traffic over the PCIe bus.

2.1.9 Accelerating SQL Database Operations on a GPU with CUDA

Bakkum and Skadron [68] show promising results in accelerating a subset of SQL queries on an in-memory database using a GPU. Experimentation shows a 20 to 70× speedup over a SQLite running on the CPU. The framework produces such big speedups by exploiting the embarrassingly parallel nature of the queries selected, such as, COUNT, SUM, MIN, MAX, etc., which perform the same operation for each row in the data. A major limitation of the work is that only databases that fit in the GPU's main memory are considered, which is significantly smaller than typical database sizes that are in the terabytes. The authors outline a possible approach that stream rows of the database through GPU memory, but concede that such data movement will become a bottleneck, likely erasing any performance benefits of using the multiprocessors of the GPU. The implementation also does not include many important SQL operations, most notably JOIN. The authors conclude that the GPU can significantly speed up select SQL queries albeit with limited applicability.

2.1.10 Efficient Data Management for GPU Databases

As a continuation of the above work, Bakkum and Chakradhar present a framework that can accelerated relational database primitive using a GPU on an arbitrary large database [69]. More moderate speedups in the 4 to 8× range are reported. The main insight of the work is to create a custom data structure, called a *tablet*, inspired by Google BigTable [10]. The rows of the table are split vertically into self-contained *tablets* which contain all the meta data and and fixed and variable with data. The authors show that the best performing strategy is to map a region of the host's main memory into the GPUs memory space, allowing the device

to read and write to the pinned pages using implicit transfers over the PCI bus. The results show that this strategy produces over 3× speedup over an explicit data copy between host and device. For maximum performance, the framework ensures that all reads and writes to the pinned memory are coalesced. Because of some restriction that require all writes to be aligned to 64 bytes in order to be coalesced, the authors use a relaxed two-stage write, where results from a query are first written to the GPUs global memory, where requirements for coalescing are less strict.

2.1.11 Rgem: A responsive GPGPU execution model for runtime engines

RGEM [70] is a responsive GPGPU execution model aimed at addressing issues faced by real-time applications executing on a GPU. As described above, CUDA, the prevalent framework used to perform general-purpose processing on a GPU, uses non-preemptive data copy and kernel execution calls. Thus, a higher priority process cannot preempt a lower priority one already executing a kernel on the GPU, which can cause prohibitively large blocking times in the context of a real-time system. To address data transfers, RGEM break each non-preemptive DMA transfer to and from the GPU in small chunks and makes scheduling decisions at chunk boundaries. RGEM also uses a single process, or *context*, to run all kernels on the GPU in order to take advantage of the fact that NVIDIA Fermi GPUs can execute multiple kernels at the same time if they are launched from the same process. Thus, multiple processes can timeshare the GPU using the RGEM queuing which respects the process' priorities.

2.1.12 Operating Systems Challenges for GPU Resource Management

Kato et al. [71] present the state-of-the-art in GPU resource management. The work provides an overview of challenges faced with GPU programming and resource management models, GPU channel, context and memory management, as well as scheduling and virtualization. The system stack for General Purpose GPU (GPGPU) computing is described, including interactions between GPU device driver, user-space runtime driver and popular GPU programming frameworks like CUDA and OpenCL. GPU command submission paths are traced, describing the location and size of the associated buffers. The work includes a discussion of trade-offs in different GPU scheduling approaches and their implications to preemption, parallelization, and data movement. The paper concludes by stressing the importance of open-source tools to further research in the GPGPU domain.

2.1.13 PTask

PTask [72] is an OS abstraction to manage GPUs as compute devices. The PTask API supports a dataflow programming model consisting of *ptasks* and input and output *ports* connected by *channels*. All the objects are OS-managed, which allows the kernel to provide system wide guarantees like fairness and performance isolation. The framework can be used to compose an application into a DAG of PTasks, possibly eliminating some data movement otherwise required by successive PTasks. The system is evaluated by offloading AES encryption of a FUSE-based file system called EncFS [73] using the GPU. Speedup of up to 28% are reported for sequential writes and 17% for read over a CPU implementation.

2.1.14 GDev

Gdev [58] is a framework that allows user-space, as well as kernel process to use a GPU as a "first-class" computing resource. The work provides some important contributions. First, it provides an alternative to the NVIDIA proprietary runtime system and its user-space only APIs. Gdev exposes the smae APIs directly to OS processes, so that the file system and network stack can take advantage of the GPU directly. Gdev also addresses another limitations of the proprietary drivers by allowing virtual memory size to exceed the physical memory size of the GPU. Moreover, it defines APIs to allow memory to be shared between GPU contexts, which also is not supported otherwise. Finally, Gdev features a scheduling scheme that can virtualize a GPU, thus enhancing isolation between processes.

One notable optimization presented in the paper, is the so-called split transaction, which addresses the fact that memory transfers between the host and the GPU often require two copies, one into a memory-pinned buffer on the host side and one from the pinned buffer into GPU memory. Rather then treating the end-to-end transfer as one transaction, Gdev splits the transfer into small chunks and overlaps the two transfers. Thus, only the first and last chunk incur the full length of the transfer. This optimization is only used for large transfer when the DMA engine is used. The authors note that for small transactions (below 4 KB on the hardware used) mapping the device memory space onto host memory and performing direct I/O can be much faster (around 4× for host to device transfer).

The evaluation shows the performance of Gdev in accelerating the hashing primitives in the eCryptfs [61] encrypted file system. Gdev peforms on par with a system described previously in this section called StoreGPU. However, Gdev is able to outperform StoreGPU when there are competing task for the GPU, as Gdev can assign priorities to the tasks independently. Finally, the effects of shared device memory are evaluated. The results show that shared memory can produce up to 50% speedup for an application pipeline that needs to pass large amounts of data, because this approach save an explicit copy of the data from device to host and back to device. For small intermediate data sizes, however, the speedup of this technique is minimal.

2.1.15 Shredder: GPU-accelerated incremental storage and computation

Shredder [74] is a high performance content-based chunking framework that that supports incremental storage systems. Shredder offloads the Rabin fingerprint [75] computation performed to detect chunk boundaries to the GPU. Several common optimizations, such as asynchronous execution and memory coalescing, are introduced. Speedups of up to 5× are reported over a parallel chunking algorithm on the CPU. Two usecases of the system are presented: an extension of HDFS that support incremental MapReduce computation and an incremental cloud backup system.

2.1.16 Limitations of Previous Work

The frameworks presented fall into several categories: OS level management for GPUs [71, 72], storage primitive acceleration [29,31,46,60,64,69], GPU kernel space development [58,61]. The main drawback in the described research is that it frames problems as compute-intensive storage kernels, such as encryption and compression. That allows a straightforward comparison to a CPU-based system, but puts the GPU at a great disadvantage as it is bottlenecked at the PCIe bus. The result is modest acceleration or indeed a slow down, as is the case with the compression work described. What is lacking in previous research is a framework tailored around the GPU itself to solve storage related problems in a fundamentally different way, rather than plugging in a GPU into the existing storage stack. What are needed are novel data structures and methods that alter storage systems to make them amenable for GPU acceleration.

2.2 Key-Value Stores

In the following section we give a brief overview of the state-of-the-art in key-value stores, with a focus on memory-efficient flash-backed KV stores.

2.2.1 Bufferhash

Bufferhash [16] divides the flash space into a number of partitions, such that a key can reside only in one of the partitions. Each partition maintains a small in-RAM hash table for the KV pairs stored in the partition. The hash tables are implemented by using the cuckoo hashing with two hash functions that help to improve the space utilization efficiency at the cost of more hash table lookups per key. Once the in-RAM hash table of a partition becomes

full, it is flushed to flash. Subsequently, a new hash table of the same size is instantiated in RAM for the incoming KV pair insertions. A given key may reside in any of the *incarnations* of the hash table, either the one currently in-RAM or any of the ones stored on flash. To minimize the number of flash reads each Bufferhash maintains a Bloom filter for every *incarnation* of every partition. Thus, unnecessary flash reads only occur on flash positive. In order to look up a key in a partition, the BufferHash identifies a specific partition where the key resides by using a hash function. Next, it examines the chain of the Bloom filters linked to the partition in the reverse order of their creation times. Finally, for each of the Bloom filters where the key is found, the BufferHash looks up the key from the associated hash tables stored either in the memory buffer or on the flash. Note that BufferHash can consume considerable amount of RAM (and flash) because hash tables have low load factors (around 50%) and all the Bloom filters for all the partitions need to be kept in RAM.

2.2.2 Bloomstore

Bloomstore [14] is closely related to Bufferhash. It too creates disjoint partitions and a chain of Bloom filters for each flash page. The key difference is that Bloomstore buffers not only the most recently inserted KV pairs, but the Bloom filters as well. Thus, Bloomstore can achieve sub-byte amortized in-RAM index size per KV pair and the cost of multiple flash reads per lookup. As an optimization, Bloomstore can issue parallel reads in order to obtain the relevant Bloom filter chains, in order to take advantage of the multiple read/write channels exposed by some high-performance SSDs. The overall design, however, is heavily bottlenecked at the SSD throughput, as a single lookup can results in tens of flash reads.

2.2.3 Skimpystash

Skimpystash [15] is RAM space-efficient key-value store that trades low RAM usage for multiple flash reads per lookup by moving pointers that locate key-value pairs to flash. Skimpystash maintains an in-RAM hash table, called the hash table directory, to map keys to their locations in flash. The in-RAM hash table consists of a set of buckets that contain a flash pointer and a bloom filter. Collisions caused by multiple keys mapping to the same hash are resolved by linear chaining. The in-RAM pointer points to the KV pair that was most recently inserted. Each KV pair on flash contains a flash pointer pointing to its predecessor in addition to the KV pair itself. SkimpyStash regards the flash as an append-log and appends the inserted KV pairs to the log sequentially. Insertions are performed into an in-RAM data buffer of a flash page size, which is periodically flushed to flash. The bloom filter associated with each bucket is critical in the SkimpyStash design as it reduces flash reads for lookups for non-existent keys. Note that SkimpyStash may incur multiple flash page reads for a key lookup if the key does exist. Thus, the average bucket size is a critical design parameter as in the worst case the number of flash reads required to perform a lookup

is equal to the number of KV pairs chained a bucket. A final optimization compacts record chains onto flash pages, reducing the number of flash reads by half on average.

2.2.4 SILT

SILT [13] is a high-performance flash-backed key-value store that incorporates three different KV stores all optimized for a different purpose. KV pairs are initially inserted into an on-flash, write-optimized store, called LogStore, and from there are gradually migrated to increasingly more memory-efficient stores. In the LogStore, KV pairs are ordered by their insertion time and are indexed by an in- RAM hash table. To improve the RAM space utilization, the hash table is built with cuckoo hashing [66] and only stores a 15-bit tag rather than full keys. Moreover, SILT boosts the hash table occupancy to about 93% by increasing its associativity. Each hash table entry consumes 6 bytes, consisting of a 15-bit tag, a single valid bit, and a 4-byte offset pointer. Once a LogStore fills up, it is converted in the background into a more memory-efficient, static data structure, called SortedStore. SortedStore maintains KV pairs in a sorted key order on flash and indexes them with a very compact in-RAM index representation, called an entropy-coded trie. In contrast to the 6 bytes/key RAM indexing overhead of LogStore, SortedStore achieves 0.4 bytes/key RAM indexing overhead. SortedStore, however, does not allow insertions or deletions. Therefore, every time a LogStore fills up it needs to be merged with an already existing and potentially very large SortedStore and the resulting SortedStore re-written to flash. This process can result in prohibitively large write amplification, which is why SILT first converts a LogStore into a more memory efficient HashStore. After a configurable number of HashStore have been created they get merged into a large SortedStore. In order to keep the average index cost per key low, most KV pairs ($> 80\%$) are stored in SortedStore, with the rest in several HashStores and a single LogStore.

2.2.5 Flashstore

Flashstore [18] is a high throughput key-value store that uses flash as a non-volatile cache between RAM and disk. It is designed to store the working set of KV pairs on flash and as the working set changes over time, Flashstore destages unused KV pairs to disk. KV pairs are stored in a log-structure on flash to exploit faster sequential write performance. Flashstore uses an in-memory hash table for indexing using a variant of cuckoo hashing [66]. The in-memory hash table stores compact key signatures instead of full keys so as to strike trade-offs between RAM usage and false flash read operations. The RAM usage is 6 bytes per KV stored on flash and uses one flash read per lookup.

2.2.6 FAWN

FAWN [19] uses an array of embedded processors equipped with small amounts of flash to build a power-efficient cluster architecture for data-intensive computing. The FAWN-KV store uses an in-memory has index to map 160-bit keys to values stored on flash. Similar to other designs described above, FAWN stores an in-RAM hash table that contains a 15-bit key fragment, a valid bit, and a 4-byte pointer into into flash, for a total of 6-bytes per entry. The key fragment is used to avoid with high probability triggering unnecessary flash accesses. Consistent hashing is used to distributed the key ranges between the nodes in the system. FAWN-KV also allows configurable replication and can efficiently perform joins and leaves of nodes from the system.

2.2.7 Limitations of Previous Work

The previous research described in this section argues against DRAM only KV stores and instead advocates a KV store that features an efficient index for data in memory and minimizes accesses to flash, where both keys and values are stored. The proposed solutions, however, all have shortcomings. For example, the indexes either have a large footprint in memory or compromise performance by keeping a part of the index on flash or disk, thus incurring many flash reads per key-value lookup in the worst case. Some designs, like SILT, involve complicated and repeated conversion and merge operations running in background. Each conversion operation requires reordering KV pairs on flash, which competes for a significant amount of I/O resources.

2.3 Out-of-core I/O Processing Systems

In this section, we describe previous research related to systems that perform out-of-core data analytics by using an accelerator or the computational resources of the storage device itself and thus freeing CPU and memory resources of the host.

2.3.1 Zero-Copy I/O Processing for Low-Latency GPU Computing

Kato et al. [76] present a new zero-copy I/O processing scheme for GPU computing. The framework allows I/O devices to directly transfer data to and from the main memory of the GPU by directly mapping the I/O address space of a device, such as a sensor, into the virtual address space of the GPU. The framework is designed for Cyber-physical systems,

which monitor and control real world phenomena. In this context, real-time constraints are a major challenge. By eliminating the extra data copy usually required to transfer data from sensors and other I/O device to a GPU, the overall latency is brought down acceptably low to where even a real-time system requiring microsecond response times can take advantage of the computational power of the GPU. The results show a 33% reduction in overall computational cost including sending and receiving data to the GPU.

2.3.2 Fast, Energy Efficient Scan inside Flash Memory SSDs.

Kim et al. [77] study offloading a key database operation, a scan, to the computing module inside an SSD. In the proposed framework, the scan is applied on-the-fly as data is accessed, which drastically reduces the amount of data that needs to be transfered to the host. The experimental results show that the proposed approach can deliver a 13× speedup for the scan operation over a conventional host side implementation. Additionally, the in-storage processing systems offers up 7× energy savings as well.

2.3.3 ActiveFlash

ActiveFlash [78] is a novel framework, which performs data analysis on the embedded controller of an SSD device. Performing analysis so near the data and with minimal involvement of the host machine has many benefit. For one, it eliminates redundant reads and writes to the storage system in a typical HPC setup with multiple rounds of simulations followed by data analysis. Second, by performing analysis in-situ the system is not constrained by the typically limited bandwidth to the central storage system. Finally, the decreased data movement combined with energy efficiency of flash, has a great energy and performance/energy characteristics. On-the-fly vs. idle time data analysis scheduling is examined, showing that on-the fly policy offers significantly reduced I/O traffic, while idle-time maximizes storage utilization given the bursty I/O patterns of typical HPC applications. The authors also investigate a hybrid ActiveFlash model, where only a fraction of the data analysis is performed by the controlled and rest done by the host's CPU, showing that this technique when combined with the idle-time policy can sustain high application data generation rate.

Chapter 3

Flash-Accelerated Systems

In this chapter, we describe our contribution in the flash-accelerated storage systems space. Today's large-scale, high-performance data-intensive applications have put a tremendous stress on data centers to store, index, and retrieve large amounts of data. Key-value (KV) stores, in particular, have been employed by both industry and academia to address the performance and scalability requirements of these workloads. A recent trend for improving the efficiency and performance of KV stores is to utilize flash that stores both the keys and values, in conjunction with DRAM that stores indexing data to minimize flash accesses.

In the following, we present the design and implementation of the CONA KV store, a comprehensive solution that uses novel data structures to deliver low read and write amplification on flash using fast, yet memory-efficient indexing and lookups. We design the Cache Oblivious No-Lookahead Array (CONA) structure to address shortcomings of state-of-the-art KV stores, such expensive sorting and excessive read and write amplification. Writes to the CONA's log-like structure are never in place, which suits current flash technology, while the CONA's efficient in-memory index only requires a single flash read per lookup. Additionally, CONA offers a range of customization options that can be used to tune read and write amplifications, as well as tailor insertion versus lookup performance to a given application's requirement.

The main innovation of this work is in developing data structures which feature:

- *Memory efficient index.* We utilize several (de)compression techniques, such as entropy encoded tries [13], to minimize the memory footprint of the key lookup index.
- *Low read amplification.* Random read throughput of flash is a bottleneck in a flash-based system such as ours, so minimizing the number of flash reads is essential for good performance. We design the system such that it will perform at most a single flash read per lookup.
- *Controllable write amplification.* The data structures that power CONA provide predictable flash write amplification. We demonstrate how to bound this amplification, even as the key-value store grows very large.

- *Efficient use of flash.* We have chosen a log like structure to store key-value pairs in flash. The structure performs its writes in bulk and never in place, thus making it favorable for use with current flash technology.
- *Outperforming State-of-the-art KV-stores.* On average, CONA outperforms SILT [13] and leveldb [24] by 2.5x and 2.8x in write throughput, respectively. While simultaneously outperforming them in read throughput by 2.9x and 1.14x.

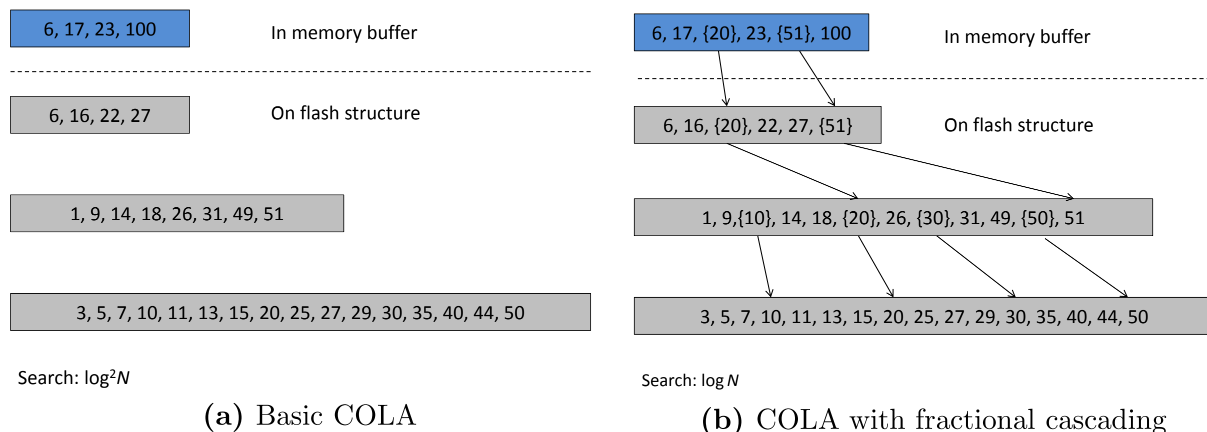
We focus on designing a system that performs best under a read-heavy workload as many KV stores are deployed as caches. Caches once warmed up service many more GET than SET requests. For example, a recent analysis of workloads running on deployed key-value stores show a GET to SET ratio of 30 to 1, which is significantly higher than what most synthetic workloads typically assume [55]. We show that CONA can outperform SILT and leveldb in read throughput under a read-heavy workloads by 3.4x and 1.3x, respectively. This performance is not achieved at the expense of insertion throughput. We demonstrate that CONA can also outperform other KV stores by as much as 5x in insertion throughput.

3.1 Design

In this section, we outline the design of our system and highlight some of its key features that distinguish it from state-of-the-art KV stores.

3.1.1 Background

The data structure that we design for CONA is based on the Cache-Oblivious Lookahead Array (COLA). COLA was proposed by Bender et al. [79] as an alternative to the traditional B-tree, and offers several orders of magnitude better insertion performance at the expense of a small degradation of lookup performance over a B-tree. Figure 3.1a depicts the structure of COLA. The basic COLA (on the left) consists of an in-memory buffer of size B ($B = 4$ in this example) and a number of *levels* stored on disk or flash, with each level doubling in size ($B, 2B, 4B$, etc.). Unlike the in-memory portion of the COLA, each level of flash contains two buffers or *cells*, at least one of which is visible (the figure shows one visible cell at each level). Visible in this context means that data is served from these cells during a read. A COLA can also have up to one "shadow" cell at each level, which is used for merging. Data in each level is sorted in ascending order. An insertion is performed into the in-memory buffer, which is also kept sorted. As a consequence, every insertion incurs a sort of the buffer. Once the buffer is full, it is copied into one of the cells of the first level. Once two cells of any given level become full, they are merged into a "shadow" cell of double the size at the next level, while still keeping entries sorted by key. The merging process can be



(a) Basic COLA

(b) COLA with fractional cascading

Figure 3.1 Cache Oblivious Lookahead Array (COLA).

deamortized by performing $2k + 2$ merge operations per insertion, where k is the number of levels. Deletions are implemented by inserting a new key-value pair, so every level of the COLA can potentially hold any given key with possibly a different value association. To perform a search, every level needs to be searched sequentially and the results from the top most level returned. This can be implemented as a binary search at each level, for a total complexity of $\log^2 N$, where N is the number of entries stored in the COLA.

Search performance is improved by using fractional cascading [80] (Figure 3.1b). A portion of the data at each level (1/3 in this example) is replaced by a *lookahead pointer* into the next level. With this optimization, in the worst case, only a constant number of elements need to be searched through at each level (the ones between the pointers to the left and right). Thus, the search performance is improved to $\log N$.

This version of COLA fits many of the requirements listed in the previous section: (i) writes into the flash are never in place and can be performed in bulk on a merge; (ii) write amplification can be adjusted based on size of the in-memory buffer as well as the number of levels of the COLA; and (iii) the structure changes relatively infrequently, only when a merge is completed.

