UNIVERSITY OF CRETE
DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF SCIENCES AND ENGINEERING

Memory-Mapped I/O for Fast Storage

by

Anastasios Papagiannis

B.Sc., Computer Science, University of Crete, Greece, 2010

M.Sc., Computer Science, University of Crete, Greece, 2013

PhD Dissertation

Presented

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

Heraklion, December 2020

UNIVERSITY OF CRETE

DEPARTMENT OF COMPUTER SCIENCE

Memory-Mapped I/O for Fast Storage

PhD Dissertation Presented

by **Anastasios Papagiannis**

in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy

APPROVED BY:

Author: Anastasios Papagiannis

Supervisor: Angelos Bilas, Professor, University of Crete

Committee Member: Manolis Katevenis, Professor, University of Crete

---

Committee Member: Polyvios Pratikakis, Assistant Professor, University of Crete

---

Committee Member: Kostas Magoutis, Associate Professor, University of Crete

---

Committee Member: Dimitrios Nikolopoulos, Professor, Virginia Tech

---

Committee Member: Christos Kozyrakis, Professor, Stanford University

---

Committee Member: George Amvrosiadis, Assistant Research Professor, Carnegie Mellon University

---

Department Chairman: Antonis Argyros, Professor, University of Crete

Heraklion, December 2020

Dedicated to my father Eleftherios,
my mother Eleni,
and
my sister Anna-Maria.

Acknowledgments

During this amazing trip called PhD, I was fortunate to collaborate with amazing people. First of all, I am grateful to my supervisor Prof. Angelos Bilas. He gave me the space, the time, the resources, and most importantly the guidance to sharpen my technical skills as well as develop my research taste. He was always being there to discuss my ideas and concerns, providing directions and invaluable advises on my research. His mentorship and his interdisciplinary approach were a source of inspiration during my PhD studies.

I am grateful to my thesis committee members Manolis Katevenis, Polyvios Pratikakis, Kostas Magoutis, Dimitrios Nikolopoulos, Christos Kozyrakis, and George Amvrosiadis for their feedback during my defense and for their comments that helped me prepare the final version of this thesis.

Furthermore, I am grateful to my first mentor Dimitrios Nikolopoulos during his years in University of Crete. I did my first steps in research under his supervision during my B.Sc. and M.Sc. studies and I am thankful for the time and energy he put into this work.

I want to thank every single co-author I had all these years: Giorgos Saloustros, Pilar González-Férez and Giorgos Xanthakis. A big thanks also goes to Manolis Marazakis who is always willing to discuss new directions of my research and all the creative brainstorming time we spent together. We worked together, we got rejected together, we resubmitted together. I really enjoyed working with all of you.

After spending a bit more than 10 years in Computer Architecture and VLSI Systems (CARV) lab of the Institute of Computer Science (ICS) in Foundation for Research and Technology-Hellas (FORTH), I feel like I have found a second family there. I met and worked alongside great people in this lab and I want to express my deepest gratitude to Yannis Sfakianakis, Manos Pavlidakis, Stelios Mavridis, Iacovos G. Kolokasis, Nikos Papakonstantinou, Foivos Zakkak, Michalis Vardoulakis, Stella Mikrou, Nikos Batsaras, Thanos

Batzelios, Antonis Chazapis, Christos Kozanitis, and all other past and present members of CARV for preserving the balance between work and real life, making the lab a fun place to be.

I want to express my sincere thanks to Antonis Papaioannou, Vassilis Papakonstantinou, Serafeim Chrisovergis, Ilias Batzelios and all other friends for their support, love, tolerance, and all the great moments we have shared. Additionally I whould like to thank my friends from my hometown Megara, Ilias Moraitis, Akis Stamoulis, Georgios Papadopoulos, Giannis Moustrakas, Giannis Theodosiou, and Nikos Monios for their support all of these years.

I am grateful to my wife Ria Vrouva, for her patience, continuous support, and encouragement during the period of my doctoral thesis' research.

At last but not least, I would like to express my deepest and most sincere gratitude to my parents, Eleftherios and Eleni, and to my little sister Anna Maria, for their love and support throughout all these years. Without your support I would have never been able to complete the dissertation.

I would also like to thank the the Institute of Computer Science (ICS) in Foundation for Research and Technology-Hellas (FORTH), which supported me with graduate scholarships throughout my doctoral studies. Funding comes from several European and Greek funded projects that include: CoherentPaaS (FP7-ICT-611068), LeanBigData (FP7-ICT-619606), Vineyard (GA 687628), ExaNeSt (GA 671553), EVOLVE (GA 825061), and Sentitour at Scale (T1EDK-02857). I am pleased to be awarded with the Maria Michael Manasaki legacy's fellowship for the academic year 2018-2019. Finally, I am honored to receive the very competitive Facebook Graduate Fellowship 2019 in the "Compute Storage and Efficiency" research team.

Abstract

Applications typically access storage devices using read/write system calls. Additionally, they use a storage cache to reduce expensive accesses to the devices. Fast storage devices provide high sequential throughput and low access latency. Consequently, the cost of cache lookups and system calls in the I/O path becomes significant at high I/O rates.

In this dissertation, we propose the use of memory-mapped I/O to manage storage caches and remove software overheads in the case of hits. With memory-mapped I/O (i.e. *mmap*), a user can map a file in the process virtual address space and access its data using processor load/store instructions. In this case, the operating system is responsible for moving data between DRAM and the storage devices, creating/destroying memory mappings, and handling page evictions/writebacks. Hits in memory-mapped I/O are handled entirely in hardware through the virtual memory mappings.

First, we design and implement a persistent key-value store that uses memory-mapped I/O to interact with storage devices, and we show the advantages of memory-mapped I/O for hits compared to explicit lookups in the storage cache. Then we show that the Linux memory-mapped I/O path suffers from several issues in the case of data-intensive applications over fast storage devices when the dataset does not fit in memory. These include: (1) the lack of user control for evictions of I/Os, especially in the case of writes, (2) scalability issues with increasing the number of threads, and (3) the high cost of page faults that happen in the common path for misses.

Next, we propose techniques to deal with these shortcomings. We propose a mechanism that handles evictions in memory-mapped I/O based on application needs. To show the applicability of this mechanism, we build an efficient memory-mapped I/O persistent key-value store that uses this mechanism. Subsequently, we remove all centralized contention points and provide scalable performance with increasing I/O concurrency

and number of threads. Finally, we separate protection and common operations in the memory-mapped I/O path. We leverage CPU virtualization extensions to reduce the overhead of page faults and maintain the protection semantics of the OS.

We evaluate the proposed extensions using mainly persistent key-value stores that are a central component for many analytics processing frameworks and data serving systems. We show significant benefits in terms of CPU consumption, performance (throughput and average latency), and predictability (tail latency).

Keywords: Memory-Mapped I/O, Fast Storage, Key-Value Store, *mmap*

Supervisor: Angelos Bilas

Professor

Computer Science Department

University of Crete

Περίληψη

Οι εφαρμογές συνήθως προσπελαύνουν τις συσκευές αποθήκευσης χρησιμοποιώντας κλήσεις συστήματος (system calls) για ανάγνωση και εγγραφή. Επιπλέον, χρησιμοποιούν μια κρυφή μνήμη για να μειώσουν τις ακριβές προσπελάσεις στις συσκευές αποθήκευσης. Ωστόσο, συσκευές αποθήκευσης υψηλής ταχύτητας, παρέχουν πλέον πρόσβαση στα δεδομένα σε χαμηλό χρόνο. Κατά συνέπεια, το κόστος των αναζητήσεων στην κρυφή μνήμη (που γίνετε σε λογισμικό) αλλά και των κλήσεων συστήματος γίνεται σημαντικό υπό αυτές τις συνθήκες.

Σε αυτή τη διατριβή, προτείνουμε την διαχείριση της κρυφής μνήμης που χρησιμοποιείται για είσοδο/έξοδο (E/E) μέσω απεικόνισης των συσκευών στη μνήμη (memory mapped I/O), με στόχο την εξάλειψη του κόστους στις περιπτώσεις ευστοχίας (hits) στην κρυφή μνήμη. Με την απεικόνιση των συσκευών αποθήκευσης στη μνήμη (Linux mmap), ο χρήστης μπορεί να απεικονίσει ένα αρχείο στον χώρο των εικονικών διευθύνσεων μιας διεργασίας και να αποκτήσει πρόσβαση στα δεδομένα του χρησιμοποιώντας τις εντολές φόρτωσης και εγγραφής (load/store) του επεξεργαστή. Σε αυτήν την περίπτωση, το λειτουργικό σύστημα είναι υπεύθυνο για τη μεταφορά δεδομένων μεταξύ της κυρίας μνήμης και των συσκευών αποθήκευσης, τη δημιουργία/καταστροφή αντιστοιχίσεων εικονικής με φυσική μνήμη και τον χειρισμό της απόρριψης και εγγραφής σελίδων της κρυφής μνήμης στις συσκευές αποθήκευσης. Επομένως η διαχείριση των προσβάσεων στην κρυφή μνήμη που είναι εύστοχες (hits) γίνεται εξ ολοκλήρου από το υλικό μέσω του μηχανισμού για την μετάφραση εικονικών διευθύνσεων.

Αρχικά, σχεδιάζουμε και υλοποιούμε ένα σύστημα αποθήκευσης ζευγαριών κλειδιού-τιμής που χρησιμοποιεί την απεικόνιση στη μνήμη για την διαχείριση της κρυφής μνήμης EE και για να αλληλεπιδρά με τις συσκευές αποθήκευσης. Στην συνέχεια, παρουσιάζουμε τα πλεονεκτήματα του σε σύγκριση με τις αναζητήσεις στην κρυφή μνήμη που είναι υλοποιημένη σε λογισμικό. Δείχνουμε ότι το μονοπάτι στο λειτουργικό σύστημα Linux για την απεικόνιση στη μνήμη των συσκευές αποθήκευσης έχει πολλά προβλήματα στην περίπτωση εφαρμογών με μεγάλες ανάγκες

σε πρόσβαση δεδομένων πάνω από συσκευές γρήγορης αποθήκευσης, όταν το σύνολο των δεδομένων τους δεν χωρά στη κύρια μνήμη. Σε αυτά περιλαμβάνονται: (1) η έλλειψη ελέγχου για την απόρριψη σελίδων, ειδικά κατά την περίπτωση των εγγράφων, (2) η μη επαρκής κλιμάκωση σε συνάρτηση με την αύξηση του αριθμού των νημάτων και (3) το υψηλό κόστος των σφαλμάτων σελίδας που συμβαίνουν κατά την διάρκεια των αστοχιών στην κρυφή μνήμη.

Κατόπιν, προτείνουμε τεχνικές για την αντιμετώπιση αυτών των μειονεκτημάτων. Προτείνουμε έναν μηχανισμό που χειρίζεται την απόρριψη και αντικατάσταση σελίδων μνήμης με βάση τις ανάγκες της εφαρμογής. Για να δείξουμε τη δυνατότητα εφαρμογής του, σχεδιάζουμε και υλοποιούμε ένα σύστημα αποθήκευσης ζευγαριών κλειδιού-τιμής που χρησιμοποιεί αυτόν τον μηχανισμό. Στη συνέχεια, καταργούμε όλα τα κεντρικά σημεία συνωστισμού κατά τον συγχρονισμό στο μονοπάτι της πραγματοποίησης E/E μέσω απεικόνισης στη μνήμη. Ο σχεδιασμός μας παρέχει κλιμακώσιμη απόδοση με τις συσκευές αποθήκευσης καθώς αυξάνεται ο αριθμός των πυρήνων και νημάτων στους εξυπηρετητές. Τέλος, διαχωρίζουμε την προστασία και τις κοινές λειτουργίες στο μονοπάτι κατά την πρόσβαση σε γρήγορες συσκευές αποθήκευσης μέσω απεικόνισης στη μνήμη. Αξιοποιούμε επεκτάσεις εικονικοποίησης (virtualization extensions) του επεξεργαστή για να μειώσουμε το κόστος των σφάλματων σελίδας (page faults) και να διατηρήσουμε την ισχυρή προστασία μεταξύ χρηστών που παρέχει το λειτουργικό σύστημα.

Αξιολογούμε τις προτεινόμενες επεκτάσεις χρησιμοποιώντας κυρίως συστήματα αποθήκευσης ζευγαριών κλειδιού-τιμής που αποτελούν σημαντικό κομμάτι για πολλά συστήματα επεξεργασίας και εξυπηρέτησης δεδομένων και δείχνουμε σημαντικά οφέλη όσον αφορά την κατανάλωση σε κύκλους του επεξεργαστή, την απόδοση και την προβλεψιμότητα.

Λέξεις κλειδία: Απεικόνιση συσκευών στη μνήμη, Συσκευές αποθήκευσης υψηλής ταχύτητας, Σύστημα αποθήκευσης ζευγαριών κλειδιού-τιμής

Επόπτης: Άγγελος Μπίλας
Καθηγητής
Τμήμα Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

Bibliographic Notes

The publications related to this dissertation (ordered by date) are:

(i) Anastasios Papagiannis, Giorgos Saloustros, Pilar Gonzalez-Ferez, and Angelos Bilas. 2016. *Tucana: design and implementation of a fast and efficient scale-up key-value store*. In Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference **(USENIX ATC '16)**. USENIX Association, USA, 537 – 550.

(ii) Anastasios Papagiannis, Giorgos Saloustros, Pilar Gonzalez-Ferez, and Angelos Bilas. 2018. *An Efficient Memory-Mapped Key-Value Store for Flash Storage*. In Proceedings of the ACM Symposium on Cloud Computing **(ACM SoCC '18)**. Association for Computing Machinery, New York, NY, USA, 490 – 502.

(iii) Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. 2020. *Optimizing Memory-mapped I/O for Fast Storage Devices*. In Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference **(USENIX ATC '20)**. USENIX Association, USA

(iv) Anastasios Papagiannis, Giorgos Saloustros, Giorgos Xanthakis, Giorgos Kalaentzis, Pilar Gonzalez-Ferez, and Angelos Bilas. 2020. *Kreon: An Efficient Memory-Mapped Key-Value Store for Flash Storage*. ACM Transactions on Storage **(ACM TOS)** ***(accepted)***

(v) Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. 2020. *Memory-Mapped I/O on Steroids*. ***(under review)***

More specifically, Chapter 2 is based on *(i)*. Chapter 3 based on *(ii)* & *(iv)*, Chapter 4 based on *(iii)*, and finally Chapter 5 based on *(v)*.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Flash-based storage devices provide high sequential throughput, high random IOPS, and low access latency. As a result, they introduce new opportunities by narrowing the gap between random and sequential throughput, especially at higher queue depths (number of concurrent I/Os). Despite these technology trends, modern data-intensive applications do not see the full potential of fast storage devices. Consequently, as datasets grow, the I/O path is becoming a significant bottleneck in terms of overhead (CPU cycles) and scalability with the number of cores. Ideally, modern and future servers should consume precious CPU cycles for performing application processing and not I/O to and from devices. Today, we are far from this ideal situation.

Given the performance gap between memory and storage, the I/O path uses a storage cache to leverage locality and reduce the number of accesses to devices. A data access results in a lookup and potential updates in the cache eviction metadata. Even in the case of hits, cache lookups result in high CPU overhead spend for cache management. Authors in [58] claim that about one-third of the total CPU cycles of a database system running OLTP workloads is spent in managing the user-space cache when the dataset fits in memory. In the case of misses, cache replacements move data between memory and the storage devices. Typically these operations use synchronous or asynchronous read and write system calls.

In this dissertation, we propose the use of memory-mapped I/O to manage storage caches and remove software overheads in the case of hits. Hits in memory-mapped I/O

are handled via virtual memory mappings and do not incur any software overhead.

In memory-mapped I/O (i.e. mmap), the application maps a storage device (or a file) in the process virtual address space and the user can access it using processor load and store instructions. In the case of a hit, a valid mapping in the page table already exists. The virtual to physical translation is handled entirely in hardware. In the case of a miss, a page fault happens, which is responsible for adding a new translation in the page table. In this case, the kernel is responsible for reading data from the devices and evicting dirty data when the memory is not enough, when a specific (configurable) amount of time has passed, or when an application explicitly asks for synchronization between memory and the devices. In memory-mapped I/O, the data transfer unit between the devices and memory depends on the hardware-defined page size. Given that the regular page size is $4KB$, performing $4KB$ I/Os in the case of HDDs result in drammatic reduction in the peak device throughput. Fast storage devices address this concern for small I/Os that occur during page faults. However, even in this case, memory-mapped I/O has several shortcomings.

This dissertation addresses limitations of memory-mapped I/O for data-intensive applications over fast storage devices. We show that the use of memory-mapped I/O provides significant improvements compared to read/write system calls and user-space caching.

First, we examine the potential benefit of our approach in persistent key-value stores. Persistent key-value stores are a central component for analytics processing frameworks and data serving systems [41, 5, 30, 49, 54, 36]. We design and implement a persistent key-value store that uses memory-mapped I/O to interact with storage devices. We apply specific optimizations to show the benefits of memory-mapped I/O. Our evaluation shows that hits, which are handled entirely in hardware, provide significant performance improvements compared to a traditional user-space cache and read/write system calls. On the other hand, we also identify several issues of memory-mapped I/O in Linux where the dataset does not fit in memory and, more importantly, in the case of writes, where the user cannot control the timing of I/Os, resulting in high tail latencies.

To overcome the lack of application control in the case where the dataset does not fit in memory, we design and implement a custom memory-mapped I/O path in Linux that handles evictions based on application needs. Our approach uses a priority-based

FIFO replacement policy, and during memory pressure, a page with a higher priority is preferred for eviction. To show the applicability of our approach, we use our mechanism in a persistent key-value store designed to take advantage of memory-mapped I/O.

Next, we show that Linux memory-mapped I/O does not scale beyond eight threads. This is a significant limitation for modern multicore servers. To overcome this limitation, we (a) separate clean and dirty-trees to avoid all centralized contention points, (b) use full reverse mappings instead of Linux object-based reverse mappings to reduce CPU processing, and (c) introduce a scalable DRAM cache with per-core data structures to reduce latency variability. Our design achieves both higher scalability and higher I/O concurrency by (1) avoiding all centralized contention points that limit scalability, (2) reducing the amount of CPU processing in the common path, and (3) using dedicated data-structures to minimize interference among processes, thus improving tail latency.

Finally, we tackle miss overheads in memory-mapped I/O managed caches. Memory-mapped I/O introduces page faults instead of system calls in the case of misses. However, page faults incur high overhead. We observe that the main operations of memory-mapped I/O occur at different frequencies: virtual memory management and device access are common path operations. On the other hand, dynamic cache resizing and virtual address range management are uncommon path operations. Based on this observation, we design and implement an efficient library operating system for storage applications that places the application in a virtual machine context. This approach eliminates the cost of protection domain crossings required during page faults. We leverage hardware virtualization extensions to provide full protection semantics while providing the full memory-mapped I/O functionality.

Our approach results in significant performance improvements in terms of performance (throughput and latency), predictability (tail latency), and CPU consumption.

Thesis statement: We propose the use of memory-mapped I/O to manage storage caching and remove software overheads in the case of hits. Also, we provide techniques to overcome issues in misses and make memory-mapped I/O practical.

1.1 Leveraging Memory-mapped I/O

This dissertation contributes on optimizing the memory-mapped I/O path. To take advantage of our contributions, applications should be re-designed to use memory-mapped I/O instead of a user-space cache and system calls. Our work on persistent key-value stores provides general principles to build an efficient system using memory-mapped I/O. These include: (i) a common data layout for in-memory and persistent representation, (ii) a common allocator for memory and storage, and (iii) the use of Copy-on-Write (CoW) for persistence.

A fundamental design decision on applications that require persistence is the data layout both in memory and device. Today, it is common to have different data representation for memory and devices and translate between them using serialization and deserialization. With memory-mapped I/O an application can map directly parts of the file/device to the user's virtual addresses. This enables data addressing with simple pointer arithmetic operations. Based on this, applications can use the same data layout for both in-memory and persistent representation. Furthermore, applications commonly use different memory and storage allocators due to the different data representation. For the latter, users generally rely on file systems. Our persistent key-value stores, use a common allocator for both memory and persistent data. Both of these techniques reduce by a large factor the number of memory copies, and this results in more CPU cycles and higher memory bandwidth available for user processing. Finally, our work on persistent key-value stores, show that CoW fits well with memory-mapped I/O and it also reduces I/O amplification in the common path. In the next chapters, we provide more details and reasoning about these general principles to design a system that leverages memory-mapped I/O for storage cache management.

1.2 Contributions

The specific contributions of this dissertation are:

1. We apply specific optimizations in a memory-mapped key-value store to show the

benefits and deal with the drawbacks of memory-mapped I/O. We show that hits, which are handled entirely in hardware, provide significant performance improvements compared to a traditional user-space cache and read/write system calls. On the other hand, we identify the issues of memory-mapped I/O in Linux, especially when the dataset does not fit in memory.

2. We provide a mechanism that handles evictions in memory-mapped I/O based on application needs. We use a priority-based FIFO replacement policy, and during memory pressure, a page with a higher priority is preferred for eviction. We show the applicability of this mechanism on a persistent key-value store that uses memory-mapped I/O to interact with storage devices.

3. We show that Linux memory-mapped I/O path fails to scale with increasing the number of concurrent I/Os and application threads. To overcome this issue, we (a) separate clean and dirty-trees to avoid all centralized contention points, (b) use full reverse mappings instead of Linux object-based reverse mappings to reduce CPU processing, and (c) introduce a scalable DRAM cache with per-core data structures to reduce latency variability. Our approach provides scalable performance with large numbers of threads and a large number of concurrent I/Os to the devices, which is essential to achieve peak device throughput.

4. We show that a page fault has higher overhead than a system call for the same I/O size. The increased cost can lead to performance degradation in the case of misses. We propose running the application in a privileged domain to reduce page fault cost, similar to where the guest OS runs in virtual machines. We provide full protection semantics by leveraging hardware virtualization extensions. This approach eliminates the cost of protection domain crossings required during page faults. A privileged domain provides direct access to virtual memory hardware, including the page table and TLB that are necessary to handle page faults. Our approach also provides the full memory-mapped I/O functionality.

1.3 Organization

The rest of this dissertation is organized as follows. Chapter 2 presents a *m*emory-mapped I/O use case, in the context of a persistent write-optimized key-value named *Tucana*. Chapter 3 presents a mechanism for user applications to define hot and cold data and affect the eviction policy. We use the *Kreon* key-value store to evaluate our approach. Chapter 4 presents a scalable *m*emory-mapped I/O path inside the Linux kernel named *FastMap*. Chapter 5 presents a design, *Aquila*, to reduce page fault cost in the common path. Chapter 6 reviews related work. Finally, Chapter 7 presents directions for future work and Chapter 8 concludes this thesis.

Chapter 2

An Efficient Memory-Mapped I/O Key-Value Store

Our goal in this chapter is to draw a different balance between device and CPU efficiency. We start from a B^ϵ–tree [22] approach to maintain the desired asymptotic properties for inserts, which is important for write-intensive workloads. B^ϵ–trees achieve this amortization by buffering writes at each level of the tree. In our case, we assume that the largest part of the tree (but not the data items) fit in memory and we only perform buffering and batching at the lowest part of the tree. Then, we develop a design that manages variable size keys and values, deals with persistence, and stores data directly on raw devices.

Although we still use the buffering technique of B^ϵ–trees to amortize I/Os, we take a different stance with respect to randomness of I/Os. Unlike LSM-trees [102], we do not make an effort to generate large I/Os. LSM-trees produce large I/Os by maintaining large sorted containers of data items in memory, which can then be read or written as a whole. These large sorted containers are maintained via a compaction technique that relies on sorting and merging smaller pieces. Although this approach has proven extremely effective for HDDs, it results in high CPU overheads and I/O amplification, as we show in our evaluation for LSM-trees.

We design a full featured key-value store, *Tucana*, that achieves lower host CPU overhead per operation than other state-of-the-art systems. *Tucana* provides persistence and recovery from failures, arbitrary dataset sizes, variable key and value sizes, concurrency,

multithreading, and versioning. We use copy-on-write (CoW) to achieve recovery without the use of a log, we directly map the storage device to memory to reduce space (memory and device) allocation overhead, and we organize internal and leaf nodes similar to traditional approaches [29] to reduce CPU overhead for lookup operations.

To evaluate our approach, we first compare with RocksDB, a state-of-the-art key-value store. Our results show that *Tucana* is up to $9.2\times$ better in terms of cycles/op and between $1.1\times$ *to* $7\times$ in terms of ops/s, across all workloads. This validates our hypothesis that randomness is less important for SSD devices, when there is an adequate degree of concurrency and relatively small I/O requests.

To examine the impact of our approach in the context of real systems, we use *Tucana* to improve the throughput and efficiency of HBase [5], a popular scale-out NoSQL store. We replace the LSM-based storage engine of HBase with *Tucana*. Data lookup, insert, delete, scan, and key-range split and merge operations are served from *Tucana*, while maintaining the HBase mapping of tables to key-value pairs, client API, client-server protocol, and management operations (failure handling and load balancing). The resulting system, H-*Tucana*, remains compatible with other components of the Hadoop ecosystem. We compare H-*Tucana* to HBase and Cassandra using YCSB and we find that, compared to HBase, H-*Tucana* achieves between $2-8\times$ better CPU cycles/op and $2-10\times$ higher operation rates across all workloads. Compared to Cassandra, H-*Tucana* achieves even higher improvements.

Our specific contributions in this work are:

- The design and implementation of a key-value data store that draws a different balance between device behavior and host overheads.
- Practical B^ϵ–tree extensions that leverage mmap-based allocation, copy-on-write, and append-only logs to reduce allocation overheads.
- An evaluation of existing, state-of-the-art, persistent key-value stores and a comparison with *Tucana*, as well as an improved implementation of HBase.

The rest of this chapter is organized as follows: Section 2.1 provides an overview of persistent data structures. Section 2.2 describes our design. Section 2.3 presents our evalua-

tion methodology and our experimental analysis. Finally, Section 2.4 concludes the chapter.

2.1 Background

In this work, we use as a basis a variant of B-trees, broadly called B$^\epsilon$–trees [22]. B$^\epsilon$–trees are B-trees with an additional per-node buffer. By using these buffers, they are able to batch insert operations to amortize their cost. In B$^\epsilon$–trees the total size of each node is B and ϵ is a design-time constant between [0,1]. ϵ is the ratio of B that is used for buffering, whereas the rest of the space in each node (1-ϵ) is used for storing pivots.

Buffers contain messages that describe operations that modify the index (insert, update, delete). Each such operation is initially added to the tree's root node buffer. When the root node buffer becomes full, the structure uses the root pivots to propagate a subset of the buffered operations to the buffers of the appropriate nodes at the next level. This procedure is repeated until operations reach a leaf node, where the key-value pair is simply added to the leaf. Leaf nodes are similar to B-Trees and they do not contain an additional buffer, beyond the space required to store the key-value pairs. The cost of an insertion in terms of I/Os is $O(\frac{\log_B N}{\epsilon B^{1-\epsilon}})$, where a regular B-Tree has $O(\log_B N)$ [22, 67].