However, there are several drawbacks of using the COLA described so far as a generalized key-value store. For example, $\log N$ flash reads are needed per lookup, which can add significant cost as the COLA grows large. Additionally, a significant portion of the useful data storage in the structure is wasted due to pointers; 50% of the COLA space in the original formulation of the structure is reserved for pointers. The main reason that the pointers are so dense is to ensure that the nearest pointer in each direction can be located with a short, constant traversal. Finally, write performance can be significantly improved if the deamortized $2k + 2$ merge operations are taken off the write path and instead are performed in the background.

In the following section, we describe how we improve the COLA structure to address all these shortcomings.

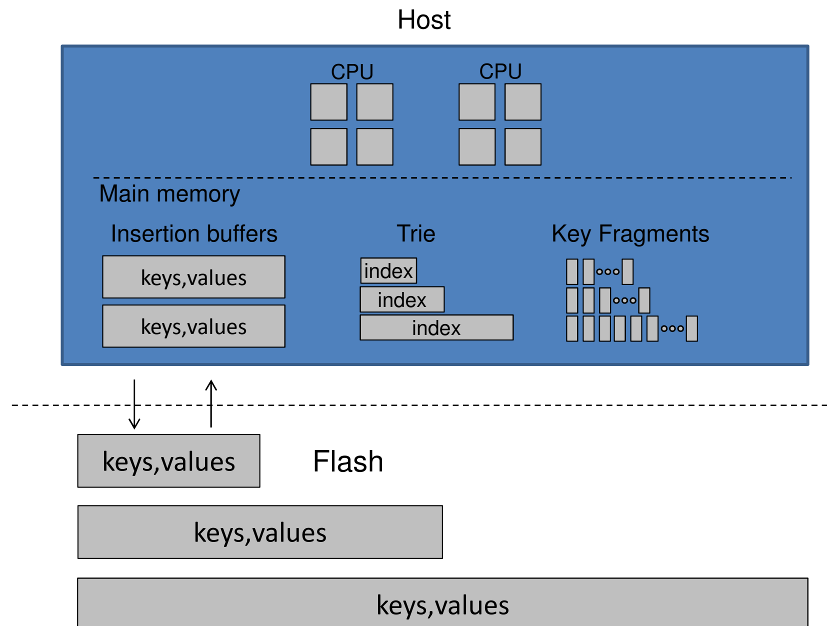


Figure 3.2 The Cache Oblivious No-lookahead Array (CONA).

3.1.2 CONA Architecture Overview

Figure 3.2 illustrates our system, which we call a *Cache Oblivious No-lookahead Array (CONA)*. It has "no lookahead" because we remove the lookahead pointers and replace them with an in-memory index. To address the other shortcomings of the COLA we create a false positive filter to improve read throughput and introduce extra memory buffering in order to remove merging operations from the write path.

In-memory Index The biggest design change we make to the COLA is to eliminate the dense pointers stored inline with the data and replace them with an in-memory index structure. Eliminating the pointers has several benefits. First, it frees up a significant amount of space inside the COLA for data. This results in a COLA of lower depth and, hence, in fewer levels that need to be searched for a particular key. Second, due to the way the COLA and pointers are constructed, certain insertion workloads can eliminate any benefits of storing pointers. For example, inserting items in ascending key order means that for any given level of the COLA, all entries at the next level have lower key values. This means that all pointers are at the front of the cell rather than randomly dispersed between data. This effectively degrades lookup performance from logarithmic to linear; in the worst case of searching for the lowest key in the store, every single entry needs to be sequentially considered.

The original COLA formulation has very strict memory constraints that allow only a single in-memory buffer of size B. We are able to remove the pointers by relaxing the memory

constraints on the COLA to accommodate an in-memory index. In order to make this approach viable, we choose a very memory efficient index called an entropy encoded trie [13]. A trie is a prefix tree data structure that stores an array of keys, with each leaf representing a key. The internal nodes represent the longest common prefix for a subset of the keys in the trie. This data structure can be used as an index into a cell of the COLA, whose entries are sorted in key order. A common trie implementation can be very memory inefficient if it relies on memory pointers, each 2 to 8 bytes long. We opt for an entropy encoded trie, which can store an index with as little as 0.45 bytes/entry on average. In Section 3.2.1 we show how we optimize the tries for every level of the CONA to extract maximum performance at a cost of only 0.67 bytes of memory per entry.

False-positive Filter The next optimization necessary to significantly improve the COLA's read performance is to reduce the number of flash reads per lookup. At $\log N$ flash reads per lookup where N is the number of entries, the COLA is too slow even for a moderate size N. Most prior approaches use Bloom filters for the probabilistic membership test [14–16, 24]. Instead of going with a Bloom filter, we leverage the trie index and store a small k-bit fragment of each key. Key fragments are stored in an in-memory array which has the same order as the sorted data on flash. During a lookup flash is only read if the key fragment matches.

This solution is more memory-efficient than a Bloom filter at low false positive rates and is much faster to construct [13]. The probability of a false positive retrieval for a k-bit fragment is $1/2^k$, which is better than a comparable sized Bloom filter [13]. We utilize a 16-bit fragment of the lowest order bits of the key, so the probability of a false positive is 0.0015%, i.e., on average 1 in 65536 flash reads will be unnecessary. The filter is very efficient to construct as all that is required is copying the least significant order bits from each key. Building a Bloom filter from scratch would be much slower because it requires hashing every key multiple times. This would have added a lot of unnecessary overhead for the frequently changing top most levels of the CONA.

The structure of the CONA provides an opportunity to minimize the memory necessary for the filter. Our targeted characteristics are a single flash read per lookup. In this case, all but the last levels of the CONA require a false positive filter. As each level of the CONA doubles is size, the last level contains about half of the keys. Hence, only half the keys require an in memory filter, so on average the 2-byte filter costs just 1 byte per key.

This approach can be extended to provide a false positive filter for all keys. Such a filter may be necessary if the target workload is anticipated to perform many lookups for keys not stored in the CONA. Optionally, the filter for the last level can be created with higher false positive probability by storing just a 1-byte fragment. This is possible because filters for any level of the CONA can be configured independently. This will result in a filter with an average fragment memory consumption of just 1.5 bytes, which can determine that a key is not present in the store with close to zero flash reads on average.

Background Merge The final design change that completes our CONA data structure is the addition of an extra in-memory buffer. A second insertion buffer has several benefits. First, doubling the buffer allows us to reduce the deammortization cost per insert by half. Hence, instead of performing $2k + 2$ merges per insertion, the CONA requires just $k + 1$, where k is the number of levels in the CONA. More importantly, these $k + 1$ merges can now be performed in the background by a dedicated merging thread. In Section3.2.1 we show that this optimization has a significant effect on insertion performance, especially in a read-mostly workload. Note that simply doubling the size of the insertion buffer is not as beneficial. The buffer needs to be sorted after every insert, so keeping its size minimal is essential for good write throughput. In Section 3.2.1 we detail the performance implications of varying the buffer size of the CONA.

3.1.3 Read and Write Workflow

Figures 3.3 shows how entries are inserted into a four level CONA. Note that only the key-value cells of each level are depicted. As discussed above, the CONA also maintains an in-memory trie index and key fragment array for each occupied cell. The CONA in Figure 3.3a has both in-memory buffers, as well as a single cell at the first two levels full of entries (represented by blue). The remaining visible cells as well as all shadow cells are empty (represented by gray). The arrows connecting the occupied cells follow the order in which a read query would search the cells. A read performs a binary search of the in-memory buffers in the sequence showed. If the desired entry is not found, the CONA consults the trie for the cell in the first level. If the key fragment stored at the index pointed to by the trie matches the search key, the associated value is returned from flash. If the fragment does not match, the search follows to the next cell and the process repeats until either the key is found or the last occupied cell in the chain is reached.

Both in-memory buffers are full in Figure 3.3a, so before the CONA can accept any more writes *Buffer 2* needs to be copied into an empty cell at the first level and then the buffers need to be swapped. The result is show in Figure 3.3b. Now *Buffer 1* is empty and ready to receive new writes. At this stage *Level 1* has two occupied cell, so the background merge process starts merging these two cell into the "shadow" cell of *Level 2*. Once *Buffer 2* is filled with new entries, the background process has completed the merge into the "shadow" cell at *Level 2*. As that creates two full cells at *Level 2*, the background process has also partially merged those cells into the "shadow" cell at *Level 3* (Figure 3.3c).

At this stage, *Buffer 1* is once again full. This time, however, there is no empty cell in which to copy *Buffer 2*. In order to make space, both cell at *Level 1* are discarded and the shadow cell at *Level 2* which they merged into gets linked in their place. Now *Buffer 2* can be copied into *Level 1* and the buffers once again swapped (Figure 3.3d). By the time *Buffer 1* is filled, the two cells at *Level 2* have been completely merged into the shadow cell at *Level 3* (Figure 3.3e). Notice that unlike in Figure 3.3c, this time 50% more merges had to be performed as the CONA now has an extra level. As there is free space at *Level 1*,

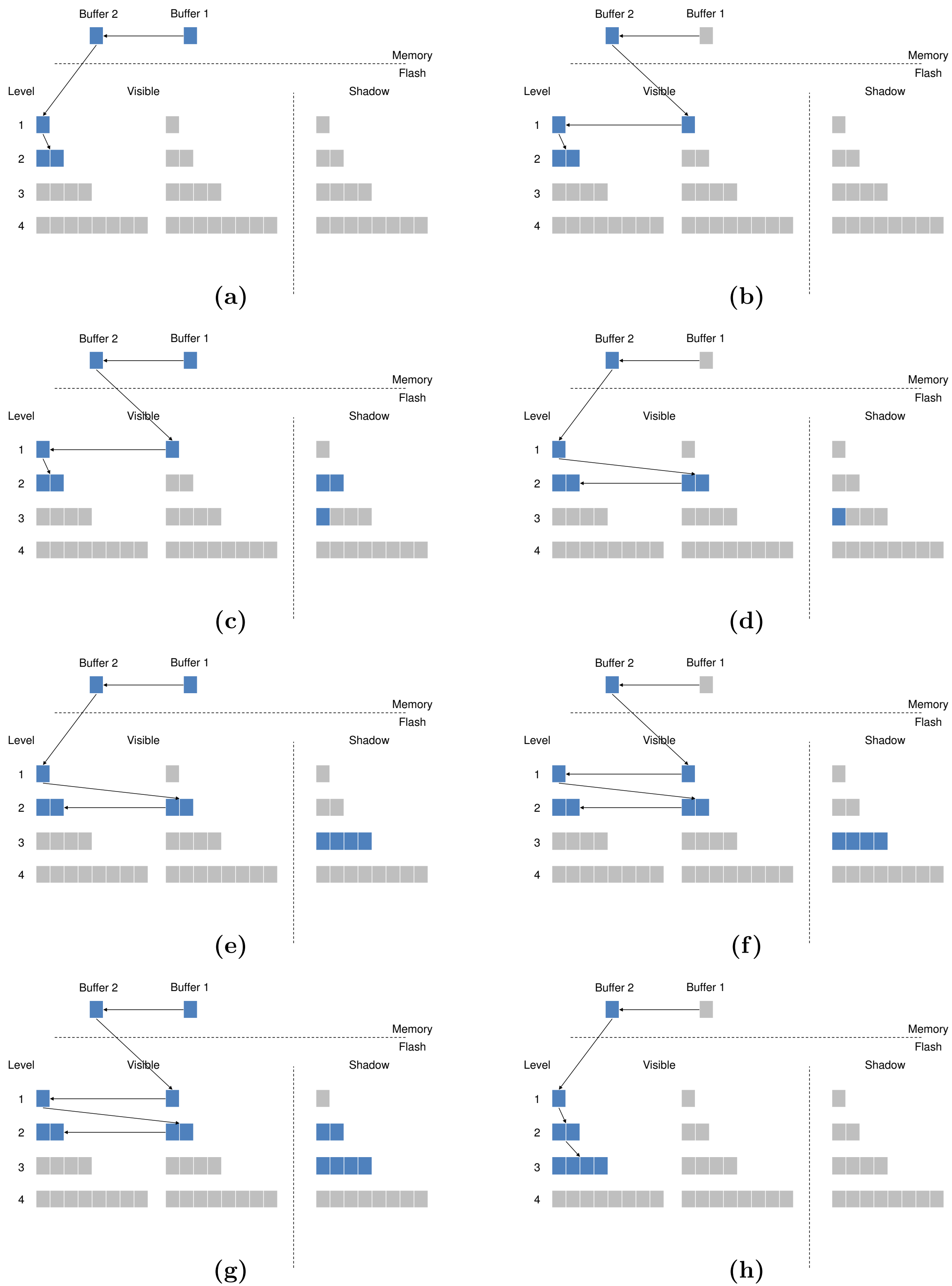


Figure 3.3 Merge process absorbing writes into the CONA

Buffer 2 is copied down and the in-memory buffer once again swapped (Figure 3.3f). During the next insertion round, the cell at *Level 1* are merged into the "shadow" cell of *Level 2* (Figure 3.3g). Figure 3.3h completes the process by discarding all four visible cells at *Level 1* and *2* and replacing them with the two shadow cell. *Buffer 2* is then copied into *Level 1* and the buffer once again swapped.

Upon completing these steps, the KV store has been transformed from a 2-level to a 3-level CONA with a single cell linked at each level. This process can repeat indefinitely, adding an extra level to the CONA. One important thing to note is the transformation that happens between Figure 3.3g and 3.3h. At this stage, a 2-level CONA with two occupied cells at each level is replaced by a 3-level CONA with a single cell occupied. In Section 3.2.2 we show that this process has significant implications for the read and write throughput of the CONA.

3.1.4 Design Considerations

There are a number of other possible design choices which we considered. For example, one control knob that can be used to adjust write amplification vs. read performance is the *growth factor* of the CONA. The CONA described so far is a 2-CONA, where each successive level doubles in size. We can also realize other flavors, e.g., 3-CONA, or 4-CONA, where each level triples or quadruples in size, respectively, thus significantly changing properties of the data structure. For example, in a 4-CONA, 80% of the keys reside in the lowest level, which means that 80% of the keys do not require an in memory filter, significantly decreasing the memory footprint. A 4-CONA is shallower than a 2-CONA, which can improve read performance as there fewer levels that need to be search for a key. Memory reduction and better read performance does not come for free though, as a 4-CONA significantly increases write amplification. Each level can be merged into multiple times with each merge causing a rewrite of all the entries already stored in the CONA. This increased read/modify/write traffic can degrade the overall read throughput as well. For these reasons we opt to implement a 2-CONA. In Section 3.2.2 we show that the 2-CONA can outperform leveldb in part because leveldb [24] utilizes a higher growth factor.

3.2 Evaluation

In this section, we evaluate CONA using micro- and macro-benchmarks. We analyze the performance implications of various design knobs, such as buffer size, number of buffers, background merges, trie block size, etc. Next, we optimize these knobs for our test system and analyze CONA's performance characteristics using the YCSB benchmark suite [81]. Finally, we perform a detailed performance comparison between CONA and two state-of-the-art KV stores, SILT [13] and leveldb [24]. We show that CONA can outperform them by 2.5x in

insertion throughput, while simultaneously outperforming them in lookup throughput by 2.9x and 1.14x, respectively.

Experimental Setup We have implemented CONA in approximately 6K lines of C++ code. For the backing store we use a *mmap*ed file and defer all buffer cache management to the kernel. As described in Section 3.1, we maintain two in-memory buffers that accept all writes. Where appropriate, all buffers are flushed to disk, so that all insertion time that is reported includes flushing the data. We evaluate CONA on Linux using a Desktop equipped with: Intel 2.3 GHz Core i7-3610QM, 8 GB of DDR SDRAM, and a Crucial M500 240 GB SSD. The drive is connected using SATA and is formatted with the ext4 filesystem.

Table 3.1 summarizes the characteristics of the five YCSB workloads we use to test CONA. All workloads use a zipfian request distribution. By default, we use 20-byte keys and 200-byte values, which are typical for cloud and web object workloads simulated by YCSB [81]. All reported performance numbers are averages of at least 5 runs.

Workload A	**Update heavy workload**
Application example	Session store recording recent actions
Read/update ratio	50/50
Workload B	**Read mostly workload**
Application example	photo tagging; add a tag is an update, but most operations are to read tags
Read/update ratio	95/5
Workload C	**Read only**
Application example	user profile cache, where profiles are constructed elsewhere (e.g., Hadoop)
Read/update ratio	100/0
Workload D	**Read latest workload**
Application example	user status updates; people want to read the latest
Read/update/insert ratio	95/0/5
Workload F	**Read-modify-write workload**
Application example	user database, where user records are read and modified by the user or to record user activity
Read/read-modify-write ratio	50/50

Table 3.1 Yahoo Cloud Serving Benchmark.

3.2.1 Microbenchmarks

In this section, we study how CONA's read and write throughput are affected by the insertion buffer size, background merging, and trie block size. We show how these parameters can be optimized for our test system.

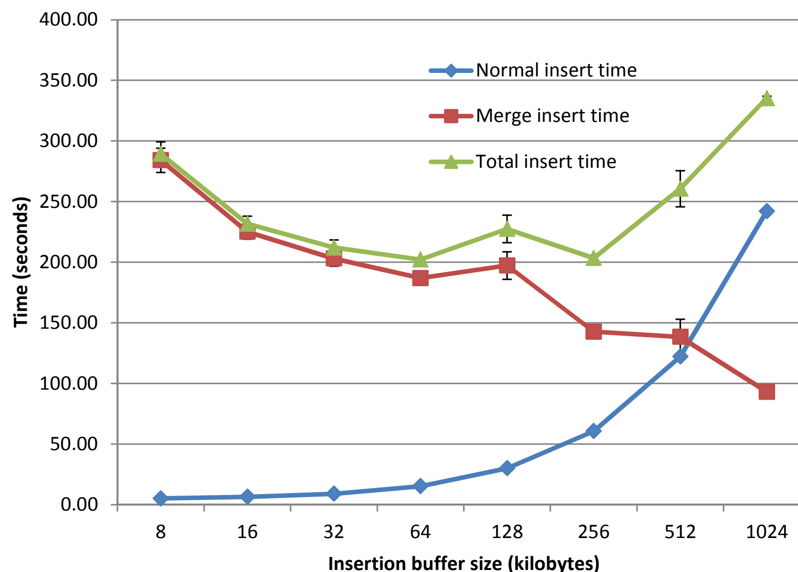


Figure 3.4 CONA insertion performance.

Insertion buffer size In the following we analyze CONA's insertion properties as affected by CONA's main performance knob, the size of the in-memory buffer. In the first experiment we insert 25 million key-value pairs into the CONA. Figure 3.4 plots insertion time as the size of the insertion buffer varies from 8 KB to 1 MB. In general, the total insertion time follows a U-shaped curve, with optimal performance achieved with a 64 KB insertion buffer.

As described in Section 3.1, each level of the CONA doubles in size. The first level is of size B, where B is the size of the insertion buffer. Hence, increasing B produces a shallow, "fat" CONA, while decreasing it yields a deep but "thin" one. For example, inserting 25 million key-value pairs into a CONA with 8 KB insertion buffer requires 18 levels, while a CONA with a 1 MB buffer needs only 10 levels.

Inserts into the CONA can be broken down into two types, "normal" and "merge" (Figure 3.4). A "normal" insert occurs when the initial insertion buffer has free space, which means that the CONA can copy the newly inserted key-value pair in the buffer and return to the client immediately. The insertion buffer along with all the levels of the CONA needs to remain sorted at all times, which means that every insert incurs a sort of the buffer. Hence, as the insertion buffer size increases, so does the normal insertion time. A "merge" insert occurs when the insertion buffer is full. In this case, the CONA needs to swap buffers and may need to wait for the merge task running in the background to free up a cell in the first level, so that the one of the buffers can be copied into the first level. The amount of work that the background merge task needs to complete is directly related to the number of levels in the CONA. As the buffer size increases the number of levels decreases along with the time to complete the merge. Hence, the total insertion performance follows the U-shaped curve seen in Figure fig:cona1. Using a 64 KB insertion buffer optimizes the CONA's insertion throughput on our test system.

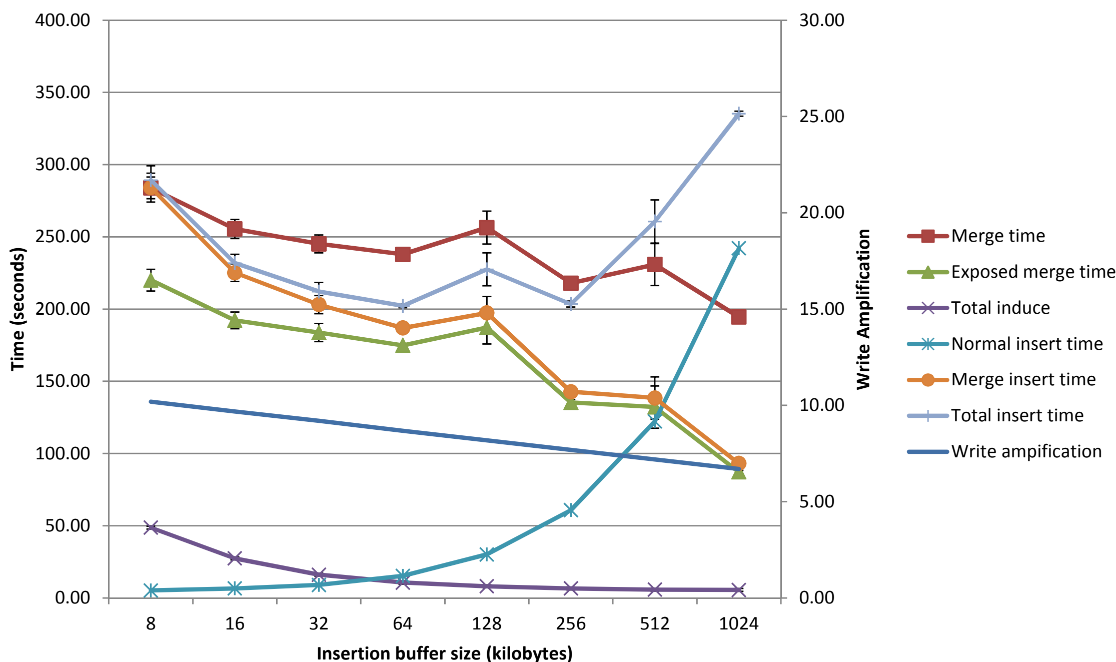


Figure 3.5 CONA insertion performance.

Figure 3.5 further breaks down the "merge" insertion time. As described in the design section, we introduce a second insertion buffer to absorb writes while merging in the background. Figure 3.5 shows that with one extra buffer about 20% of merging time (merge time vs. exposed merge time) can be hidden during construction. The number of buffers can be adjusted depending on the expected workload. For example, in later experiments we show that one additional buffer is sufficient to hide all merging cost in a predominantly READ workload. Once the insertion buffer fills up it is swapped and copied (induced) into the first level of the CONA. The backing store is a *memmap*ed file, so this involves a *memcpy* of the entire buffer. As the buffer size increases, inducing can be achieved in fewer larger operations. Thus, the performance of induce increases as the buffer size increases until it plateaus once it hits the maximum memory subsystem throughput on the test machine. The performance of the background merge operation is proportional to the number of writes required to complete the merge (the write amplification), which decrease with the number of levels in the CONA.

Next we examine insertion performance under several workloads form the YCSB suite. In all experiments a CONA is constructed with 25 million key-value pairs before the workload is executed.

Figure 3.6 shows the insertion performance under *Workload A* performing 25 million operations (50% read, 50% update). In general, performance characteristics are similar to that of the construction workload, so the best performance is once again achieved with a 64 KB insertion buffer. One key difference is that now the background merges can be performed not only while writing to the second buffer, but also during all reads. Hence, now only about 40%

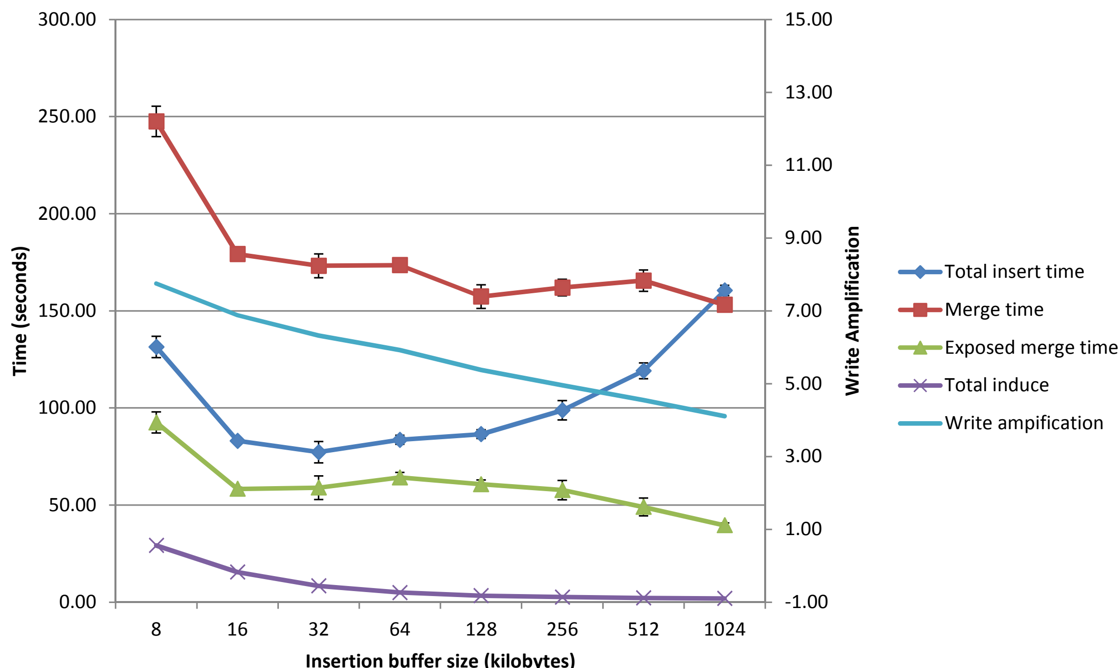


Figure 3.6 CONA Insertion performance under *Workload A* (50/50 read/write).

of the total merge cost is directly exposed to the client. As expected, the background merge performance under *Workload B* follows the write amplification curve (Figure 3.7). However, only about 5% of the merge time is exposed to the client, since all of the merging can be performed during the read operations. As a result, the U-shape of the total performance is asymmetrical in this case, with only the *induce* time contributing to the left side of the curve. Again, optimal performance is achieved with a 64 KB buffer.

Insertion buffer size and read throughput Figure 3.8 show the read performance of 25 million read operations into a CONA with 25 million entries. Unlike insertion, lookup performance remains largely unaffected by the in-memory buffer size and the resulting structure of the CONA. This is counterintuitive as the number of levels that need to be searched through sequentially decreases from 18, with a 8 KB buffer size, to just 10, when the buffer size is 1 MB. One would expect a proportional increase in throughput. Figure 3.9 reveals why this is not the case. As every level of the CONA doubles in size, about half of the entries reside in the last level and about 25% in the second to last level. Hence, a large percentage of all reads are serviced from the last two level irrespective of how many levels there are. The last two levels reside in flash, so the read cost is dominated by the flash reads rather then the sequential index lookups. As the reads are mostly served from only two levels in all cases, the benefits from caching the *memmap*ed backing store are also similar.

One benefit of increasing the buffer size is that popular entries can be cached in memory and as the buffer size increases this benefit increases. The workload follows a zipfian distribution, where popular entries are significantly more popular than average and hence the

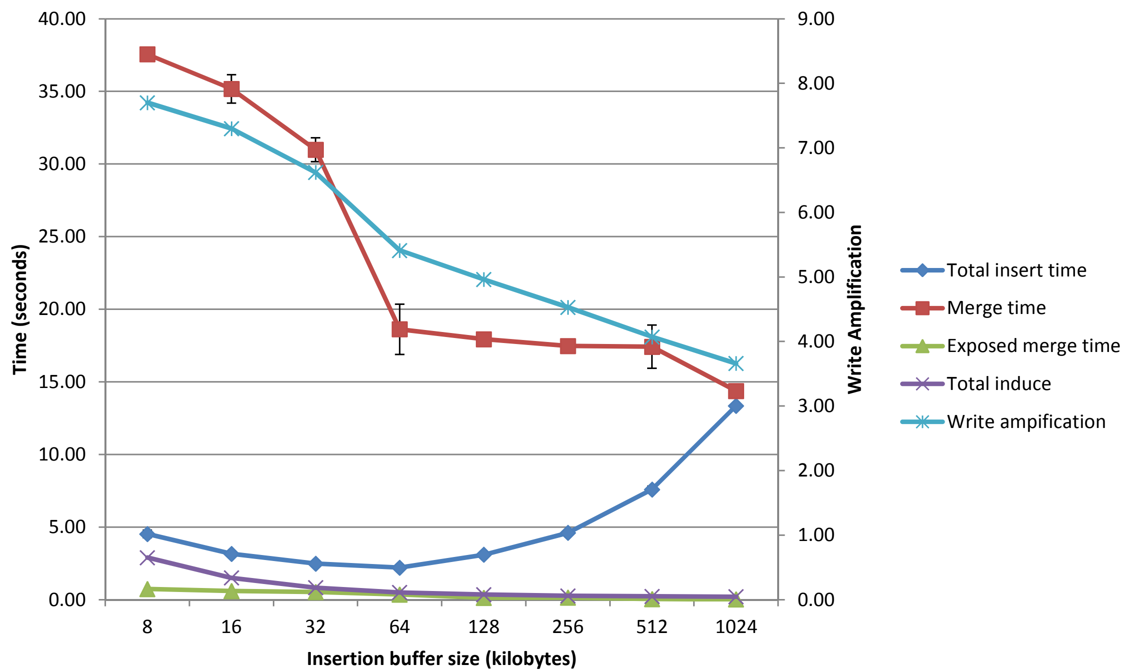


Figure 3.7 Insertion performance under *Workload B* (95/5 read/write).

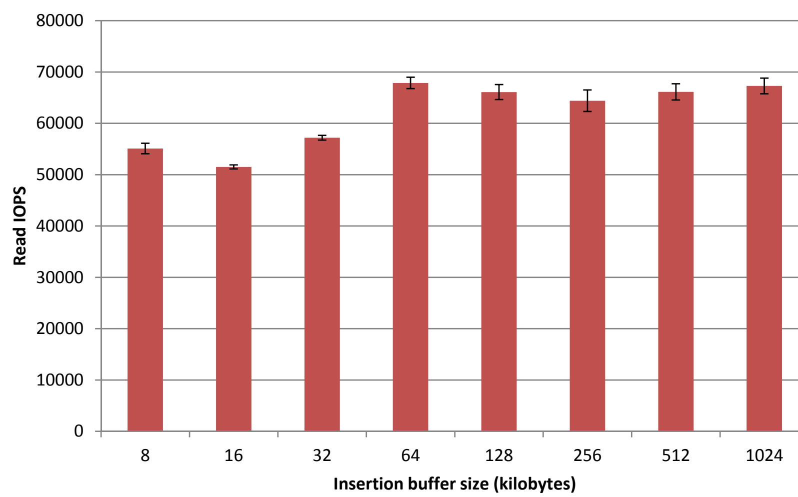


Figure 3.8 Lookup performance under *Workload C* (100/0 read/write).

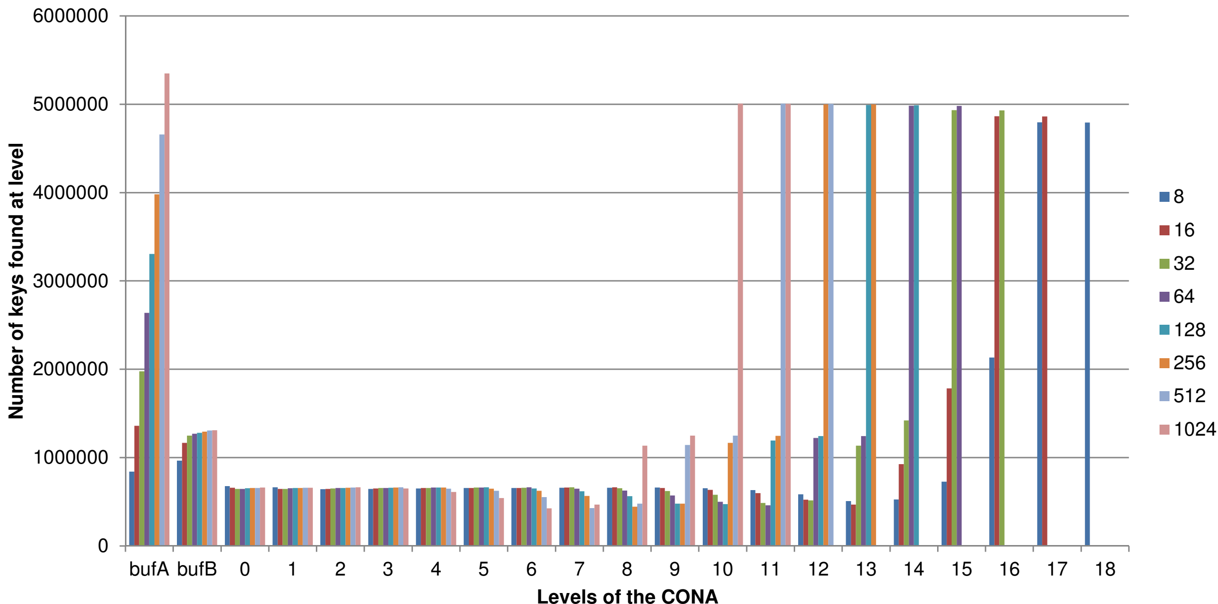


Figure 3.9 Read distribution among the levels of the CONA.

benefits from caching is amplified. Figure 3.9 shows that *Buffer A* serves significantly more reads when its size is 1 MB, which is why read IOPS increase from about 55,000 to well over 65,000, at 1 MB buffer size.

A 64 KB buffer CONA achieves a throughput within a few percent of optimal for reads as well as write. Hence, we chose this buffer size for all the rest of the experiments below.

Trie block size As shown in the previous section, lookups into the trie index at each level are not the main performance bottleneck. However, the lookup indexes are a major part of the memory footprint of the CONA. In this section, we detail how we optimize the tries to minimize memory consumption with a minimal drop in performance.

Trie block size	Average memory footprint per key (bytes)
4	~ 2
16	~ 0.74
64	~ 0.45
Optimal	~ 0.67

Table 3.2 Trie index average memory footprint per key.

The trie block size is the main performance tuning knob available in the trie. The block size affects how many bits can be used at each level of the trie to encode entries. Smaller block size results in more bytes per entry on average, but improves performance by producing

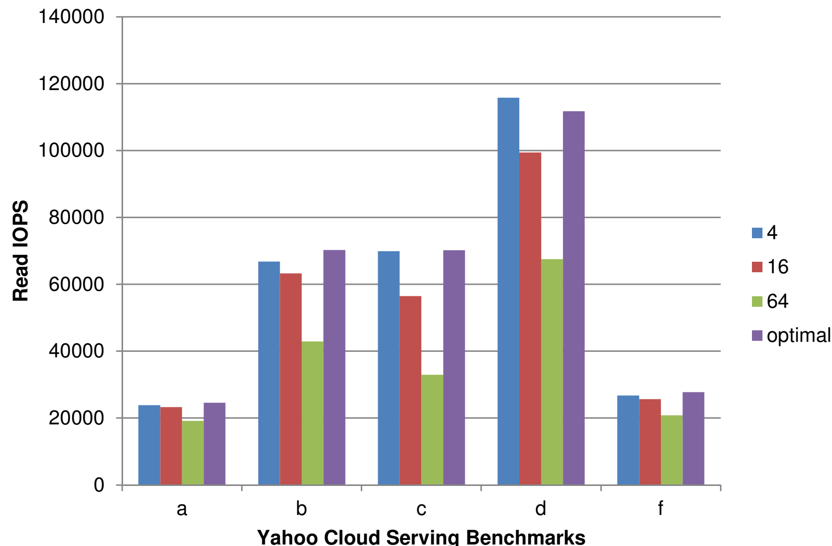


Figure 3.10 Lookup throughput of YCSB workloads under varying trie block sizes.

shallower tries. Table 3.2 shows the average memory footprint per key of a range of trie block sizes. Our memory footprint optimization is informed by the read distribution shown in Figure 3.9 and the discussion in Section 3.1. The vast majority (over 94%) of entries reside in the last four levels of the CONA and around 50% of reads are served from the last level alone. Hence, we divide the levels into three categories and optimize the tries in each separately. In the first $N-4$ levels where N is the number of levels of the CONA we optimize the trie for performance by choosing a block size of 4. In level $N-3$ to $N-1$ we use a block size of 16, while at the last level where most of the entires reside, we optimize for memory usage and use a block size of 64. Table 3.3 shows the resulting overall memory footprint of 0.678 bytes per key on average.

	Percentage of keys (%)	Bytes per key	Overall bytes
Level 0 to N-4	6.3	2.00	0.125
Level N-3	6.3	0.75	0.046
Level N-2	12.5	0.75	0.093
Level N-1	25	0.75	0.187
Level N	50	0.45	0.225
Sum	100	-	0.678

Table 3.3 Average memory footprint per key for optimal trie index configuration.

Figure 3.10 compares the read throughput of our trie optimization against using a constant block size at all levels. As expected, using a block size of 4 produces the best read performance under all the YCSB benchmarks. Block size of 16 reduces the throughput by about 5% on average, while 64 can reduce performance as much as 50% on some workloads. Optimal

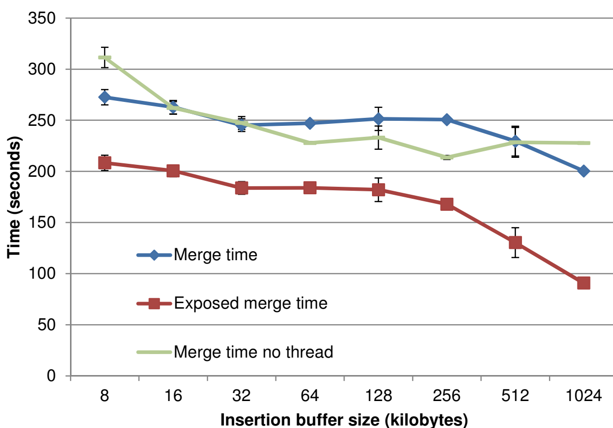


Figure 3.11 Threaded merging performance.

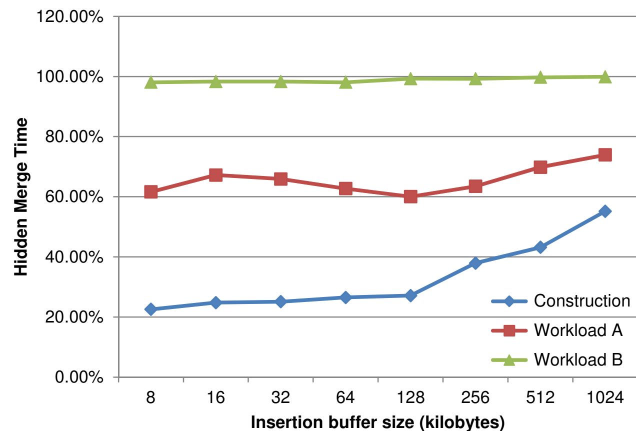


Figure 3.12 Percentage merge time hidden by threading.

trie block size is especially important in workloads where in-memory caching plays a significant role, such as workloads C and D. There are no background merges performed during the read-only *Workload C*, so the structure of the CONA remains unchanged and hence can be better cached. In this case flash reads do not dominate the read cost as much and having an better performing trie index lookup becomes more important. Similarly, *Workload D* is a read-latest workload, where most reads are served from the in-memory buffers and first few levels of the CONA, so trie lookup performance is again important. Our trie configuration optimizes for all these scenarios and its resulting throughput is within a percent of optimal, with a 66% decrease in memory consumption.

Background Merges In this section we examine how introducing of an extra in-memory buffer and performing merges in the background can improve insertion throughput. Figure 3.11 shows that using an additional memory buffer that absorbs writes while a background thread merges entries can improve construction performance by around 20%. This benefit increases to around 60% for a 50/50 read write workload and almost 100% for a 95/5 read write one (Figure 3.12). As CONA's target workload as a flash-backed cache is a read-mostly workload, a single additional buffer is sufficient. Moreover, using extra buffers would increase the memory footprint unnecessarily. Using a large buffer is also not an option as it needs to remain sorted after every insert, which will have a negative effect on performance, as we have shown in Figure 3.4.

3.2.2 Macrobenchmarks

In this section, we present a detailed analysis full-system performance of the optimized, threaded CONA under the YCSB suite of benchmarks. We investigate the read and write

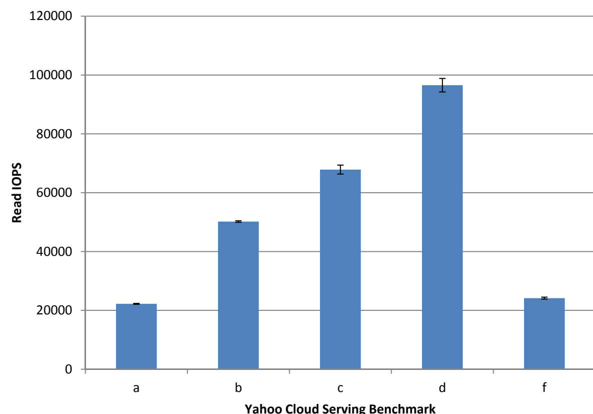


Figure 3.13 CONA read performance under YCSB workloads.

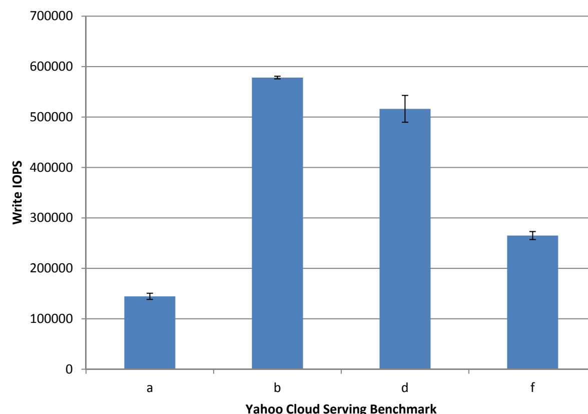


Figure 3.14 CONA write performance under YCSB workloads.

throughput of CONA along with its background merging performance throughout an execution run. We perform a similar optimization and analysis of two state-of-the-art key-value stores and conclude the section with a head-to-head performance comparison under the same workloads.

CONA full-system performance Figure 3.13 and Figure 3.14 show the read and write throughput of CONA on our test system under the five YCSB benchmarks. In all cases, a CONA is constructed using 25 million key-value pairs, followed by a benchmark workload that performs 25 million operations, whose read/write/update distribution is described in Table 3.1. All reported numbers are averages of five runs.

As discussed earlier, CONA write performance is maximized in a read mostly workload, where the number of writes is significantly smaller compared to the number reads, as well as the number of entries already present in the CONA. Such workload like *Workload B* and *Workload D* achieve over 500K write IOPS on our test system. Workloads like *A* and *B* where the number of writes is on the order of entries already present in the CONA (50% and 33%, respectively) achieve 150K and 250K write IOPS, respectively.

CONA's read performance has similar characteristics to the write. It performs best under read only/mostly workloads like *B, C, and D* and worse under more even read/write ratio. *Workload D* is by far the best performing at just under 100K read IOPS. It is a "read latest" workload, which suits the CONA because it naturally stores the latest entries at the top. The seconds fastest workload is the read-only one, which benefits from caching of the unchanged CONA and as a result achieves over 65K read IOPS. *Workload B* also benefits from caching albeit not as much as *C and D*, which produces 50K read IOPS. Workloads *A* and *F* come in last at just over 20K read IOPS.

In the following sections we investigate the reasons behind the performance under each YCSB workload. We divide them into two categories: write heavy (Workloads *A* and *F*) and read heavy (Workloads *B*, *C* and *D*).

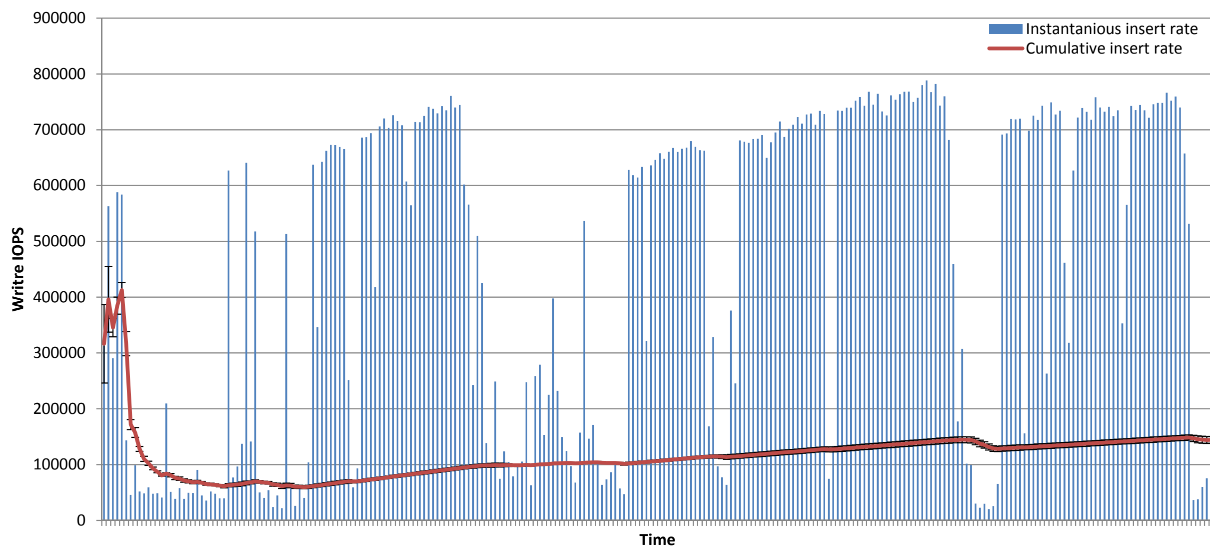


Figure 3.15 *Workload A* write performance.