A get operation is similar to the B-Tree data structure. It traverses the path from the root to the corresponding leaf. This results in similar complexity to B-trees, regarding I/O operations. The main difference is that in a B$^\epsilon$–tree we also need to search the buffers of the internal nodes along the path. A range scan is similar to a get, except that messages for the entire range of keys must be checked and applied as the appropriate subtree is traversed. Therefore, buffers are frequently modified and searched. For this reason, they are typically implemented with tree indexes rather than sorted containers.

Next, we present the design of *Tucana*, a key-value store that aims to significantly improve the efficiency of data access.

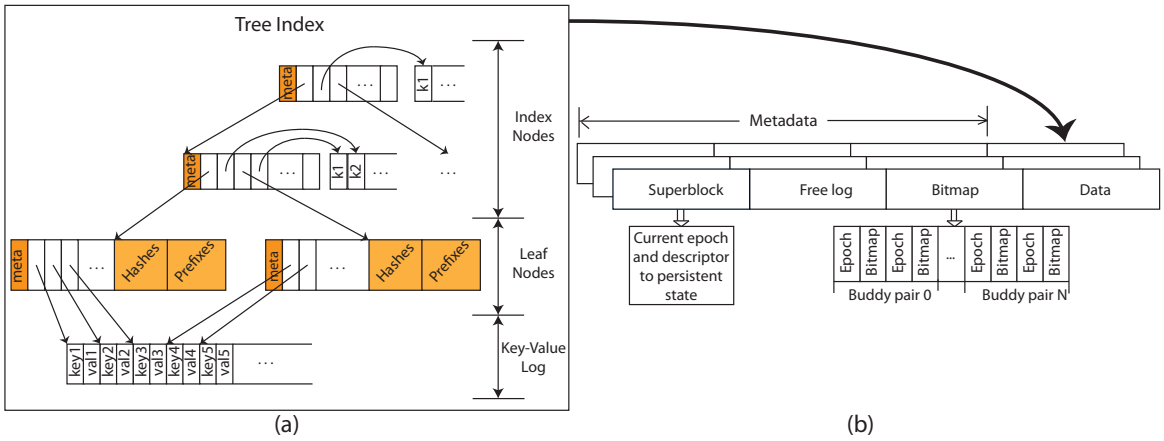


Figure 2.1: The top-level design of *Tucana*. The left part (a) of the figure shows the tree index. The right part (b) shows the volume layout.

2.2 Design

Figure 2.1 shows an overview of *Tucana*. More specifically, Figure 2.1a shows the index organization, which uses B$^\epsilon$–trees as a starting point (Section 2.2.1). In Figure 2.1b we depict *Tucana*'s approach for allocation and persistence, which we discuss in Sections 2.2.2 and 2.2.3, respectively.

2.2.1 Tree Index

Figure 2.2 shows the differences between *Tucana* and a B$^\epsilon$–tree. On the left side of the figure we show a B$^\epsilon$–tree, which we explain in Section 2.1. On the right side of the figure we show *Tucana*, where we distinguish nodes that fit in main memory from those that do not. To improve host-level efficiency (in terms of cycles/op), *Tucana* limits buffering and batching to the lowest part of the tree. In many cases today, the largest part of the index structure (but not the actual data) fits in main memory (DRAM today and byte-addressable NVM in the future) and therefore, we do not buffer inserts in intermediate nodes. *Tucana* design provides desirable asymptotic properties for random inserts, where a single I/O is amortized over multiple insert operations.

Figure 2.1a shows the index organization in *Tucana*. The index consists of internal

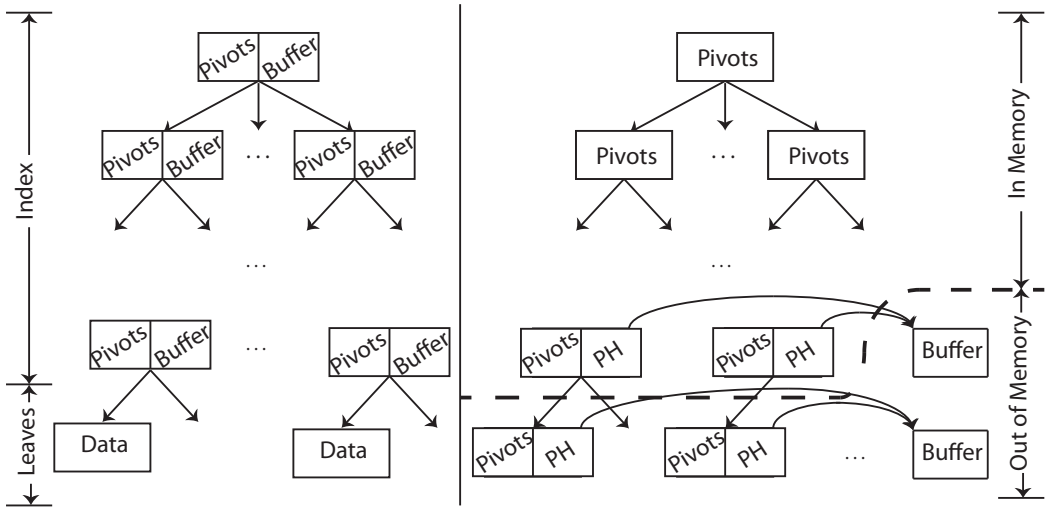


Figure 2.2: Comparison of B$^\epsilon$–tree (left) and *Tucana* (right). In *Tucana* we distinguish the part of the tree that fits in memory above the dashed line and the rest that does not. PH stands for Prefix-Hash.

nodes with pointers to next level nodes and pointers to variable size keys (pivots). We use a separate space per internal node to store the variable size keys themselves. Pointers to keys are sorted based on the key, whereas keys are appended to the buffer. The leaf nodes contain sorted pointers to the key-value pairs. We use a single append-only log to store both the key and values. The design of the log is similar to the internal buffers of B$^\epsilon$–trees.

Insert operations traverse the index in a top down fashion. At each index node, we perform a binary search over the pivots to find the next level node to visit. When we reach the leaf, we append the key-value pair to the log and we insert the pointer in the leaf, keeping pointers sorted by the corresponding key. Then, we complete the operation. Compared to B$^\epsilon$–trees we avoid the buffering at intermediate nodes. If a leaf is full, we trigger a split operation prior to insert. Split operations, in index or leaf nodes, produce two new nodes each containing half of the keys and they update the index in a bottom-up fashion. Delete operations place a tombstone for the respective keys, which are removed later. Deletes will eventually cause rebalancing and merging [13].

Point queries traverse the index similar to inserts to locate the appropriate leaf. At the leaf, we perform a binary search to locate the pointer to the key-value pair. Since there are no intermediate buffers as in B$^\epsilon$–trees, we do not need to perform searches in the

intermediate levels. Finally, range queries locate the starting key similar to point queries and subsequently use the index to iterate over the key range. It is important to notice that in contrast to B^{ϵ}–trees we do not need to flush all the intermediate buffers prior to a scan operation.

We note that binary search in the leaf nodes and index nodes is a dominant function used by all operations. To reduce memory footprint for metadata, *Tucana* does not store keys in leaves. This means that keys during binary search need to be retrieved from the device. To avoid this, *Tucana* uses two optimizations, prefixes and hashes.

We store as metadata, a fixed-size prefix for each key in the leaf block. Binary search is performed using these prefixes, except when they result in ambiguity, in which case the entire key is fetched from the log. Prefixes improve performance of inserts, point queries, and range queries. In our tuning of prefixes we find that for different types of keys, prefixes eliminate 65%–75% of I/Os during binary search in leaves.

Additionally, a hash value for each key is stored in the leaf nodes. Hashes help with point queries. For a point query we first do a binary search over prefixes. If this results in a tie, then we linearly examine the corresponding (so not all) hashes. We use Jenkins hash function (one-at-a-time) [69] to produce 4-byte hashes. Then the key is read to ensure there is no collision. In our experiments we find that hashes identify the correct key-value pair in more that the 98% of the cases.

Complexity analysis on the memory footprint and dataset to DRAM ratio are out of the scope of this dissertation and more detail can be found at [105].

2.2.2 Device layout and access

Figure 2.1b depicts the data layout in *Tucana*. *Tucana* manages a set of contiguous segments of space to store data. Each segment can be a range of blocks on a physical, logical, virtual block device, or a file. To reduce overhead, segments should be allocated directly on virtual block devices, without the use of a file system. Our measurements show that using XFS as the file system results in a 5-10% reduction in throughput compared to using a virtual block device directly without any file system.

Each segment is composed of a metadata portion and a data portion. The metadata portion contains the superblock, the free log, and the segment allocator metadata (bitmap). The superblock contains a reference to a descriptor of the latest persistent and consistent state for a segment. Modifying the superblock commits the new state for the segment. Each segment has a single allocator common for all databases (key ranges) in a segment. The data portion contains multiple databases. Each database is contained within a single segment and uses its own separate indexing structure.

The allocator keeps persistent state about allocated blocks of a configurable size, typically set to 4 KB, and multiples of it. For this purpose, it uses bitmaps because in key-value stores allocations can be in the order of KBs, as opposed to filesystems that typically do larger allocations. Moreover, allocator bitmaps are accessed directly via an offset and at low overhead, while for searches there are efficient bit parallel techniques [23]. It also maintains state about free operations and performs them lazily in a log structure named *Free log*.

In all persistent key-value stores, including *Tucana*, the index includes pointers to data items in the storage address space. During system operation, part of the index and data are cached in memory. When traversing the index to serve an operation, there is a need to translate storage pointers to pointers in memory. This leads to frequent cache lookups that cannot be avoided easily. Essentially, the cache serves as a mechanism to translate pointers from the storage to the memory address space. Previous work [58] indicates that when all data and metadata fit in memory, managing this cache requires about one-third of the index CPU cycles.

Most key-value stores today follow this caching approach [54, 49, 5, 82, 101]. This allows the key-value store to also control the size and timing of I/O operations between the memory cache and the storage devices, as well as the cache policy.

Instead, *Tucana* uses an alternative approach based on `mmap`. `mmap` uses a single address space for both memory and storage and virtual memory protection to determine the location (memory or storage) of an item. This eliminates the need for pointer translation at the expense of page faults. We note that pointer translation occurs during index operations regardless of whether items are in memory or not, whereas page faults occur

only when items are not in memory. The use of `mmap` also allows *Tucana* to use a single allocator for memory and device space management. Additionally, `mmap` eliminates data copies between kernel and user space.

The use of `mmap` has three drawbacks. First, each write operation of variable size is converted to a read-modify-write operation, increasing the amount of I/O. In our design, due to the copy-on-write persistence (see Section 2.2.3), all writes modify eventually the full page and there can be no reads to unwritten parts of a page. Therefore, we use a simple filter block device in the kernel, which filters read-before-write operations and merely returns a page of zeros. Write and read-after-write operations are not filtered and are forwarded to the actual device. The filter module uses a simple, in-memory bitmap and is initialized and updated by *Tucana* via a set of ioctls. The size of the in-memory bitmap is proportional to the block device size (for 1 TB of storage we need 32 MB of memory).

Second, `mmap` results in the loss of control over the size and timing of I/O operations. `mmap` generates page-sized I/Os (4 KB). To mitigate the impact of small I/Os we use `madvise` to instruct `mmap` to generate larger I/Os. To control their timing we use `msync` for specific items and memory ranges during commit operation.

Third, `mmap` introduces page faults for fetching data. The number of page faults depends on `mmap` kernel page eviction policy. *Tucana* would benefit from custom eviction policies that keep the index and the tail of the append log in memory. In this work, we do not make an attempt to control these policies. However, future work should examine this issue in more detail.

2.2.3 Copy-on-write persistence

Tucana uses a Copy-on-Write (CoW) approach for persistence instead of a Write-Ahead-Log (WAL). WAL produces sequential write I/Os at the expense of doubling the amount of writes (in the log and later in place). CoW performs only the necessary writes, however, it generates a more random I/O pattern. Therefore, although a WAL is more appropriate for HDDs, CoW has more potential for fast devices. The use of CoW is also motivated by three additional reasons; (a) It is amenable to supporting versioning. (b) It allows instantaneous

recovery, without the need to redo or undo a log. (c) It helps increase concurrency by avoiding lock synchronization for different versions of each data item [93], as we discuss in the next subsection.

The state of a segment consists of the allocator, tree metadata, and buffers. CoW is used to maintain the consistency of both allocator and tree metadata. The bitmap in each segment is organized in *buddy pairs*, as shown in Figure 2.1b. Each *buddy pair* consists of two 4 KB blocks that contain information about allocated space. Each buddy is marked with a global per segment increasing counter named ***epoch***. The epoch field is incremented after a successful commit operation and denotes the latest epoch in which the buddy was modified. At any given point only one buddy of the pair is active for write operations, whereas the other buddy is immutable for recovery. Commits persist and update modified buddy pairs.

The allocator defers free operations with the use of the free log [20]. Directly applying a free operation that could be rolled back in the presence of failures is more complicated as it can corrupt persistent state. We log free operations using their epoch id, and we perform them later after their epoch becomes persistent.

To maintain the consistency of the tree structure during updates, each internal index and leaf node uses epochs to distinguish its latest persistent state. During an update, the node's epoch indicates whether a node is immutable, in which case a CoW operation takes place. After a CoW operation for inserting a key, the parent of the node is updated with the new node location in a bottom-up fashion. The resulting node belongs to epoch+1 and will be persisted during the next commit. Subsequent updates to the same node before the next commit are batched by applying them in place. Since we store keys and values in buffers in an append-only fashion, we need to only perform CoW on the header of each internal node.

Tucana's persistence relies on the atomic transition between consistent states for each segment. Metadata and data in *Tucana* are written asynchronously to the devices. However, transitions from state to state occur atomically via synchronous updates to the segment's superblock with `msync` (commits). Each commit creates a new persistent state for the segment, identified by a unique epoch id. The epoch of the latest persistent state of a

segment is stored in a descriptor to which the superblock keeps a reference.

Commits can take place in parallel with read and write operations. To achieve this, a commit is performed in two steps: (1) Initially, it marks the current state as persistent by increasing the epoch of the system. This state includes the bitmap and the tree indexes for this segment. (2) It flushes the state of the segment to the device. In case of a failure during a commit, the segment simply rolls back to the latest persistent state by ignoring any writes that have reached the device but were not committed via the metadata epoch states.

During a commit operation, the bitmap cannot be modified by new allocations (a subset of the write operations) because this may change the state on the device (`mmap` may propagate any write from memory to the device asynchronously). In case the current commit fails, then both *buddy pairs* will be inconsistent. To avoid this, allocations during a commit are buffered in a temporary location in memory and are applied at the end of the commit.

2.2.4 Concurrency in *Tucana*

Concurrency in key-value stores is important for scaling up as server density increases in terms of CPU, storage, and network throughput. Key-value stores typically operate under high degrees of concurrency, due to the large numbers of client requests.

Similar to most key-value stores, *Tucana* partitions datasets in multiple databases (key ranges). Requests in different ranges can be served without any synchronization. The only exception in *Tucana* is insert operations in different regions that are stored in the same segment. In this case the existence of a single segment allocator requires synchronization across ranges during allocation operations. To reduce the impact of such synchronization, the allocator operates in a batched mode, where a request reserves more space than required for the current operation. Subsequent inserts to the same database do not need to request space from the allocator.

Within each range, *Tucana* allows any number of concurrent reads and a single write without synchronization. To achieve this, *Tucana* uses the versions of the segment cre-

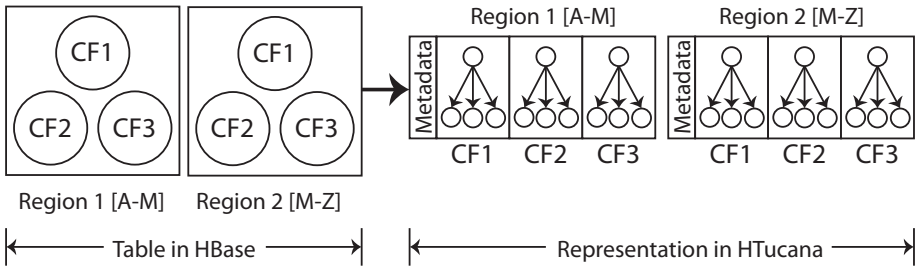


Figure 2.3: Table storage in HBase and H-*Tucana*. CF stands for column family.

ated through commits, similar to read-copy-update synchronization [95]. In particular, we serve read operations from the latest persistent version of the segment, which is immutable. Writes on the other hand are served from the modified root which contains all modifications.

Updates applied by an application are visible to readers after a commit. *Tucana*'s API offers additional fence operations to allow higher layers to control when updates become visible.

Finally, in the current state of the prototype, *Tucana* does not allow multiple concurrent writes in the same range. Although there are possible optimizations, especially to allow non-conflicting writes via copy-on-write, or dynamic partitioning of the key-space, we leave these for future work.

2.2.5 H-*Tucana*

HBase [5] is a scale-out columnar store which supports a small and volatile schema. HBase offers a table abstraction over the data, where each table keeps a set of key-value pairs. Each table is further decomposed into regions, where each region stores a contiguous segment of the key space. Each region is physically organized as a set of files per column, as shown in Figure 2.3.

At its core HBase uses an LSM-tree to store data [102]. We use *Tucana* to replace this storage engine, while maintaining the HBase metadata architecture, node fault tolerance, data distribution and load balancing mechanisms. The resulting system, H-*Tucana*, maps HBase regions to segments (Figure 2.3), while each column maps to a separate tree in the

segment. In our work, and to eliminate the need for using HDFS under HBase, we modify HBase so that a new node handles a segment after a failure. We assume that segments are allocated over a reliable shared block device, such as a storage area network (SAN) or virtual SAN [97, 132] and are visible to all nodes in the system. In this model, the only function that HDFS offers is space allocation. *Tucana* is designed to manage space directly on top of raw devices, therefore, it does not require a file system. H-*Tucana* assumes the responsibility of elastic data indexing, while the shared storage system provides a reliable (replicated) block-based storage pool.

2.3 Experimental Analysis

In this section we evaluate Tucana against RocksDB [49] and H-*Tucana* to HBase [5] and Cassandra [82]. Our goal is to examine the following aspects of *Tucana*:

1. How does *Tucana* compares to RocksDB in terms of efficiency and absolute performance and where does these improvements come from?

2. What is the impact of *Tucana* in NoSQL stores?

Tucana and RocksDB support similar features including persistence and recovery, arbitrary size keys and values and versions. In the same category there are other popular key-value stores, such as LevelDB, KyotoDB, BerkeleyDB, and PerconaFT (based on Fractal Index Trees). In our experiments we find that RocksDB outperforms all of them [52, 55] and therefore, we present only the comparison between *Tucana* and RocksDB. HBase and Cassandra are NoSQL databases that are widely used as a back-end for high throughput systems. HBase and Cassandra use LSM-trees [102].

2.3.1 Methodology

Our experimental platform consists of two systems (client and server) each with two quad-core Intel(R) Xeon(R) E5520 CPUs running at 2.7 GHz. The server is equipped with 48 GB DDR-III DRAM, and the client with 12 GB. Both nodes are connected with a 10 Gbits/s

	Workload
A	50% reads, 50% updates
B	95% reads, 5% updates
C	100% reads
D	95% reads, 5% inserts
E	95% scans, 5% inserts
F	50% reads, 50% read-modify-write

Table 2.1: Workloads evaluated with YCSB. All workloads use a query popularity that follows a Zipf distribution except for D that follows a latest distribution.

network link. As storage devices, the server uses four Intel X25-E SSDs (32 GB) and we make a RAID-0 with them using the standard *md* Linux driver. *Tucana* is implemented in C and can be accessed from applications as a shared library. H-*Tucana* is cross-linked between the Java code of HBase and the C code of *Tucana*.

We use the open-source Yahoo Cloud Serving Benchmark (YCSB) [37] to generate synthetic workloads. The default YCSB implementation executes gets as range queries and therefore, exercises only scan operations. For this reason, we modify YCSB to use point queries for get operations. Range queries are still exercised in Workload E, which uses scan operations.

When comparing RocksDB and *Tucana* we use a low-overhead C++ version of YCSB-C [37, 114]. The original Java YCSB benchmark requires JNI to run with RocksDB and *Tucana*, which are written in C++ and C respectively, incurring high overheads.

In all cases, we run the standard workloads proposed by YCSB with the default values. Table 2.1 summarizes these workloads. We run the following sequence proposed by the YCSB author: Load the database using workload A's configuration file, run workloads A, B, C, F, and D in a row, delete the whole database, reload the database with workload E's configuration file, and run workload E.

When comparing *Tucana* to RocksDB we use 256 YCSB threads and 64 databases (unless noted otherwise) and we choose the appropriate database by hashing the keys. When comparing H-*Tucana* to HBase and Cassandra we use 128 YCSB threads and 8 regions for HBase and H-*Tucana*. Cassandra is hash-based and does not support the notion of region, so we use a single table.

We use a small dataset that fits in memory and a large dataset that does not. The small dataset is composed of 60M or 100M records when using *Tucana* and H-*Tucana*, respectively. The large dataset has 300M or 500M records when using *Tucana* and H-*Tucana*, respectively.

In all the cases, the load phase creates the whole dataset and the run phases issue 5 million operations, bounded also by time (one hour max). With *Tucana*, even in the case of the large dataset the index nodes fit in memory as per our assumptions.

We measure efficiency as cycles/op, which shows the cycles needed to complete an operation on average. We calculate efficiency as:

$$cycles/op = \frac{\frac{CPU\_utilization}{100} \times \frac{cycles}{s} \times cores}{\frac{average\_ops}{s}}, \tag{2.1}$$

where *CPU_utilization* is the global average of CPU utilization among all processors, excluding idle and I/O time, as given by `mpstat`. As cycles/s we use the per-core clock frequency. *average_ops/s* is the throughput reported by YCSB and *cores* is the number of cores including hyperthreads.

2.3.2 Experimental Results

Tucana compared to RocksDB

Figure 2.4 shows the improvement over RocksDB in efficiency. In workloads Load A and Load E that are insert intensive, *Tucana* is similar to RocksDB for both small and large datasets, since both use write-optimized data structures. In all other workloads *Tucana* outperforms RocksDB by 0.75× to 7.01× for the small dataset and by 1.07× to 9.24× for the large dataset.

We note that increased efficiency can also be achieved with low absolute performance, which is not desirable. Figure 2.5 shows ops/s for the two systems. We see that for the small dataset *Tucana* outperforms RocksDB by 2× to 7× and by 4.47× on average in absolute performance (throughput) as well. For the large dataset, where both systems are limited by device performance, *Tucana* outperforms RocksDB by 1.1× to 2.1× and by 1.35× on

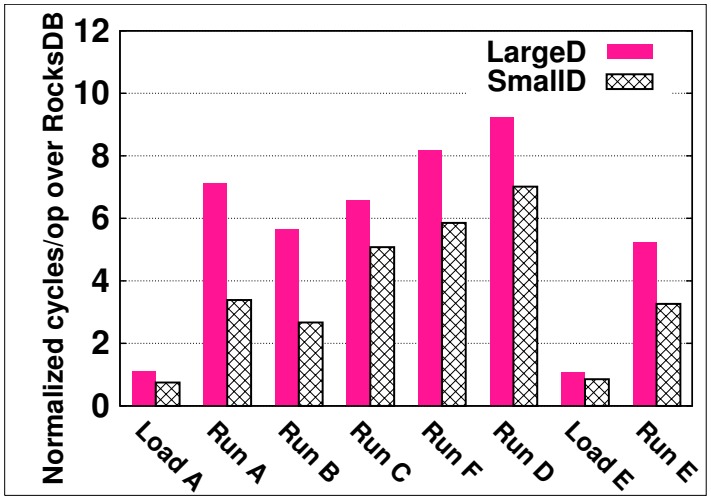
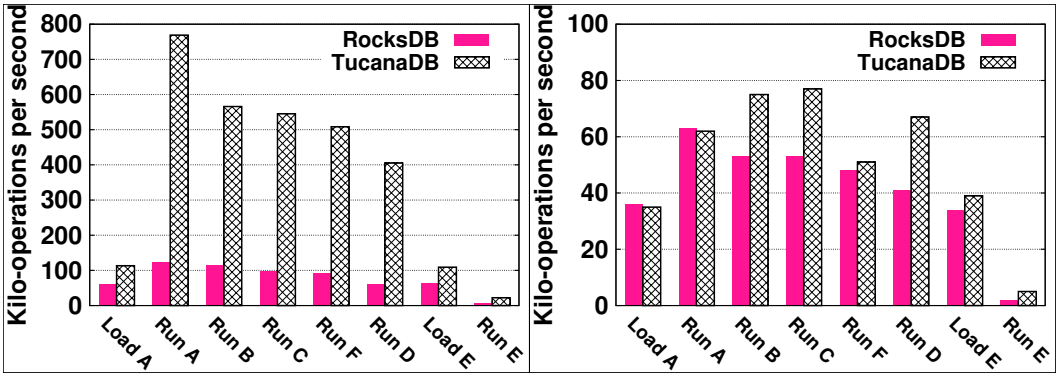


Figure 2.4: *Tucana* improvement compared to RocksDB in cycles per operation.



(a) Small dataset (b) Large dataset

Figure 2.5: Performance of *Tucana*DB and RocksDB in ops/s.

average. Average SSD utilization for all workloads is 93% for *Tucana* and 78% for RocksDB. *Tucana* has on average smaller request size, 86 KB compared to 415 KB for RocksDB. As next generation SSDs close the gap between sequential and random performance, we expect even larger performance improvements over RocksDB and similar stores.

Next, we examine I/O amplification and randomness. We run an insert-only benchmark (random distribution) using a *single* database of size 36.3 GB. RocksDB writes 435 GB while *Tucana* writes 123 GB, thus 3.5× less than RocksDB. Due to compaction operations, RocksDB also reads 2.3× the amount of data read by *Tucana*, 69 GB vs. 29 GB. Table 2.2

Inserts	Write (GB)	rq_sz	SSD (2010) time (s)	SSD (2015) time (s)
Tucana	123	18K	133	31
RocksDB	435	884K	623	100
Speedup			4.68	3.22

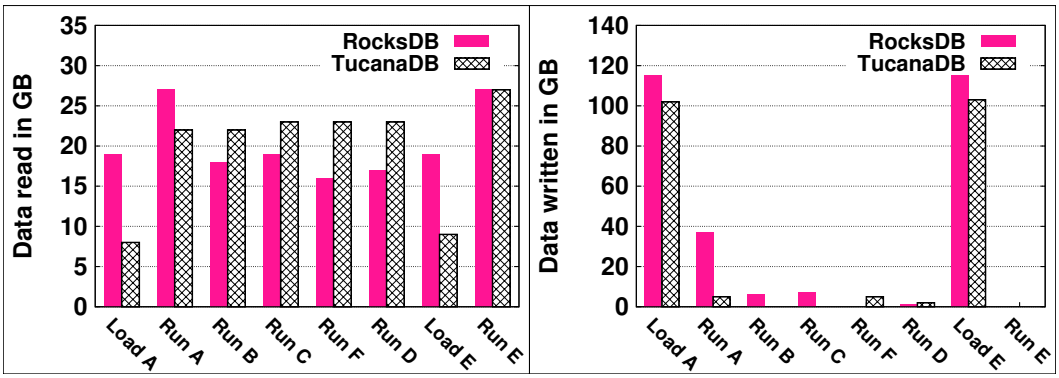
Inserts	Read (GB)	rq_sz	SSD (2010) time (s)	SSD (2015) time (s)
Tucana	26	4K	256	140
RocksDB	29	6K	229	171
Speedup			0.89	1.22

Table 2.2: Performance for the traffic pattern induced by *Tucana* and RocksDB as modeled with FIO to isolate device behavior.

shows the performance difference between these two patterns on two different SSD generations, using FIO (Flexible I/O) [6] to generate each pattern. For inserts, *Tucana*'s I/O pattern is $4.68\times$ faster on the older SSD (2010) and $3.22\times$ faster on the newer SSD (2015), compared to RocksDB's I/O pattern and volume. For gets, the difference in volume size and request size is lower and performance differences are smaller. The I/O pattern of RocksDB is better by 11% for the older SSD, whereas the I/O pattern of *Tucana* is better by 22% for the newer SSD.