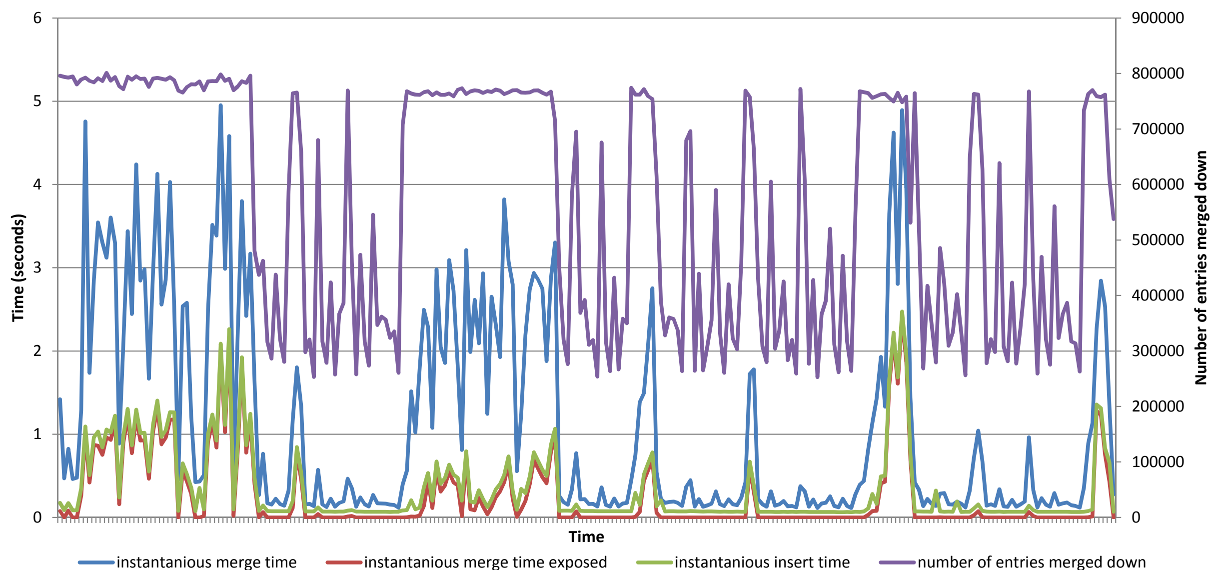


Figure 3.16 *Workload A* merge performance.

Write heavy workloads Figures 3.15 through 3.17 show a time-series of the performance of Workload *A* while performing 25 million operations. The granularity of the sampling in all cases is at 100K operations.

Figure 3.15 reveals that the (instantaneous) insertion performance of CONA can vary significantly under this workload as compared with the (cumulative) average. Performance varies between periods where both cumulative and instantaneous are around 100K IOPS, followed by ones where instantaneous jumps up to 700K IOPS. The reason for this behavior is two-fold. First, is the state of the CONA when the workload begins, is such that the

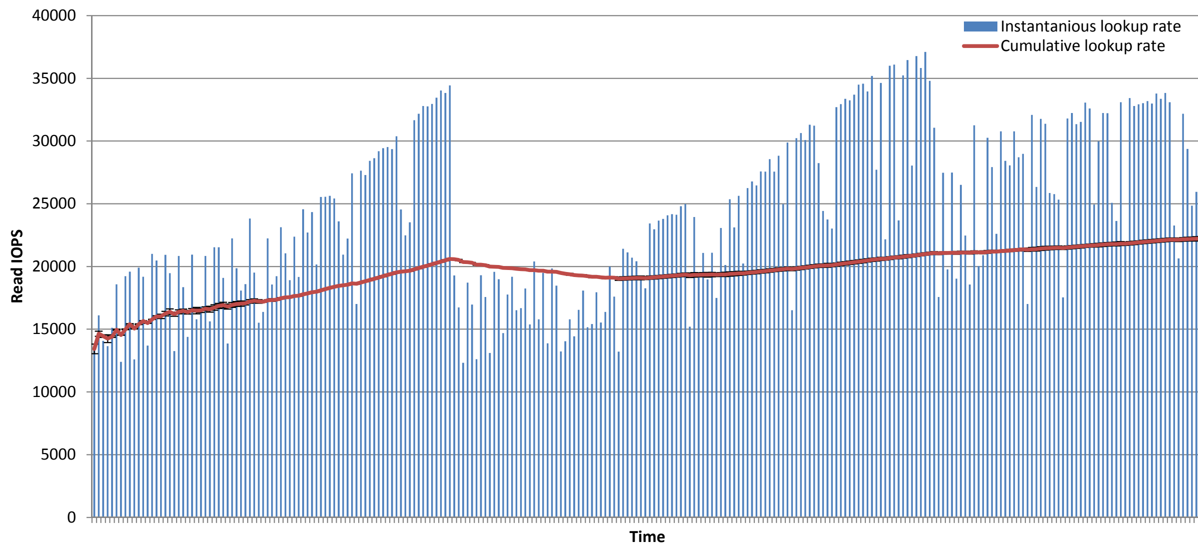


Figure 3.17 *Workload A* read performance.

second-to-last level of the CONA contains two full cell. In order to free up space for new entries, these two cell are being continuously merged in the background into a new cell at the lowest level. This is the reason for the decreased write performance observed during the first quarter of execution of the benchmark. This prolonged merging occurs only when a workload increases the total number of entries by 50%. This means that every time the insertion buffer is filled up, the CONA has to wait for the background merge to complete before returning to the client, resulting in lower write performance. The second and more important reason for the lower insertion performance is the write-heavy nature of the workload. 50% of operations are writes, which means that write buffering and read operations do not provide sufficient time for the background merge to finish. The structure of the CONA does not allow for the number of merge operations to be equally distributed throughout the entire execution run, because a merge can only occur when a level has two full cells. The levels of the CONA alternate between having two and one cell full, which results in this fluctuation of performance under a write heavy workload.

This is best seen in Figure 3.16, which shows the number of merge operations, as well as the total and exposed merge time and its effect on the insertion performance. At the beginning of the execution the number of merge operations hover around the maximum $k+1$ per insert. There are 50,000 writes during each sampling period and the number of levels, k, is 15, so number of merges is around 750,000. This continuous merge workload causes some of the merge time to become exposed, which increases the insertion time and the consequent decrease in performance seen in Figure 3.15. Once the cells merge into the last level, the merge workload drops to less than half of the maximum, which is enough for the merge operations to remain completely hidden. This pattern continues as cells of different sizes need to be merged, causing periods of maximum merge operations, which has implications for the write as well as the read performance.

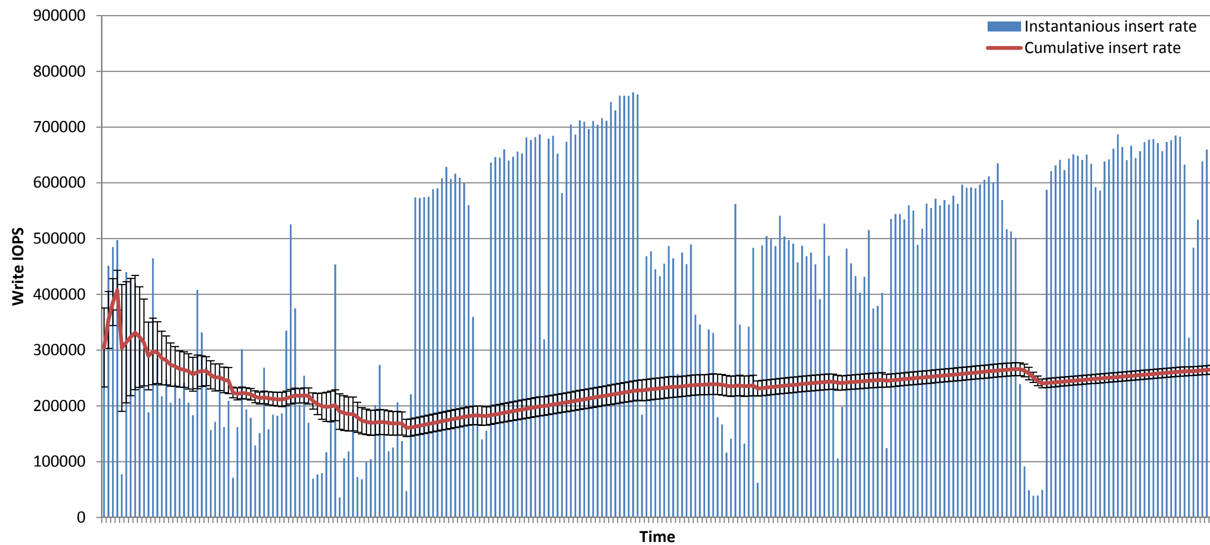


Figure 3.18 *Workload F* write performance.

For the read, the performance impact of the write heavy workload comes in the form of loss of locality (Figure 3.17). The two cells at the second to last level merge into the a new "shadow" cell at the lowest level of the CONA. However, reads are still serviced from the two cells for some time. Eventually, enough entries are inserted into the CONA causing the "shadow" cell at the last level to become "visible" replacing the two smaller cells. This results in temporary loss of read performance as pages of the new cell are reintroduced into the page cache. Figure 3.17 shows two epochs in which read performance slowly increases followed by a sharp drop. The first drop from about 35K IOPS to under 15K corresponds the "shadow" cell at the new lowest level becoming "visible". The second 35K peak about 3/4 into the execution is followed by a smaller drop in performance caused by a "shadow" cell in the second-to-last level becoming "visible". Apart from the two major trends, the execution run is dotted with several smaller drops in performance, caused by interior cells becoming "visible". As these cells are much smaller in size, performance rebounds almost instantly in these cases.

The performance of the other write heavy workload, *Workload F*, is very similar to that of *Workload A*. Only about 1/3 of operations are writes, as compared to 50% for *Workload A*. As a result, a little over 4 million fewer entries are inserted during the workload, shifting the time series to the right. This is best seen in Figure 3.20, which has only the first read performance drop that now occurs around the midpoint of the workload, as the 4 million fewer inserts are not enough to cause the second-to-last "shadow" cell to become "visible". Figures 3.18 and 3.19 are similarly shifted to the right as compared to Figures 3.15 and 3.16.

One way to improve performance is to increase the number of insertion buffers significantly. This will improve performance, but at the expense of a corresponding significant increase in memory footprint. However, we show that even the two-buffer configuration CONA can outperform state-of-the-art stores.

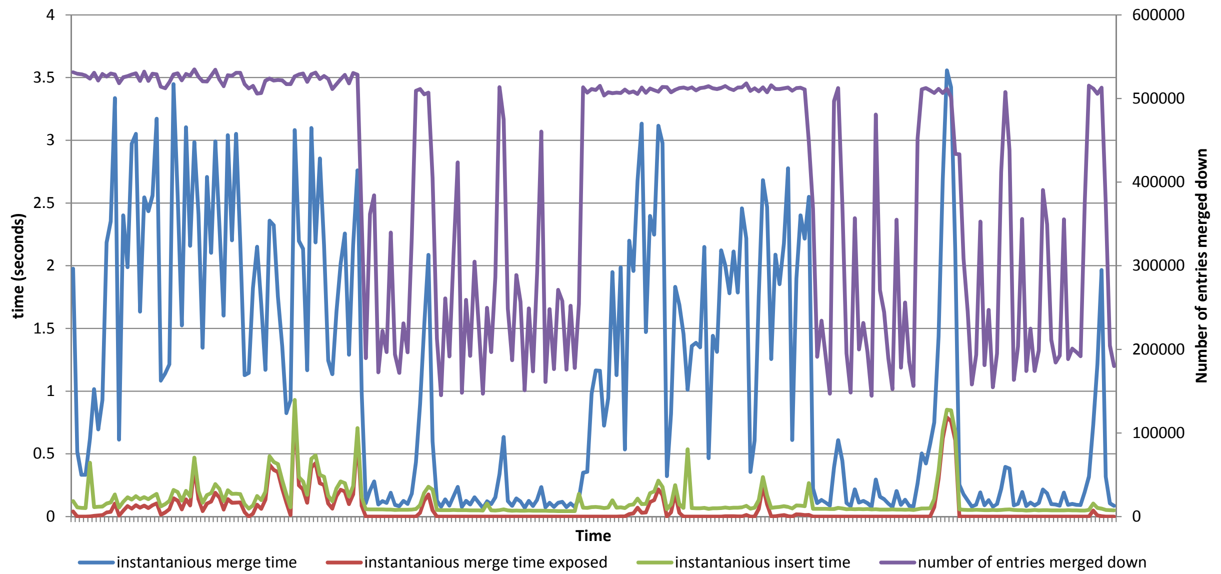


Figure 3.19 *Workload F* merge performance.

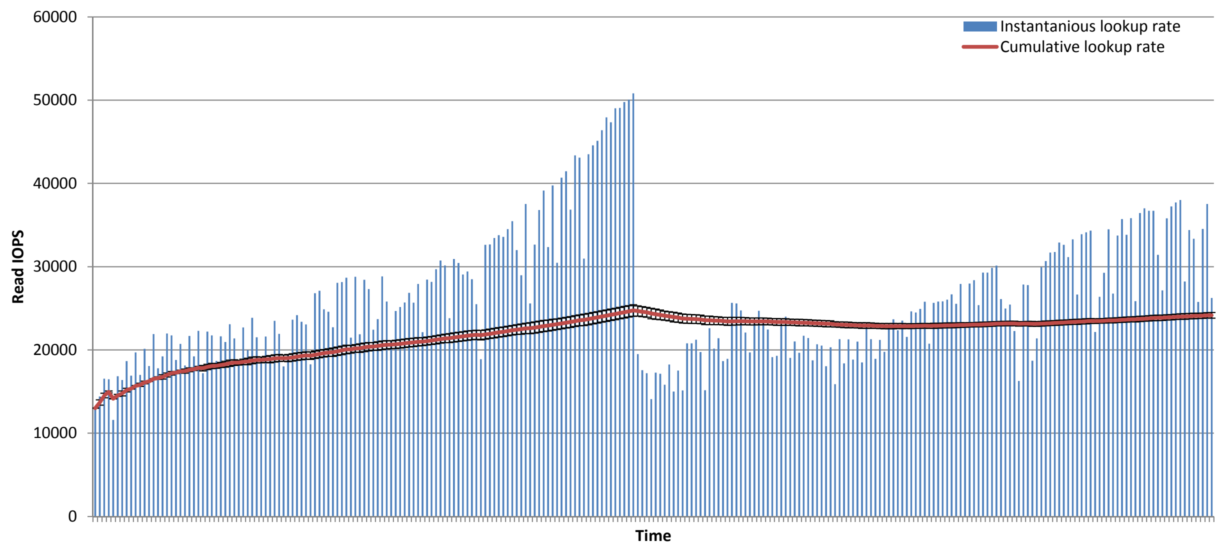


Figure 3.20 *Workload F* read performance.

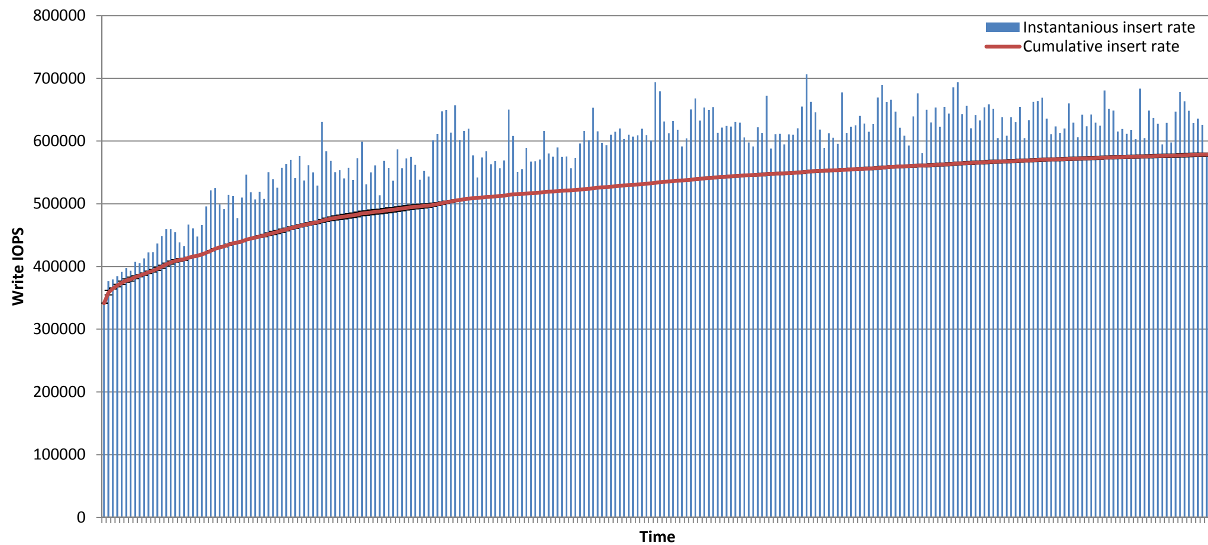


Figure 3.21 *Workload B* write performance.

Read-heavy workloads The write performance of the read-heavy workloads is much more stable than that of the *Workload A* and *F*. The reason for that is the 95/5 read/write distributions of the operations, which allows for plenty of time for the background merges to complete before the insertion buffer fills up. As a result, the insertion performance seen in Figures 3.21 and 3.23 is much more stable that that of the write heavy workload. It is also significantly higher, at close to 4x and 2x better than that of *A* and *F*, respectively. Note that this performance increase is not due to the fact that these inserts occur during a time when no background merges occur. On the contrary, Figures 3.22 and 3.24 show that the number of merges are at the maximum during the entire benchmark runs. However, in this case no merge time is exposed to the client.

The best read performance is achieved by *Workload D* at just under 100K IOPS. This read-latest workload is best suited for the CONA, which naturally stores the latest entries at the top (Figure 3.25). Read performance is also stable throughout the entire execution. The small number of writes produce only a few cell merges at the higher levels of the CONA, which result in short lived performance degradations.

The read-only *Workload C* produces the second-best read performance at around 70K IOPS. In contrast to *Workload D*, in this case read performance changes significantly during the course of the benchmark. As the workload is read-only, the structure of the CONA remains constant throughout resulting in no loss of locality. Due to the zipfian distribution of the workload, the popular entires get quickly cached, which significantly increase the read performance. Toward the second half of the run performance levels off, as the page cache cannot accommodate the entire read set.

The read performance characteristic of *Workload B* (Figure 3.27) are similar to that of *C*. However, read performance in this case levels off much sooner and at a lower level, because the 1.25 million entries inserted by *Workload B* continuously change the structure of the

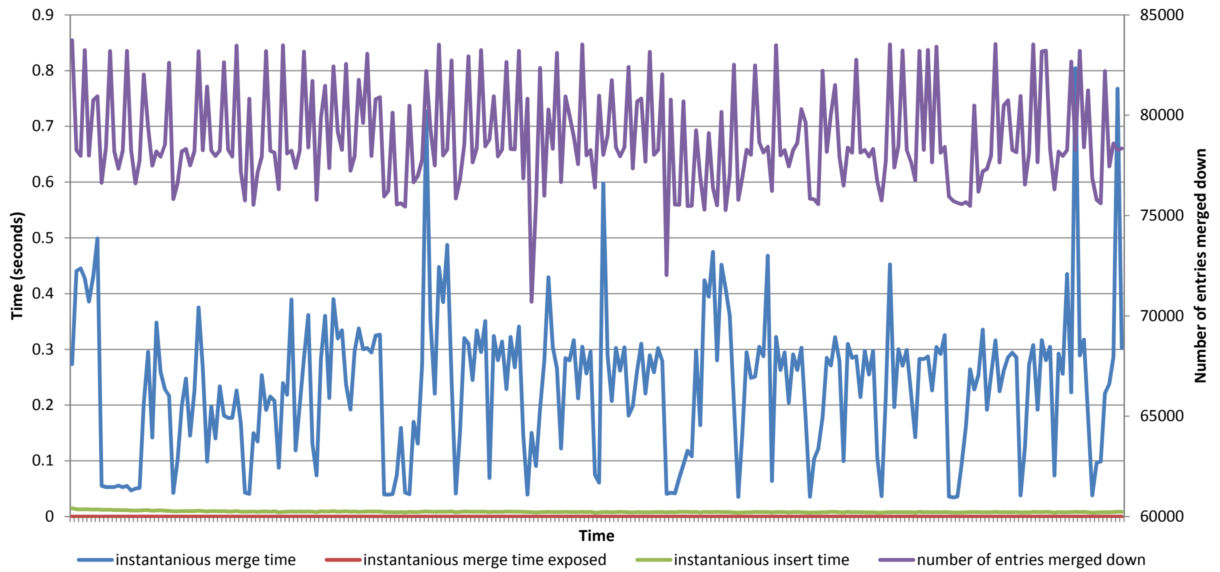


Figure 3.22 *Workload B* merge performance.

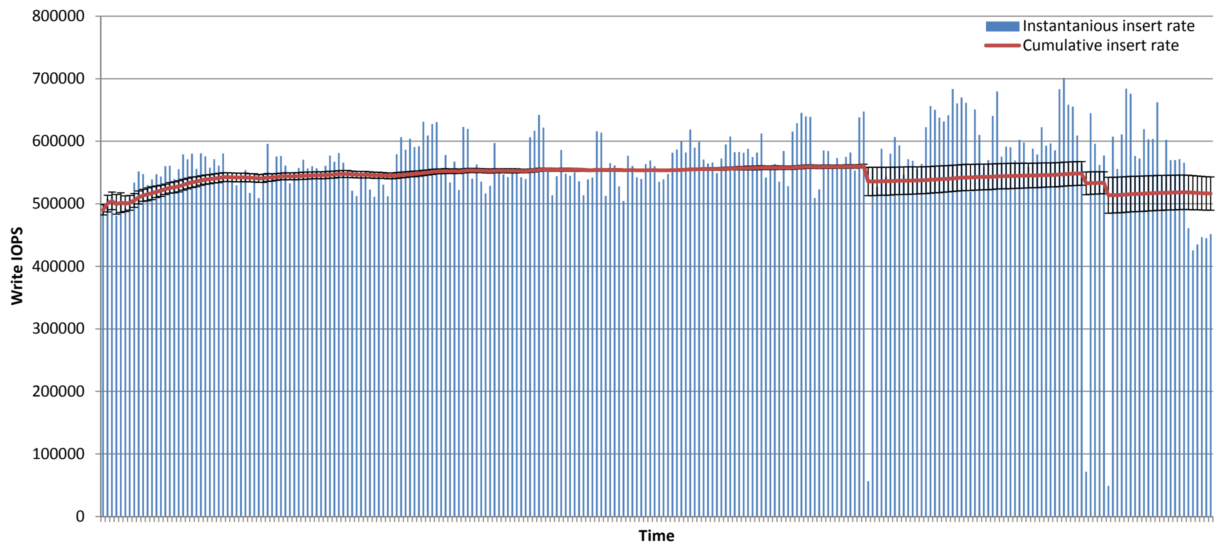


Figure 3.23 *Workload D* write performance.