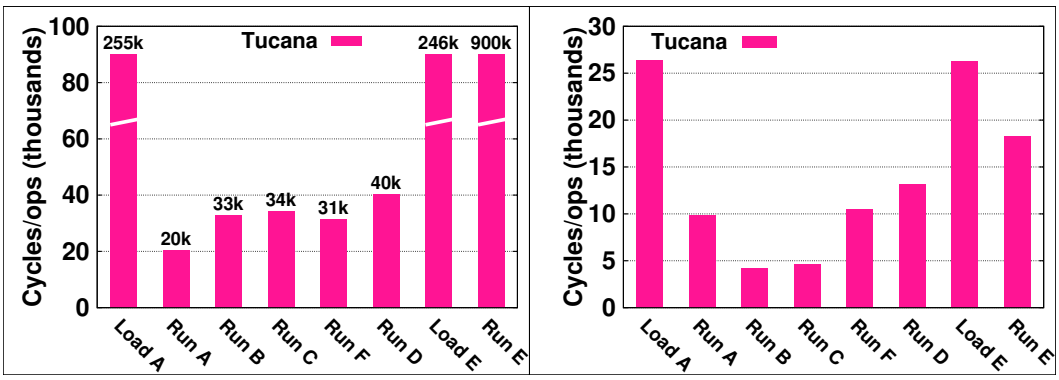
Figure 2.6 shows read and write amplification using 64 databases. Although *Tucana* incurs less I/O on average for both read and write, the difference with RocksDB in this case is smaller. On average RocksDB writes $3.33\times$ and reads $1.25\times$ more data.

Next, we examine the absolute number of cycles/op for each workload (Figure 2.7(a)). Each operation is a composite operation over a row with ten qualifiers and therefore a get operation performs ten lookup operations. For this reason, we also present numbers for the same workloads, with one qualifier per row in Figure 2.7(b). In addition, in the case of Workload E the default average length of a range query is fifty. In Figure 2.7(b) we change the scan length to five. On average, an insert operation takes about 26K cycles (Load A & Load E), a point query (get) 4K cycles (Run C) and a range query (scan), including initialization and five rows of one qualifier about 18K cycles (Run E). The other workloads are mixes of these operations. If we examine a breakdown of cycles, we see that on average



(a) Reads (b) Writes

Figure 2.6: Total amount of data read and written during each YCSB workload.



(a) 10 qualifiers (b) 1 qualifier

Figure 2.7: Number of cycles needed for YCSB workloads.

15% is used by YCSB, 43% by *Tucana*, 38% by the OS kernel, and 4% by other server processes. More specifically, for an insert-only benchmark 35% is used by *Tucana* and 60% by OS kernel. On the other hand for a get-only benchmark 66% is used by *Tucana* and 26% by kernel. System time is due to `mmap` that handles page faults, mappings, and the swapper that evicts dirty pages to devices.

Finally, Figure 2.8 shows scalability of *Tucana* and RocksDB with the number of server cores. We use the small dataset that fits in memory, partitioned in 64 databases, and we increase the load by increasing the number of YCSB threads that issue requests. For gets, *Tucana* is able to scale and it saturates the full server at 16 YCSB threads. *Tucana* pro-

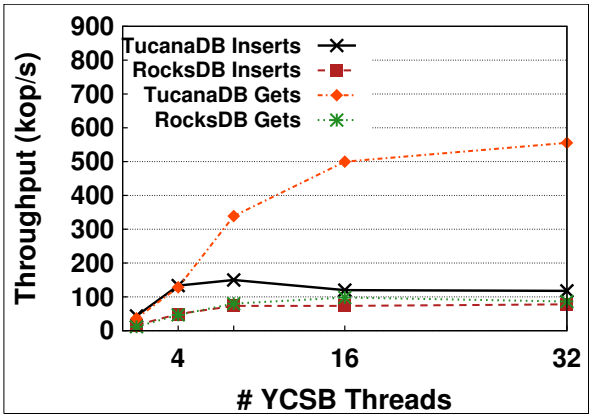


Figure 2.8: Scalability of RocksDB and *Tucana*DB with increasing threads, using the small dataset.

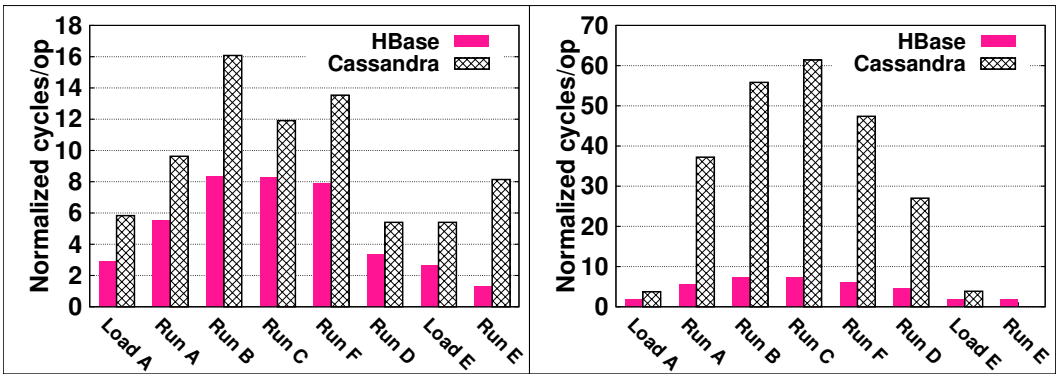
vides lock-less gets and therefore uses all available cores. After warm-up, where data is brought in memory, system utilization is about 100% at 16 YCSB threads. On the other hand, RocksDB, even after warm up, still has about 25% idle CPU time at 8 or more YCSB threads, indicating synchronization bottlenecks.

For inserts, *Tucana* saturates the server at about 8 YCSB threads, where CPU is utilized at 90-95%. RocksDB scales up to 8 threads also, where it saturates the server. Due to its more random I/O pattern, *Tucana* incurs higher device utilization, about 50% vs. 20% for RocksDB. Generally, scaling for puts in both systems is related to the number of databases. In this work, we do not explore this dimension further.

Impact on NoSQL store performance

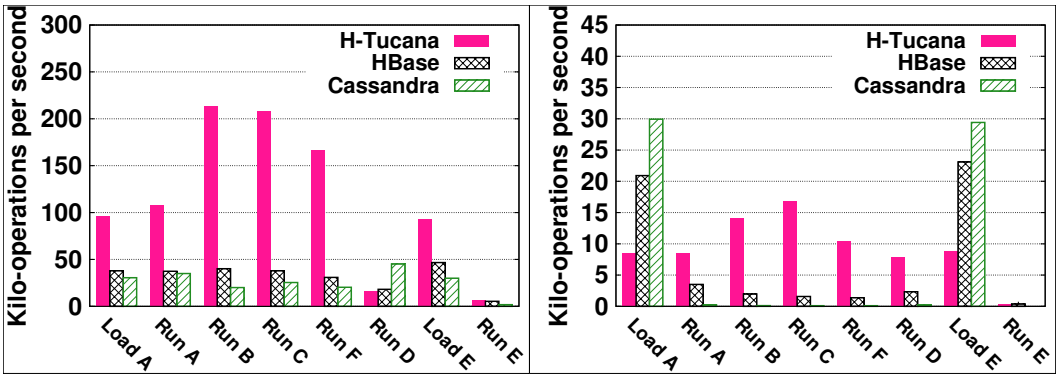
In this section, we analyze the efficiency and performance of H-*Tucana*, compared to HBase [5] and Cassandra [82].

Figure 2.9 depicts the speedup in efficiency (cycles/op) achieved by H-*Tucana* over HBase and Cassandra. We see that H-*Tucana* significantly outperforms both HBase and Cassandra. Compared to HBase, H-*Tucana* uses fewer cycles/op by up to 2.9×, 8.4×, and 5.6× for write-intensive, read intensive, and mixed workloads. Compared to Cassandra, the improvement depends on the size of the dataset. With the small dataset H-*Tucana* outperforms Cassandra by up to 5.8×, 16.1×, and 13.5× for the write, read intensive, and



(a) Small dataset (b) Large dataset

Figure 2.9: Improvement in efficiency (cycles/op) achieved by H-*Tucana* over HBase and Cassandra.



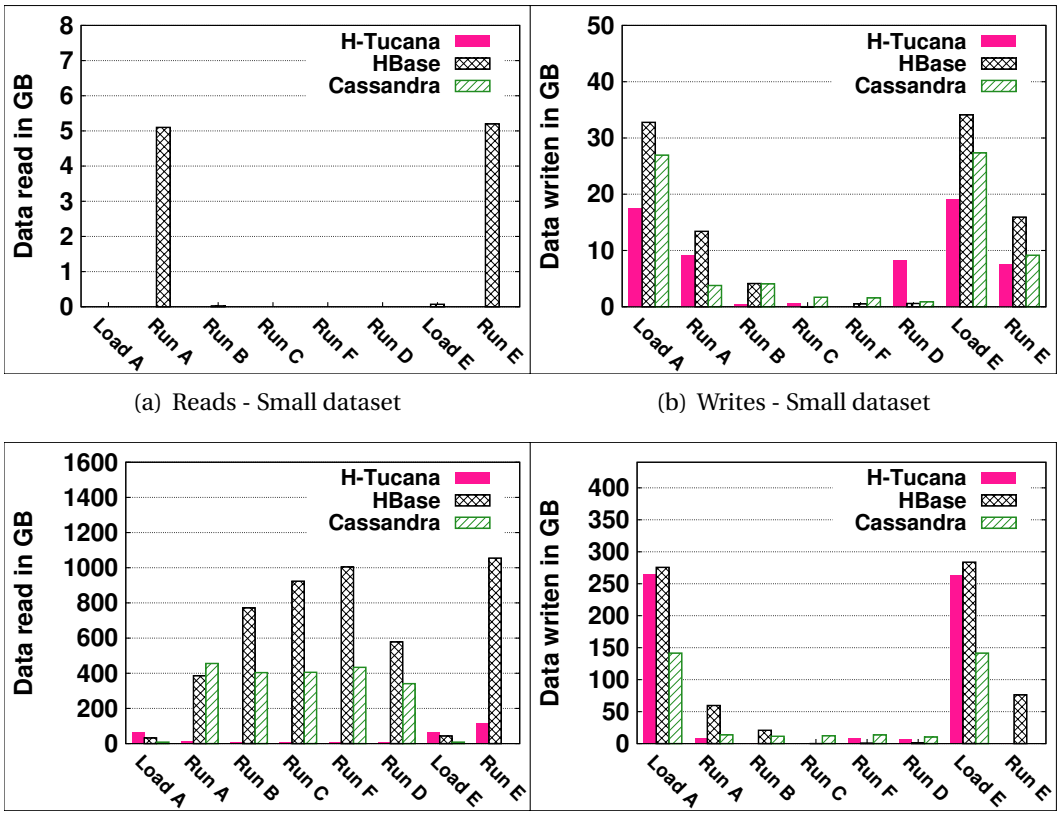
(a) Small dataset (b) Large dataset

Figure 2.10: Throughput (ops/s) achieved by H-*Tucana*, HBase and Cassandra.

mix workloads, respectively. With the large dataset, H-*Tucana* improves cycles/op over Cassandra by up 3.9×, 61.4×, and 37.2× write, read-intensive and mixed workloads respectively.

Next, we examine throughput in terms of ops/s. Figure 2.10 shows performance in kilo-operations per second whereas Figure 2.11 depicts the amount of data read and written by each workload.

For the small dataset, H-*Tucana* has up to 5.4× higher throughput compared to HBase, and up to 10.7× compared to Cassandra. In addition, H-*Tucana* does not perform any reads during the run phases. Cassandra does not read any data either, whereas HBase



(a) Reads - Small dataset

(b) Writes - Small dataset

(c) Reads - Large dataset

(d) Writes - Large dataset

Figure 2.11: Amount of data, in GB, read/written by H-*Tucana*, HBase, and Cassandra.

reads 5.1 GB and 5.2 GB when running workloads A and E, respectively. The amount of data written to the device is significantly reduced by H-*Tucana* by 38% and 17% compared to HBase and Cassandra.

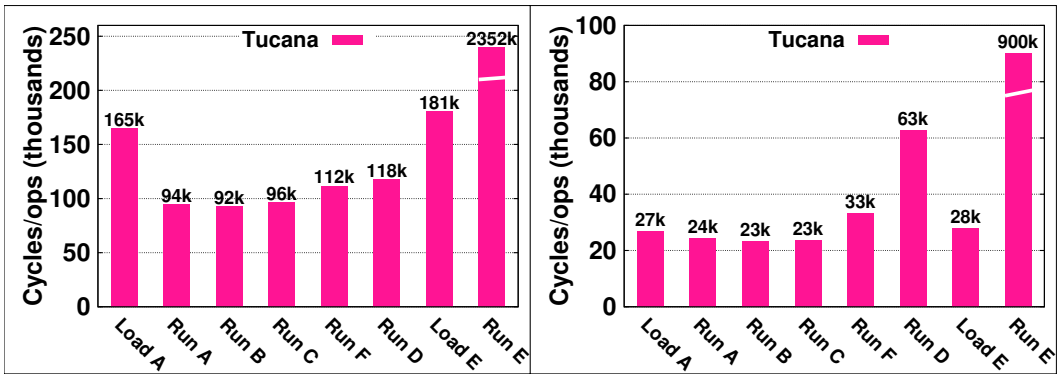
For the large dataset, during the run phase, H-*Tucana* outperforms HBase and Cassandra by up to 10.7× and 153.3×, respectively. This improvement is reflected in a significant reduction of the amount of data read from the storage device, by up to 16× and 6.9× compared to HBase and Cassandra, respectively. For read-intensive and mixed workloads, H-*Tucana* is more lightweight not only in CPU utilization but also in the amount of data read. Our modified B$^\epsilon$–tree performs faster lookups than the LSM-trees used by HBase and Cassandra, obtaining significant improvement in throughput.

During the load phase (write intensive workloads) for the large dataset H-*Tucana* exhibits up 2.5× and 3.7× worse throughput than HBase and Cassandra, as shown in Fig-

ure 2.10b. Figure 2.11 shows that during the load phase, H-*Tucana* writes 264 GB and reads 69 GB, although the size of the dataset including metadata is 77.2 GB. This is not inherent to the design of *Tucana*, as shown by the results in Section 2.3.2, but rather due to `mmap`, as follows.

With `mmap` modified disk blocks are written to the device not only during *Tucana*'s commit operations, but also periodically, by the flush kernel threads when they are older than a threshold or when free memory shrinks below a threshold, using an LRU policy and `madvise` hints. We believe that due to the increased memory pressure in H-*Tucana* compared to *Tucana* due to the Java HBase front-end, `mmap` evicts not only log pages, but also leaf pages. This reduces the amount of I/Os that can be amortized for inserts due to the limited buffering in our B^ϵ–tree. To solve this problem, we need to (a) control better which pages are evicted by `mmap`, which will be effective up to roughly the 10-15% ratio of memory to SSD capacity (see Section 2.2.1), and (b) add buffering one level higher in the B^ϵ–tree. In the same figure, we notice that in run D phase, using the small dataset, we write more data than the other systems. This is because workload D inserts new key-value pairs and then searches for them. YCSB always searches for keys that exist in the database. In *Tucana* newly inserted keys appear in searches only after a commit operation. If a key is not found, we issue a commit operation to read it. These commit operations cause increased traffic to/from the device. However the other systems retrieve the new values directly from memory. In the large dataset case all systems write them to devices and all of them write about the same amount of data.

Figure 2.12 shows the cycles/op in H-*Tucana* to execute all the workloads with ten (default configuration) and with one qualifier. With ten qualifiers, write intensive workloads require on average 172K cycles/op and read intensive and mix workloads require on average 115K cycles/op. Workload E that performs scans uses more than 2.3M cycles/op (for retrieving 50 key-value pairs). Figure 2.12b shows that with a single key, all write-intensive, read-intensive, and mixed workloads require on average 27K cycles/op, whereas workload E requires 900K cycles/op. In more detail, we see that on average 40% of the time is used by the HBase component in H-*Tucana*, 23% by *Tucana*, 33% by the system, and 5% by other processes.



(a) 10 qualifiers (b) 1 qualifier

Figure 2.12: Number of cycles needed by H-*Tucana* for YCSB workloads.

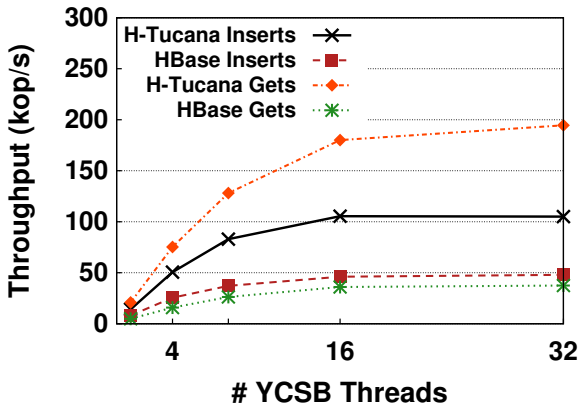


Figure 2.13: Scalability of H-*Tucana* and HBase with the small dataset.

Figure 2.13 shows the scalability for H-*Tucana* and HBase when increasing the number of YCSB threads at the client. We have not measured the scalability for Cassandra because it is not as competitive. We use the small dataset to avoid accesses to the storage device.

For gets, both systems scale up to 16 YCSB threads. At this point CPU utilization for H-*Tucana* on the server side is 52% and for HBase is 79%, while H-*Tucana* achieves higher throughput. In both cases there is a single thread that reaches 100% CPU utilization. We find that this server thread performs HBase network processing. For inserts, H-*Tucana* scales up to 16 YCSB threads and HBase scales up to 8 YCSB threads. In H-*Tucana*, server CPU utilization is 53%, whereas in HBase 63%. Similar to gets, a single server thread in the HBase front-end limits further scalability.

2.4 Summary

In this chapter we present *Tucana*, a key-value store that is designed for fast storage devices, such as SSDs, that reduces the gap between sequential and random I/O performance, especially under high degree of concurrency and relatively large I/Os (a few tens of KB). Unlike most key-value stores that use LSM-trees to optimize writes over slow HDDs, *Tucana* starts from a B^ϵ–tree approach to maintain the desired asymptotic properties for inserts. It is a full-feature key-value store that supports variable size keys and values, versions, arbitrary data set sizes, and persistence. The design of *Tucana* centers around three techniques to reduce overheads: copy-on-write, private allocation, and direct device management.

Our results show that *Tucana* is up to $9.2\times$ more efficient in terms of CPU cycles/op for in-memory workloads and up to $7\times$ for workloads that do not fit in memory. In addition, *Tucana* outperforms RocksDB for in memory workloads up to $7\times$, whereas for workloads that do not fit in memory both systems are limited by device I/O throughput. Also, H-*Tucana* is able to improve up to $8\times$ the efficiency of HBase and on average $22\times$ the efficiency of Cassandra.

This chapter shows the potential and drawbacks of memory-mapped I/O as a way to manage storage caching and therefore access storage devices. We choose to use a persistent key-value store, which is a significant building block for larger systems. *Tucana* shows that hits, which are handled entirely in hardware, provide significant performance improvements compared to a traditional user-space cache and read/write system calls. On the other hand, we also identify several issues of memory-mapped I/O in Linux where the dataset does not fit in memory and, more importantly, in the case of writes, where the user cannot control the size and timing of I/Os.

Chapter 3

Optimizing Writes With User Policies

With current technology limitations and trends, the issues of high CPU utilization and I/O amplification are becoming a significant bottleneck for keeping up with data growth. Server CPU is the main bottleneck in scaling today's infrastructure due to power and energy limitations [90, 81, 116]. Therefore, it is important to increase the amount of data each CPU can serve, rather than rely on increasing the number of CPUs in the datacenter. In this context, flash-based storage, such as solid state drives (SSDs), introduces new opportunities by narrowing the gap between random and sequential throughput, especially at higher queue depths (number of concurrent I/Os). Figure 3.1 shows the throughput of an SSD and two NVMe devices with random I/Os and increasing request size. At a queue depth of 32, an I/O request size of 32 KB for SSDs and 8 KB for NVMe achieve almost the maximum device throughput. Therefore, increased traffic due to I/O amplification is becoming a more significant bottleneck than I/O randomness. This trend will be even more pronounced with emerging storage devices that aim to achieve sub-μs latencies.

In this chapter we present *Kreon*, a key-value store that aims to reduce CPU overhead and I/O traffic by trading I/O randomness. *Kreon* combines ideas from LSM [102] (multi-level structure), bLSM [118] (B-Tree index), and Tucana/WiscKey [105, 91] (separate value log). Additionally, it uses a fine-grain spill mechanism which partially reorganizes levels to provide high insertion rates and reduce CPU overhead and I/O traffic. *Kreon* uses a write optimized data structure that is organized in N levels, similar to LSM-Tree, where each level i acts as a buffer for the next level $i+1$. To reduce I/O amplification, *Kreon* does not

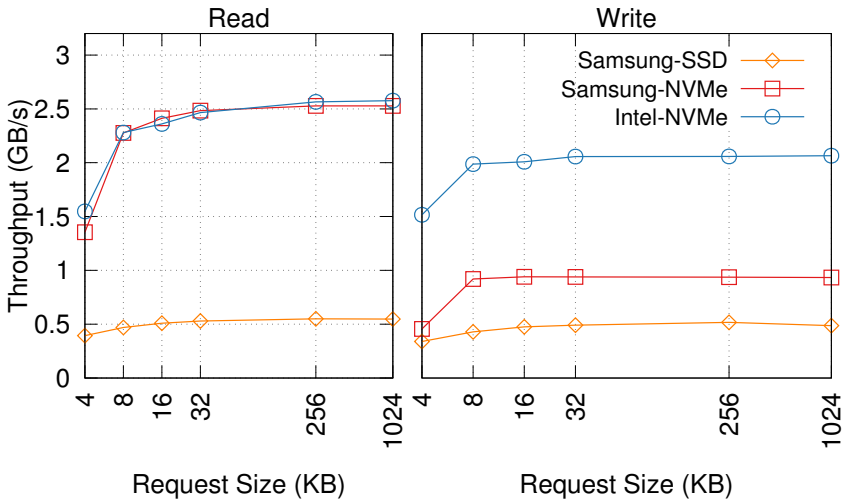


Figure 3.1: Throughput vs. block size (using iodepth 32) for Samsung SSD 850 Pro 256 GB, Samsung 950 Pro NVMe 256 GB, and Intel Optane P4800X NVMe 375 GB devices, measured with FIO [6].

operate on sorted buffers, but instead it maintains a B-tree index within each level. As a result, it generates smaller I/O requests in favor of reduced I/O amplification and CPU overhead. *Kreon* still requires and uses multiple levels to buffer requests and amortize I/O operations.

Furthermore, *Kreon* uses *memory-mapped I/O* to perform all I/O between memory and (raw) devices. *Memory-mapped I/O* essentially replaces cache lookups with valid memory mappings, eliminating the overhead for data items that are in memory. Misses incur a page fault and require an I/O operation that happens directly from memory without copying data between user and kernel space. However, the asynchronous nature of *memory-mapped I/O* means that I/O happens at page granularity, resulting in excessive and small I/Os, especially for read operations. In addition, *memory-mapped I/O* does not provide any type of consistency and recoverability nor the ability to tune page evictions for specific needs. To overcome these limitations, we implement a custom *memory-mapped I/O* path, *kmmap*, as a Linux kernel module. *kmmap* provides all the benefits of memory-mapped storage, such as, it removes the need to use DRAM caching both in kernel and user space, eliminates data copies between kernel and user space, and removes the need

for pointer translation.

We implement *Kreon* and evaluate its performance by using YCSB and large datasets of up to 6 billion keys. We compare *Kreon* with RocksDB [49], a state-of-the-art, LSM-Tree based, persistent key-value store which has lately been optimized for SSDs [42]. Our results show that using both datasets that stress I/O and datasets that fit in memory, *Kreon* reduces the amount of cycles/op by up to 8.3x. Additionally, *Kreon* reduces I/O amplification for insert-intensive workloads by up to 4.6x and increases ops/sec by up to 5.3x. Finally, our analysis of CPU overheads shows that a saturated *Kreon* server can achieve up to 2.4M insert requests/s.

Overall, the contributions of this chapter are:

1. The combination of multilevel data organization with full indexes at each level and a fine-grain spill mechanism that all together reduce CPU overhead and I/O traffic at the expense of increased I/O randomness.

2. The design and implementation of *kmmap* a custom *memory-mapped I/O* path to reduce the overhead of explicit I/O and address shortcomings of the native *mmap* path in Linux for modern key-value stores.

3. The implementation and detailed evaluation of a full key-value store compared to a state-of-the-art key-value store in terms of absolute performance, efficiency, execution time breakdown, tail latencies and device behavior.

The rest of this chapter is organized as follows: Section 3.1 provides an overview of persistent data structures. Section 3.2 presents our design. Section 3.3 presents our evaluation methodology and experimental results. Section 3.4 concludes this chapter.

3.1 Background

B-trees [13] are asymptotically optimal in the number of block transfers required for point and range queries. However, their main issue is that write performance degrades as the index grows [78]. The increasing interest for systems that are able to absorb bursty writes has led to the emergence and broad use of write-optimized data structures, which aim to

improve writes while keeping read performance close to B-trees. A popular data structure in this group is LSM-Tree [102]. LSM-Tree organizes its index in multiple hierarchical levels to amortize write operations. O'Neil *et al.* [102] do not provide specific information on how each level is organized and two alternatives exist: (a) use sorted arrays per-level or (b) use a full index per-level. HDDs favor the use of the first alternative.

Inserts to LSM-Tree are served from memory and data is gradually moved to lower levels, as the current level fills up. To move data between levels, LSM-Tree uses an important operation, called *compaction*. Compaction moves data from L_i to L_{i+1} by reading and sorting large buffers in memory and subsequently writing them to storage at L_{i+1}. Compactions have the advantage that they generate only large I/O requests because they read full tables, which makes LSM-Tree preferable to other index structures for hard disk drives (HDDs). On the other hand compactions result both in I/O amplification and CPU overheads due to moving data from one level to another. *Kreon* uses a different approach and introduces a full index per-level rather than sorted arrays, in order to reduce I/O amplification and CPU overheads.

3.2 Design

Kreon, similar to Tucana [105] and Wisckey [91], stores key-value pairs in a log to avoid data movement during reorganization from level to level. It organizes its index in multiple levels of increasing size and transfers data between levels in batches to amortize I/O costs, similar to LSM-Tree. Unlike LSM-Tree, within each level, it organizes keys in a B-tree with leaves of page granularity similar to bLSM [118]. However, unlike bLSM, *Kreon* transfers data between levels via a *spill operation*, rather than full reorganization of the data in the next level. Spills are a form of batched data compaction that merge keys of two consecutive levels [L_i, L_i + 1]. However, spills do not read the entire L_{i+1} during merging with L_i and do not reorganize data and keys on a sequential part of the device [118]. Instead, *Kreon* spills read/write level L_{i+1} partially using the full B-tree index of each level.

The trade-off is that during batched spills, *Kreon* generates random read I/O requests at large queue depth (high I/O concurrency) to significantly reduce I/O traffic and CPU

overhead. On the other hand write requests for writing updated parts of L_{i+1} index produce relative large write I/O requests. This is because *Kreon* B-tree uses Copy-on-Write for persistence [56] and a custom segment allocator so updated leaves are written close on the device. We believe that trading I/O randomness for I/O traffic and CPU utilization is the right tradeoff, given current technology trends for power limitations in datacenters and storage device technology with SSDs and NVM.

Furthermore, *Kreon* uses memory mapped I/O to eliminate redundant copies between kernel and user space and constant pointer translation. *Kreon*'s *memory-mapped I/O* path is designed to provide efficient support for managing I/O memory addressing shortcomings of the default *mmap* path in the Linux kernel. These shortcomings are: (a) It does not provide explicit control over data eviction, as with an application-specific cache, (b) it results in an I/O even for pages that include garbage, and (c) it employs eager evictions to free memory, which results in excessive I/O, in order to avoid starving other system components.

Figure 3.2 depicts the architecture of *Kreon* showing two levels of indexes, the key-value log, and the device layout. Next, we discuss our design for the system index and *memory-mapped I/O* in detail.

3.2.1 Index Organization

Kreon offers a dictionary API (insert, delete, update, get, scan) of arbitrary sized keys and values. *Kreon* stores keys in groups named regions. Each region can map either to a table or shards of the same table. For each region it stores key-value pairs in a single append-only *key-value log* [105, 91] and keeps a multilevel index. The index in each level is a B-tree [13], which consists of two types of nodes: internal and leaf nodes. Internal nodes keep a small log where they store pivots, whereas leaf nodes store key entries. Each key entry consists of a tuple with a pointer to the key-value log and a fixed-size key prefix. Prefixes are the first M bytes of the key used for key comparisons inside a leaf. They reduce significantly I/Os to the log since leaves constitute the vast majority of tree nodes. If the effectiveness of prefixes is reduced due to low entropy of the keys, existing techniques

discuss how they can be recomputed [18].

During inserts, *Kreon* appends the key-value pair to the key-value log, then it performs a top-down traversal in its L_0 B-tree, from the root to the corresponding leaf, and adds a key entry to the leaf. Get operations examine hierarchically levels from L_0 to L_N, and return the first match. Since inserts propagate with the same order as get operations, the version of the retrieved key is the most recent. Delete operations mark keys with a tombstone and defer the actual delete operation. During system operation we use the marked key entries for subsequent inserts that reuse the index entry and mark as free the deleted (old) key-value pair in the log. Marked and unused entries in the index are reclaimed during spills. Marked space in the log is reclaimed asynchronously, as discussed in Section 3.2.1. Update operations are similar to a combined insert and delete. Scan operations create a scanner per-level and use the index to fetch keys in sorted order. They combine the results of each level to provide a global sorted view of the returned keys.

Each region supports a single-writer/multiple-readers concurrency model. Readers operate concurrently with writers using Lamport counters [83] per tree node for synchronization. Scans, similar to other systems [49], access all data inserted to the system up to the scanner creation time and they operate on an immutable version of each tree which is facilitated by the Copy-On-Write approach used by *Kreon* (Section 3.2.3).

Similar to LSM-Tree, L_0 always resides entirely in memory. Portions of $levels \geq 1$ are brought in memory on demand. *Kreon* enforces memory placement rules for different levels by using *kmmap* and explicit priorities (Section 3.2.2). Spill operations, data reorganization in scans and number of levels are outside of the scope of this dissertation and more details on them can be found in [106].

Device Layout and Access

Kreon manages storage space as a set of segments. Each segment is a contiguous range of blocks on a device or a file. To further reduce overhead we access devices directly rather than use a file system in between. Our measurements show that files result in a 5-10% reduction in throughput due to file system overhead. Each segment hosts multiple regions and it has its own allocator to manage free space.

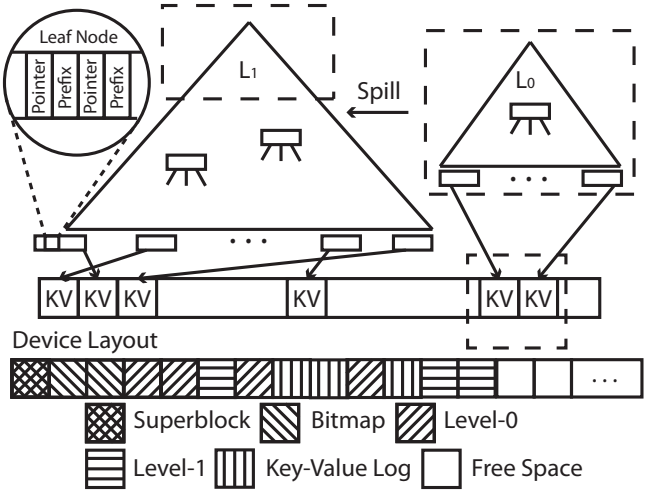


Figure 3.2: The main structures of *Kreon* showing two levels of indexes, the key-value log, and the device layout. Dashed rectangles include portions of the data structures that are kept in memory via *kmmap*.

Kreon's allocator stores its metadata at the beginning of each segment, which consists of a *superblock* and a *bitmap*. The *superblock* keeps pointers to the latest consistent state of the segment and its regions. The *bitmap* contains information about the allocation status (free or reserved) of each 4 KB block. The *bitmap* is accessed directly via an offset and at low overhead, while for searches we use efficient bit parallel techniques [23].

Kreon allocates space eagerly for regions, at large units, currently 2 MB, consuming them incrementally in smaller units. This approach avoids frequent calls to the allocator that is shared across regions in each segment. It also improves average write I/O size by letting each region grow in a contiguous part of the device.

Similarly, the key-value log in *Kreon* is organized in large chunks, currently 2 MB. At the start of each chunk we keep metadata about the garbage bytes as done in other systems [101]. Delete operations update the deleted bytes counter of the corresponding chunk. When this counter reaches a threshold the valid key-value pairs are moved to the end of the log. We locate these keys in the index via normal lookups and we update the leaf pointers accordingly. Finally we release the chunk to be available for next allocations.

3.2.2 Memory-Mapped I/O

Most key-value stores and other systems that handle data use explicit I/O to access storage devices or files with read/write system calls. In many cases, they also employ a user-space cache as part of the application to minimize accesses to storage devices and user-kernel crossings for performance purposes. The use of a user-space cache is important to avoid frequent system calls for lookup operations that need to occur for every data item, regardless if it eventually hits or misses. However, even the use of an application user-level cache incurs significant overhead in the common path [58, 61, 105].

The use of *memory-mapped I/O* in *Kreon* reduces CPU overhead related to the I/O cache in three ways: (a) It eliminates cache lookups for hits by using valid virtual page mappings. Memory-mapped I/O does not require cache lookups because virtual memory mappings distinguish data that are present in memory from data that are only located on the device. All device data are mapped to the application address space but only data that are present in memory have valid virtual memory mappings. Accesses to data that are not present in memory result in page faults that are then handled by *mmap*. Given that many operations in key-value stores, such as get operations with a Zipf distribution, complete from memory, *Kreon* avoids all related cache lookup overheads. (b) There is no need to copy data between user and kernel space when performing I/O. Pages used for data in memory are used directly to perform I/O to and from the storage devices. (c) There is no need to serialize/deserialize data between memory and the storage devices. Finally, *memory-mapped I/O* uses a single address space for both memory and storage, which eliminates the need for pointer translation between memory and storage address spaces and therefore, the need to serialize and deserialize data when transferring between the two address spaces.

Kreon's memory-mapped I/O

Kreon provides its own custom *memory-mapped I/O* path to address the shortcomings of *mmap* in Linux.

First, in *mmap* there is no explicit control over data eviction, as with an application-

specific cache. Linux uses an LRU-based policy, which may evict useful pages, for instance, pages of L_0 instead of L_1 pages. We assume, by our design, that L_0 always resides in main memory in order to amortize write I/O operations. Linux kernel *mmap* does not provide a mechanism to achieve this. A possible solution to this problem is to lock important pages with *mlock*. However, Linux does not allow a large number of pages to be locked by a single process because this affects other parts of the system.

Second, each write operation in an empty page is effectively translated to a read-modify-write because *mmap* does not have any information about the status (allocated or free) of the underlying disk page and the intended use. This results in excessive device I/O. Instead, if applications can inform *mmap* whether a page contains garbage and will be written entirely, *mmap* can map this page without reading it first from the device, eliminating unnecessary read traffic.

Third, *mmap* employs aggressive evictions based on memory usage and time elapsed since pages marked as dirty to free memory and avoid starving other system components. Mapping large portions of the application virtual address space creates pressure to the virtual memory subsystem and results in unpredictable use of memory and bursty I/O. Furthermore, eager and uncoordinated evictions do not facilitate the creation of large I/Os through merging. Empirically, we have often observed large intervals (of several 10s of seconds) where the system freezes while it performs I/O with *mmap* and applications do not make progress. Furthermore we observed similar behaviour with *msync* call. This unpredictability and large periods of inactivity are an important problem for key-value stores that serve data to online, user-facing applications.

To overcome these limitations, we implement a custom *mmap*, as a Linux kernel module, called *kmmap*. Figure 3.3 shows the overall design and data structures of *kmmap*.

Kmmap bypasses the Linux page cache and uses a priority-based FIFO replacement policy. As priority we define a small, per-page number (0 to 255). During memory pressure, a page with a higher priority is preferred for eviction. Priorities are kept only in memory and are set explicitly by *Kreon* with *ioctl* calls. Priorities are set as follows: *Kreon* assigns priority 0 to index nodes of L_0, 1 to index nodes of L_1, 2 to leaf nodes of L_1, and 3 to the log. L_0 fits in memory and it will not be evicted. Generally if we have more than two levels L_0

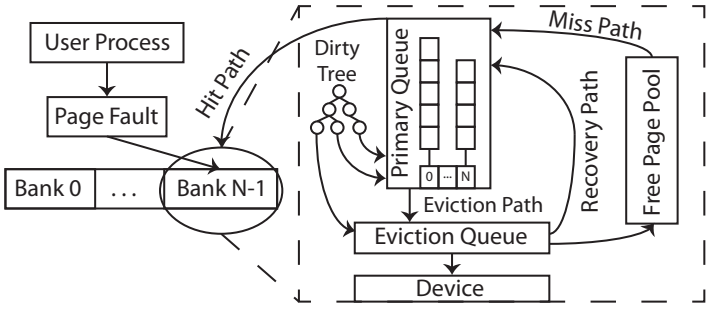


Figure 3.3: The main structures of *kmmap*.

always uses priority 0 and the log maximum priority. We calculate the priority of level L_N as $(2 * N - 1)$ for index nodes and $(2 * N)$ for leaves.

To increase parallelism, *kmmap* organizes memory in independent banks, similar to DI-MMAP [48]. Pages are mapped to banks by hashing the page fault address. To place consecutive pages in the same bank, the page fault address is first shifted. Unlike DI-MMAP, *kmmap* uses fine-grain locking inside banks, which results in higher parallelism and eliminates periods of inactivity (long freezes).

When *Kreon* accesses a page (for read or write), that does not reside in main memory, a page fault occurs. On a page fault, *kmmap* retrieves a free page from an in-memory list (*Free Page Pool*), it reads the data from the device if required, and finally enqueues the page to the *Primary Queue* based on its priority. *kmmap* keeps a separate FIFO per priority inside the *Primary Queue*. In the case where the *Primary Queue* is full of pages, it dequeues a fixed number of entries for batching purposes, with preference to entries with higher priority. Then it unmaps them from the process address space and moves them into the *Eviction Queue*. The *Eviction Queue* is organized as an in-memory red-black tree structure, keeping keys sorted based on page offset at the device. For evictions, it traverses the *Eviction Queue* and merges consecutive pages to generate as large I/Os as possible. It keeps dirty pages that belong to the *Primary Queue* or the *Eviction Queue* in another in-memory red-black tree structure (*Dirty Tree*) sorted by their device offset. The *Dirty Tree* is used by *msync*, to avoid scanning unnecessary (clean) pages.

Kmmap compared to *mmap* keeps pages in memory for a longer period of time and

does not evict them, unless there is a need to do so. This allows *Kreon* to generate larger I/Os during spill operations by merging more requests. When a spill is completed, *Kreon* sets the priority of pages from the previously spilled L_0 to 255 (smallest priority) so they get evicted as soon as possible.

To avoid unnecessary reads that occur when a new page is written in *Kreon*, *kmmap* detects and filters these read-before-write operations, whereas write and read-after-write operations are forwarded to the actual device. To achieve this, it uses an in-memory bitmap, which is initialized and updated by *Kreon* via a set of *ioctl* calls. The bitmap uses a bit per device block, so a 1 TB SSD requires 32 MB of memory for the bitmap.

Kmmap provides a non-blocking *msync* call that allows the system to continue operation while pages are written asynchronously to the devices. For this purpose we keep a timestamp for each page that indicates when it became dirty. To write dirty pages, we iterate the *Dirty Tree* and write only pages with timestamp older than the timestamp of *msync*. We use fine grain locking in *Dirty Tree* and we allow to add new dirty pages into it during *msync*. However, there can be pages that are already dirty and changed after *msync*, which should not be written. *Kreon* uses Copy-On-Write to ensure that after a commit dirty pages will not change again as we need to allocate new pages.

Finally, *Kreon* significantly reduces unpredictability with respect to memory management during system operation by limiting the maximum amount of memory it occupies throughout its operation. It uses a configuration parameter to calculate the size of L_0 in memory and based on this it preallocates all *memory-mapped I/O* structures.

3.2.3 Persistence

Kreon uses Copy-On-Write (CoW) [115] to maintain its state consistent and recoverable after failures. *Kreon*'s state includes the data section of each segment (metadata and data of the tree) and the allocator metadata. To persist a consistent version of its state *Kreon* provides a commit operation. This operation first writes the dirty (in-memory) data into the device and then switches atomically from the old state to the new state. More specifically, *Kreon* stores a pointer to the latest persistent state in the superblock. At the end

of a commit operation, *Kreon* updates this pointer to the newly created persistent state which becomes immutable. In case of a failure, the new state that is not committed will be discarded during startup, resulting in a rollback to the last valid state.

In *Kreon* we use CoW for different purposes at L_0 and the rest of the levels. The index of all levels except L_0 is kept on the device and only brought to memory on demand. Therefore, typically, only a small part of these indexes is in memory. For these indexes, *Kreon* uses CoW to ensure consistency of the index on the device during failures. These levels are only written to the device during spills. Therefore, the only time when commits occur (besides L_0), is at the end of each spill operation.

L_0 is different and can always be recovered by replaying a subset of the key-value log. This subset is always the latest portion of the log and is easy to identify via markers placed in the log during the spill operation from L_0 to L_1. Therefore, after a failure, L_0 can be reconstructed. However, L_0 can grow significantly due to the large amount of memory available in modern servers. *Kreon* uses CoW to checkpoint L_0 to the device and to reduce recovery time. Therefore, *Kreon*'s commits of L_0 are not critical for recovery. L_0 checkpoints do not have to be very frequent. Infrequent L_0 commits do not lead to data loss because the L_0 index can be reconstructed through the replay of the key-value log. The log is written to the device more frequently, when a log segment (2 MB) becomes full.

Essentially, *Kreon* uses L_0 commits at a coarse granularity to improve recovery time, without however, a negative impact on the recovery point. The tradeoff introduced is that commits incur overhead during failure free operation. Overall, we expect that *Kreon* L_0 commits will be issued periodically at a time scale of minutes, which has a low impact on performance. Section 3.3.2 evaluates commit overhead in *Kreon*.

3.3 Experimental Analysis

In this section we evaluate *Kreon* against RocksDB [49, 52]. Our goal is to examine the following aspects of *Kreon*:

1. What is the efficiency in cycles/op achieved by *Kreon* compared to LSM-based key-value stores? Does higher efficiency come at the cost of worse absolute throughput

	Workload
A	50% reads, 50% updates
B	95% reads, 5% updates
C	100% reads
D	95% reads, 5% inserts
E	95% scans, 5% inserts
F	50% reads, 50% read-modify-write
G	100% scans

Table 3.1: Workloads evaluated with YCSB. All workloads use a query popularity that follows a Zipf distribution except for D that follows a latest distribution.

or latency?

2. How much does the new index design and *memory-mapped I/O* contribute to reducing overheads?

3. How does *Kreon* improve I/O amplification? How much does it increase I/O randomness?

4. How do the growth factor across levels and L_0 checkpoint interval affect performance?

Next, we discuss our methodology and each aspect of *Kreon* in detail.

3.3.1 Methodology

Our testbed consists of a single server which runs the key-value store and the YCSB client. The server is equipped with two Intel(R) Xeon(R) CPU E5-2630 v3 CPUs running at 2.4 GHz, with 8 physical cores and 16 hyper-threads, for a total of 32 hyper-threads and with 256 GB DDR4 at 2400 MHz. It runs CentOS 7.3 with Linux kernel 4.4.44. During our evaluation we scale-down DRAM as required by different experiments. The server has six Samsung 850 PRO 256 GB SSDs, organized in a RAID-0 using Linux *md* and 1 MB chunk size.

We use RocksDB[1] v5.6.1, atop of *XFS* with disabled compression and jemalloc [68], as recommended. We configure RocksDB to use direct I/O because we evaluate experimentally that in our testbed results in better performance. Furthermore, we use RocksDB's

[1]Options file: `https://goo.gl/NJNLNr`.

user-space LRU cache, with 16 or 192 GB depending on the experiment. We use a C++ version of YCSB [114] with the standard workloads proposed by YCSB [37, 35]. Table 3.1 summarizes these workloads. To further enrich our analysis, we add a new workload named *G* which is similar to *E* but consists only of scans. In all cases we use 128 YCSB threads for each client and 32 regions.

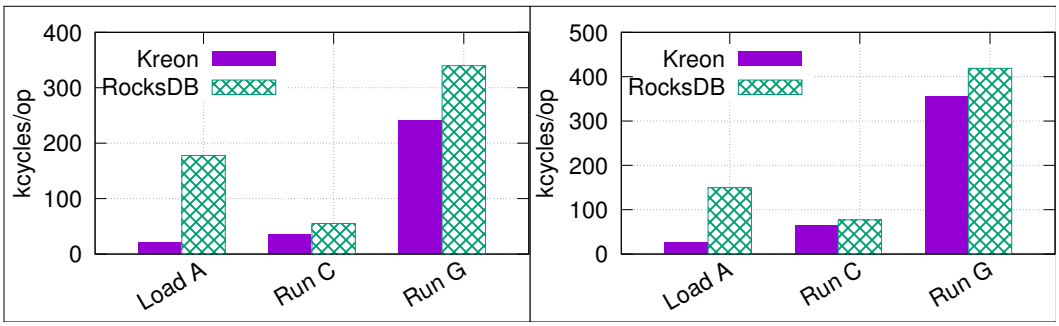
We use two datasets, a small dataset that fits in memory and a large dataset that does not. Both datasets consist of 100M records and require about 120 GB of storage. YCSB by default generates 10 columns for each key. We keep these 10 columns inside a single value. We use a 100M keys (recordcount and operationcount equals to 100M) * 10 columns which results in 1 billion columns. In the small dataset we boot the server with 194 GB of memory, 192 GB for key-value store and 2 GB for the OS. For the large dataset, and to further stress I/O we boot the server with 18 GB of memory, 16 GB for key-value store and 2 GB for the OS.

In the small dataset, both the key-value log and the indexes fit in memory, so I/O is generated by commit operations. In the large dataset, neither the key-value log nor the indexes fit in memory and only L_0 is guaranteed to reside in memory. Therefore, the small dataset demonstrates more clearly overheads related to memory accesses whereas the large dataset stresses the I/O path.

We calculate efficiency in cycles/op as follows:

$$cycles/op = \frac{\frac{CPU\_utilization}{100} \times \frac{cycles}{s} \times cores}{\frac{average\_ops}{s}},$$

where *CPU_utilization* is the average of CPU utilization among all processors, excluding idle and I/O wait time, as given by *mpstat*. As *cycles/s* we use the per-core clock frequency. *average_ops/s* is the throughput reported by YCSB and *cores* is the number of system cores including hyperthreads.



(a) Small dataset (b) Large dataset

Figure 3.4: Efficiency of *Kreon* and RocksDB in cycles/op.

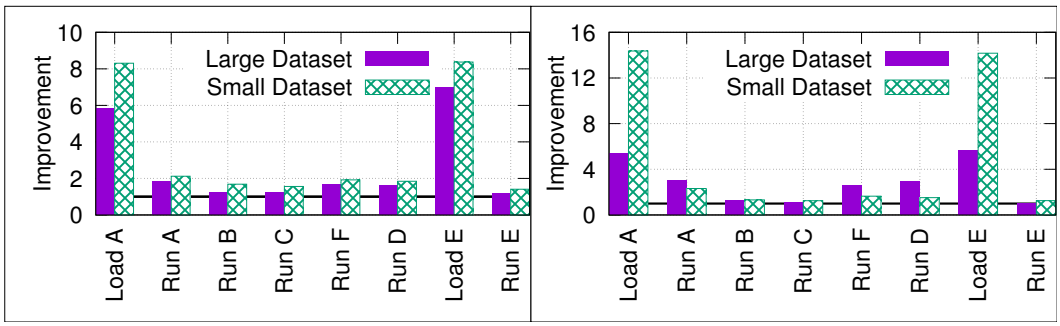
3.3.2 Experimental Results

CPU Efficiency and Performance

We evaluate the efficiency of *Kreon* in terms of cycles/op required to complete each operation, excluding YCSB overhead. To exclude the overhead of the YCSB client, we profile the average cycles/op required by YCSB and we subtract this overhead from the overall value for both RocksDB and *Kreon*.

Figure 3.4 shows efficiency results for *Kreon* and RockDB. For the small dataset *Kreon* requires 8.3x, 1.56x, and 1.4x fewer cycles/op for *Load A*, *Run C*, and *Run G*, respectively. For the large dataset *Kreon* requires 5.82x, 1.2x, and 1.18x fewer cycles/op for *Load A*, *Run C*, and *Run G*, respectively. In addition, for the small dataset and *Load A* we compare *Kreon* when using *kmmap* and when using vanilla *mmap*. Although we do not show these results for space purposes, using *kmmap*, *Kreon* achieves 1.47x fewer cycles/op compared to vanilla *mmap*, indicating the importance of proper and customized *memory-mapped I/O* for key value stores.

In terms of absolute numbers, we see that *Kreon* requires 21, 35, and 241 kcycles/op for each of *Load A*, *Run C*, and *Run G* for the small dataset and 25, 64, and 354 kcycles/op for each of *Load A*, *Run C*, and *Run G* for the large dataset.



(a) Efficiency (b) Throughput

Figure 3.5: Efficiency and throughput improvement of *Kreon* compared to RocksDB for all YCSB workloads.

YCSB Workloads

We now show results from a complete run for all YCSB workloads. We run the workloads in the recommended sequence [35]: Load the database using the configuration file of workload A, run workloads A, B, C, F, and D in a row, delete the whole database, reload the database with the configuration file of workload E and then run workload E.

For both the small and large dataset, Figure 3.5(a) shows the improvement in efficiency compared to RocksDB, whereas Figure 3.5(b) shows the improvement in throughput. Regarding efficiency, *Kreon* improves RocksDB efficiency, on average, by 3.4x and 2.68x, for the small and large dataset, respectively. Regarding throughput, the improvement in *Kreon* compared to RocksDB is, on average, 4.72x and 2.85x for the small and large datasets, respectively.

Latency analysis

First, we examine the average latency per operation for the small dataset. For *Load A*, RocksDB achieves 1162 μs/op, *Kreon* with vanilla *mmap* achieves 346 μs/op, and *Kreon* with *kmmap* achieves 72 μs/op. This shows that *kmmap* provides significant reduction in latencies compared to vanilla *mmap*. For *Run C*, RocksDB achieves 174 μs/op, *Kreon* with vanilla *mmap* achieves 119 μs/op, and *Kreon* with *kmmap* achieves 109 μs/op. Generally, *Kreon* with *kmmap* achieves 16.1x and 1.5x lower latency on average for *Load A* and *Run C*

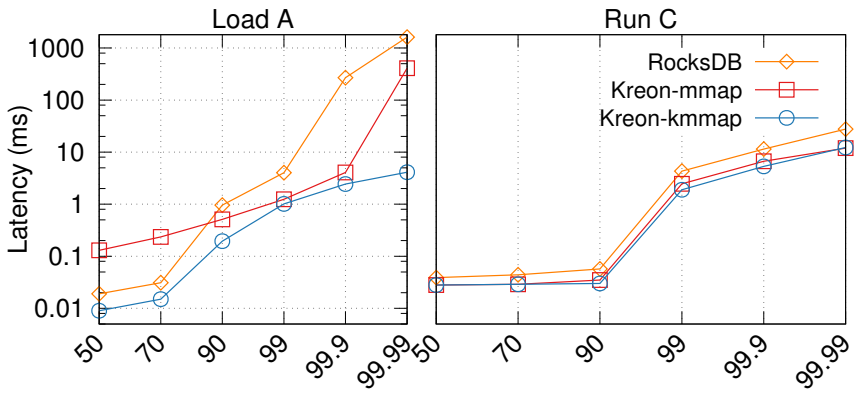


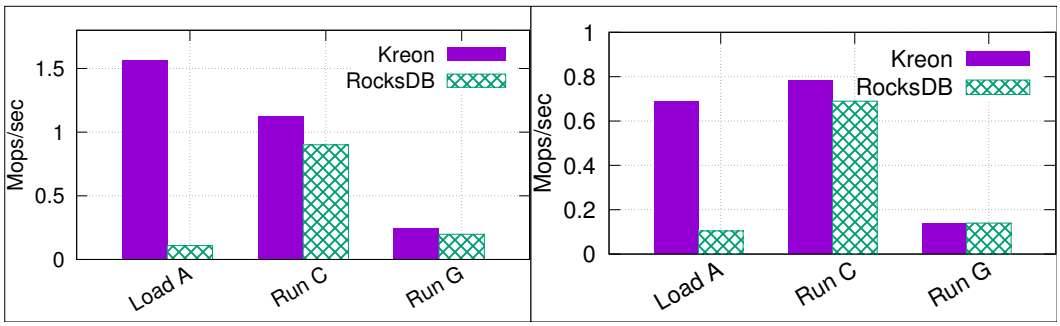
Figure 3.6: Tail latency for Load A and Run C for RocksDB, *Kreon* with vanilla *mmap*, and *Kreon* with *kmmap*.

compared to RocksDB.