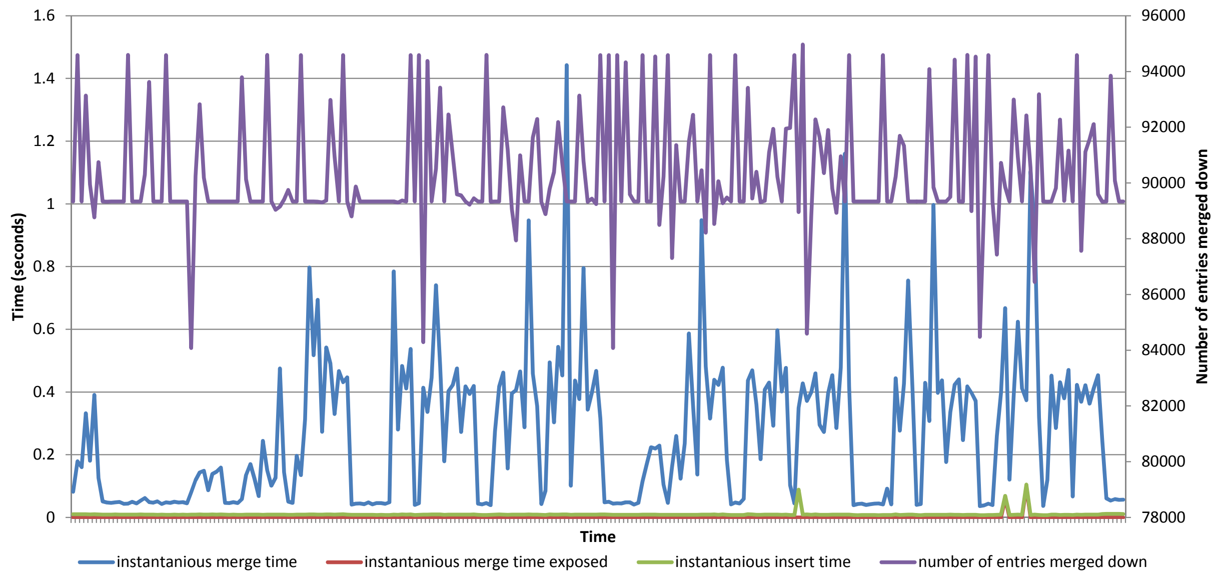


Figure 3.24 *Workload A* merge performance.

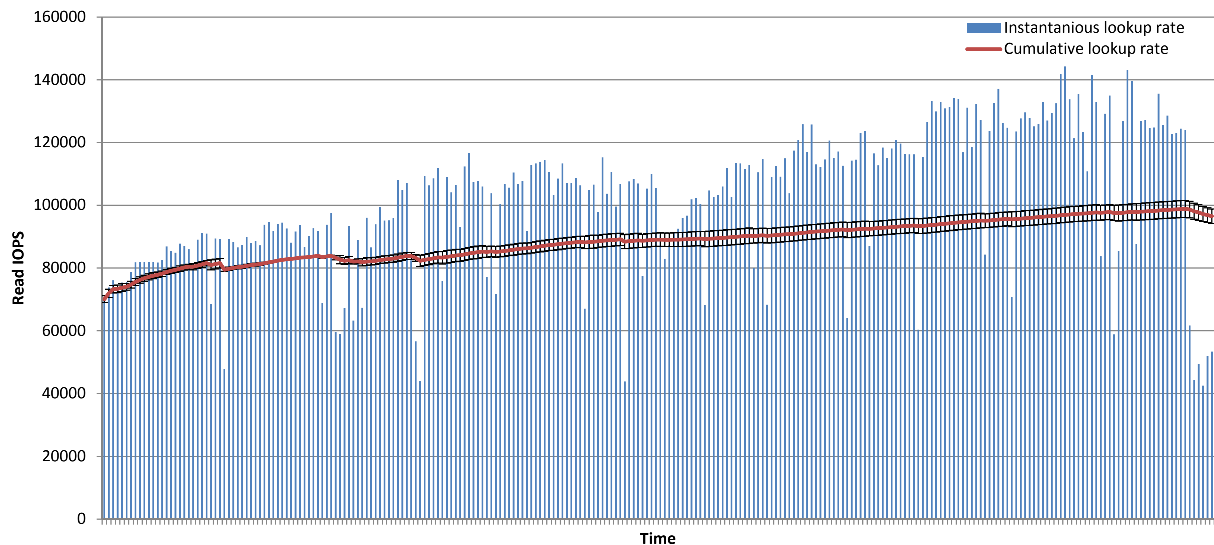


Figure 3.25 *Workload D* read performance.

CONA. As a result, entries cannot be cached as well, producing just over 50K IOPS.

Performance Comparison with SILT and leveldb

In this section, we compare CONA's performance to that of two state-of-the-art KV stores, SILT and leveldb. Both of these have a tiered storage structure similar to that of CONA. Open source implementation are available for both stores, which allows us to perform a head-to-head comparison on our test system.

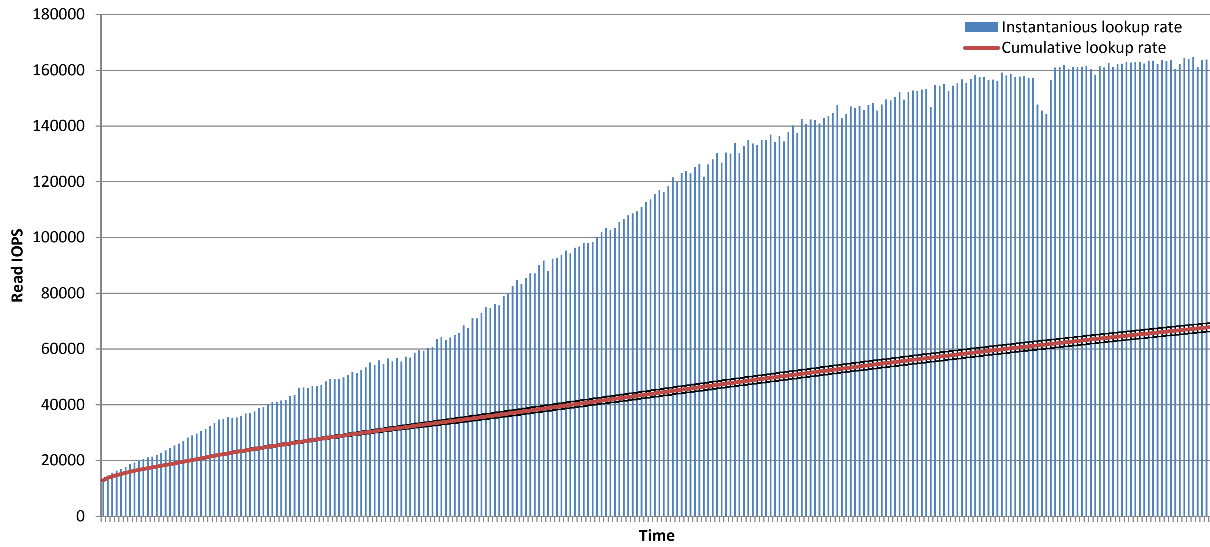


Figure 3.26 *Workload C* read performance.

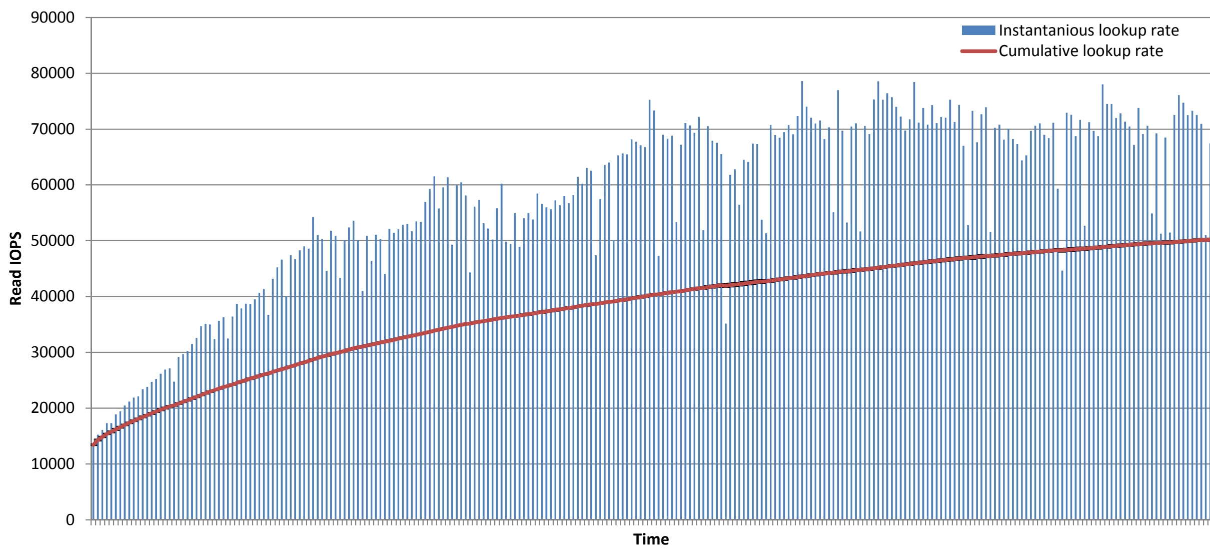


Figure 3.27 *Workload B* read performance.

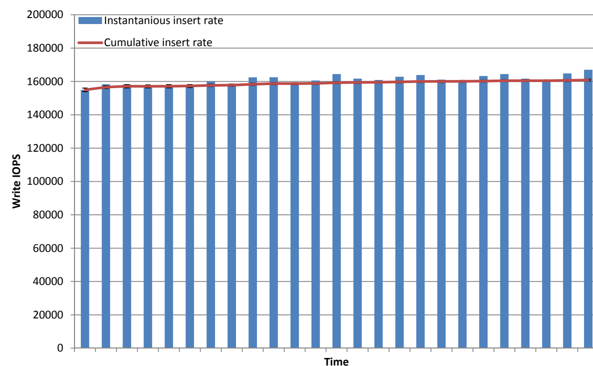


Figure 3.28 SILT write performance under *Workload A*.

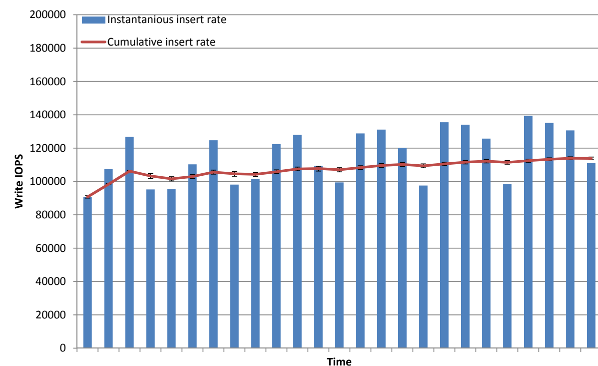


Figure 3.29 SILT write performance under *Workload B*.

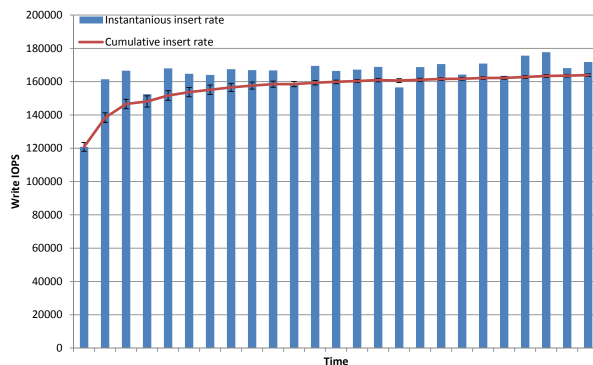


Figure 3.30 SILT write performance under *Workload D*.

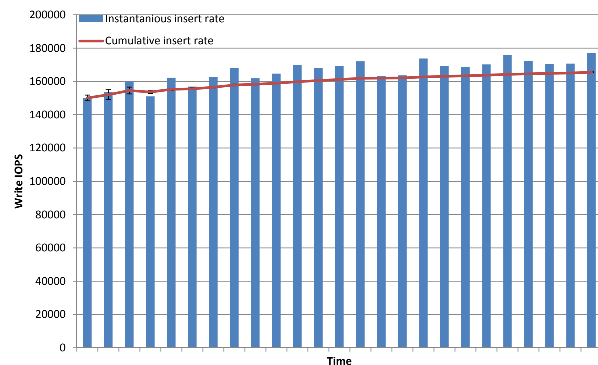


Figure 3.31 SILT write performance under *Workload F*.

Performance of the SILT KV store Figures 3.28 through 3.31 show SILT's write performance under the five YCSB benchmarks described in the previous section, while figures 3.32 through 3.36 show the corresponding read performance. We have configured SILT with a index block size of 64 to provide a comparable memory efficiency to performance characteristics to that of CONA.

Unlike CONA, SILT exhibits a more stable performance during the execution run. The reason for that is that SILT is constantly performing a background sort in order to convert its hash stores into a more memory efficient sorted store. This expensive operation competes with resources to perform the read operations. This is especially evident under *Workload B* where read and write performance fluctuates, as during one time-slice read performance increase while the write decreases and vice-versa. In contrast, CONA does not have to perform an expensive sort, resulting in a much higher performance compared to SILT.

Performance of the leveldb KV store Figures 3.37 through 3.45 show the write and read performance of leveldb. Performance is similarly stable throughout the execution runs, as, like SILT, leveldb also relies on an asynchronous background merging mechanism. This

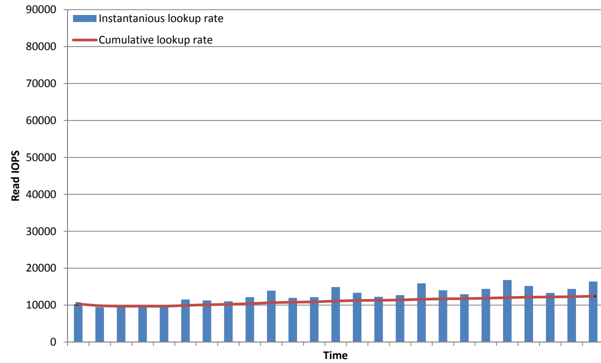


Figure 3.32 SILT read performance under *Workload A*.

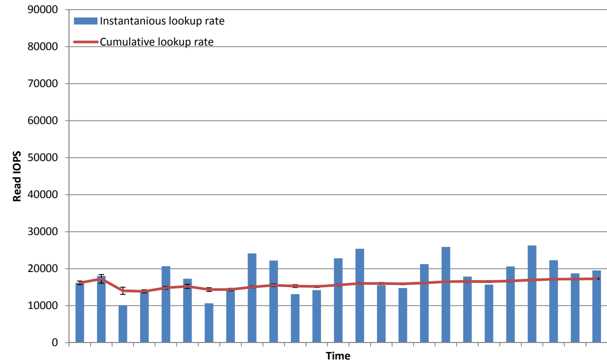


Figure 3.33 SILT read performance under *Workload B*.

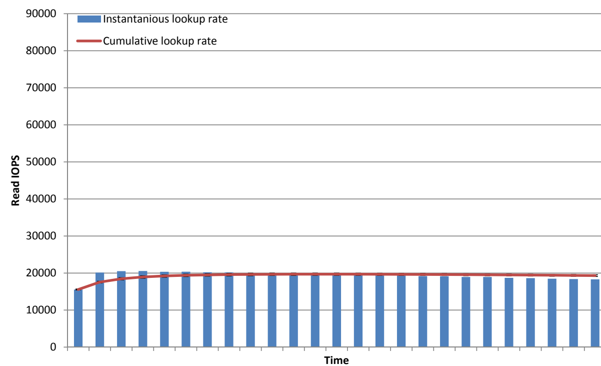


Figure 3.34 SILT read performance under *Workload C*.

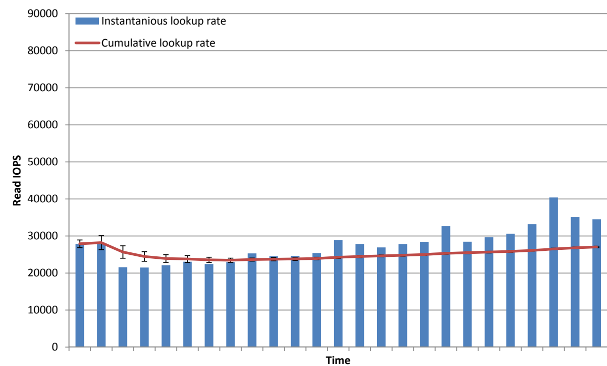


Figure 3.35 SILT read performance under *Workload D*.

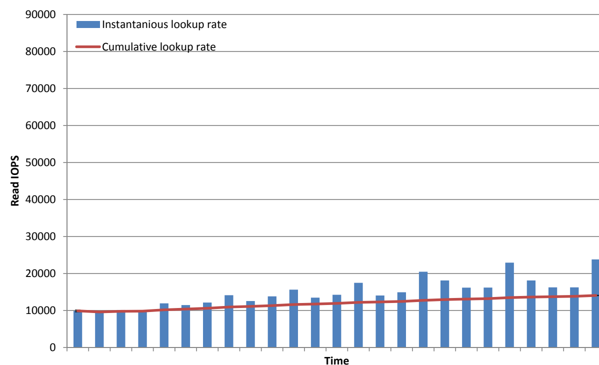


Figure 3.36 SILT read performance under *Workload F*.

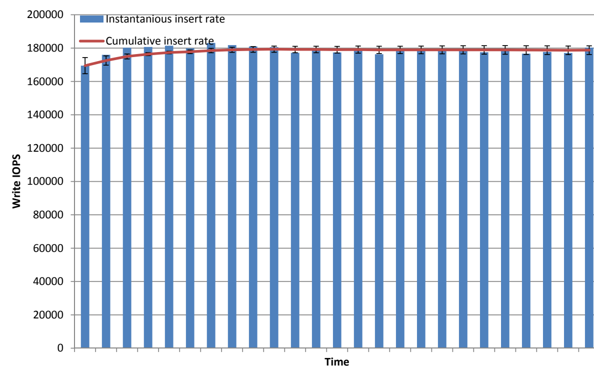


Figure 3.37 leveldb write performance under *Workload A*.

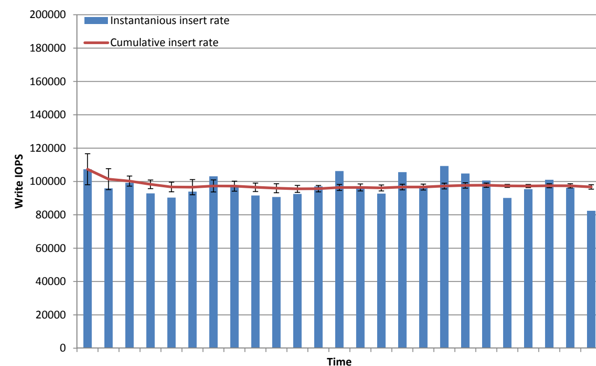


Figure 3.38 lleveldb write performance under *Workload B*.

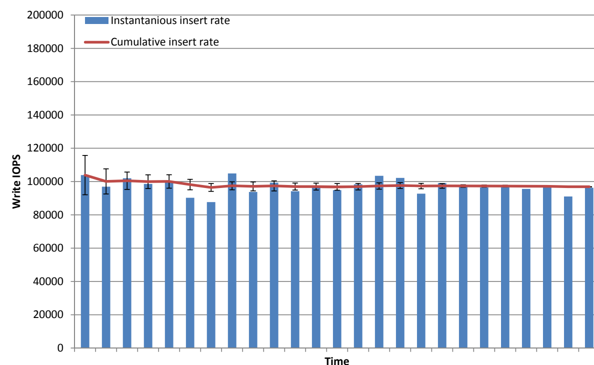


Figure 3.39 leveldb write performance under *Workload D*.

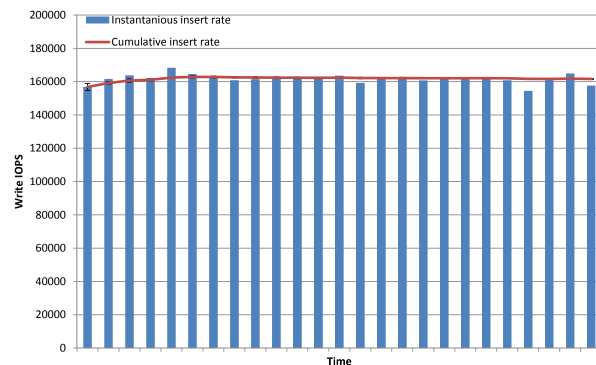


Figure 3.40 leveldb write performance under *Workload F*.

ensures stable performance, but as levels have to be constantly re-merged into, this background workload competes with the client read operations. Leveldb has to perform more merges because of its growth factor of ten. Thus, *B* and *D* write performance suffers at the expense of better read throughput. The read-only *Workload C* benefits from caching, resulting in a sharp increase in read throughput which quickly levels off.

Overall Performance Comparison. Figure 3.46 and 3.47 show the overall read and write throughput of CONA as compared to that of SILT and leveldb. As another point of comparison, we also show the performance of the single thread implementation of the CONA, which performs merges inline with writes. Note that both SILT and leveldb have a dedicated thread to perform sorting in the background.