Figure 3.6 shows the tail latency for *Kreon* using both *kmmap* and vanilla *mmap* and RocksDB. For *Load A*, for 99.99% of requests, *Kreon* with *kmmap* achieves 393x lower latency compared to RocksDB. Furthermore, *kmmap* results in 99x lower latency compared to vanilla *mmap*. In our design we remove blocking for inserts during *msync* and during spilling of L_0. Unlike *Kreon*, RocksDB blocks inserts during compaction operations for longer periods. For *Run C*, *Kreon* results in almost the same latency with and without *kmmap* and about 2x better than RocksDB. This is because in a read-only workload most overheads comes from the data structure, as we use a dataset that fits in memory and removes the need for I/O. In the case of RocksDB this overhead includes also a cache lookup while in *Kreon* it only accesses already mapped memory. The use of *mmap* and *kmmap* results in almost the same performance as this experiment does not stress *memory-mapped I/O* path.

Very large dataset

To examine *Kreon*'s behavior with a very large dataset we run *Load A* using 6 billion keys with one column per key (key size of 30 bytes and value size of 100 bytes). For this experiment we use 192 GB of DRAM for both *Kreon* and RocksDB. *Kreon* reduces cycles/op by 8.75x, increases ops/s by 12.11x, and reduces write volume by 4.25x and read volume by



(a) Small dataset (b) Large dataset

Figure 3.7: Throughput for *Kreon* and RocksDB in ops/s.

3.14x.

Absolute operation throughput

Next, we examine if *Kreon*'s increased efficiency in cycles/op comes at the expense of reduced absolute performance. This is important for understanding if *Kreon* trades device and host CPU efficiency in the right manner. For *Kreon* and RocksDB, Figure 3.7 shows the throughput (ops/s), achieved by YCSB. For the small dataset, *Kreon* achieves 14.35x, 1.24x, and 1.25x more ops/s for *Load A*, *Run C*, and *Run G*, respectively.

For the large dataset, *Kreon* achieves 5.33x and 1.05x more ops/s for *Load A* and *Run C*, respectively, than RocksDB. However, *Kreon* is 2% worse for *Run G*. In this case, both RocksDB and *Kreon* are limited by device throughput and this is the reason that both systems are comparable. On the other hand, *Kreon* results in much lower CPU utilization: on average *Kreon* has a utilization of 13.8% while RocksDB has a utilization of 39.5%. Therefore, *Kreon* is able to support more clients given an adequate number of storage devices.

For the small dataset and *Load A*, we compare *Kreon* with *kmmap* and with vanilla *mmap*. We see that *kmmap* improves throughput by 4.34x compared to vanilla *mmap*.

Execution Time Breakdown

In this section we examine the main components that contribute to overhead in *Kreon* and RocksDB. Our purpose is to identify what are the main sources of improvement in *Kreon*

kcycles/ operation	Load A (16GB)		Load A (192GB)	
	RocksDB	Kreon	RocksDB	Kreon
YCSB	26.67	25.34	22.79	21.37
index	24.15	13.46	26.76	13.1
cache	0.33	0.56	0.82	0.45
I/O pfault	2.92	5.84	1.66	2.61
I/O syswrite	12.20	0	11.91	0
compaction/spill	63.41	0.78	60.87	0.64

Table 3.2: Breakdown of cycles per operation for workload Load A (write only). Numbers are in kcycles.

compared to RocksDB and what are the remaining sources of overhead.

We examine two workloads a write-intensive (Load A) and a read-intensive (Run C) using both the small and large datasets. We profile *Load A* and *Run C* workloads and we use stack traces from *perf* and Flamegraph [57] to identify where cycles are spent. We divide overhead in the following components: index operations (updates/traversals for put/get operations), caching, I/O, and compaction/spill. I/O refers to explicit I/O operations in RocksDB and *memory-mapped I/O* in *Kreon*. Caching refers to the cycles needed for cache lookups, fetching new data for misses and page evictions when the cache becomes full. RocksDB uses a user-space LRU cache whereas in *Kreon* cache resides in kernel-space as part of *kmmap*.

Table 3.2 shows the breakdown for the write-intensive *Load A* workload. The number of cycles used by the YCSB client is roughly the same in all cases. In the small workload, index manipulation incurs about 44% lower overhead in *Kreon* (~13K cycles/op in *Kreon* vs. 24K cycles/op in RocksDB). Caching overhead is low in all cases for the write-intensive workload. For I/O *Kreon* requires 52% fewer cycles. For compaction/spill *Kreon* dramatically reduces the cycles required per operation from 63.41K to 0.78K. In the large workload, index manipulation requires 51% fewer cycles in *Kreon* (from 26K to 13K) and for I/O 78% fewer cycles. Similarly to the small dataset, *Kreon* significantly reduces the number of cycles per operation for compaction/spill from 60.87K to 0.64K.

Table 3.3 shows the breakdown for the read-intensive workload (*Run C* benchmark). In the small dataset, index manipulation incurs 12% fewer cycles (from 4.87K in RocksDB

kcycles/ operation	Run C (16GB)		Run C (192GB)	
	RocksDB	Kreon	RocksDB	Kreon
YCSB	13.9	12.11	54.04	53.11
index	4.87	4.28	25.59	10.29
cache	8.61	0.41	9.79	0.74
I/O pfault	0.12	3.16	0.54	5.9
I/O sysread	2.86	0	7.21	0

Table 3.3: Breakdown of cycles per operation for workload Run C (read only). Numbers are in kcycles.

to 4.28K in *Kreon*). Caching overhead is reduced by 95% (from 8.61K cycles/op in RocksDB to 0.41K cycles/op in *Kreon*) whereas I/O requires 9% more cycles in *Kreon*. In the large dataset, index manipulation overhead is reduced by 59% in *Kreon*, caching overhead by 92%, and I/O by 18%.

Overall, we see that *Kreon*'s design significantly reduces overheads for index manipulation, spills, and I/O. We also see that all proposed mechanisms for indexing, spills that involve only metadata, and *memory-mapped I/O*-based caching, have important contributions. Finally, we see that in *Kreon* the largest number of cycles is consumed by index manipulation (up to 13K cycles/op) both for both datasets in both workloads and secondarily by page faults (up to 5.9K cycles/op).

I/O Amplification and Randomness

In this section we evaluate how *Kreon* reduces amplification at the expense of reduced I/O size and increased I/O randomness and what is the overall impact on performance.

Table 3.4 shows the total amount of traffic to the device using the large dataset. We see that for *Load A Kreon* reduces both read traffic by 5.9x and write traffic by 3.9x, while the total traffic reduction is 4.6x. *Kreon* reads 1.08x less data for *Run C*. On the other hand, *Kreon* reads 4.1x more data for *Run G*, due to data re-organization. However, this cost comes only during scans and for leaves that are not re-organized. On the other hand, in RocksDB data reorganization takes place in every compaction.

To reduce amplification, *Kreon* generates by design smaller and more random I/Os

	Load A	Run C	Run G
RocksDB-Read	669	138	296
Kreon-Read	112	127	1237
RocksDB-Write	869	0	8
Kreon-Write	221	0	139

Table 3.4: Total I/O volume (in GB) for all benchmarks using the large dataset.

compared to RocksDB and traditional LSM trees. We measure the average request size for *Load A* using the large dataset. For writes, *Kreon* has an average request size of 94 KB compared to 333.2 KB for RocksDB. However, even at 94 KB, most SSDs exhibit high throughput with a large queue depth (Figure 3.1). For reads, *Kreon* produces 4 KB I/Os, compared to 126 KB for RocksDB. Because of compactions, RocksDB reads large chunks of data in order to merge them. This enables RocksDB to have large request size but it also results in high read amplification (4.8x more data compared to *Kreon*).

To examine randomness, we implement a lightweight I/O tracer as a stackable block device in the Linux kernel that keeps the device offset and size for *bios* issued to the underlying device. The tracer stores this information to a ramdisk to reduce overhead and avoid interfering with the key-value store I/O pattern. Tracing reduces average throughput of YCSB by about 10%. We analyze traces after each experiment and calculate a metric for I/O randomness based on the distance and size of successive *bios*, as follows:

$$R = \frac{\sum_{i=0}^{nb-1} |bs[i+1].off - (bs[i].off + bs[i].size)| + bs[i].size}{device\_size\_in\_pages * \sum_{i=0}^{nb-1} bs[i].size},$$

where *bs* is the array that contains bio information and *nb* its length. *R* is the randomness metric and takes values between [0,1]. The larger *R* is, the more random the I/O pattern. Finally, we compute three versions of *R*, one for all *bios* (R_t), one for reads (R_r), and one for writes (R_w).

Table 3.5 shows our results for *Kreon* and RocksDB. For calibration purposes, we run *fio* with queue depth of 1 and block size of 4 KB: a sequential pattern is 0 and a random pattern is close to 0.33. *Kreon* produces overall about 5.53x more random I/O patterns

	R_t	R_r	R_w
RocksDB	0.001780	0.003878	0.000112
Kreon	0.009851	0.033648	0.000325

Table 3.5: I/O randomness using the large dataset and *Load A*. The higher the value of R, the more random the I/O pattern.

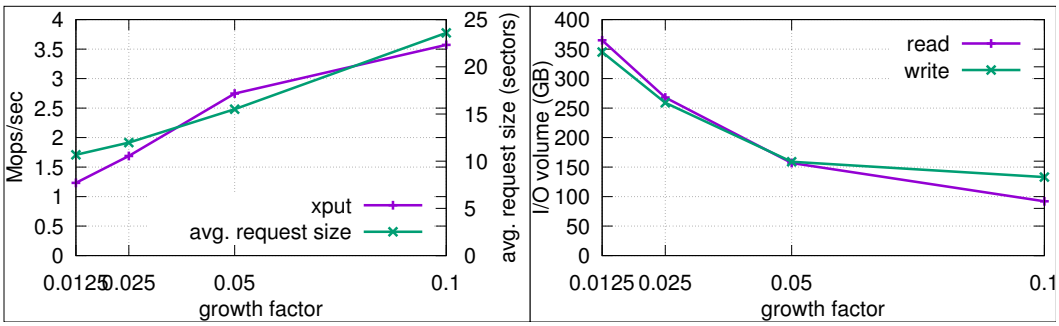
than RocksDB. Reads exhibit a larger difference in randomness, about 10x, because *Kreon* moves data between levels at smaller granularity than RocksDB. For writes, *Kreon* exhibits a 3x more random pattern.

Overall, during inserts, *Kreon* reduces write traffic by 2.8x and read traffic by 4.8x. In both cases, queue depth is about 30 on average. Figure 3.1 shows that, at this queue depth, commodity SSDs achieve their maximum throughput with at 32 KB requests, so *Kreon*'s 94 KB write requests result in little or no loss of device efficiency, while there is a 2.8x gain from reduced write traffic. For read traffic, *Kreon*'s 4K requests result in a small percentage drop of SSD throughput at a queue depth of 32, but at a 4.8x gain in traffic. Therefore, *Kreon* properly trades randomness and request size for amplification. The calculation is somewhat different for our NVMe devices, but still favorable to *Kreon*.

Finally, *Kreon* achieves an average read throughput of 123 MB/s and an average write throughput of 743 MB/s at an average queue depth of 21.2. On the other hand RocksDB achieves 707 MB/s for reads and 889 MB/s for writes at an average queue size of 26.2. In both cases queue depth is large enough for devices to operate at high throughput, although *Kreon* exhibits lower throughput for reads due to the smaller request sizes it generates. This loss of device efficiency is compensated by the reduced amplification (by 4.6x) and the reduced CPU overhead, eventually resulting in higher performance and data serving density.

Growth Factor and Commit Interval

An important parameter for key value stores that use multi-level indexes is the ratio of the size between two successive levels (growth factor). The growth factor in *Kreon* represents the amount of buffering that happens for inserts in one level before keys are spilled to



(a) xput & avg. request size (b) I/O volume

Figure 3.8: Results with varying growth factor from 1.25% to 10% (x-axis) using the large dataset.

the next level. This affects how effectively I/Os are amortized across several inserts and reduces write amplification.

Figure 3.8 shows *Load A* with varying growth factor using the large dataset. A growth factor of 0.1 means that L_1 is 10x larger than L_0 and therefore L_0 can buffer about 10% of the keys in L_1. Figure 3.8(b) shows that a growth factor between 0.05 and 0.1 is appropriate, meaning that each level should buffer between 5-10% of the next level. A smaller growth factor results in significant increase in traffic and reduces device efficiency. Increasing the growth factor beyond 0.1 reduces traffic further, however, this also requires more memory for L_0. Figure 3.8(a) (right y-axis) shows that average request size increases as buffering increases and combined with the reduced traffic, results in increasing throughput (ops/s), as shown in Figure 3.8(a) (left y-axis).

Figure 3.9 shows how the commit interval for L_0 affects ops/s, read volume, and write volume in *Kreon*. For *Run C* the commit interval does not affect any of the metrics, therefore, we examine only *Load A* with the large dataset.

Increasing the commit interval decreases the total amount of data read and written to the device. This is due to Copy-on-Write. For each commit we create a read-only version of our tree, thus an insert has to allocate new nodes and copy data from the immutable copy. Additionally, we see that commit intervals longer than 120s have a small impact for read and write volume.

For throughput, a small commit interval results in larger read and write volume which

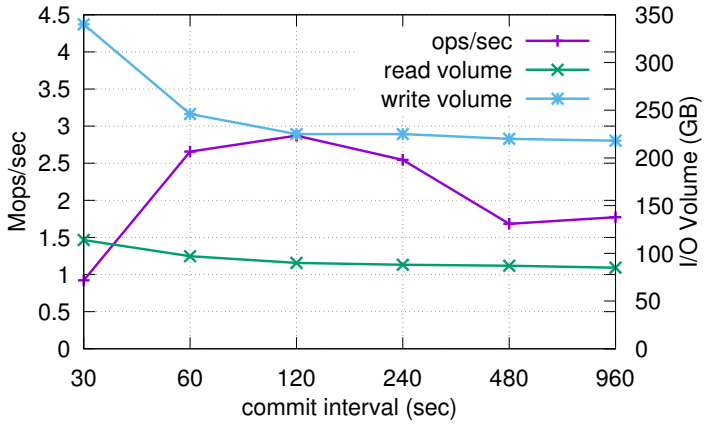


Figure 3.9: Results with varying the commit interval (x-axis) for *Load A* and the large dataset.

reduces performance. Interestingly, a value larger than 240 seconds reduces throughput significantly as well. This is due to the behavior of *msync*. In *kmmap*, *msync* is optimized to generate many large and asynchronous I/Os from all dirty pages, which means that it is more efficient compared to the eviction path *mmap* where we evict less amount of data. Overall, we see that a good value for the commit interval is about 2 minutes, which we use in all our other experiments.

3.4 Summary

We design *Kreon*, a persistent key-value store based on LSM trees that uses an index within each level to eliminate the need for sorting large segments and uses a custom *memory-mapped I/O* path to reduce the cost of I/O. *Kreon* reduces CPU overhead by up to 8.3x, I/O amplification by up to 4.6x at the expense of increasing randomness of I/Os. Both index organization and *memory-mapped I/O* contribute significantly to the reduction of CPU overhead, while index manipulation and page faults emerge as the main components of per operation cost in *Kreon*.

This chapter provides a way for user applications to define custom eviction and write-back operations over memory-mapped I/O. Our approach uses a priority-based FIFO replacement policy, and during memory pressure, a page with a higher priority is preferred

for eviction or writeback. To show the applicability of this approach, we use a multi-level persistent key-value store. The upper levels of our data structure have low priorities (i.e. hot data during insertions), whereas the lower levels have higher priorities (i.e. cold data during insertions). This approach shows significant improvements in terms of predictability, especially during writes. On the other hand, we observe substantial scalability issues in the memory-mapped I/O path of Linux. This limits performance, especially in multi-core servers that are common today in modern datacenters.

Chapter 4

Increasing Page Fault Concurrency

A major reason for the limited use of *memory-mapped I/O*, despite its advantages, has been that *mmap* may generate small and random I/Os. With modern storage devices, such as NVMe and persistent memory, this is becoming less of a concern. However, Figure 4.1 shows that the default *memory-mapped I/O* path (*mmap* backed by a device) for random page faults does not scale well with the number of cores. In this experiment (details in Section 4.4), we use *null_blk*, a Linux driver that emulates a block device but does not issue I/Os to a real device (we use 4TB dataset and 192GB of DRAM cache). Using *null_blk* allows us to stress the Linux kernel software stack while emulating a low-latency, next-generation storage device. Linux *mmap* scales up to only 8 cores, achieving 7.6 GB/s (2M random IOPS), which is about 5× less compared to a state-of-the-art device [65]; servers with multiple storage devices need to cope with significantly higher rates. We observe that from Linux kernel 4.14 to 5.4 the performance and the scalability of the *memory-mapped I/O* path has not improved significantly. Limited scalability also results in low device queue depth. Using the same micro-benchmark for random read page faults with 32 threads on an Intel Optane SSD DC P4800X, we see that the device queue depth is 27.6 on average. A large device queue depth is essential for fast storage devices to provide their peak device throughput.

In this chapter, we propose *FastMap*, a novel design for the *memory-mapped I/O* path that overcomes these two limitations of *mmap* for data intensive applications on multi-core servers with fast storage devices. *FastMap* (a) separates clean and dirty-trees to avoid

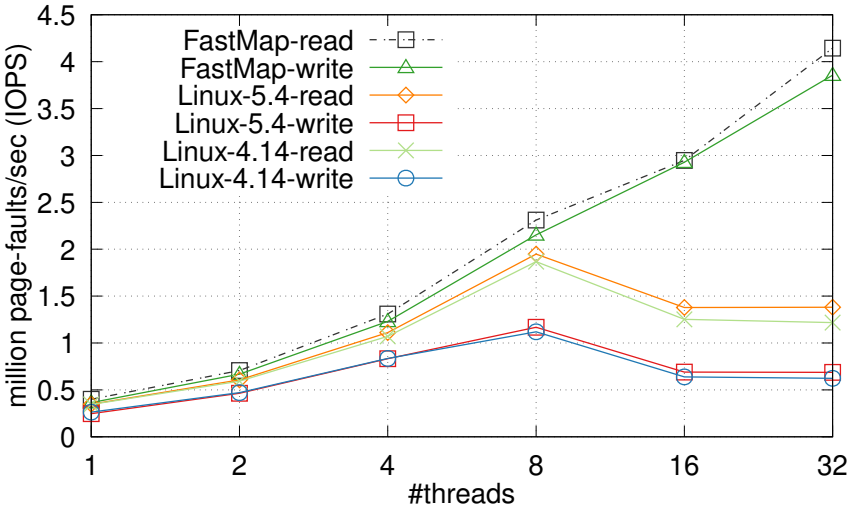


Figure 4.1: Scalability of random page faults using two versions of Linux *memory-mapped I/O* path (v4.14 & v5.4) and *FastMap*, over the *null_blk* device.

all centralized contention points, (b) uses full reverse mappings instead of Linux object-based reverse mappings to reduce CPU processing, and (c) introduces a scalable DRAM cache with per-core data structures to reduce latency variability. *FastMap* achieves both higher scalability and higher I/O concurrency by (1) avoiding all centralized contention points that limit scalability, (2) reducing the amount of CPU processing in the common path, and (3) using dedicated data-structures to minimize interference among processes, thus improving tail latency. As a further extension to *mmap*, we introduce a user-defined read-ahead parameter to proactively map pages in an application's address space and reduce the overhead of page faults for large sequential I/Os.

We evaluate *FastMap* using both micro-benchmarks and real workloads. We show that *FastMap* scales up to 80 cores and provides up to 11.8× more random IOPS compared to Linux *mmap* using *null_blk*. *FastMap* achieves 2× higher throughput on average for all YCSB workloads over Kreon [106], a persistent key-value store designed to use *memory-mapped I/O*. Moreover, we use *FastMap* to extend the virtual address space of memory intensive applications beyond the physical memory size over a fast storage device. In this case we achieve up to 75× lower average latency for TPC-C over Silo [128] and 5.27× better

performance when using the Ligra graph processing framework [120]. Finally, we achieve 6.06% higher throughput on average for all TPC-H queries over MonetDB [19] that mostly issues sequential I/Os.

In summary, our work optimizes the *memory-mapped I/O* path in the Linux kernel and makes the following three specific contributions:

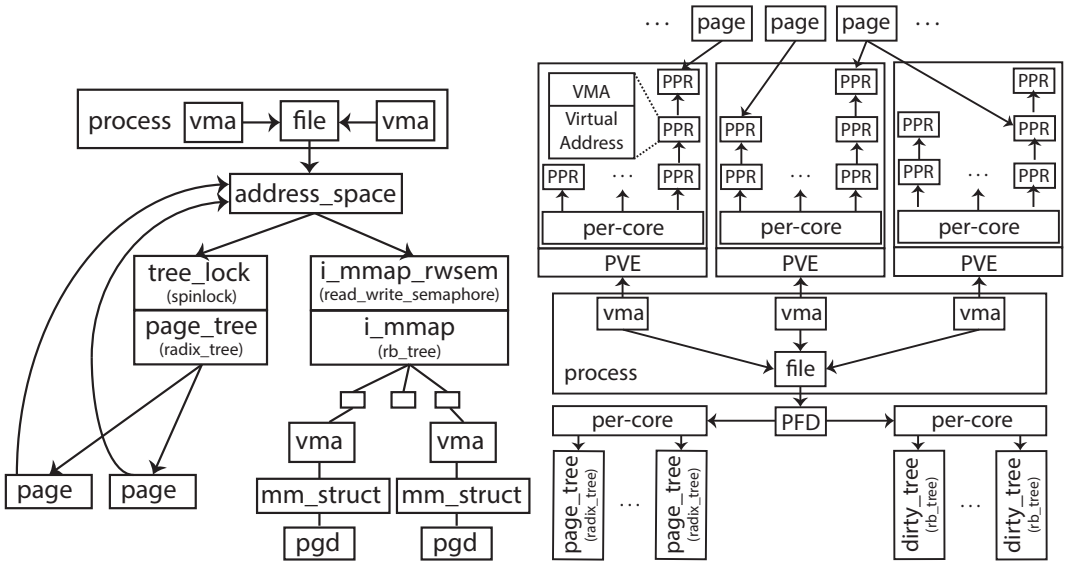
1. We identify severe performance bottlenecks of Linux *memory-mapped I/O* in multi-core servers with fast storage devices.
2. We propose *FastMap*, a new design for the *memory-mapped I/O* path.
3. We provide an experimental evaluation and analysis of *FastMap* compared to Linux *memory-mapped I/O* using both micro-benchmarks and real workloads.

The rest of the chapter is organized as follows. Section 4.1 provides the motivation behind *FastMap*. Section 4.2 presents the design of *FastMap* along with our design decisions. Sections 4.4 and 4.4.2 present our experimental methodology and results, respectively. Finally, Section 4.5 concludes this chapter.

4.1 Motivation

With storage devices that exhibit low performance for random I/Os, such as hard disk drives (HDDs), *mmap* results in small (4KB) random I/Os because of the small page size used in most systems today. In addition, *mmap* does not provide a way for users to manage page writebacks in the case of high memory pressure, which leads to unpredictable tail latencies [106]. Therefore, historically the main use of *mmap* has been to load binaries and shared libraries into the process address space; this use-case does not require frequent I/O, uses read-mostly mappings, and exhibits a large number of shared mappings across processes, e.g. libc is shared by almost all processes of the system. Reverse mappings provide all page table translations for a specific page and they are required in order to unmap a page during evictions. Therefore, Linux *mmap* uses object-based reverse mappings [94] to reduce memory consumption and enable fast *fork* system calls, as they do not require copying full reverse mappings.

With the introduction of fast storage devices, where the throughput gap between ran-



(a) Linux (b) *FastMap*

Figure 4.2: Linux (left) and *FastMap* (right) high-level architecture for memory-mapped files (acronyms: **PFD**=**P**er-**F**ile-**D**ata, **PVE**=**P**er-**V**ma-**E**ntry, **PPR**=**P**er-**P**ve-**R**map).

dom and sequential I/O is small, *memory-mapped I/O* has the potential to reduce I/O path overhead in the kernel, which is becoming the main bottleneck for data-intensive applications. However, data intensive applications, such as databases or key-value stores, have different requirements compared to loading binaries: they can be write-intensive, do not require large amount of sharing, and do not use *fork* system calls frequently. These properties make the use of full reverse mappings a preferred approach. In addition, data intensive applications use datasets that do not fit in main memory and thus, the path of reading and writing a page from the device becomes the common case. Most of these applications are also heavily multithreaded and modern servers have a large number of cores.

These trends and characteristics make the design of *FastMap* appropriate for data-intensive applications in multi-core servers with fast storage devices, as discussed next.

4.2 Design

The Linux kernel provides the *mmap* and *munmap* system calls to create and destroy memory mappings. Linux distinguishes memory mappings in shared vs. private. Mappings can also be anonymous, i.e. not backed by a file or device. Anonymous mappings are used for memory allocation. In this chapter we examine I/O over persistent storage, an inherently shared resource. Therefore, we consider only shared memory mappings backed by a file or block device, as also required by Linux *memory-mapped I/O*.

Figure 4.2(a) shows the high-level architecture of shared memory mappings in the Linux kernel. Each virtual memory region is represented by a *struct vm_area_struct* (*VMA*). Each *VMA* points to a *struct file* (*file*) that represents the backing file or device and the starting offset of the memory mapping to it. Each *file* points to (a shared between processes) *struct address_space* (*address_space*) which contains information about mapped pages and the backing file or device.

Figure 4.2(b) illustrates the high-level design of *FastMap*. The most important components in our design are *per_file_data* (*PFD*) and *per_vma_entry* (*PVE*). Combined, these two components provide equivalent functionality as the Linux kernel *address_space* structure. Each *file* points to a *PFD* and each *VMA* points to a *PVE*. The role of a *PFD* is to keep metadata about device blocks that are in the *FastMap* cache and metadata about dirty pages. *PVE* provides full reverse mappings.

4.2.1 Separate Clean and Dirty Trees in *PFD*

In Linux, one of the most important parts of *address_space* is *page_tree*, a radix tree that keeps track of all pages of a cacheable and mappable file or device, both clean and dirty. This data structure provides an effective way to check if a device block is already in memory when a page fault occurs. Lookups are lock-free (*RCU*) but inserts and deletes require a spinlock (named *tree_lock*). Linux kernel radix trees also provide user-defined *tags* per entry. A *tag* is an integer, where multiple values can be stored using bitwise operations. In this case *tags* are used to mark pages as dirty. Marking a previously read-only page as writable requires holding the *tree_lock* to update the *tag*.