On average, CONA outperforms SILT and leveldb by 2.5x and 2.8x in insertion throughput, respectively, while simultaneously outperforming them in lookup throughput by 2.9x and 1.14x. The key-value stores are more evenly matched under the write-heavy workloads *A* and *F*, where CONA outperforms SILT and leveldb in write throughput by 1.4x and 1.3x on average. CONA is best suited for the read-heavy workloads *B* and *D*, delivering as much a

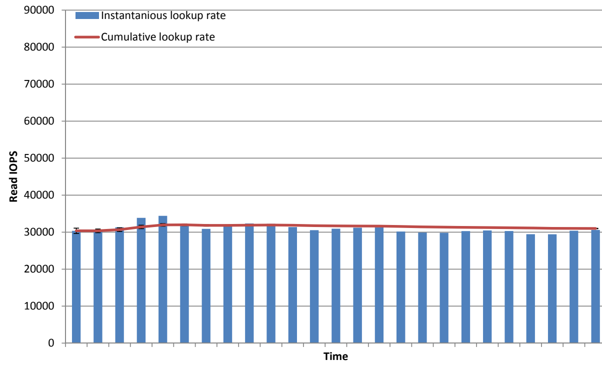


Figure 3.41 leveldb read performance under *Workload A*.

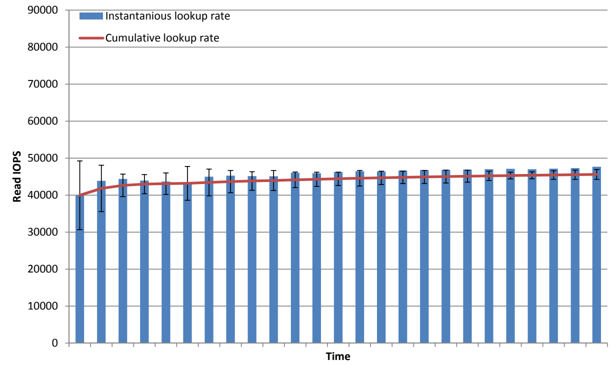


Figure 3.42 leveldb read performance under *Workload B*.

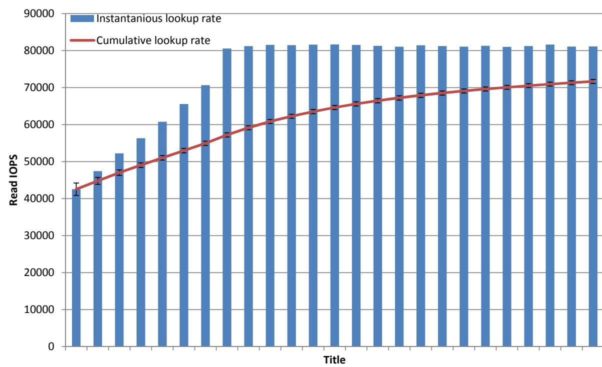


Figure 3.43 leveldb read performance under *Workload C*.

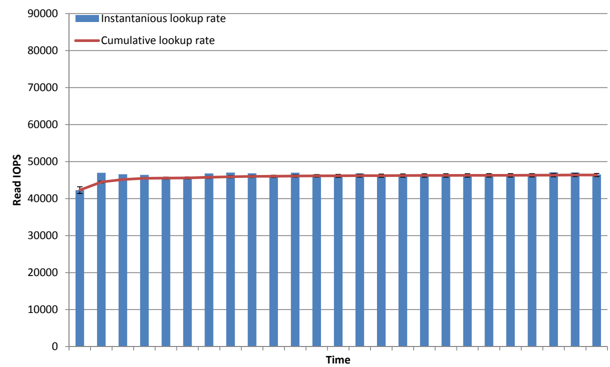


Figure 3.44 leveldb read performance under *Workload D*.

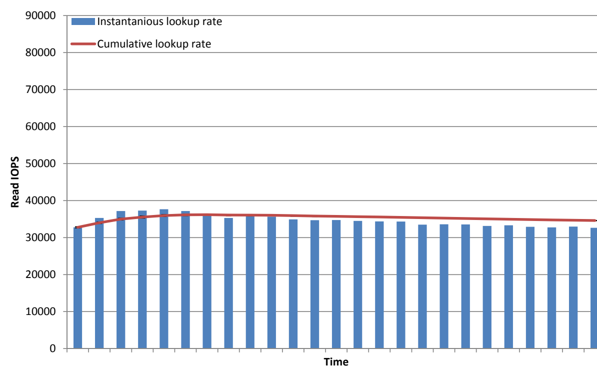


Figure 3.45 leveldb read performance under *Workload F*.

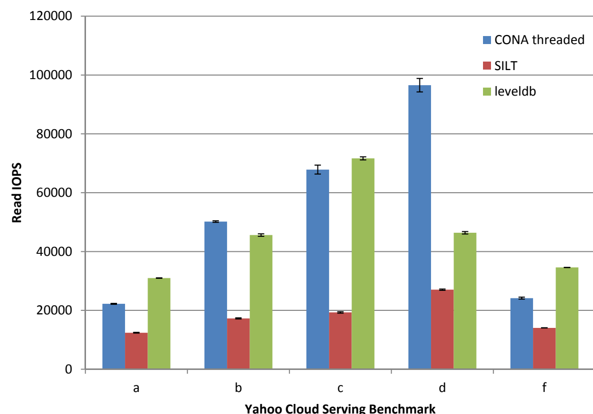


Figure 3.46 KV store read performance comparison.

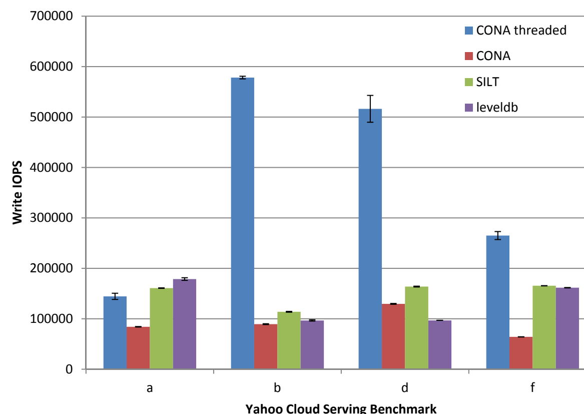


Figure 3.47 KV store write performance comparison.

5.6x better insertion performance over leveldb.

CONA and leveldb are similar in read throughput, with CONA outperforming leveldb by 14% on average. At the write heavy benchmarks, leveldb actually comes ahead by about 39% as CONA has to perform several major cell merges. SILT is a distant third in read throughput, at just over a third the throughput of CONA.

3.3 Chapter Summary

In this Chapter, we presented a KV store, CONA, which is designed as a read-mostly flash-backed cache. The Cache Oblivious No-lookahead Array that powers the CONA is a data structure tailored around the specific requirements of flash. It arranges data on flash in such a way as to minimize write amplification, which is detrimental to the flash cells. The CONA also delivers outstanding read amplification through the use of a trie index and false positive filter. We analyze the performance characteristics of CONA using the YCSB benchmark suite and show how it can be optimized for our test system. Our experiment show that compared to two state-of-the-art key-value stores, CONA can deliver 2.5x more insertion throughput, while simultaneously outperforming them in lookup throughput by 2.9x and 1.14x on average.

Chapter 4

GPU-Accelerated Systems

In this chapter we describe the research done in accelerating a distributed RAID system using a GPU. Another GPU-accelerated system we have explored involves offloading address translation in a Flash Translation Layer to a GPU. We detail a proof-of-concept implementation and discuss future directions of that work in Section 5.1.

4.1 GPU-Accelerated Cost-Effective Distributed RAID

In this section we detail our work in using a GPU to accelerate a storage system. We design an innovative solution to achieve a flexible, fault-tolerant, and high-performance RAID-6 solution for a parallel file system (PFS) [82]. Our system utilizes low-cost, strategically placed GPUs — both on the client and server sides — to accelerate parity computation. In contrast to hardware-based approaches, we provide full control over the size, length and location of a RAID array on a per file basis, end-to-end data integrity checking, and parallelization of RAID array reconstruction. We have deployed our system in conjunction with the widely-used Lustre PFS, and show that client-side parity generation using a GPU improves reliability while imposing minimal overheads on the overall client performance. The system achieves a coding throughput of over 3.0 GB/s on each client.

The framework capitalize on the resources provided by the PFS, such as striping individual files over multiple disks, in conjunction with the computational power of a GPU to provide flexible and fast parity computation for encoding and rebuilding of degraded RAID arrays. It attains end-to-end data integrity by performing encoding and decoding at the compute node, where data is produced and consumed. We implement our client-driven, per-file RAID in the widely used Lustre PFS [56], which will facilitate wider adoption of our system. We evaluate our system using a medium-scale cluster based on nodes with off-the-shelf, GPUs and show that our approach: provides a customizable interface for an application to tailor the RAID array parameters and provides default values to support legacy applications. The results demonstrate that leveraging GPUs for I/O support functions, i.e., RAID parity computation, is a feasible approach and can provide an efficient alternative to specialized-hardware-based solutions.

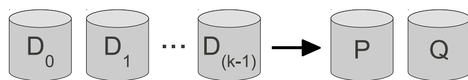


Figure 4.1 Logical overview of a RAID-6 system.

4.1.1 Enabling Technologies

4.1.1.1 Erasure Codes

In the following, we describe the enabling technologies that are used in realizing our GPU accelerated software-based RAID-6 distributed PFS.

4.1.1.2 Erasure Codes

In recent years, RAID-6 systems have become increasingly important as they can tolerate a complete failure of one drive occurring in combination with a latent failure of a block on a second drive. Such a failure scenario would result in a permanent data loss on a RAID-5 system. Unlike RAID-1 through RAID-5, which provide exact data encoding techniques, RAID-6 is only a specification and as a consequence there are a number of available coding techniques. The recently introduced Liberation codes promise to become a standard for RAID-6.

A RAID-6 system (Figure 4.1) is composed of $k+2$ data nodes and can tolerate the failure of any two devices. Devices D_0 through D_{k-1} can each store B bytes, whereas the remaining 2B bytes are in the P and Q coding devices. The P device is calculated to be the parity of all data devices, while the implementation of the Q device is left to the designer, with the sole constraint that it cannot hold more than B bytes and the resulting system must be able to recover from the failure of any two devices.

Liberation coding (Figure 4.2) is similar to Cauchy Reed-Solomon coding [83]. The system splits each data device into w words and uses a $w(k+m) \times wk$ matrix to perform the encoding, where k and m represent the number of data and encoding devices respectively. For all RAID-6 techniques, the value for m is two. All operations are performed in Galois Field (2), where addition and multiplication are bitwise XOR and AND operations, respectively. The matrix is called a Binary Distribution Matrix (BDM) and each element is either one or zero. BDM is multiplied by the vector representing device bits, to produce a vector representing the data and encoding devices. The BDM is quite restricted as the top $k(w \times w)$ portion of the matrix is the identity, $D_{0,1}$ through $D_{0,k-1}$ are also identity matrices that produce the P device, and the bottom row can be customized as per rules laid out in [83].

The encoding matrices for the Liberation codes are shown to be optimal or close to optimal. The decoding matrix is produced by inverting the portion of the encoding matrix that corresponds to the data devices that are still active. However, the resulting matrix typically has

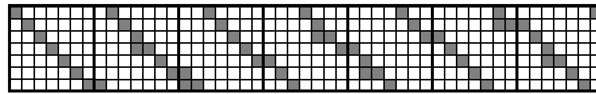


Figure 4.2 Bottom row of BDM used to compute parity for the Q device for a system with 7 devices and word size of 7. Gray boxes represent a 1, white a 0.

far more 1*s* than optimal and in some cases it is more efficient to calculate a word in one of the failed devices from a previously computed product, rather than from the original BDM matrix by data vector product. To take advantage of this, a schedule is created from the BDM that does the least number of XORs. The optimized schedule produces a significant speedup for decoding. A schedule can also be used in the encoding process as it is a more compact representation of the operations than the BDM itself [54].

4.1.1.3 The Lustre Parallel File System

Lustre [56] is a storage architecture for Linux-based clusters and provides a POSIX-compliant UNIX file system interface. It is best known for powering seven of the ten largest HPC machines worldwide, with thousands of client systems, petabytes of storage and hundreds of gigabytes per second I/O throughput. Many HPC sites use Lustre as a site-wide global file system, serving dozens of clusters on an unprecedented scale, e.g., the Spider file system [84]. A Lustre file system comprises of the following key components: *Client*, *MDS* (MetaData Server) and *OSS* (Object Storage Server). Each OSS can be configured to host several *OST*s (Object Storage Target) that manage the storage devices.

The Lustre client that runs on the compute nodes of the cluster communicates with the MDS to obtain privileges and layout for a given file. Once file metadata has been received, the client is able to directly communicate with the OSTs that house the objects associated with the file. An important feature of the Lustre file system that we exploit in our design is its ability to store files in multiple same-sized objects striped over multiple OSTs. Moreover Lustre provides extensive management and recovery features that are useful in identifying the files affected in the event of an OST failure. Thus, Lustre provides some key building blocks to turn each file into its own RAID array.

Lustre also supports hot-swappable hard-drives on each OSS. In the case of a disk failure, a new disk can easily replace the failed disk. Upon a mount, the Lustre manager node detects and recreates the objects that were present in the failed disk. During the per-file RAID array rebuild process, our system restores the data in the lost objects, while reusing the objects that have not failed.

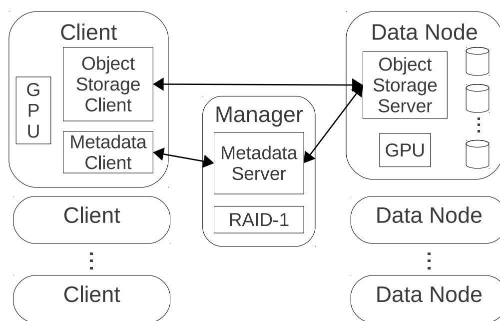


Figure 4.3 High-level architecture of the GPU-enabled RAID system.

4.1.2 GPU-Accelerated RAID

In this section, we present the design of our GPU-enabled RAID system and its realization within the Lustre PFS [56]. We also describe the use of KGPU [62], a GPU management framework, in our system.

4.1.2.1 System Overview

A high-level overview of the hardware and software components used in our system is shown in Figure 4.3. The *Data Nodes* serve as the main storage components; the *Client* provides the user-side interface to the system; and the *Manager* directs and facilitates the interactions between all components. All system components are tightly integrated with the Lustre PFS. The clients typically run on the GPU-enabled compute nodes of the cluster. All or a subset of the Data nodes are equipped with a GPU to perform parity computation during a RAID array rebuild. This hardware addition is feasible on many deployments, since modern motherboards typically have a built-in PCI-Express (PCIe) slot. For the setups where installing a GPU on Data nodes is not an option, the array rebuild process can be offloaded to idle client machines. Each Data node runs an *Object Storage Server* (OSS), which provides file I/O services and network request handling for all the *Object Storage Targets* (OSTs). The OSTs manage the disk drives that store chunks of files called objects. A file in the Lustre PFS can be striped over any number of equally sized objects.

In our design, the Manager is equipped with a hardware or software RAID-1. The Lustre guidelines suggest using RAID-1 or RAID-1+0 for the disks on the Manager, which efficiently performs frequent updates on small metadata files. The Manager runs a MDS that only stores metadata (such as file names and layout, directories, and permissions), which generally accounts for only 1% of the total storage capacity of the system [85]. This ensures that only a small number of disks are required to store the entire metadata in a typical deployment. Hence, equipping the Manager with a low-end RAID-1 controller with a small number of ports (or utilizing a software RAID) fits with our overall goal of achieving fault tolerance with minimal cost.

Each client node in our design is equipped with a programmable GPU that is used to accelerate the file encoding and decoding process. Each client node runs a *Metadata Client*, which communicates with the MDS at the Manager to serve all directory and file operations, such as opening and closing, on behalf of the client. Each client also runs an *Object Storage Client*, which interacts with the OSS at the Data node to read and write to the file objects in parallel. This enables the client to bypass the Manager for all subsequent read and write operations after opening a file and receiving its layout on the Data nodes.

We use the fault tolerant Manager to "bootstrap" the per-file RAID-6 arrays created by our system. If an OST device fails, the Manager identifies all the surviving objects of a given file, which are then used to reconstruct the lost objects.

4.1.2.2 RAID-enabled PFS Design

One of our key design objectives is to make our system compatible with Lustre so that it can be easily integrated with extant Lustre deployments. To this end, our first design choice is to keep the Lustre backend software infrastructure (Manager, OSS, etc.) intact and limit our software-level modifications to the client nodes.

One design obstacle for integrating parity acceleration on the client-side is that the NVIDIA CUDA toolkit is designed to run in user-space, while the Lustre client is implemented as a kernel module. One option is to augment `liblustre` [86], a user-space implementation of the Lustre client, to handle parity generation and storage. This approach decreases the number of context switches that are otherwise required to send data between the client module and the GPU. However, `liblustre` is not widely used in practice as it does not support many performance enhancing features of the kernel implementation, including client-side caching and the support for multi-threaded applications. Hence, we integrate all parity generation inside the Lustre client module and use KGPU to access the GPU directly from kernel space. We implement parity encoding and decoding as a service provided by the user-space component of KGPU.

Another challenge is to find the appropriate location to transparently store the extra parity information. One option is to create a separate "shadow" parity file for each file. This is promising, especially in Lustre, where a file can be striped over any collection of OSTs and by ensuring that the shadow file is stored on different OSTs from the OSTs containing the actual file contents, we can provide a complete RAID-6 array. Moreover, the data file and its attributes remains intact and can be accessed without modification. However, this approach doubles the number of files in the storage system and may introduce a bottleneck at the manager node. Additionally, updating the parity would require write locks on two different files simultaneously and would complicate the locking procedure. An alternative approach that incurs minimal bookkeeping overhead is to interleave data and parity in the same file. However, utilizing this approach requires an effective mechanism to hide the parity from the user. To this end, we modify all file and inode operations that can expose the parity information, such as `write`, `read`, `seek`, `get` and `set` attribute. For operations such as `seek`, and

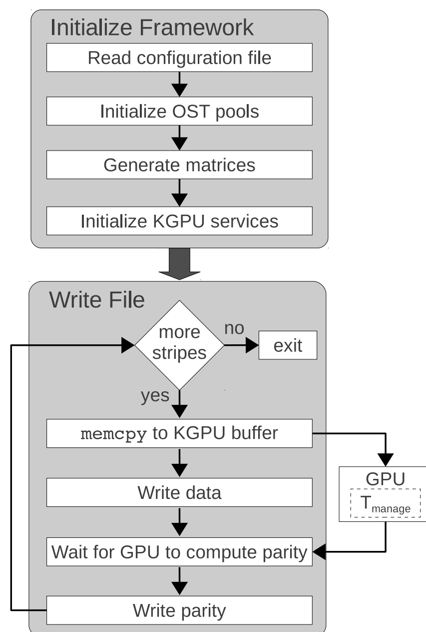


Figure 4.4 Control flow in our GPU-enabled RAID system.

`get/set` attribute, we perform a translation between the size of the actual file including the parity and size of the data. The bulk of the parity generation modifications are contained in the `write` call.

An important feature of our system that significantly decreases overheads when writing to small files is that as long as a file is smaller than a single object it is configured as a RAID-1. We achieve this by mirroring each write into the first parity object, while keeping the second one empty. If a write anywhere outside the first data object is submitted to the system it automatically locks all stripes and converts the file to a RAID-6 array. Note that while in the RAID-1 state no extra space is wasted as the empty blocks inside the second parity object are never written to disk. To maintain consistency we lock the parity object instead of the data, which ensures that a concurrent write to any portion of the stripe, would conflict with the current write and thus will be properly serialized. In this RAID-1 state the GPU is completely bypassed eliminating the expensive read-modify-write step.

4.1.2.3 Control Flow

We now describe the interactions between the different components of our system and how they come together to realize the flexible RAID-6 solution.

Figure 4.4 illustrates the control flow between different components of the system. The system is initialized by reading a configuration file, which specifies different architectural and RAID array specific parameters, such as available GPU memory, maximum supported file object size, and maximum number of disks a file can be striped over. These parameters are

used to initialize global defaults, such as the coding bit matrix used in the default parity algorithm. Some of these parameters are passed on to the KGPU framework, which spawns a GPU management daemon, T_{manage}, that we later utilize to compute the parity. T_{manage} initializes its *request* containers and allocates their associated buffers. The daemon then waits for the jobs to be submitted to the request queue.

The Manager initializes the appropriate storage pools before the Lustre file system can be mounted. In Lustre, any OST can be assigned to a number of storage pools, which we use to define default RAID arrays in our system. Storage pools can be modified at runtime to support addition or removal of storage devices. Once Lustre is mounted on the client, the root directory is assigned to the default storage pool. Files and directories created under the root are recursively assigned the default pool. On creation, each file receives a randomized order in which to write to the OSTs in its pool, which ensures that parity is spreads around the OSTs. In addition to the given defaults, applications have full control to assign files and directory trees to any other pool using standard Lustre system calls.

The bulk of the operations are performed during a write. Lustre caches data on the client side and as a consequence most data writes are processed asynchronously. Synchronous I/O is triggered when the Lustre cache fills up. Lustre breaks down the write in a loop based on the object size. In each iteration, the client asks for a lock on the object and proceeds to update the object, releases the lock, and moves on to the next object. In order to ensure consistency of the parity during simultaneous writes to the same file stripe, we acquire a lock that spans all of objects in a stripe. Thus, we increase the granularity of Lustre's locking from an object to a stripe of objects. Note that we still allow multiple clients to be simultaneously reading and writing to the same file, as long as it is to a different stripe.

After acquiring the lock on a stripe, we copy the relevant portion of the write buffer to CUDA page-locked buffer previously initialized by KGPU and send a request to the KGPU module. The copy is required to maximize the PCIe bandwidth utilization ensured by the CUDA page-locked buffer. The request is then forwarded to our parity generation service implemented in the KGPU user-space daemon that interacts with the GPU to compute the parity for the buffer and return it to the caller. KGPU's call is asynchronous with the data write to the Lustre cache and for a full stripe write completes before it, thus hiding all the latencies associated with moving data to and from the GPU and computing parity. The only overhead exposed is due to the `memcpy` call and parity write. We quantify these latencies in our evaluation.

A read operation also acquires the lock in a loop. In the common case when only data is read, the read loop skips over the parity objects in each stripe of the file. However, a user can also read parity along with the data to ensure end-to-end integrity. In that case, locking is again done at the granularity of the stripe and data and parity is sent to the GPU for validation. If data corruption is detected the read call is restarted. However, if the call fails again an error is returned, as it indicates a permanent error in one of the system components.

4.1.2.4 Degraded Array Reconstruction

Unlike a conventional hardware RAID controller, our system is capable of utilizing multiple GPUs to reconstruct a degraded array. If a disk fails, it can be replaced manually or via a hot spare. The disk is formatted if necessary and assigned the same internal Lustre ID as the failed disk. When the new disk is mounted, the Manager recreates all the missing objects and relinks them to the file objects on the surviving disks. Next, the system requests a list of files that have been affected, and based on the location of the failed disks and the availability of GPUs, mounts a Lustre client on the machines to reconstruct the lost objects. The list of affected files is then split accordingly and forwarded to the reconstructing clients to rebuild the affected files in parallel.

4.1.3 Implementation

We have implemented our system as described in Section 4.1.2 using 1272 lines of C/C++ and CUDA code. The implementation runs on Linux (kernel version 2.6.32) and is portable to CUDA-enabled GPUs. We based our parity generation implementation on the definition of Liberation Codes [54], which is provided in a freely available library, called *Jerasure* [87]. *Jerasure* provides a single threaded implementation for both Liberation and Blaum-Roth functionality.

Our analysis of Liberation and Blaum-Roth codes' single threaded implementation revealed that more than 95% of the time is spent in the function that performs the XOR operations. We also noted that the same function is used for both encoding and decoding, with the only difference being the schedule. Furthermore, the work done in this function has the potential for both coarse and fine-grain parallelism, making it a good candidate for offloading to the GPU. Therefore in our implementation, we offload only XOR operations on the data to the GPU to maximize SIMD parallelism. Note that most of the other operations in the coding process, such as creating the BDM and converting it to a schedule are computed once and sent to the KGPU service at initialization. As these operations are at most quadratic in the number of drives, which are usually in the tens in a typical RAID array, the overhead for these tasks is negligible.

4.1.3.1 Basic GPU Implementation

As described earlier, a schedule is derived from the original BDM matrix while performing the XOR operations on the given data or while copying it between different devices. The schedule is a two dimensional array of integers of size $5 \times N$, where N is the number of operations that need to be performed for encoding. The operations defined are XOR or `memcpy`. The five integers in each tuple identify which words will be operated upon. The

```
__constant__ int d_schedule[8192];
__constant__ int d_num_reads[1024];
__constant__ long d_data[64];

__global__ void xor_gpu(int packetsize,
                        int blocksize, int k, int w) {
  int dest, source, i, y = (k + 2)*2*blockIdx.y;
  const unsigned int g_tid = blockIdx.x*blockDim.x + threadIdx.x;
  int *dptr = (int *)((char *)d_data[d_schedule[y]] + d_schedule[y+1]*packetsize);
  int *sptr = (int *)((char *)d_data[d_schedule[y+2]] + d_schedule[y+3]*packetsize);
  dest = sptr[g_tid];
#pragma unroll
  for(i = 4; i < d_num_reads[blockIdx.y] * 2; i += 2) {
    sptr = (int *)((char *)d_data[d_schedule[y+i]] + d_schedule[y+i+1]*packetsize);
    source = sptr[g_tid];
    dest = dest ^ source;
  }
  dptr[g_tid] = dest;
}
```

Figure 4.5 GPU parity computation kernel.

first two integers identify the id of the device and the word that will serve as source, while the next two identify the destination. The last integer is either 1 for XOR or 0 for `memcpy`. For example, the operation $< 00700 >$ can be interpreted as the first word of device 0 is to be copied over the first word of device 7. In the case of encoding, the schedule is used to compactly represent the BDM. In the offloaded function, the schedule is also flattened to a single dimensional array for easier copying of the data from the host to the GPU memory. Furthermore, since this data is relatively small and does not change during GPU kernel execution, it is copied directly to the GPU's constant memory to enable faster access by the GPU threads. The kernel iterates over all the operations in the given schedule and each thread performs an XOR or `memcpy` operation on the corresponding words (in 4 byte chunks) in parallel. Hence, the amount of parallelism exposed depends directly on the word size, which is determined by the size of each data object.

4.1.4 Optimizations

The main drawback of the basic GPU port is that it reveals only the fine-grain parallelism that is present within a scheduled operation. We analyzed data dependencies and found that entire operations can be done in parallel as well. Specifically, the schedule produces the $2w$ words of the coding devices of a RAID-6 array, where w is 8 and 16 for the Liberation and Blaum-Roth coding, respectively. All the operations associated with computing a single word in a coding device can be performed in parallel with the ones that encode the rest of the words. To exploit this, we modify the schedule and create our optimized port shown in Figure 4.5.

We create a two dimensional grid of thread blocks and assign each of the $2w$ rows of blocks to perform the operations associated with one of the encoding words. We use an additional structure, `num_reads`, to store the number of operations needed to compute each of the $2w$

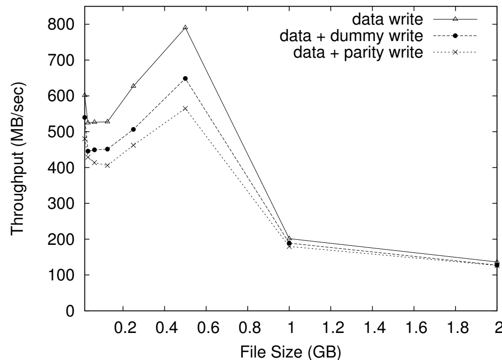


Figure 4.6 Write throughput for a file striped over 16 OSTs + 2 parity OSTs.

coding words. This enables the kernel to execute fewer iterations compared to the basic port, thus simultaneously reducing the work of each thread and exposing more parallelism.

4.1.5 Evaluation

In this section, we present the evaluation of our GPU-enabled RAID system. We first describe our testbed, and then present the I/O measurements of our system. The goal is to show the impact of different design parameters and features, such as RAID stripe size and end-to-end integrity checking, on the overall system performance. Next, we evaluate the performance of RAID array reconstruction. Finally, we quantify performance under a real workload.

4.1.5.1 Experimental Setup

We have set up a Lustre cluster, consisting of one Manager node and three OSSs, each with six OSTs. The Lustre server machines are identical with four Opteron quad-cores each, and 64 GB of main memory. Additionally, each OSS has a GeForce 9500 GT GPU with 1 GB of graphics memory connected to an 8× PCIe slot. Our client machine has two Intel Xeon quad-cores, 48 GB of RAM and a Tesla C2070 GPU with 6 GB of GDDR memory. All the machines are connected using a dedicated Gigabit switch. We use Lustre patched Linux 2.6.32 kernel, Lustre 1.8.5, and CUDA SDK 4.0.

4.1.5.2 I/O Throughput Measurement

4.1.5.3 Raw Throughput

We first measure the raw write throughput that our client machine can achieve. Figure 4.6 compares the throughput of writing a file striped over 16 OSTs with a stripe/block size of 1 MB, denoted as *data write*. The file size ranges between 16 MB and 2 GB. A Lustre client maintains a 32 MB local cache per OST, which is flushed periodically. If a write submitted by a client fits in the cache, the Lustre module returns from the write immediately after the write buffer is written to the cache. If there is no space left in the cache, it is flushed to the corresponding OST and the write returns after the write buffer has been written to the Lustre back-end. Since we are writing to 16 OSTs, the combined available cache is 512 MB, and consequently writes smaller than 512 MB exhibit throughput higher than the theoretical throughput of Gigabit Ethernet. The throughput of files larger than the cache quickly levels out to an effective available bandwidth of around 125 MB/s.

In the Figure, *data + parity write* curve shows the throughput when our RAID encoding system is turned on. In this case, the same data as in the base case is striped over 16 OSTs and concurrently the parity is generated and written to the remaining 2 OSTs. As a point of reference we also include a *data + dummy write* curve, which generates the same traffic as the RAID encoding, without computing the parity.

Writes that fit into the Lustre cache exhibit a very high throughput and as a result the overhead of `memcpy`-ing data into KGPU buffers results in around 10% overhead (difference between *data + dummy write* and parity *data + parity write*). The rest is attributed to copying the extra parity ($1/8^{th}$ of data in this case) to the Lustre cache. It is important to note that in this experiment all the overhead associated with parity generation remains completely hidden from the application.

4.1.5.4 Encoding Throughput

Next, we evaluate the parity encoding throughput delivered by the GPU. We measured throughput delivered by our high and low-end GPUs, which includes moving the data to and from the GPU's memory as well as actual parity computation on the GPU (Figure 4.7). A low-end GPU can deliver encoding throughput around 1 GB/s for 512 KB files, which quickly increases to 1.7 GB/s for files large than 8 MB. The Tesla GPU delivers encoding rates of 1.6 GB/s for 512 KB files and in excess of 3 GB/s for files larger than 8 MB. Therefore, our system using a low-end GPU can generate parity faster than the speed at which Lustre commits the data to its caches. As parity is encoded asynchronously with the commit to cache, the overhead of generating it remains hidden.

We also include the encoding rates of the Tesla GPU, while it is under heavy load from an

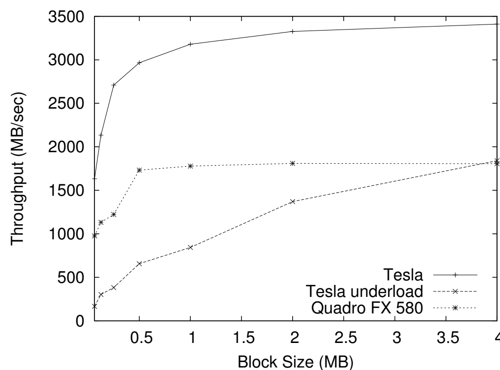


Figure 4.7 GPU encoding throughput.

N-body simulation. We used the N-body simulation from the CUDA SDK and ran multiple iterations in the benchmark mode using the default number of objects based on the specifications of the Tesla GPU. Even under heavy load, the Tesla GPU delivers sufficient throughput for all but the smallest files, for which the encoding overheads are exposed to the system as increased latencies due to the heavy load. However, the performance of the simulation is unaffected by the parity generation kernels and remains constant at 484.650 single-precision GFLOP/s. This is because Tesla GPU can perform efficient context switches at the kernel boundary and asynchronous data transfer for different contexts can run simultaneously. Therefore the parity data can be transfered to the GPU, while the simulation kernel is running and vice versa. Moreover, the parity generation kernels complete 2-3$\times$ faster than the simulation kernels.

It is important to note that not all background loads on the GPU have the same effect on the parity generation. An iterative workload with kernel execution times in the order of milliseconds, such as the N-body simulation that can be rendered in real time, would not block the parity kernels and cause an unacceptable slowdowns to our system. However, for GPU kernels with execution times in seconds, alternative techniques such as "context funneling"[1] can be used to minimize the overhead. The down side is that the GPU workloads need to be modified, e.g., as a KGPU service. This enables even a long running kernel to run parity generation concurrently.

4.1.5.5 Impact of Number of Disks on Throughput

Next, we study the effect of number of OSTs that a file is striped over on the write throughput of our system. Figure 4.8 shows the baseline write throughput and the throughput of our system for writing a 256 MB file. When the file is striped over 6 OSTs or less, it cannot fit in the client caches and as a result, raw network bandwidth is exposed to the application. If the 256 MB file is striped over more than 6 drives, parity is cached and flushed after the

[1]Context funneling uses advanced features of the Fermi architecture to execute concurrent kernels, which must be launched from the same context [88].

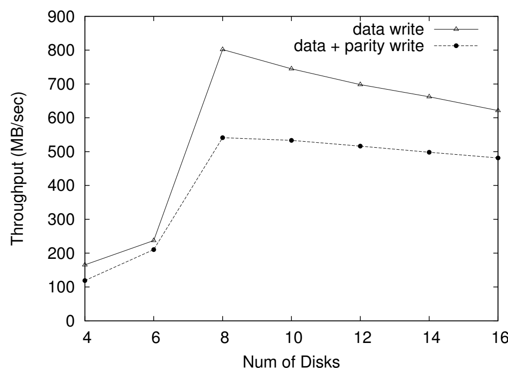


Figure 4.8 Effect of number of disks on throughput (file size = 256 MB).

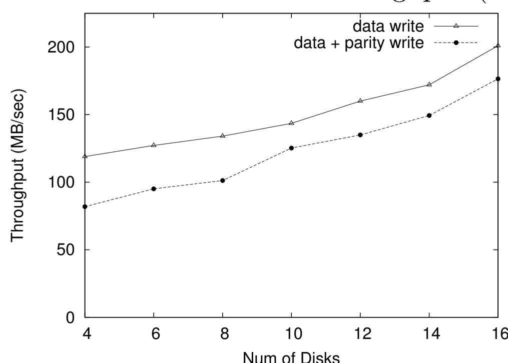


Figure 4.9 Effect of number of disks on throughput (file size = 1024 MB).

write returns. As the write fits in the caches, throughput levels out. As the file size remains constant splitting and committing it to more caches becomes less efficient, which causes the slight dip in the throughput.

Striping does not have such an effect when writing a 1024 MB file as the file and its parity cannot be cached (Figure 4.9). In the *data write* case, writing to more drives achieves better throughput because of efficient bandwidth utilization when the file is striped over all available OSTs. For the *data + parity write* case, decreasing the number of drives has the effect of decreasing the length of the RAID 6 array, e.g., striping data over four disks produces a (4,2) RAID 6 array where four objects in a stripe are data and the rest are parity, having a parity overhead of 50%. Increasing the array length decreases the relative size of the parity, e.g., in a (16,2) RAID 6 array parity is 12.5%. Thus, for the *data + parity write* case, there is a linear increase in throughput available for data with the increase in the array length.

4.1.5.6 End-to-End Data Integrity

One of the important features of our system is that it can provide end-to-end data integrity checks for the I/O operations. Figure 4.10 shows the achieved read throughput when the end-to-end integrity check is enabled. To ensure that data is not corrupted on the disk or

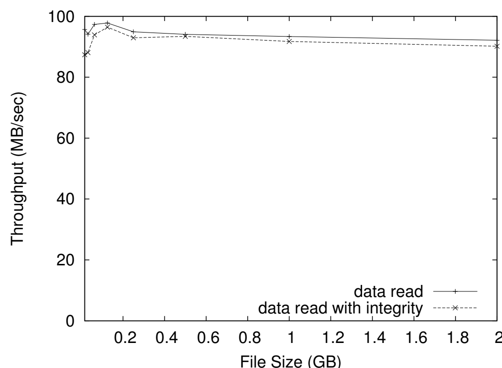


Figure 4.10 Read throughput with end-to-end data integrity.

Data Size (GB)	1 Node	2 Nodes		3 Nodes	
	Time (s)	Time (s)	Speedup	Time (s)	Speedup
5	46	30	1.53	25	1.84
25	237	151	1.57	125	1.90
50	493	345	1.43	258	1.91

Table 4.1 RAID reconstruction time and normalized speedup with respect to 1 Node.

on the network interconnects, one of the parity objects in each stripe is read along with the data. Both data and parity needs to be sent to the GPU for verification to successfully complete the `read` call. For files with single stripes, the synchronous call causes an overhead of 9%. However, when reading files with more than one stripe, the parity check for each stripe is performed in parallel with the read of the next stripe, resulting in a negligible overhead of 2%.

4.1.5.7 RAID Reconstruction Cost

It is critical to minimize the degraded RAID reconstruction time for maintaining the integrity of data, as the system is exposed to unrecoverable read errors during the reconstruction process. Table 4.1 shows the reconstruction time for rebuilding 20, 100, and 200 degraded files with a combined size of 5 GB, 25 GB, and 50 GB, respectively. Files are striped on all 18 OSTs (16 for data and 2 for parity). As the RAID arrays are defined per file, their rebuilding can be distributed between the available machines, resulting in a speedup of close to 2× when reconstructing for the 200 files case. During this test, we use all the machines in our setup, which results in the utilization of all the available network bandwidth. It is important to note that each of our low-end GPU achieves an effective reconstruction rate of 1.5 GB/s, therefore even higher speedup is possible, if network and disk throughput permit it.

These results show that our GPU-enabled RAID solution is feasible, and provides a configurable and flexible solution.

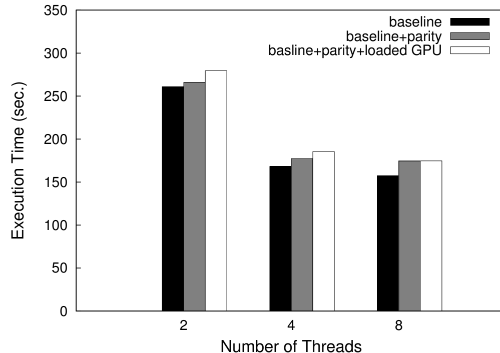


Figure 4.11 Performance of NAS DC benchmark.

4.1.5.8 Impact on Applications

Next, we examine the performance of our system under load by a real-world application, the Data Cube (DC) NAS OpenMP [89] benchmark. DC performs a data-intensive operation known in data mining as the Data Cube Operator (DCO), which computes views of a dataset represented as a set of `n` tuples and involves O(log n) memory accesses per tuple.

Figure 4.11 shows the performance of DC executing on our client machine with varying number of threads. It is configured to write out the views to disk as they are computed, thus stressing both memory and the storage subsystem. At two threads the benchmark is actually CPU-bound, thus generating and writing out the extra parity for each view introduces a small additional slowdown of 2%. Beyond four threads the benchmark becomes I/O-bound and as a result, the overheads due to parity produce a 5% and 10% slowdown for four and eight threads, respectively. We also measured performance of our parity generation system under a heavy background GPU load produced by an N-body simulation application running on the GPU. When the DC benchmark is running with eight threads, the background job does not affect performance at all, because our system is able to schedule the workload for the eight threads more effectively. With fewer threads requesting parity, the system cannot obtain enough time on the GPU and as a result exposes parity generation overheads to a portion of the write operations, resulting in a slowdown of around 5%.

4.2 Offloading Flash Address Translation to a GPU

In this section, we detail a proof-of-concept implementation that uses the CONA structure described in Chapter 3 to perform virtual to physical address translation as part of a Flash Translation Layer (FTL) of a PCIe SSD (Figure4.15). In these devices, the FTL is implemented as part of the driver, which communicates with the DMA engine running on the onboard FPGA. Translation in the FTL is usually implemented in a map table stored in main memory. While such a map table provides adequate performance, it performs all modifications to the table in place. Because the modifications are done in place and in sizes

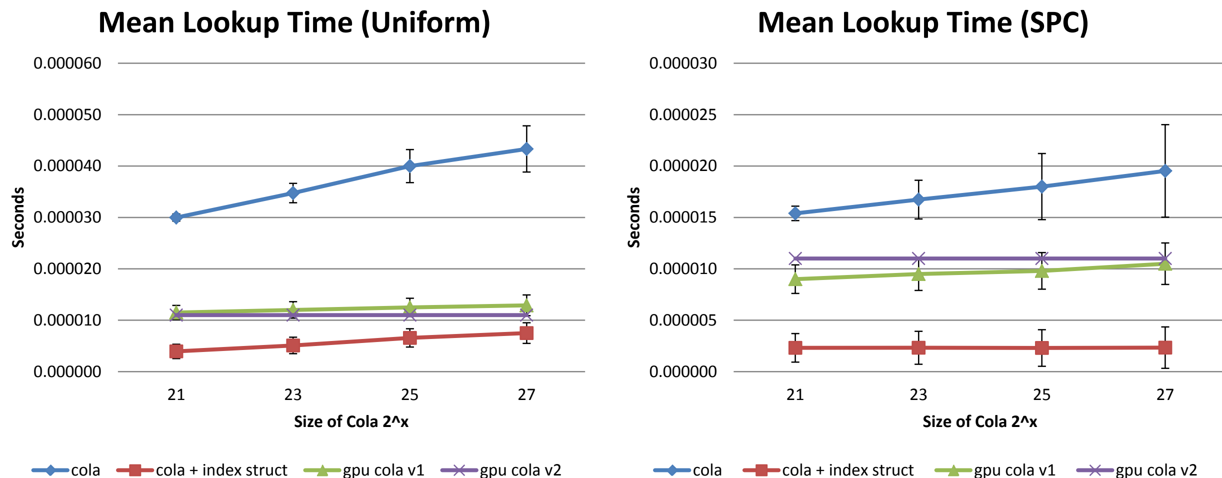


Figure 4.12 Average lookup time vs. size of the COLA.

much smaller than a write page, it is difficult to persist the table efficiently. As a results FTL implementations often opt not to persist the changes to table on every write, but only keep it memory. In the case of a power loss or unclean unloading of the driver, the entire device needs to be scanned to bring the device to consistent state, which can take prohibitively long.

Addresses are treated as 4-byte integers, which means that the structure can comfortably fit the GPUs main memory without needing any compression. Changes to mappings are accumulated in the insertion buffer on the host side. When any level of the CONA is merged into, the changes are sent to the GPU and also persisted on flash.

We implemented two version of the CONA: 1) We configure initial buffer size B to match a multiple of the SIMD elements in the GPU; and 2) We configure B so that it accounts for the height of the CONA. The first version produces a constant search at every level, while the the second produces a constant search for the entire CONA.

Figure4.12 shows average lookup time with both a uniform and an SPC workload. Both version of the GPU implementation achieve better performance than a standard host-side COLA implementation, whose lookup performance degrades as the size of the structure increases. The GPU implementation, in contrast, achieves a constant time lookup even for very large CONAs. Note that the host size implementation, which uses our novel indexing, produces slightly better results that the GPU version for single requests. Later we show that the GPU can easily batch up to 32 request, in which case the kernel launch and data transfer can be amortized, thus outperforming a CPU implementation for a large COLA.

Due to large core count on the GPU, which allows the system to perform a large number of comparisons in parallel, we can utilize a much sparser index the original implementation. To highlight effectiveness of our approach, we compare the performance of our system to

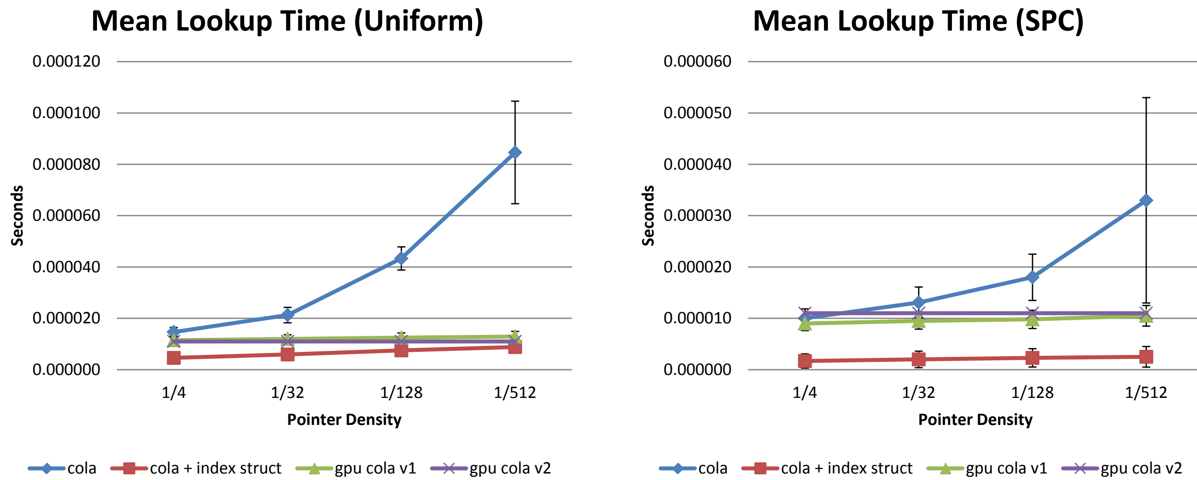


Figure 4.13 Average lookup time vs. pointer density of the CONA.