Using the experiments of Figure 4.1 and *lockstat* we see that *tree_lock* is by far the most contended lock: Using the same multithreaded benchmark as in Figure 4.1, over a single memory mapped region, *tree_lock* has 126× more contended lock acquisitions, which involve cross-cpu data, and 155× more time waiting to acquire the lock, compared to the second most contended lock. The second more contended lock is a spinlock that protects concurrent modifications in *PTEs* (4th level entries in the page table). This design has remained essentially unchanged from Linux kernel 2.6 up to 5.4 (latest stable version at the time of this writing).

To remove the bottleneck in *tree_lock*, *FastMap* uses a new structure for per-file data, *PFD*. The most important aspects of *PFD* are: (i) a per-core radix tree (*page_tree*) that keeps all (clean and dirty) pages and (ii) a per-core red-black tree (*dirty_tree*) that keeps only dirty pages. Each of these data structures is protected by a separate (per core) spinlock, different for the radix and red-black trees. We assign pages to cores in a round-robin manner and we use the page offset to identify the per-core structure that holds each page.

We use *page_tree* to provide lock-free lookups (RCU), similar to the Linux kernel. We use per-core data structures to reduce contention in the case we need to add or remove a page from it. On the other hand, we do not use *tags* to mark pages as dirty but we use the *dirty_tree* for this purpose. In the case where we have to mark a previously read-only page as read-write, we only acquire the appropriate lock of *dirty_tree* without performing any additional operations to the *page_tree*. Furthermore, having all dirty pages in a sorted data structure (red-black tree) enables efficient I/O merging for the cases of writebacks and the *msync* system call.

4.2.2 Full Reverse Mappings in *PVE*

Reverse (inverted) mappings are also an important part of *mmap*. They are used in the case of evictions and writebacks and they provide a mechanism to find all existing virtual memory mappings of a physical page. File-backed memory mappings in Linux use object-based reverse mappings [94]. The main data structure for this purpose is a red-black tree, *i_mmap*. It contains all *VMAs* that map at least one page of this *address_space*. A read-write

semaphore, *i_mmap_rwsem*, protects concurrent accesses to the *i_mmap* red-black tree. The main function that removes memory mappings for a specific page is *try_to_unmap*. Each page has two fields for this purpose: (i) a pointer to the *address_space* that belongs to and (ii) an atomic counter (*_mapcount*) that keeps the number of active page mappings. Using the pointer to *address_space*, *try_to_unmap* gets access to *i_mmap* and then iterates over all *VMAs* that belong to this mapping. Through each VMA, it has access to *mm_struct* which contains the root of the process page table (*pgd*). It calculates the virtual address of the mapping based on the *VMA* and the page, which is required for traversing the page table. Then it has to check all active *VMAs* of *i_mmap* if the specific page is mapped, which results in many useless page table traversals. This is the purpose of *_mapcount*, which limits the number of traversals. This strategy is insufficient in some cases but it requires a very small amount of memory for the reverse mappings. More specifically, in the case where *_mapcount* is greater than zero, we may traverse the page table for a *VMA* where the requested page is not mapped. This can happen in the case where a page is mapped in several different *VMAs* in the same process, i.e. with multiple *mmap* calls, or mapped in the address space of multiple different processes. In such a case, we have unnecessary page table traversals that introduce overheads and consume CPU cycles. Furthermore, during this procedure, *i_mmap_rwsem* is held as a read lock and as a write lock only during *mmap* and *munmap* system calls. Previous research shows that even a read lock can limit scalability in multicore servers [32].

The current object-based reverse mappings in Linux have two disadvantages: (1) with high likelihood they result in unnecessary page table traversals, originating from *i_mmap*, and (2) they require a coarse grain read lock to iterate *i_mmap*. Other works have shown that in multi-core servers locks can be expensive, even for read-write locks when acquired as read locks [32]. These overheads are more pronounced in servers with a NUMA memory organization [24].

To overcome these issues *FastMap* provides finer grained locking, as follows: *FastMap* uses a structure with an entry for each VMA, *PVE*. Each *PVE* keeps a per-core list of all pages that belong to this *VMA*. A separate (per core) spinlock protects each of these lists. The lists are append-only as unmapping a page from a different page table does not re-

quire any ordering. We choose the appropriate list based on the core that runs the append operation (*smp_processor_id()*). These lists contain *per_pve_rmap* (*PPR*) entries. Each *PPR* contains a tuple (*VMA*, *virtual_address*). These metadata are sufficient to allow iterating over all mapped pages of a specific memory mapping in the case of an *munmap* operation. Furthermore, each page contains an append-only list of active *PPRs*, which are shared both for *PVEs* and pages. This list is used when we need to evict a page that is already mapped in one or more address spaces, in the event of memory pressure.

4.2.3 Dedicated DRAM Cache

An *mmap address_space* contains information about the backing file or device and the required functions to interact with the device in case of page reads and writes. To write back a set of pages of a memory mapping, Linux iterates *page_tree* in a lock-free manner with RCU and writes only the pages that have the dirty tag enabled. Linux also keeps a per-core LRU to find out which pages to evict. In the case of evictions, Linux tries to remove clean pages in order not to wait for dirty pages to do the writeback [94].

The Linux page-cache is tightly coupled with the swapper. For the *memory-mapped I/O* path, this dependency results in unpredictable evictions and bursty I/O to the storage devices [106]. Therefore, *FastMap* implements its own DRAM cache, managing evictions via an approximation of LRU. *FastMap* has two types of LRU lists: one containing only clean pages (*clean_queue*) and one containing only dirty pages (*dirty_queue*). *FastMap* maintains per-core *clean_queues* to reduce lock contention. We identify the appropriate *clean_queue* as clean_queue_id = page_offset % num_cores.

When there are no free pages during a page fault, *FastMap* evicts only clean pages, similar to the Linux kernel [94], from the corresponding *clean_queue*. We evict a batch (with a configurable size, currently set to 512) of clean pages to amortize the cost of page table manipulation and TLB invalidations. Each page eviction requires a TLB invalidation with the *flush_tlb* function, if the page mapping is cached. *flush_tlb* sends an IPI (Inter-Processor-Interrupt) to all cores, incurring significant overheads and limiting scalability [3, 4]. We implement a mechanism to reduce the number of calls to *flush_tlb* function, using batch-

ing, as follows.

A TLB invalidation requires a pointer to the page table and the *page_offset*. *FastMap* keeps a pointer to the page table and a range of *page_offsets*. Then, we invoke *flush_tlb* for the whole range. This approach may invalidate more pages, but reduces the number of *flush_tlb* calls by a factor of the batch-size of page evictions (currently 512). As the file mappings are usually contiguous in the address space in data intensive applications, in the common case false TLB invalidations are infrequent. Thus, *FastMap* manages to maintain a high number of concurrent I/Os to devices and increase device throughput.

FastMap uses multiple threads to write dirty pages to the underlying storage device (writeback). Each of these manages its own *dirty_queue*. This design removes the need of synchronization when we remove dirty pages from a *dirty_queue*. During writebacks, *FastMap* merges consecutive I/O requests to generate large I/O operations to the underlying device. To achieve this, we use *dirty_trees* that keep dirty pages sorted based on the device offset. As we have multiple *dirty_trees*, we initialize an iterator for each tree and we combine the iterator results using a min-max heap. When a writeback occurs, we also move the page to the appropriate *clean_queue* to make it available for eviction. As page writeback also requires a TLB invalidation, we use the same mechanism as in the eviction path to reduce the number of calls to the kernel *flush_tlb* function. Each writeback thread checks the amount of dirty pages compared to clean pages and starts the writeback when the percentage of dirty pages is more that 75% of the total cache pages. The cache in *FastMap* currently uses a static memory buffer, allocated upon module initialization and does not create any further memory pressure to the Linux page cache. We also provide a way to grow and shrink this cache at runtime, but we have not yet evaluated alternative sizing policies.

To keep track of free pages *FastMap* uses a per-core free list with a dedicated spinlock. During a major page fault i.e., when the page does not reside in the cache, the faulting thread first tries to get a page from its local free list. If this fails, it tries to steal a page from another core's free list (randomly selected). After *num_cores* unsuccessful tries, *FastMap* forces page evictions to cleanup some pages. To maintain all free lists balanced, each evicted page is added to the free list from which we originally obtained the page.

Overall, *FastMap* with per-core data structures requires more memory compared to the native Linux *mmap*. *FastMap* requires a single *PFD* per file for all memory mappings. A single *PVE* is about 512 bytes and a single *PPR* is 24 bytes. We require a single *PVE* for each mmap call, i.e. $1:1$ with the Linux VMA struct. *FastMap* requires a single *PPR* entry per *PVE* for each mapped page, independently of how many threads access the same page. In the setups we target, there is little sharing of files across processes and we can therefore, assume that we only need one *PPR* entry for each page in our DRAM cache. *FastMap* targets storage servers with large memory spaces and can be applied selectively for the specific mount points that hold the files of data-intensive applications. While it is, in principle, possible to allow more fine-grain uses of *FastMap* in Linux, we leave this possibility for future work.

Finally, the Linux kernel also provides private, file-backed memory mappings. These are Copy-On-Write mappings and writes to them do not reach the underlying file/device. Such mappings are outside the scope of this chapter, but they share the same path in the Linux kernel to a large extent. Our proposed techniques also apply to private file-backed mappings. However, these mappings are commonly used in Linux kernel to load binaries and shared libraries, resulting in a large degree of sharing. We believe that it is not beneficial to use the increased amount of memory required by *FastMap* to optimize this relatively uncommon path.

4.3 Implementation

Figure 4.3 shows the I/O path in the Linux kernel and indicates where *FastMap* is placed. *FastMap* is above *VFS* and thus is independent of the underlying file system. This means that common file systems such as *XFS*, *EXT4*, and *BTRFS* [1] can benefit from our work.

FastMap provides a user interface for accessing both a block device but also a file system through a user-defined mount point. For the block device case, we implement a virtual block device that uses our custom *mmap* function. All other block device requests (e.g. *read*/*write*) are forwarded to the underlying device. Requests for fetching or evicting

[1] *FastMap* has been successfully tested with all of these file systems.

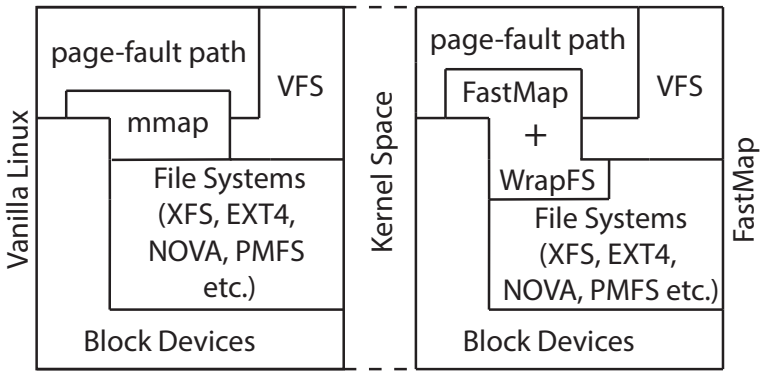


Figure 4.3: FastMap I/O path.

pages from *FastMap* are issued directly to the underlying device.

For the file system implementation we use WrapFS [142], a stackable file system that intercepts all *mmap* calls to a specific mount point so that *FastMap* is used instead of the native Linux *mmap* implementation. For fetching or evicting pages from within *FastMap* we use direct I/O to the underlying file system, bypassing the Linux page cache. All other file system calls are forwarded to the underlying file system.

4.4 Experimental Analysis

In this section, we present the experimental methodology we use to answer the following questions:

1. How does *FastMap* perform compared to Linux *mmap*?
2. How much does *FastMap* improve storage I/O?
3. How sensitive is *FastMap* to (a) file system choice and (b) false TLB invalidations?

4.4.1 Methodology

Our main testbed consists of a dual-socket server that is equipped with two Intel(R) Xeon(R) CPU E5-2630 v3 CPUs running at 2.4 GHz, each with 8 physical cores and 16 hyper-threads for a total of 32 hyper-threads. The primary storage device of the server is a PCIe-attached

Intel Optane SSD DC P4800X series [63] with 375 GB capacity. For the purposes of evaluating scalability, we use an additional four-socket server. This four-socket server is equipped with four Intel(R) Xeon(R) CPU E5-4610 v3 CPUs running at 1.7 GHz, each with 10 physical cores and 20 hyper-threads for a total of 80 hyper-threads. Both servers are equipped with 256 GB of DDR4 DRAM at 2400 MHz and run CentOS v7.3, with Linux kernel 4.14.72.

During our evaluation we limit the available capacity of DRAM (using a kernel boot parameter) as required by different experiments. In our evaluation we use datasets that both fit and do not fit in main memory. This allows us to provide a more targeted evaluation and separate the costs of the page-fault path and the eviction path. To reduce variability in our experiments, we disable swap and Transparent Huge Pages (THP), and we set the CPU scaling governor to "performance". In experiments where we want to stress the software path of the Linux kernel we also use the *null_blk* [100] and *pmem* [111] block devices. *null_blk* emulates a block device but ignores the I/O requests issued to it. For *null_blk* we use the bio-based configuration. *pmem* emulates a fast block device that is backed by DRAM.

In our evaluation we first use a custom multithreaded microbenchmark. It uses a configurable number of threads that issue *load*/*store* instructions at randomly generated offsets within the memory mapped region. We ensure that each *load*/*store* results in a page fault.

Second, we use a persistent key-value store. We choose Kreon [106], a state-of-the-art persistent key-value store that is designed to work with *memory-mapped I/O*. The design of Kreon is similar to the LSM-tree, but it maintains a separate B-Tree index per-level to reduce I/O amplification. Kreon uses a log to keep user data. It uses *memory-mapped I/O* to perform all I/O between memory and (raw) devices. Furthermore, it uses Copy-On-Write (COW) for persistence, instead of Write-Ahead-Logging. Kreon follows a single-writer, multiple-reader concurrency model. Readers operate concurrently with writers using Lamport counters [83] per node for synchronization to ensure correctness. For inserts and updates, it uses a single lock per database; however, by using multiple databases Kreon can support concurrent updates.

To improve single-database scalability in Kreon and make it more suitable for evalu-

	Workload
A	50% reads, 50% updates
B	95% reads, 5% updates
C	100% reads
D	95% reads, 5% inserts
E	95% scans, 5% inserts
F	50% reads, 50% read-modify-write

Table 4.1: Standard YCSB Workloads.

ating *FastMap*, we implement the second protocol that Bayer et al. propose [12]. This protocol requires a read-write lock per node. It acquires the lock as a read lock in every traversal from the root node to a leaf. In the case of inserts or rebalance operations it acquires the corresponding lock as a write lock. As every operation has to acquire the root node read lock, this limits scalability [32]. To overcome this limitation, we replace the read/write lock of the root node with a Lamport counter and a lock. Every operation that modifies the root node acquires the lock, changes the Lamport counter, performs a COW operation, and then writes the update in the COW node.

For Kreon we use the YCSB [37] workloads and more specifically a C++ implementation [114] to remove overheads caused by the JNI framework, as *Kreon* is highly efficient and is designed to take advantage of fast storage devices. Table 4.1 summarizes the YCSB workloads we use. These are the proposed workloads, and we execute them in the author's proposed sequence [37], as follows: LoadA, RunA, RunB, RunC, RunF, RunD, clear the database, and then LoadE, RunE.

Furthermore, we use Silo [128], an in-memory database that also provides scalable transactions for modern multicore machines. In this case, we use TPC-C [126], a transactional benchmark, which models a retail operation and is a common benchmark for OLTP workloads. We also use Ligra [120], a lightweight graph processing framework for shared memory with OpenMP-based parallelization. Specifically, we use the Breadth First Search (BFS) algorithm. We use Silo and Ligra to evaluate *FastMap*'s effectiveness in extending the virtual address space of an application beyond physical memory over fast storage devices. For this reason we convert all *malloc*/*free* calls of Ligra and Silo to allocate space over a memory-mapped file on a fast storage device. We use the *libvmmalloc* library from

the Persistent Memory Development Kit (PMDK) [109]. *libvmmalloc* transparently converts all dynamic memory allocations to persistent memory allocations. This allows the use of persistent memory as volatile memory without modifying the target application. The memory allocator of *libvmmalloc* is based on *jemalloc* [68].

Finally, we evaluate *FastMap* using MonetDB-11.31.7 [19, 99], a column-oriented DBMS that is designed to use *mmap* to access files instead of using the read/write API. We use the TPC-H [127] benchmark, a warehouse read-mostly workload.

We run all experiments three times and we report the averages. In all cases the variation observed across runs is small.

4.4.2 Experimental Results

How does *FastMap* perform compared to Linux *mmap*?

Microbenchmarks: Figure 4.1 shows that Linux *mmap* fails to scale beyond eight threads on our 32-core server. *FastMap* provides 3.7× and 6.6× more random read and write IOPS, respectively, with 32 threads compared to Linux *mmap*. Furthermore, both versions 4.14 and 5.4 of the Linux kernel achieve similar performance. To further stress *FastMap*, we use our 80-core server and the *null_blk* device. Figure 4.4 shows that with 80 threads, *FastMap* provides 4.7× and 7× higher random read and write IOPS respectively, compared to Linux *mmap*. Furthermore, in both cases *FastMap* performs up to 38% better even in the case where there is little or no concurrency, when using a single thread.

Figure 4.4 shows that not only *FastMap* scales better compared to Linux *mmap*, but also that *FastMap* sustains more page faults per second. On the other hand *FastMap* does not achieve perfect scalability. For this reason, we profile *FastMap* using the random read page faults microbenchmark. We find that the bottleneck is the read-write lock (*mmap_sem*) that protects the red-black tree of active *VMAs*. This is the problem that *Bonsai* [32] tackles. Specifically, with 10 cores the cost of *read_lock* and *read_unlock* is 7.6% of the total execution time, with 20 cores it becomes 25.4%, with 40 cores 32%, and with 80 cores 37.4%. To confirm this intuition, we apply Speculative Page Faults (SPF) [44], an attempt to use SRCU (Sleepable RCU) instead of the read-write lock to protect the red-

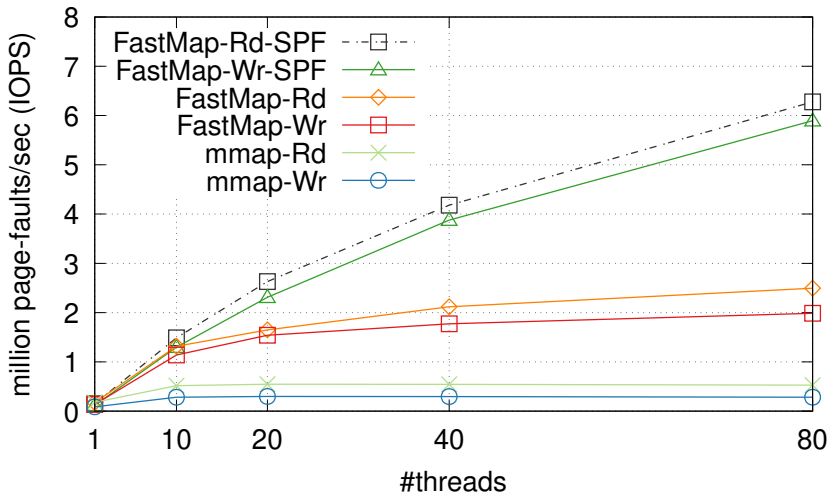


Figure 4.4: Scalability of random page faults for Linux and *FastMap*, with up to 80 threads, using the *null_blk* device.

black tree of active *VMAs*, an approach similar to *Bonsai*. We use the Linux kernel patches from [43] as at the time of this writing it has not been merged in the Linux mainline. As SPF works only for anonymous mappings, we modify it to use *FastMap* for block-device backed memory-mappings. Figure 4.4 shows that *FastMap* with *SPF* provides even better scalability: 2.51× and 11.89× higher read IOPS compared to *FastMap* without SPF and to Linux kernel, respectively. We do not provide an evaluation of *SPF* without *FastMap* as it (1) works only for anonymous mappings and (2) it could use the same Linux kernel path that has scalability bottlenecks, as we show in Section 4.2.1.

Figure 4.5 shows the breakdown of the execution time for both random reads and writes. We profile these runs using *perf* at 999Hz and plot the number of samples (y axis) that *perf* reports. First, we see that for random reads Linux *mmap* spends almost 80% of the time in manipulating the *address_space* structure, specifically in the contented *tree_lock* that protects the *radix_tree* which keeps all the pages of the mapping (see Section 4.2). In *FastMap* we do not observe a single high source of overhead. In the case of writes the overhead of this lock is even more pronounced in Linux *mmap*. For each page that is converted from read-only to read-write, Linux has to acquire this lock again

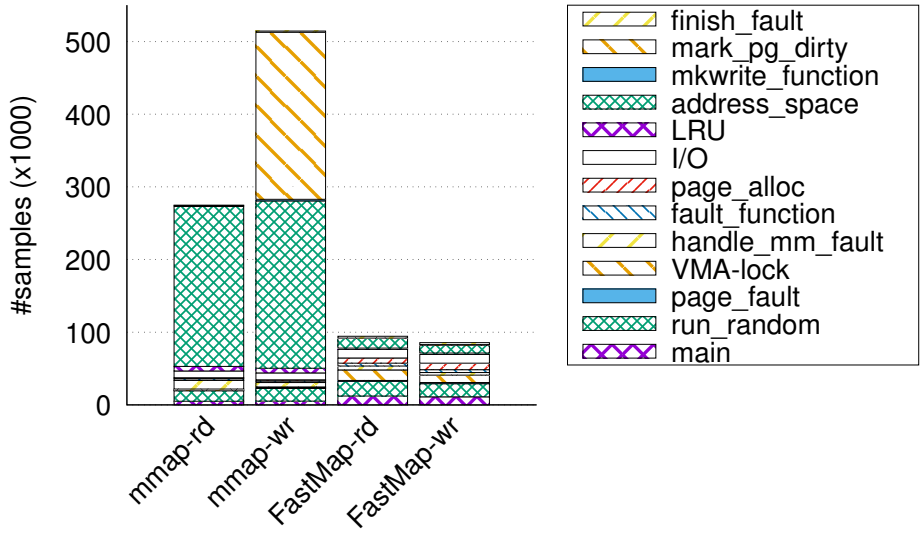


Figure 4.5: *FastMap* and Linux *mmap* breakdown for read and write page faults, with *null_blk* and 32 cores.

to set the *tag*. *FastMap* removes this contention point as we keep metadata about dirty pages only in the per-core red-black trees (Section 4.2.3). Therefore, we do not modify the *radix_tree* upon the conversion of a read-only page to a read-write page.

Figure 4.6 shows how each optimization in *FastMap* affects I/O performance. Vanilla is the Linux *mmap* and *basic* is *FastMap* with all the optimizations disabled, except the per-core red-black tree. The *per-core radix-tree* optimization is important, because with increasing core counts on modern servers (Section 4.2.1) the single *radix tree* lock is by far the most contended lock. Furthermore, *per-core cleanQ* enables the per-core LRU list for clean pages. The *per-core freelists* optimization allows for scalable page allocation, resulting in significant performance gains. Finally, the main purpose of *per-core dirtyQ* is to enable higher concurrency when we convert a page from read-only to read-write and allow for multiple eviction threads with minimal synchronization. This optimization mainly improves the write path, as is shown in Figure 4.6.

In-memory Graph Processing: We evaluate *FastMap* as a mechanism to extend the virtual address space of an application beyond the physical memory and over fast storage

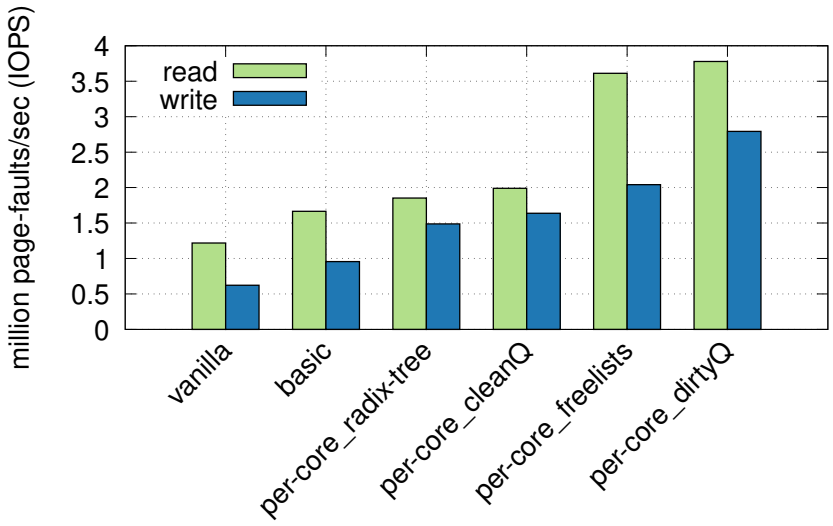


Figure 4.6: Performance gains from different optimizations in *FastMap*, as compared to "vanilla" Linux using *null_blk* and 32 cores.

devices. Using *mmap* (and *FastMap*) a user can easily map a file over fast storage and provide an extended address space, limited only by device capacity. We use Ligra [120], a graph processing framework, a demanding workload in terms of memory accesses and commonly operating on large datasets. Ligra assumes that the dataset (and metadata) fit in main memory. For our evaluation we generate a R-Mat [26] graph of 100M vertices, with the number of directed edges is set to 10× the number of vertices. We run BFS on the resulting 18GB graph, thus generating a read-mostly random I/O pattern. Ligra requires about 64GB of DRAM throughout execution. To evaluate *FastMap* and Linux *mmap*, we limit the main memory of our 32-core server to 8 GB and we use the Optane SSD device.

Figure 4.7 shows that BFS completes in 6.42s with *FastMap* compared to 21.3s with default *mmap* and achieves a 3.31× improvement. *FastMap* requires less than half the system time (10.3% for *FastMap* vs. 27.38% for Linux) and stresses more the underlying storage device as seen in iowait time (19.31% for *FastMap* vs. 9.5% for Linux). This leaves 2.11× more user-time available for the Ligra workload execution. Using a *pmem* device the benefits of *FastMap* are even higher. Linux *mmap* requires 21.9s for BFS, while *FastMap* requires only 4.15s, i.e. a 5.27× improvement. Overall, Ligra induces a highly concurrent

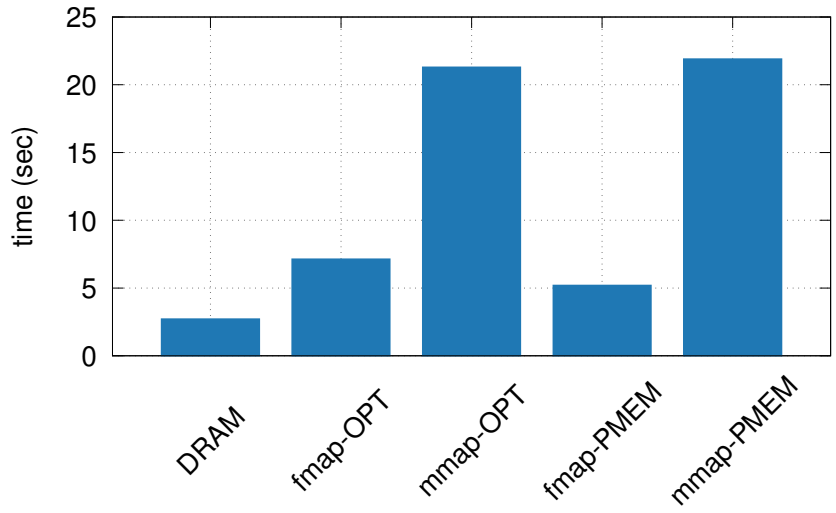


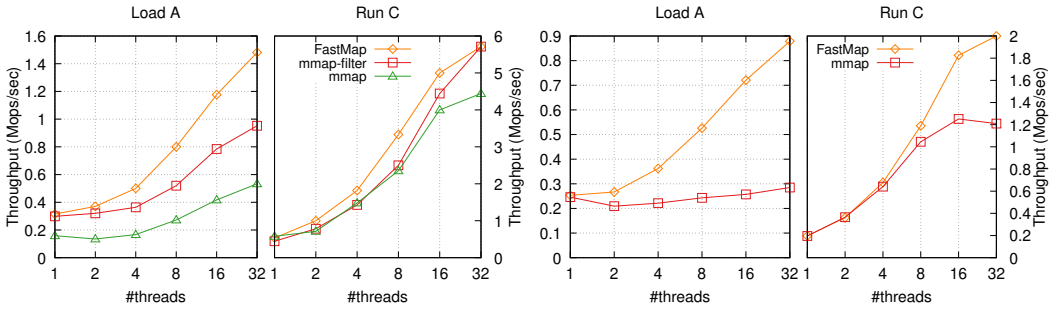
Figure 4.7: Execution time for Ligra running BFS with 32 threads and using an Optane SSD and a *pmem* device.

I/O pattern that stresses the default *mmap*, resulting in lock contention as described in Section 4.2.1 and as evidenced by the increased system time. The default *mmap* results in a substantial slowdown, even with a *pmem* device that has throughput comparable to DRAM.

How much does *FastMap* improve storage I/O?

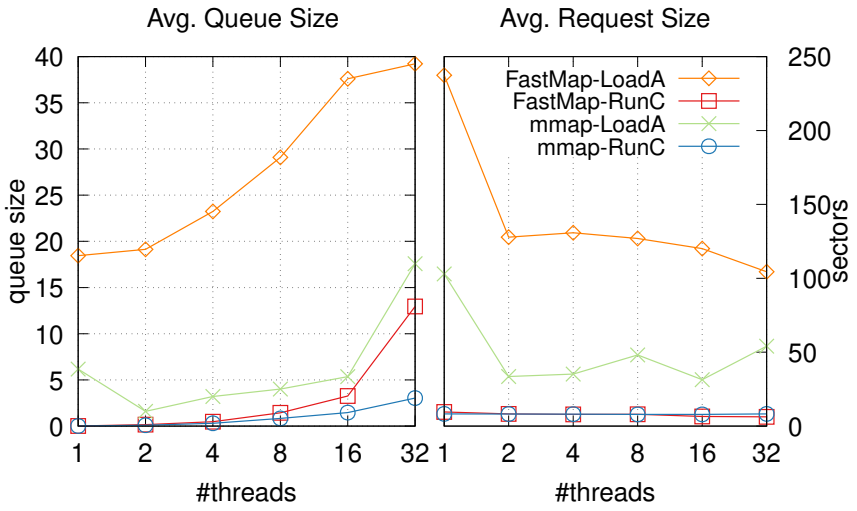
Kreon Persistent Key-value Store: In this section we evaluate *FastMap* using Kreon, a persistent key-value store that uses *memory-mapped I/O* and a dataset of 80M records. The keys are 30 bytes long, with 1000 byte values. This results in a total footprint of about 76GB. We issue 80M operations for each of the YCSB workloads. For the in-memory experiment, we use the entire DRAM space (256GB) of the testbed, whereas for the out-of-memory experiment we limit available memory to 16GB. In all cases we use the Optane SSD device.

Figure 4.8(a) illustrates the scalability of Kreon, using *FastMap*, Linux *mmap*, and *mmap-filter*, with a dataset that fits in main memory. The *mmap-filter* configuration is the default Linux *mmap* implementation augmented with a custom kernel module we have created to



(a) In-memory

(b) Out-of-memory



(c) Average queue & request size

Figure 4.8: Kreon scalability with increasing the number of threads (*(a)* and *(b)*). Average queue size and average request size for an out-of-memory experiment *(c)*. In all cases we use the Optane SSD.

remove the unnecessary read I/O from the block device for newly allocated pages. Using 32 threads (on the 32-core server), *FastMap* achieves 1.55× and 2.77× higher throughput compared to *mmap-filter* and *mmap* respectively, using the LoadA (insert only) workload. Using the RunC (read only) workload, *FastMap* achieves 9% and 28% higher throughput compared to *mmap-filter* and *mmap* respectively. As we see *mmap-filter* performs always better, therefore, for the rest of the Kreon evaluation we use this configuration as our baseline.

Figure 4.8(b) shows the scalability of Kreon with *FastMap* and *mmap-filter* (denoted as

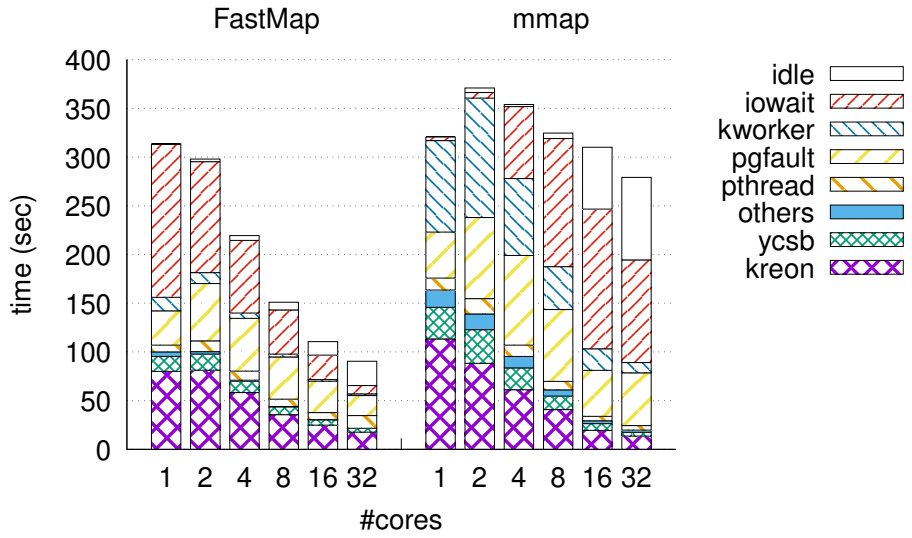


Figure 4.9: Kreon breakdown using *FastMap* and Linux *mmap* for an out-of-memory experiment for LoadA YCSB workload, with an increasing number of cores, an equal number of YCSB threads, and the Optane SSD.

mmap) using a dataset that does not fit in main memory. Using 32 threads (on the 32-core server) *FastMap* achieves 3.08× higher throughput compared to *mmap* using LoadA (insert only) workload. Using the RunC (read only) workload, *FastMap* achieves 1.65× higher throughput compared to *mmap*. We see that even for the lower core counts, *FastMap* outperforms *mmap* significantly. Next, we provide an analysis on what affects scalability in *mmap* and how *FastMap* behaves with an increasing number of cores.

Figure 4.9 shows the execution time breakdown for the out-of-memory experiment with an increasing number of threads for LoadA. *kworker* denotes the time spent in the eviction threads both for Linux *mmap* and *FastMap*. *pthread* refers to pthread locks, both mutexes and read-write locks as described in Section 4.4. First, we observe here that in the case of Linux *mmap* both *iowait* and *idle* time increases. For *iowait* time, the small queue depth that *mmap* generates (discussed in detail later) leads to sub-optimal utilization of the storage device. Furthermore, the idle time comes from sleeping in mutexes in the Linux kernel. We also observe that the *pgfault* time is lower in *FastMap* and this is more pronounced with 32 threads. In summary, the optimized page-fault path results in 2.64×

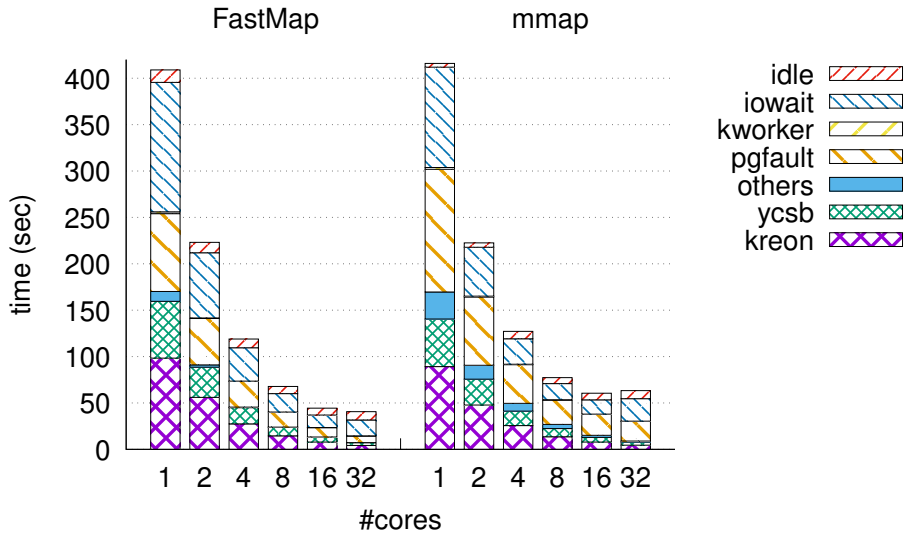


Figure 4.10: Kreon breakdown using *FastMap* and Linux *mmap* for an out-of-memory experiment with the RunC YCSB workload, with increasing number of cores (and equal number of YCSB threads) and the Optane SSD.

lower *pgfault* time and 12.3× lower *iowait* time due to higher concurrency and larger average request size. In addition, the optimized page-fault path results in 3.39× lower idle time due to spinning instead of sleeping in the common path. This is made possible as we apply per-core locks to protect our data structures, which are less contended in the common case. Similar to the previous figure, Figure 4.10 shows the same metrics for RunC. In this case the breakdown is similar both for *FastMap* and Linux *mmap*. With 32 threads the notable differences are in *pgfault* and *iowait*. Linux *mmap* spends 2.88× and 1.41× more time for *pgfault* and *iowait*, respectively. The difference in *pgfault* comes from our scalable design for the *memory-mapped I/O* path. As in this case both systems always issue 4KB requests (page size) the difference in *iowait* comes from the higher queue depth achieved on average by *FastMap*.

Figure 4.8(c) shows the average queue depth and average request size for both *FastMap* and Linux *mmap*. Using 32 threads, *FastMap* produces higher queue depths for both LoadA and RunC, which is an essential aspect for high throughput with fast storage devices. With 32 threads in LoadA *FastMap* results in an average queue size of 39.2, while

Linux *mmap* results in an average queue size of 17.5. Furthermore, *FastMap* also achieves a larger request size of 100.2 sectors (51.2KB) compared to 51.8 sectors (26.5KB) for Linux *mmap*. For RunC, the average request size is 8 sectors (4KB) for both *FastMap* and Linux *mmap*. However, *FastMap* achieves (with 32 threads) an average queue size of 13 compared to 3 for Linux *mmap*.

For all YCSB workloads, Kreon using *FastMap* outperforms Linux *mmap* between 1.25–3.65× and 2.48× on average.

MonetDB: In this section we use TPC-H over MonetDB, a column oriented DBMS that uses *memory-mapped I/O* instead of *read*/*write* system calls. We focus on out-of-memory workloads, using a TPC-H dataset with a scale factor $SF = 50$ (around 50GB in size). We limit available server memory to 16GB and we use the Optane SSD device. In all 22 queries of TPC-H, system-time is below 10%. The use of *FastMap* decreases further the system time (between 5.4% and 48.6%) leaving more CPU cycles for user-space processing. In all queries, the improvement on average is 6.06% (between −7.2% and 45.7%). There are 4 queries where we have a small decrease in performance. Using profiling we see that this comes from the *map_pages* function that is responsible for the fault-around page mappings, and which is not as optimized in the current prototype. In some cases we see greater performance improvements compared to the reduction in system time. This comes from higher concurrency to the devices (I/O depth) which also results in higher read throughput. Overall, our experiments with MonetDB show that a complex real-life memory-based DBMS can benefit from *FastMap*. The queries produce a sequential access pattern to the underlying files which shows the effectiveness of *FastMap* also for this case.

How sensitive is *FastMap* to (a) file system choice and (b) false TLB invalidations?

In this section we show how underlying file system affects *FastMap* performance. Furthermore, we also evaluate the impact of batched TLB invalidations. For these purposes we use Silo [128], an in-memory key-value store that provides scalable transactions for multicores. We modify Silo to use a memory-mapped heap over both *mmap* and *FastMap*.

Table 4.2: Throughput in kilo-operations per second and average latency in msec for TPC-C.

	xput	**latency**
mmap-EXT4-Optane SSD	4.3	7.43
mmap-EXT4-pmem	4.2	7.62
FastMap-*EXT4-Optane SSD*	226	0.141
FastMap-*EXT4-pmem*	319	0.101
FastMap-*NOVA-pmem*	344	0.009

File system choice: Table 4.2 shows the throughput and average latency of TPC-C over Silo. We use both EXT4 and NOVA. We also use XFS and BTRFS but we do not include these as they exhibit lower performance. We see that *FastMap* with EXT4 provides $52.5\times$ and $75.9\times$ higher throughput using an NVMe and a pmem device respectively, compared to *mmap*. We also see similar improvement in the average latency of TPC-C queries. With NOVA and a pmem device, *FastMap* achieves $1.07\times$ higher throughput compared to EXT4. In all cases we do not use DAX *mmap*, as we have to provide DRAM caching over the persistent device. Therefore, *FastMap* improves performance of memory-mapped files over all file systems, although the choice of a specific file system does affect performance. In this case we see even larger performance improvements compared to Ligra and Kreon. Silo requires more page faults and it accesses a smaller portion of each page. Therefore, Silo is closer to a scenario with a single large file/device and a large number of threads generating page faults at random offsets. Consequently, it exhibits more of the issues we identify with Linux *mmap* compared to the other benchmarks: Kreon performs mostly sequential I/O for writes and a large part of a page is indeed needed when we do reads. From our evaluation we see that Ligra has better spatial locality compared to Silo and this explains the larger improvements we observe in Silo.

Figure 4.11 shows the breakdown of execution time for the previous experiments. In the case of Linux *mmap* with EXT4, most of the system time goes to buffer management: allocation of pages, LRUs, evictions, etc. In *FastMap*, this percentage is reduced from 74.2% to 10.3%, for both NOVA and EXT4. This results in more user-time available to TPC-C and increased performance. Finally, NOVA reduces system time compared to EXT4 and results in the best performance for TPC-C over Silo.

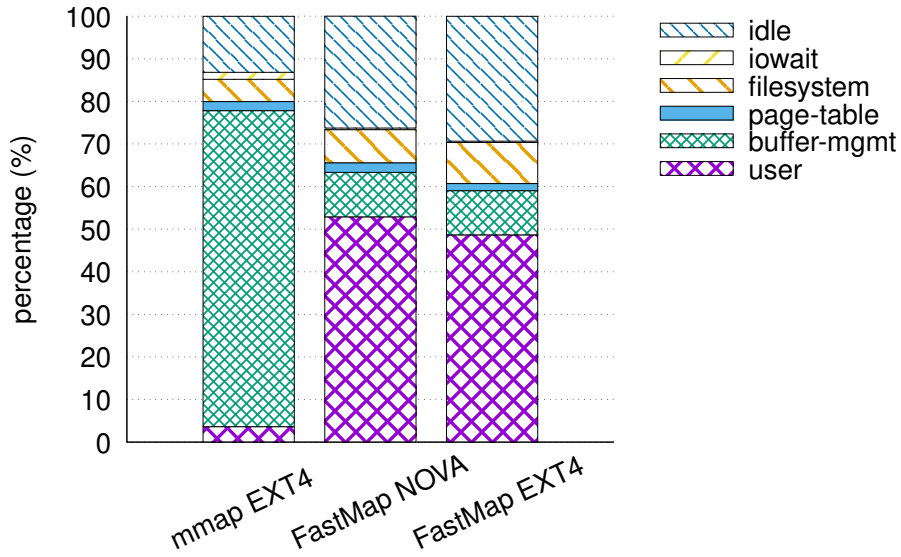


Figure 4.11: Execution time breakdown for Silo running TPC-C using different file systems and the *pmem* device.

False TLB invalidations: *FastMap* uses batched TLB invalidations to provide better scalability and thus increased performance. Our approach reduces the number of calls to *flush_tlb_mm_range()*. This function uses Interprocessor Interrupts (IPI) to invalidate TLB entries in all cores and can result in scalability bottlenecks [33, 3, 4]. Batching of TLB invalidations can potentially result in increased TLB misses. In TPC-C over Silo, batching for TLB invalidations results in 25.5% more TLB misses (22.6% more load and 50.5% more store TLB misses). On the other hand, we have 24% higher throughput (ops/s) and 23.8% lower latency (ms). Using profiling, we see that without batching of TLB invalidations the system time spent in *flush_tlb_mm_range()* increases from 0.1% to 20.3%. We choose to increase the number of TLB misses in order to provide better scalability and performance. Other works [33, 3, 4] present alternative techniques to provide scalable TLB shootdown without increasing the number of TLB invalidations and can be potentially applied in *FastMap* for further performance improvements.

4.5 Summary

In this chapter we propose *FastMap*, an optimized *memory-mapped I/O* path in the Linux kernel that provides a low-overhead and scalable way to access fast storage devices in multi-core servers. Our design enables high device concurrency, which is essential for achieving high throughput in modern servers. We show that *FastMap* scales up to 80 cores and provides up to 11.8× more random IOPS compared to *mmap*. Overall, *FastMap* addresses important limitations of Linux *mmap* and makes it appropriate for data-intensive applications in multi-core servers over fast storage devices.

We show that the lack of scalability in the Linux memory-mapped I/O path results in performance issues and non-optimal pattern to the devices. Fast storage devices require a high concurrency of I/Os to achieve peak device throughput, especially with small request sizes. Our work overcomes all of these limitations and provides significant performance gains in multithreaded applications. On the other hand, single-thread performance remains about the same. There are cases where the single-thread performance matters, especially with low-latency devices where the software overhead is comparable to the device access latency.

Chapter 5

Reducing Protection Costs

In this chapter, we propose a way to reduce the page fault cost in the *memory-mapped I/O* path. In order to achieve that, first we quantify the overhead of page faults and the associated mechanisms, such as trap, TLB misses, I/O, handler, etc, in *memory-mapped I/O*. We instrument the *memory-mapped I/O* path in Linux and we run *FIO* [6] using random reads with one outstanding I/O over a $100GB$ file. All reads result in a page cache miss that translates to an I/O to the device. We use a *pmem*[111] device (backed by DRAM) to emulate a next-generation fast storage device (NVM). In this experiment, a $4KB$ read system call has 18.5% lower latency on average compared to a page fault. Moreover, page faults can only be done using predefined page sizes. Therefore, an I/O may require multiple page faults, while a single system call suffices for explicit I/O. To make *memory-mapped I/O* practical, we need to reduce as much as possible the overhead of page faults.

Figure 5.1 shows the average overhead breakdown for a page fault from the previous experiment. On average a page fault over a memory-mapped file costs 5380 cycles. Two major components of this overhead are: (1) 49% is due to device I/O overhead (2) 24% is due to the cost of protection domain crossing (trap). Even if we exclude device I/O, e.g. when the page exists in the page cache, the cost of a page fault (2724 cycles or $1.13\mu s$) is comparable to accessing an NVMe device and about $4\times$ higher compared to accessing an NVM device. We note that *memory-mapped I/O* needs to change protection domains and to move data efficiently and transparently between DRAM and storage devices. These are fundamental operations for *memory-mapped I/O*, given that device capacity is signifi-

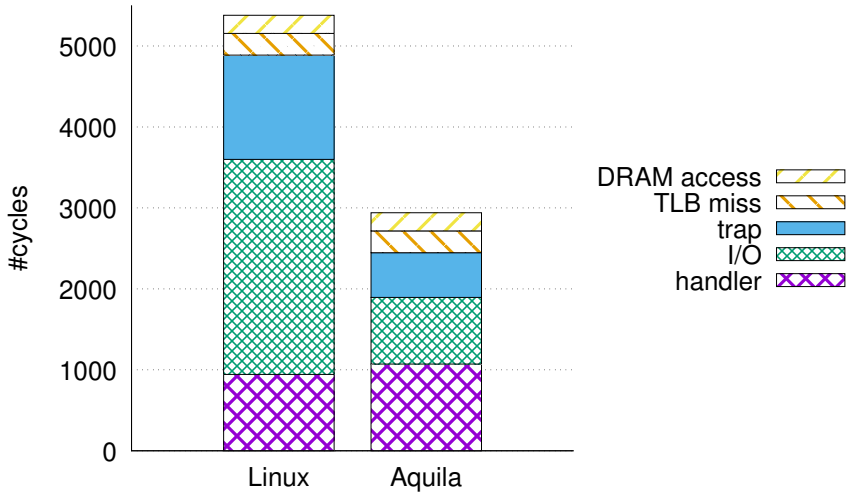


Figure 5.1: Page fault latency breakdown for Linux and *Aquila*, using a *pmem* device (backed by DRAM).

cantly higher than available DRAM.

In this work we observe that *memory-mapped I/O* performs generally the following operations, initiated from page faults:

❶ Modify virtual to physical page mappings.

❷ Device I/O to block devices or files.

❸ DRAM cache lookups and updates.

❹ Virtual memory region lookups and updates.

❺ DRAM allocation and deallocation.

We observe that in *memory-mapped I/O* the most common operations are mainly ❶, ❷, and ❸. They are performed when we need to bring in new data or when we need to writeback dirty pages, if the dataset does not fit in memory, which we assume is the common case for many data-intensive applications. Virtual memory region lookups are common path operations, however updates are not in the common path. Finally, allocation and deallocation of DRAM, mainly used to resize the DRAM cache, typically occur at coarser grain. Based on this observation we design a system that optimizes the first three operations as follows.

In this work we propose running the application in a privileged domain, over a hypervisor similar to where the guest OS runs in virtual machines. We design and implement *Aquila*, a library OS for high-performance storage applications. *Aquila* utilizes hardware virtualization extensions [129] and: ❶ Eliminates the cost of protection domain crossings required during page faults, maintaining full protection [129, 14]. A privileged domain provides direct access to virtual memory hardware, including the page table and TLB. ❷ Reduce the cost of I/O in the expensive kernel I/O path under *memory-mapped I/O* by allowing direct access to devices. Typically, direct device access is performed with frameworks such as SPDK [124] for block-addressable devices and DAX [87] for byte-addressable devices. ❸ Provides better scalability in the DRAM cache, compared to Linux *mmap*. The DRAM cache of Linux *mmap* is frequently accessed by multiple paths of *memory-mapped I/O* and currently does not scale with the number of cores [107]. *Aquila* properly addresses ❹, virtual memory range lookups in the common path, while it performs virtual memory range updates. Finally, it supports dynamic ❺ DRAM allocation in synergy with the host OS, in the uncommon path, along with all other system calls, e.g. networking.

We implement *Aquila* in Linux and evaluate its efficiency with micro-benchmarks and real-life applications. *Aquila* reduces the average page-fault latency by 1.83× compared to Linux *memory-mapped I/O*. We show that our optimized *memory-mapped I/O* path is more efficient compared to the explicit I/O path for accessing fast storage devices. We also show that *Aquila* achieves both low average and tail latency, while maintaining high device throughput. Using RocksDB [49], we show that *Aquila* achieves up to 1.65× higher throughput, compared to user-space caching and *read*/*write* system calls. Finally, we use Kreon [106] and we show that *Aquila* achieves up to 1.22× higher throughput and up to 13.72× lower tail latency for all YCSB workloads over an optimized domain-specific *memory-mapped I/O* path.

The specific contributions of this chapter are:

1. We redesign *memory-mapped I/O* to separate common from uncommon path operations.
2. We use hardware virtualization to remove protection domain switches for page faults and enable direct device access in the common path by leveraging hardware virtual-

ization.

3. We show the effectiveness of our approach using real applications, including RocksDB, which is broadly used today for data storage and access.

The rest of the chapter is organized as follows. Section 5.1 presents an overview of Linux *memory-mapped I/O* and *x86_64* virtualization extensions. Section 5.2 presents the design and implementation of *Aquila*. Section 5.4 presents our experimental results. Finally, Section 5.5 concludes this chapter.

5.1 Background

In this section we provide a brief summary of Linux *memory-mapped I/O* [94] and Intel VT-x [129] virtualization extensions.

5.1.1 Linux *mmap*

The Linux kernel *mmap* system call creates new virtual memory mappings to physical memory for a specific process. Mappings can be either anonymous or backed by files. Anonymous mappings are private to each process and are mainly used for user heap-based memory allocation, i.e. *malloc*. Therefore, anonymous mappings are not bound to a storage device and cannot be recovered after a failure.

File-backed mappings can be either private to each process or shared among processes. Private file mappings are used to load executables and shared libraries. These are typically mapped with read-execute permissions in the text segment and include portions mapped with read-write permissions in the data segment. Any modification to the data segment does not reach the underlying file. For this reason, these are Copy-On-Write mappings.

On the other hand, shared file mappings are persistent and exist after a process exits or a failure occurs. These characteristics make them appropriate for storage purposes and in this chapter we target only file-backed shared mappings.

5.1.2 VT-x CPU Virtualization

Intel VT-x [129] is a set of processor hardware extensions to accelerate the operation of virtual memory machines (VMMs). The CPU has two major operating modes, VMX root and VMX non-root. VMX root is similar to non-virtualized CPU operation: The intention is to run the hypervisor and the host OS in this mode. VMX non-root runs the guest operating system. In this mode, CPUs have protection limitations, e.g. for accessing directly hardware resources. Instead, they need to go through the hypervisor, which will perform the required access with the help of the host OS.

CPUs provide instructions to change their mode. Executing *vmlaunch* or *vmresume* from VMX root, changes the mode to VMX non-root and starts or continues to execute guest OS code. This transition is named *vmentry*. The opposite transition is needed when a privileged instruction should be handled by the hypervisor and is named *vmexit*. *vmexits* occur upon events predefined by the hypervisor. These events and several other configuration options are stored in the per-CPU VM Control Structure (VMCS) memory buffer. Except from the predefined events (privileged instructions) the guest can generate a *vmexit* explicitly by issuing a *vmcall* instruction. For both *vmentry* and *vmexit* events, processor hardware handles the steps for saving and restoring architectural state. This state information is stored in VMCS.

Each of the two modes, VMX root and VMX non-root, supports a separate set of protection rings, where ring-0 is the most privileged and ring-3 is the least privileged. Commonly the operating system (host and guest) runs in ring-0 and user applications run in ring-3. Ring-1 and ring-2 are not used in modern operating systems.