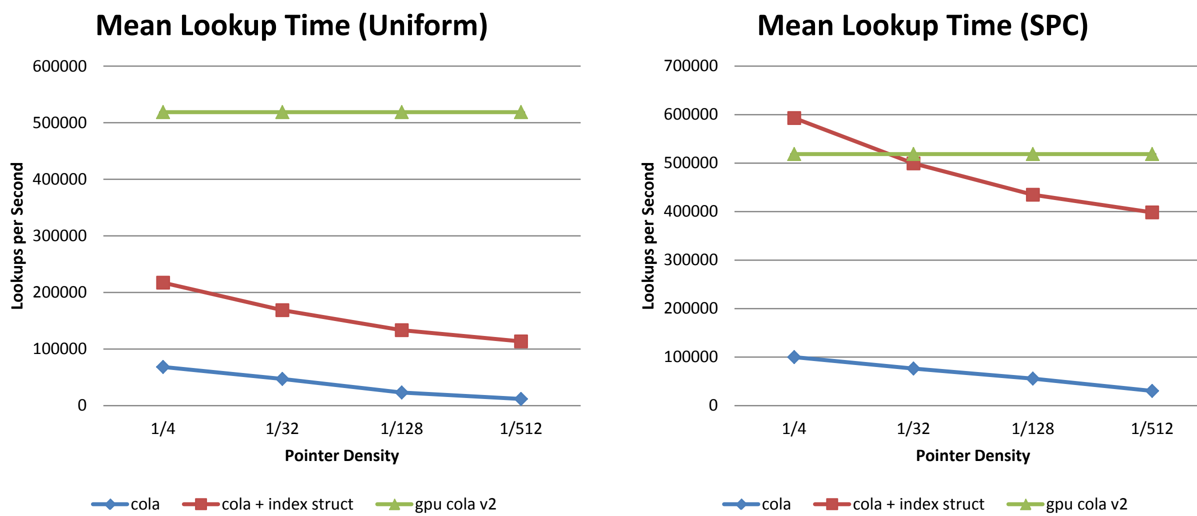


Figure 4.14 Lookup per Second vs. size of the CONA.

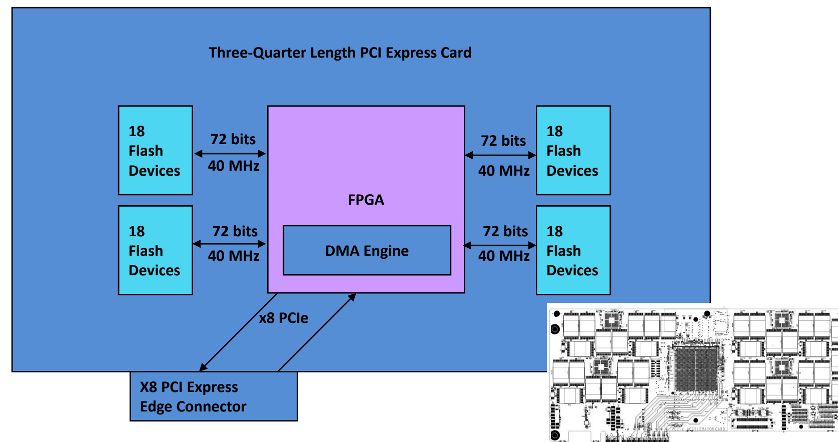


Figure 4.15 PCIe SSD.

a CPU implementation with a similar indexing overhead. Figure 4.13 plots average lookup time of both a uniform access and SPC workload. As the pointer density decreases in the traditional COLA, average lookup time increases dramatically due to the sequential search required to locate the increasingly sparse indexes into the next level.

Finally we compare lookup performance of CPU implementation vs a GPU implementation that batches 32 request for every kernel launch (Figure 4.14). As the GPU implementation achieves a near linear lookup time, even for moderately sized COLAs the GPU outperforms even the optimized CPU code, whose performance drops as the size of the COLA increases.

4.3 Chapter Summary

In this Chapter, we have presented a cost-effective alternative that uses commodity GPUs to implement RAID-6 in software, in conjunction with the Lustre PFS. Our solution leverages low-cost GPUs on the client and server nodes to accelerate minimum-density RAID-6 coding schemes. We have shown, through a prototype implementation, that our software-controlled parity computation scheme imposes acceptable overhead on application performance, and constitutes, overall, a feasible, low-cost, and efficient alternative to specialized hardware-based solutions. We also provide a brief description of a proof-of-concept implementation that uses that uses the CONA structure described in Chapter 3 to perform virtual to physical address translation as part of an FTL. In Section 5.1, we outline a direction of future work to extend this proof-of-concept work.

Chapter 5

Conclusion

In this dissertation, we presented comprehensive frameworks that leverage emerging technologies, including GPUs and flash-based SSDs, to accelerate modern storage systems. Our accelerator-based solution features deep integration of the GPU in a distributed PFS. The system builds on the resources available in the file system and coordinates the workload to minimize data movement across the PCIe bus, while exposing data parallelism to maximize the potential for acceleration on the GPU. We improved the overall reliability of the PFS by developing a distributed per-file parity generation that provides end-to-end data integrity and unprecedented flexibility.

To tackle the challenges associated with unlocking the full potential of flash-based SSDs to accelerated I/O workloads, we designed and implemented a comprehensive solution that delivers low read and write amplification on flash using fast, yet memory-efficient indexing and lookups. Our proposed Cache Oblivious No-Lookahead Array (CONA) structure addresses shortcomings of state-of-the-art KV stores, such as excessive read and write amplification. Writes to the CONA's log-like structure are never in place, which suits current flash technology, while the CONA's efficient in-memory index only requires a single flash read per lookup. Additionally, CONA offers a range of customization options that can be used to tune read and write amplifications, as well as tailor insertion versus lookup performance to a given application's requirement.

5.1 Future Research

In this dissertation, we addressed the challenges of integrating emerging technologies to accelerate modern storage systems. Nevertheless, there exist a number of open questions related to the efficient deployment and integration of accelerator based solutions. In the following, we outline a vision that follows natural extensions of the techniques discussed in this dissertation.

As flash technology matures, so do the Flash Translation Layers (FTLs) that power current state-of-the-art SSDs. Recently, FTLs expose an increasingly richer set of features to applications, opening the door for a host of optimizations that bring applications closer to the

flash [78, 90]. For example, one area of opportunity is eliminating redundancies by tightly integrating a storage applications, such as a key-value store with an FTL. Another promising area is performing data analysis out-of-core, where an accelerator, such as a GPU, interacts directly with the storage device without impacting the host's memory and CPU resources.

5.1.1 Out-of-core Data Analysis on a GPU

One major limitation of utilizing an accelerator, such as a GPU, to offload any storage operations is that the GPU cannot directly interact with the I/O device. This results in unnecessary overhead on the host side to orchestrate the process. One possible direction to perform GPU analysis of data stored on a flash device is to tightly integrated the GPU with the FTL. The FTL can then stream relevant flash pages through the GPU, which will run the appropriate data analysis kernels. There are several possible usage scenarios for such a system: 1) Out-of-core analysis, where data analysis is performed during low I/O periods when the main applications is performing computations on data that fits in main memory, which is a typical scenario in HPC deployments. In this case, all the I/O bandwidth of the device will be utilized to stream relevant data for analysis through the GPU, without interfering with the main application. And 2) In-line analysis, where all relevant I/O traffic is first streamed through the GPU for analysis, and then sent to the flash device to be persisted.

The main challenge will be to ensure that the overall I/O rate as seen by the main application is not affected by the data analysis kernels. This can be addressed by performing analysis during low I/O periods, so that the GPU does not compete for I/O throughput, in the out-of-core usage case. For in-line analysis, an appropriate streaming framework is required which overlaps kernel execution on the GPU with CPU-to-GPU and GPU-to-flash transfers, so that available bandwidth to the multiple channels of the PCIe SSD are fully utilized. Such framework will be able to utilize the processing power of the GPU to perform even highly computationally intensive data analysis kernels, by streaming appropriate flash pages directly to and from the GPU's main memory, without affecting the host system.

5.1.2 Tight Integration of a KV store with the FTL

Another possible research direction is exploiting the recent expansion in Flash Translation Layer (FTL) capabilities to implement a storage application, such as key-value store, as a thin layer running closer to the flash. Modern FTLs are beginning to provide functionality similar to the data structures presented in this thesis, with their log-like structure and out-of-place updates [91–93]. We envision a storage architecture where the logic running on the host side is responsible for scale out, load balancing across multiple SSDs that natively support a KV store like interface.

Bibliography

[1] Top500 supercomputer sites. http://www.top500.org/.

[2] Bob Yirka. Supercomputer Titan to get world's fastest storage system, Apr 2013. http://phys.org/news/2013-04-supercomputer-titan-world-fastest-storage.html.

[3] Brooke Crothers. DARPA 'exascale' supercomputer in the works, August 2010. http://news.cnet.com/8301-13924_3-20013088-64.html.

[4] Philip Schwan. Lustre: Building a File System for 1,000-node Clusters. In *Proc. Ottawa Linux Symposium*, Ottawa, Canada, July 2003.

[5] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end data integrity for file systems: a zfs case study. In *Proceedings of the 8th USENIX conference on File and storage technologies*, FAST'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[6] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[7] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 2:1–2:17, Berkeley, CA, USA, 2008. USENIX Association.

[8] Apache Software Foundation. Apache Cassandra, Feb 2011. http://cassandra.apache.org/.

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

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[12] Apache Software Foundation. Hadoop, May 2007. `http://hadoop.apache.org/core/`.

[13] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. Silt: a memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 1–13, New York, NY, USA, 2011. ACM.

[14] Guanlin Lu, Young Jin Nam, and D.H.-C. Du. Bloomstore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–11, 2012.

[15] Biplob Debnath, Sudipta Sengupta, and Jin Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 25–36, New York, NY, USA, 2011. ACM.

[16] Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. Cheap and large cams for high performance data-intensive networked systems. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 29–29, Berkeley, CA, USA, 2010. USENIX Association.

[17] Anirudh Badam, KyoungSoo Park, Vivek S. Pai, and Larry L. Peterson. Hashcache: cache storage for the next billion. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, pages 123–136, Berkeley, CA, USA, 2009. USENIX Association.

[18] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: high throughput persistent key-value store. *Proc. VLDB Endow.*, 3(1-2):1414–1425, September 2010.

[19] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: a fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 1–14, New York, NY, USA, 2009. ACM.

[20] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *OSDI'10*, pages 47–60, 2010.

[21] Jure Petrovic. Using memcached for data distribution in industrial environment. In *Proceedings of the Third International Conference on Systems*, ICONS '08, pages 368–372, Washington, DC, USA, 2008. IEEE Computer Society.

[22] Memcached: A distributed memory object caching system, 2011. http://www.danga.com/memcached/.

[23] Facebook Engineering (Notes). Mcdipper: A key-value cache for flash storage, March 2013. http://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-fl

[24] Dean, J. and Ghemawat, S. LevelDB, Jan 2015. http://code.google.com/p/leveldb.

[25] Wenbin Fang, Ka Keung Lau, Mian Lu, Xiangye Xiao, Chi Kit Lam, Philip Yang Yang, Bingsheng He, Qiong Luo, Pedro V. S, and Ke Yang. Parallel data mining on graphics processors. Technical report, 2008.

[26] HPCwire. Jp morgan speeds risk calculations with nvidia gpus, August 2011. http://www.hpcwire.com/hpcwire/2011-08-04/jp_morgan_speeds_risk_calculations_with_nv

[27] Ling Sing Yung, Can Yang, Xiang Wan, and Weichuan Yu. Gboost: a gpu-based tool for detecting genegene interactions in genomewide case control studies. *Bioinformatics*, 27(9):1309–1310, 2011.

[28] Amazon Web Services. Announcing Cluster GPU Instances for Amazon EC2, Apr 2013. http://aws.amazon.com/about-aws/whats-new/2010/11/15/announcing-cluster-gpu-instance

[29] Matthew L. Curry, H. Lee Ward, Anthony Skjellum, and Ron Brightwell. A lightweight, gpu-based software raid system. *Parallel Processing, International Conference on*, 0:565–572, 2010.

[30] Changxin Li, Hongwei Wu, Shifeng Chen, Xiaochao Li, and Donghui Guo. Efficient implementation for md5-rc4 encryption using gpu with cuda. In *Anti-counterfeiting, Security, and Identification in Communication, 2009. ASID 2009. 3rd International Conference on*, pages 167–170. IEEE, 2009.

[31] Ritesh A Patel, Yao Zhang, Jason Mak, Andrew Davidson, and John D Owens. *Parallel lossless data compression on the gpu*. IEEE, 2012.

[32] J. Michalakes and M. Vachharajani. Gpu acceleration of numerical weather prediction. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–7, april 2008.

[33] Cole Trapnell and Michael C. Schatz. Optimizing data intensive gpgpu computations for dna sequence alignment. *Parallel Comput.*, 35:429–440, August 2009.

[34] Massimiliano Fatica. Accelerating linpack with cuda on heterogenous clusters. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 46–51, New York, NY, USA, 2009. ACM.

[35] Timothy D.R. Hartley, Umit Catalyurek, Antonio Ruiz, Francisco Igual, Rafael Mayo, and Manuel Ujaldon. Biomedical image analysis on a cooperative cluster of gpus and multicores. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 15–25, New York, NY, USA, 2008. ACM.

[36] M. Mustafa Rafique, Ali R. Butt, and Dimitrios S. Nikolopoulos. A capabilities-aware framework for using computational accelerators in data-intensive computing. *J. Parallel Distrib. Comput.*, 71:185–197, February 2011.

[37] M.L. Curry, A. Skjellum, H.L. Ward, and R. Brightwell. Arbitrary dimension reed-solomon coding and decoding for extended raid on gpus. In *Petascale Data Storage Workshop, 2008. PDSW '08. 3rd*, nov. 2008.

[38] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Real-time parallel hashing on the gpu. *ACM Trans. Graph.*, 28:154:1–154:9, December 2009.

[39] Georgia Institute of Technology. Keenland, 2010. `http://keeneland.gatech.edu/`.

[40] Damon Poeter. Cray's Titan Supercomputer for ORNL Could Be World's Fastest, 2011. `http://www.pcmag.com/article2/0,2817,2394515,00.asp`.

[41] M. Mustafa Rafique, Ali R. Butt, and Dimitrios S. Nikolopoulos. Designing accelerator-based distributed systems for high performance. In *Proc. IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID'2010)*, Melbourne, Australia, May. 2010.

[42] M. A. Clark. Qcd on gpus: cost effective supercomputing, 2009.

[43] NVIDIA Corporation. Science & Education, 2011. `http://www.nvidia.com/object/nvidia_userful_success.html`.

[44] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

[45] Samer Al-Kiswany, Matei Ripeanu, Sudharshan S. Vazhkudai, and Abdullah Gharaibeh. stdchk: A checkpoint storage system for desktop grid computing. In *Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems*, ICDCS '08, pages 613–624, Washington, DC, USA, 2008. IEEE Computer Society.

[46] Samer Al-Kiswany, Abdullah Gharaibeh, Elizeu Santos-Neto, George Yuan, and Matei Ripeanu. Storegpu: exploiting graphics processing units to accelerate distributed storage systems. In *Proceedings of the 17th international symposium on High performance distributed computing*, HPDC '08, pages 165–174, New York, NY, USA, 2008. ACM.

[47] Abdullah Gharaibeh, Samer Al-Kiswany, Sathish Gopalakrishnan, and Matei Ripeanu. A gpu accelerated storage system. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC'10, pages 167–178, New York, NY, USA, 2010. ACM.

[48] Samer Al-Kiswany, Abdullah Gharaibeh, Elizeu Santos-Neto, and Matei Ripeanu. On gpu's viability as a middleware accelerator. *Cluster Computing*, 12:123–140, June 2009.

[49] Gabriel Falcão, Leonel Sousa, and Vitor Silva. Massive parallel ldpc decoding on gpu. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 83–90, New York, NY, USA, 2008. ACM.

[50] Owen Harrison and John Waldron. Practical symmetric key cryptography on modern graphics hardware. In *Proceedings of the 17th conference on Security symposium*, pages 195–209, Berkeley, CA, USA, 2008. USENIX Association.

[51] Andrew Moss, Daniel Page, and Nigel P. Smart. Toward acceleration of rsa using 3d graphics hardware. In *Proceedings of the 11th IMA international conference on Cryptography and coding*, Cryptography and Coding'07, pages 364–383, Berlin, Heidelberg, 2007. Springer-Verlag.

[52] I S Reed and G Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[53] M. Blaum and R.M. Roth. New array codes for multiple phased burst correction. *Information Theory, IEEE Transactions on*, 39(1):66 –77, jan 1993.

[54] James S. Plank. The raid-6 liberation codes. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, pages 7:1–7:14, Berkeley, CA, USA, 2008. USENIX Association.

[55] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[56] Sun Microsystems, Inc. Lustre file system - High-performance storage architecture and scalable cluster file system, 2007.

[57] NVIDIA Corporation. NVIDIA CUDA Programming Guide, May 2013.

[58] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev: first-class gpu resource management in the operating system. In *USENIX ATC*, volume 12, pages 37–37, 2012.

[59] Samer Al-Kiswany, Abdullah Gharaibeh, and Matei Ripeanu. The case for a versatile storage system. *SIGOPS Oper. Syst. Rev.*, 44(1):10–14, March 2010.

[60] Abdullah Gharaibeh, Samer Al-Kiswany, and Matei Ripeanu. Crystalgpu: Transparent and efficient utilization of gpu power. *CoRR*, abs/1005.1695, 2010.

[61] Weibin Sun, Robert Ricci, and Matthew L Curry. Gpustore: harnessing gpu computing for storage systems in the os kernel. In *Proceedings of the 5th Annual International Systems and Storage Conference*, page 6. ACM, 2012.

[62] KGPU. KGPU: enabling GPU computing in Linux kernel, 2011. `http://code.google.com/p/kgpu`.

[63] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: Operating System Abstractions To Manage GPUs as Compute Devices. In *Proc. ACM SOSP*, 2011.

[64] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Real-time parallel hashing on the gpu. *ACM Trans. Graph.*, 28(5):154:1–154:9, December 2009.

[65] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, June 1984.

[66] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[67] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE, 2009.

[68] Peter Bakkum and Kevin Skadron. Accelerating sql database operations on a gpu with cuda. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 94–103, New York, NY, USA, 2010. ACM.

[69] Peter Bakkum and Srimat Chakradhar. Efficient data management for gpu databases.

[70] Shinpei Kato, Karthik Lakshmanan, Aman Kumar, Mihir Kelkar, Yutaka Ishikawa, and R Rajkumar. Rgem: A responsive gpgpu execution model for runtime engines. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 57–66. IEEE, 2011.

[71] Shinpei Kato, Scott Brandt, Yutaka Ishikawa, and R Rajkumar. Operating systems challenges for gpu resource management. In *Proc. of the International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 23–32, 2011.

[72] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. Ptask: operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 233–248, New York, NY, USA, 2011. ACM.

[73] Valient Gough. EncFS, May 2013. `www.arg0.net/encfs`.

[74] Pramod Bhatotia, Rodrigo Rodrigues, and Akshat Verma. Shredder: Gpu-accelerated incremental storage and computation. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, pages 14–14, Berkeley, CA, USA, 2012. USENIX Association.

[75] Michael O Rabin. *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[76] Shinpei Kato, Jason Aumiller, and Scott Brandt. Zero-copy i/o processing for low-latency gpu computing. *4th ICCPS*, 2013.

[77] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, and Sang-Won Lee. Fast, energy efficient scan inside flash memory ssds, 2011.

[78] Simona Boboila, Youngjae Kim, Sudharshan S Vazhkudai, Peter Desnoyers, and Galen M Shipman. Active flash: Out-of-core data analytics on flash storage. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–12. IEEE, 2012.

[79] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming b-trees. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, pages 81–92, New York, NY, USA, 2007. ACM.

[80] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1:133–162, 1986.

[81] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[82] Aleksandr Khasymski, M Mustafa Rafique, Ali R Butt, Sudharshan S Vazhkudai, and Dimitrios S Nikolopoulos. On the use of gpus in realizing cost-effective distributed raid. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pages 469–478. IEEE, 2012.

[83] James S. Plank and Lihao Xu. Optimizing cauchy reed-solomon codes for fault-tolerant network storage applications. In *Proceedings of the Fifth IEEE International Symposium on Network Computing and Applications*, pages 173–180, Washington, DC, USA, 2006. IEEE Computer Society.

[84] Galen M. Shipman, David A. Dillow, Sarp Oral, and Feiyi Wang. *The Spider center wide file system: From concept to reality*. 2009.

[85] Oracle Corporation. Lustre Documentation, 2011. `http://wiki.lustre.org/index.php/Lustre_Documentation`.

[86] Sun Microsystems, Inc. LibLustre How-To Guide, 2010. http://wiki.lustre.org/index.php/LibLustre_How-To_Guide.

[87] J. S. Plank, S. Simmerman, and C. D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications - Version 1.2. Technical Report CS-08-627, University of Tennessee, August 2008.

[88] Lingyuan Wang, Miaoqing Huang, and Tarek El-Ghazawi. Towards efficient gpu sharing on multicore processors. In *Proceedings of the second international workshop on Performance modeling, benchmarking and simulation of high performance computing systems*, PMBS '11, pages 23–24, New York, NY, USA, 2011. ACM.

[89] Michael A. Frumkin and Leonid V. Shabanov. Benchmarking memory performance with the data cube operator. In *ISCA PDCS'04*, pages 165–171, 2004.

[90] Leonardo Mármol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. Nvmkv: A scalable and lightweight flash aware key-value store. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'14, pages 8–8, Berkeley, CA, USA, 2014. USENIX Association.

[91] Mohit Saxena, Michael M. Swift, and Yiying Zhang. Flashtier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 267–280, New York, NY, USA, 2012. ACM.

[92] Xiangyong Ouyang, David Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K. Panda. Beyond block i/o: Rethinking traditional storage primitives. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 301–311, Washington, DC, USA, 2011. IEEE Computer Society.

[93] William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. Dfs: A file system for virtualized flash storage. *Trans. Storage*, 6(3):14:1–14:25, September 2010.