An important aspect of VT-x is the use of *Extended Page Tables (EPTs)* that accelerate address translation, as follows. When the guest is executing, there are two levels of address translations. First, from Guest Virtual Address (GVA) to Guest Physical Address (GPA). This is done using regular page tables and does not require a *vmexit*. Second, from Guest Physical Address (GPA) to Host Physical Address (HPA). EPTs accelerate this second step under the control of the hypervisor. During the access of a GPA, if the translation does not exist, an EPT-fault occurs and the hypervisor handles the fault in a way similar to regular page

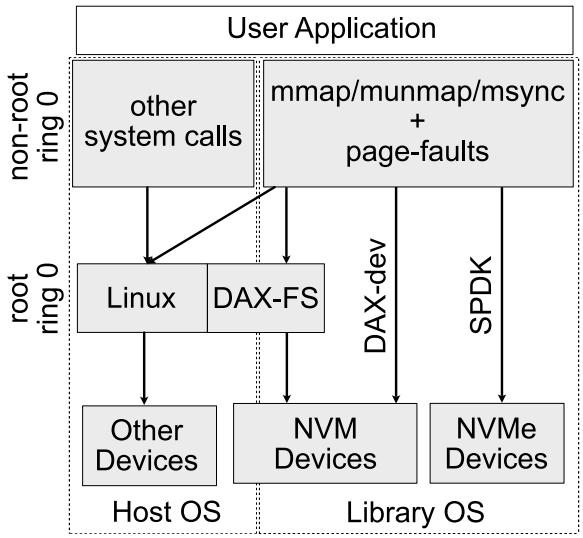


Figure 5.2: *Aquila* high-level design.

faults.

5.2 Design

Today, *memory-mapped I/O* is based on the legacy separation of kernel and user space responsibilities, which dictates that all privileged operations should happen inside the OS. In this chapter, first we present our observation that *memory-mapped I/O* performs generally five operations. Based on these observations, in this chapter, we redesign *memory-mapped I/O* to optimize for the most common operations ❶, ❷, and ❸. We also provide efficient mechanisms for ❹ and ❺ as they are necessary, however, they occur at coarser granularity.

We design and implement *Aquila*, a library operating system that optimizes *memory-mapped I/O* for storage applications that utilize fast storage devices. Figure 5.2, shows the high-level design of *Aquila*, which handles all virtual memory related operations. It also provides direct access to fast storage devices, bypassing the host operating system in the common path of *memory-mapped I/O*. In *Aquila*, the uncommon path uses the host operating system to handle initialization of *Aquila*, to perform runtime management op-

erations, e.g. for resizing the available DRAM for *memory-mapped I/O*, networking, and handling other system calls.

5.2.1 Virtual to Physical Mappings

Aquila proposes the use of *memory-mapped I/O* to eliminate cache lookup overheads, which requires costly page faults and virtual to physical mappings. *Aquila* reduces this cost as follows.

Protection domain crossings: Figure 5.1 shows that page fault handling incurs significant cost for switching protection from ring-3 to ring-0 as follows. For user applications, page fault exceptions occur in ring-3. The page fault initiates a protection domain crossing to ring-0, changes the stack, and stores state information about the process exception in the kernel stack. Then, execution branches to the appropriate fault handler. Upon page fault completion, control returns to the user-space code in ring-3 with the *iret* instruction.

We measure this protection domain switch cost (excluding the handler itself) to be 1287 cycles ($536ns$). In this measurement, if we exclude I/O cost, then switching rings is 1287 out of 2724 cycles, so 52.7% of the page fault cost. Reducing the protection domain crossing overhead will affect all page faults that happen in the common path.

Aquila eliminates the overhead to switch protection domains (rings) by placing the application in VMX non-root ring-0 and executing page faults in the same ring. The important observation is that page faults can be handled entirely in VMX non-root ring-0, if they only require manipulating virtual memory mappings over already allocated physical memory. We measure the trap cost in VMX non-root ring-0 to be 552 cycles ($230ns$), which is $2.33\times$ lower compared to exceptions from ring-3.

Stack management: Using the same stack for both user data and exception handling in *Aquila* (non-root ring-0) can cause corruption due to the *red zone* compiler optimization. The *red zone* is a fixed-size area in each function stack frame, beyond the current stack pointer. A function may use the *red zone* to store local variables without the additional overhead of modifying the stack pointer. The x86-64 ABI uses a 128-byte *red zone*, starting

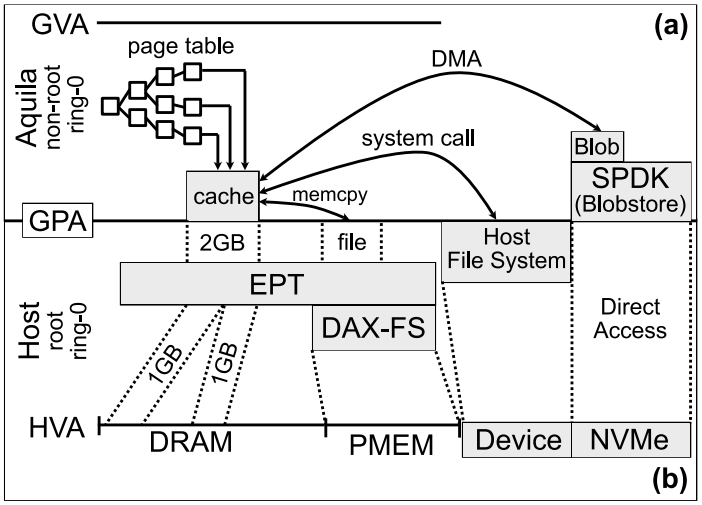


Figure 5.3: Page tables, device I/O, DRAM cache, and DRAM allocation in *Aquila*.

directly under the stack pointer.

Corruption may occur when an interrupt/exception is triggered and the user code is using the *red zone*: The handler will overwrite the *red zone* and will corrupt user data. For this reason, operating systems are compiled with the *red zone* feature disabled to avoid this type of data corruption.

A simple but impractical way to address this problem would be to require all user applications to be compiled without the *red zone* optimization. This approach would make it necessary to also compile in the same manner all third-party libraries, including the standard C library *libc*. This is not practical, as we want to run as much as possible unmodified applications.

To overcome this limitation we use a *x86_64* feature which provides the ability to change the stack in hardware when an interrupt/exception occurs. In *x86_64* there can be up to 7 alternative stacks. The Linux kernel currently uses 4 of these for Double Fault Exceptions, non-maskable interrupts, hardware debug interrupts, and Machine Check Exceptions.

Aquila provides two handlers: one for page faults and one for Inter-Processor Interrupts (IPIs), each with its own stack. In both page faults and IPIs, we start by disabling interrupts and then the exception frame uses an alternative stack. We take care of the *red zone* and copy the exception frame back to the currently running function stack before

re-enabling interrupts. We cannot keep interrupts disabled during page faults, because in memory-mapped I/O a page fault can lead to an I/O operation, which can take several thousands of cycles to complete. Delaying IPIs during a prolonged interval would negatively affect performance of other threads.

Page table updates & TLB shootdown: Similar to modern OSes, we use a shared page table for all threads of each process. Figure 5.3 shows that *Aquila*'s page table resides in non-root ring-0 and translates application virtual addresses (GVAs) to DRAM cache pages (GPAs). To add a new mapping, we traverse the page table from its root and add the appropriate page translation in the last level (PTE). To deal with concurrent accesses, we use atomic operations to update and add new levels similar to Linux [94].

Besides adding new translations in the page table during page faults, another important operation is to modify or remove mappings. It is required in the case of page evictions or when updating protection flags in existing mappings (i.e. *mprotect*). These operations require a TLB invalidation. In *x86_64*, each CPU can only invalidate its local TLB. *x86_64* provides Inter-Processor Interrupts (IPIs) so the OS can notify other cores to invalidate their TLB (aka TLB shootdown). Other work [4, 3] has shown that for anonymous mappings this can limit scalability with high core counts. Handling TLB shootdowns in VMX non-root ring-0 introduces challenges for both correctness and performance.

To reduce this cost *Aquila* uses a batched TLB shootdown approach. We remove the mappings for multiple pages (512 in our evaluation) and send a single TLB invalidation for all pages. We use posted IPIs as provided by hardware virtualization extensions, together with a mechanism similar to Shinjuku [71]. Shinjuku sends and receives IPIs without the need of a *vmexit*. To remove *vmexit* in the send path, it maps the *APIC* directly to the user.

An additional problem to address in *Aquila*, as it target unmodified user applications, is the case where a malicious process performs a denial-of-service attack by issuing a large number of interrupts to a specific core. To address this, we choose to use a *vmexit* in the send path by writing to an *MSR* register. Requiring a *vmexit* on the send path results in increasing the cost from 298 to 2081 cycles [71]. However, due to batching this cost is amortized for the whole batch and we show in our evaluation that this is negligible com-

pared to other costs (Figure 5.4). Finally, similar to Shinjuku, *Aquila* uses the *vmexit*-less receive path.

5.2.2 Device I/O

Device I/O typically requires kernel involvement and incurs high overhead. To reduce overhead, user-space frameworks, such as SPDK [124] and DAX [87] allow direct access to certain types of dedicated devices, typically NVMe and NVM respectively. *Aquila* allows *memory-mapped I/O* to use such frameworks in non-root ring-0. Figure 5.3*(a)* shows the I/O path in *Aquila* and how it provides direct access to block-addressable NVMe and byte-addressable NVM devices.

Direct access to NVMe: First, *Aquila* can bypass the host OS and issue I/O operations directly to devices. Figure 5.3*(a – right)* shows this path. Direct access requires that the devices are not shared with other processes and also that they are visible directly from *Aquila* at non-root ring-0. This is the case, e.g. with modern modern NVMe devices attached to PCIe, because device configuration registers the user can be mapped directly to the user space.

In order to provide the user applications with a file abstraction, we leverage SPDK [124] and *Blobstore* [123]. *Blobstore* provides a flat namespace of *blobs*, where each *blob* is identified by a unique number and can be created/resized/deleted at runtime while it also supports extended attributes. *Aquila* also supports the translation from files to blobs transparently. In order to do this we intercept *open* and *mmap* calls in non-root ring-0. On the other hand, this does not provide POSIX semantics but it is enough for data intensive applications (we have successfully run RocksDB without modifications using this path). *Aquila* uses the direct I/O path of *Blobstore*, which does not buffer access, as opposed to *BlobFS* [122], which buffers data in its local DRAM cache.

Direct device access requires dedicated devices for protection purposes. This can also take the form of dedicated device partitions. Today, such an approach is common for systems, such as key-value stores and data processing frameworks, that entirely manage

their storage and data.

Direct access to NVM: Second, *Aquila* allows direct access to byte-addressable storage as a backing device for DRAM. Figure 5.3*(a – left)* shows this path. NVM devices [62] provide performance closer to DRAM than block-addressable storage devices, such as NVMe and SSDs. However, they provide higher latency and lower throughput compared to DRAM [65]. Therefore, NVM devices can still benefit from DRAM caching in cases where the working set fits in cache.

NVM is typically accessed via a *Direct Access (DAX)* file system. DAX maps the NVM device directly to the user address space, as physical memory, next to the available DRAM. Data accesses can occur with *load*/*store* operations, similar to DRAM. Metadata operations to DAX files still go through the host operating system but they are not in the common path.

Aquila ensures protected sharing of NVM between different processes. We use DAX to map the NVM device to process memory in non-root ring-0. For I/O we use memory copy between DAX-*mmap*ed files and our DRAM cache. Finally, we forward all metadata operations to the host OS.

Aquila uses $4KB$ memory copies (*memcpy*) for reads, equal to the page size. An important difference between memcpy in Linux and *Aquila* is the following. The Linux kernel cannot use SIMD instructions for memcpy because this requires a full FPU state save and restore, which is extremely costly. For *SSE* it has to save 512 bytes and for *AVX* 832 bytes. We measure the cost to save and restore *AVX* state using the *XSAVEOPT* and *FXRSTOR* instructions to be around 300 cycles. Furthermore, we measure the cost of a $4KB$ memcpy, without using SIMD instructions to be about about 2400 cycles. Instead, an optimized memcpy of $4KB$ using *AVX2* streaming (i.e. cache bypass) instructions requires about 900 cycles. With the cost of save and restore FPU state this increases to 1200 cycles, i.e. $2\times$ faster than non-SIMD memcpy. For these reasons *Aquila* uses an *AVX2* optimized memcpy and pays the cost of saving and restoring FPU state.

Finally, *Aquila* can use other approaches for I/O as well. These include the common synchronous *read*/*write* calls and asynchronous approaches, such as *libaio* or *io_uring* [70]

(Figure 5.3*(a)*). All strike a different balance between protection and performance. We leave the evaluation of different configurations for future work.

5.2.3 DRAM Cache

A DRAM cache is necessary for hosting data fetched from I/O devices. Previous work [107] has shown that the kernel buffer cache used by Linux *mmap* does not scale well with the number of threads. Therefore, *Aquila* aims to reduce contention and improve cache scalability. The main observation is that, unlike the Linux kernel, dirty pages needs to be maintained in a separate structure from the clean pages. *Aquila* uses the following structures and operations that target specifically the caching functionality required by *memory-mapped I/O*.

DRAM cache lookup: Cache lookups are an important component of the *memory-mapped I/O* path. Page faults eliminate the software cost for cache lookups in the case where a page resides in the cache. In the case of a page fault, the handler first checks if the requested page is in the DRAM cache. Although this is a page fault, it may still happen that at the time of the page fault check the page has been brought in the cache. For this reason, the handler uses a lock-free hash table to perform a fast lookup, similar to [38]. If the requested page resides in the DRAM cache, *Aquila* simply creates a mapping to the page table between the faulting VM address and the physical page. If the page is not present in the cache, it (1) allocate new cache pages from the freelist, (2) evict pages from the cache, (3) clean dirty pages, and (4) issues write and read I/O to the underlying device, as follows.

Freelist & Evictions: *Aquila* uses a hierarchical 2-level freelist to manage DRAM cache pages, as follows. The first level consists of a queue per NUMA node while the second level of a queue per core. When a page is required, the core checks in order, its local (core) queue, the local NUMA node queue, and the remote NUMA node queues. All of these queues contain free pages. In the case where all of these queues are empty, *Aquila* tries to evict a batch of pages (512) synchronously. We choose which pages to evict via an approximation of *LRU*. When a page is evicted from the cache, it is placed in the local core queue.

If the number of pages in the local core queue exceeds a threshold, they are moved to the appropriate NUMA queue. All page movement between first and second level queues are performed in batches (4096 pages in our evaluation). Freelist queues are lock free and using our two level allocator and the movements in batches between levels, we do not observe high contention.

Dirty page write-back: *Aquila* maintains dirty pages in a separate from the hash table data structure to accelerate writeback and *msync* operations. Dirty pages need to be sorted by device offset and this is not facilitated by our hash table. For this reason, and to reduce contention to a single lock, we use a per-core red-black tree. We write pages back based on their page offset, when a page is selected for replacement. Having multiple sorted red-black trees allows easy merging of pages in larger I/Os for writebacks similar to Linux kernel [94]. For reads, we require synchronous I/Os and we cannot apply any batching. Based on the *madvise* arguments we also perform read-ahead to improve sequential reads.

5.2.4 Virtual Memory Lookups and Updates

Virtual address range update operations, such as *mmap*, *munmap*, and *mremap* are used to create, destroy, expand, or shrink mappings to files or devices. In *memory-mapped I/O* virtual address range lookups are common path operations because every page fault checks if the faulting address refers to a valid, properly mapped, virtual address. On the other hand, update operations happen at a coarse grain. Therefore, the challenge in *Aquila* is supporting efficient virtual memory range lookups.

Linux, uses a red-black tree to keep all active Virtual Memory Areas (VMA), protected by a read-write lock. Operations that modify address ranges, (*mmap*, *munmap*, and *mremap*) need to acquire this as a write lock, while page faults acquire it as read lock. Other work [32, 21, 33] has shown that this lock can limit scalability in servers with a large number of cores, even in case where it is acquired as a read lock.

For this reason, *Aquila* uses a radix tree, similar to RadixVM [33], instead of a balanced tree to avoid contention and provide scalable manipulation and access of virtual address

ranges. In the case of page faults, the radix tree is used for two purposes: (1) check if the page fault occurred in a valid address and (2) lock the specific entry to avoid concurrent modifications for the same page.

RadixVM uses per-core page tables in order to keep metadata about which TLB contain specific mappings and enable targeted TLB invalidations. RadixVM targets anonymous mappings where the design tradeoffs are different. We choose to have a single page table shared for all cores similar to what common OSes do. This approach reduces the number of total page faults and as we use a batched TLB shootdown approach it does not negatively affect the performance.

Finally, we also do not use RadixVM *refcache* mechanism for page reference counting. We provide explicit page management as described in the previous section. In the cases where reference counting is required (i.e. radix tree metadata), we use a single shared reference count but this does not incurs and performance bottlenecks as it is not in the common path.

5.2.5 DRAM Allocation

DRAM allocation and deallocation occurs less frequently, compared to other operations of *memory-mapped I/O*. Therefore, its cost is of secondary importance. Furthermore, to reduce the frequency of DRAM allocations, and given the nature of *memory-mapped I/O*, we use huge pages. *Aquila* supports both $2MB$ and $1GB$ pages; in our evaluation we only use $1GB$ pages. However, *Aquila* needs to cooperate with the host OS to allow for dynamic DRAM allocation, as follows.

Figure 5.3*(b)* shows the DRAM allocation path. *Aquila* allocates pages using (anonymous mapping) *mmap* calls to the host operating system. *Aquila* can also ask the host operating system to deallocate memory with *munmap* operations.

The host operating system is responsible to allocate additional DRAM to *Aquila* and reclaim it, when needed. *Aquila*'s page table is responsible for translating GVAs to GPAs. The host operating system then translates GPAs to HPAs, via the *EPT*. In *Aquila*, similar to common OSes, all threads of a process share the same page table and for this reason we

use a single *EPT* per process. *Aquila* uses Dune [14] for *EPT* management and we modify it to replace its one *EPT* per thread with one *EPT* per process.

Accesses to a GPA where an *EPT* mapping does not exist result in an *EPT* fault in the host OS. This is similar to common page faults but has higher cost due to the required *vmexit*. Similar to Dune [14], during an *EPT* fault it checks the normal page table if the access is valid and then it adds the translation to the *EPT*. *munmap* calls end up removing the corresponding mappings from the host *EPT*.

5.3 Implementation

Dune[14] uses hardware virtualization extensions and provides direct but safe access to hardware features, such as ring protection, page tables, and tagged TLBs. *Aquila* uses Dune to access and configure Intel VT-x virtualization extensions.

Similar to Dune, *Aquila* implements a subset of the system calls and redirects the rest to the host operating system, using *vmcall*. This requires a custom handler for system calls in the *MSR_LSTAR* address. We modify Dune to also enable system call interception in ring-0. Then, we intercept all virtual memory related system calls, specifically *mmap*, *munmap*, *mremap*, *madvise*, *mprotect* and *msync*. These calls are handled in *Aquila* and do not result in a *vmcall*. Therefore, they incur the overhead of a regular function call, as they do not trigger a protection domain crossing.

Aquila consists of about $20K$ lines of code of both *C* and *C++* source code excluding third-party libraries. This code handles virtual address ranges management, DRAM caching, including the dedicated page allocator, dirty page management, the *LRU* eviction policy, and I/O to/from the devices. It also handles page faults and intercepts system calls. We are able to run user applications, such as RocksDB, with minimal changes. Similar to Dune, we have to initialize *Aquila* during the application startup. Additionally we have to call a single function for every new thread to make it enter in the *Aquila* mode.

Dependent on architecture and OS: *Aquila* leverages hardware virtualization support to provide fast and protected access to virtual memory hardware. *Aquila* is not tightly

coupled with specific CPU architecture, OS, or hypervisor, as follows. We use *x86_64* to make our case, however, other architectures, such as ARM [130], Intel Itanium [64], and IBM Power [60] provide hardware-assisted virtualization. In addition, *memory-mapped I/O* is supported in most commonly used OSes. OSes based on Linux and BSD support *mmap* system calls, while Windows provide similar functionality with the *MapViewOfFile* system call. Finally, *Aquila* can also leverage nested virtualization [16] that modern hypervisors [138, 88, 131, 98] provide to run within a virtual machine.

Security implications: *Aquila* provides a similar security model to a guest virtual machine running on a host operating system. This assumption is also valid for Dune [14], which we use in our implementation.

5.4 Experimental Analysis

In this section, we evaluate *Aquila* experimentally to answer how does *Aquila*:

1. Reduce overheads compared to Linux *mmap*?
2. Scale with the number of threads compared to *mmap*?
3. Perform compared to explicit *read*/*write* I/O calls for real applications?
4. Perform compared to Linux *mmap* for real applications?

Next, we discuss our experimental evaluation methodology.

5.4.1 Methodology

Our testbed consists of a dual-socket server that is equipped with two Intel(R) Xeon(R) E5-2630 v3 CPUs running at 2.4 GHz, each with 8 physical cores and 16 hyperthreads, for a total of 32 hyperthreads. The storage device used in our experiments is a PCIe-attached Intel Optane SSD DC P4800X series [63] with a capacity of 375 GBs. The server is equipped with 256 GB of DDR4 DRAM at 2400 MHz and runs CentOS v7.3, with Linux kernel 4.14.72.

During our evaluation we limit available capacity of DRAM (using *cgroups* [74]) as required by different experiments. To reduce variability in our experiments, we disable swap and Transparent Huge Pages (THP), and we set the CPU scaling governor to *performance*. In experiments where we want to stress the software path of the Linux kernel we also use a *pmem* [111] block device. This emulates a fast byte-addressable (NVM) block device backed by DRAM.

In our evaluation we first use a custom multithreaded microbenchmark. It uses a configurable number of threads that issue *load*/*store* instructions at randomly generated offsets within the memory mapped region. We ensure that each *load*/*store* results in a page fault.

Second, we use RocksDB [49](v6.8.0), a persistent key-value store developed by Facebook and widely used in production systems. It is based on LSM-trees [102] with each level organized in fixed sized files ($64MB$ by default). These files, named Static-Sorted-Tables (*SST*s), and are placed in the mount point specified by the user and organized in a flat namespace. RocksDB provides different ways to read/write data from files: include direct I/O with a user-space cache, buffered read/write in the Linux kernel, and *mmap*. The recommended mode of operation is to use direct I/O explicit *read*/*write* calls combined with a user-space cache [25].

Finally, we use Kreon [106], a persistent key-value store designed from the ground-up to use *memory-mapped I/O* in the common path. Kreon is also based on LSM-trees [102] but instead of *SST*s it uses a log to store all keys and values and a B-Tree index per level for indexing. This approach increases random accesses to devices but reduces I/O amplification and CPU cycles in the common path. Kreon provides a custom *memory-mapped I/O* path in the Linux kernel, named *kmmap*, and organizes its data in a single large file/device, using a custom allocator for space management.

We port RocksDB and Kreon to *Aquila* with small changes to their initialization code. We use the original YCSB workloads [37] with a C++ implementation of YCSB [114] to eliminate high JNI overheads. We run all experiments three times and report averages across runs. In our experiments the variation we observe across runs is negligible.

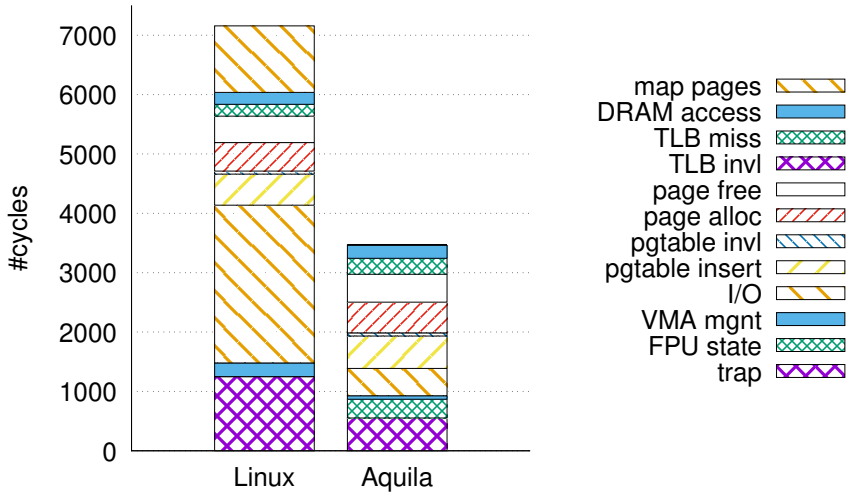


Figure 5.4: *Aquila* execution time breakdown (in cycles) for reads, with a dataset that does not fit in memory and 1 thread.

5.4.2 Experimental Results

In the next sections we provide our experimental results addressing the questions listed above.

How does *Aquila* reduce overheads compared to Linux *mmap*?

Figure 5.1 shows the average overhead breakdown for a memory-mapped file over a *pmem* device. In this case we use a $100GB$ dataset with a $100GB$ DRAM cache. Therefore, no page evictions are required. *Aquila* achieves $1.83\times$ lower overhead compared to Linux *mmap*. *Aquila* reduces (1) the protection domain crossing cost by $2.33\times$ and (2) data access by $3.22\times$. These are the main parts that *Aquila* improves in the common path. Furthermore, we see that the software cost of the handler is similar in both *Aquila* and Linux. Finally, *Aquila* does not change the DRAM accesses and TLB miss cost as they are handled in hardware.

Figure 5.4 shows the average overhead breakdown for the case where the dataset does not fit in main memory and evictions happen in the common path. In this case we use $8GB$ for the DRAM cache and a $100GB$ dataset. *Aquila* achieves $2.06\times$ lower overhead

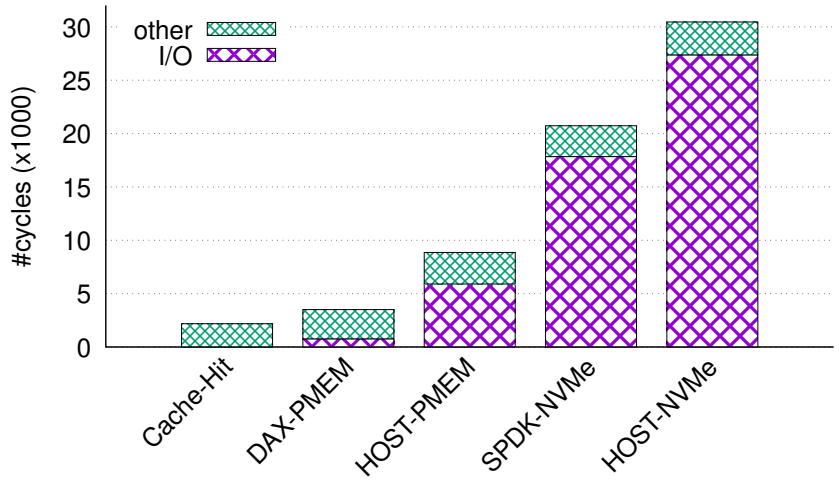


Figure 5.5: *Aquila* execution time breakdown (in cycles) using different approaches for I/O.

compared to Linux *mmap*. We observe that in this case the major sources of overhead are protection domain switching and I/O to the underlying device. Finally, we observe that in *Aquila*, no single source of overhead dominates in the common path, even in the case where evictions occur in the common path: Each component of the *Aquila mmap* path accounts for less than 10% of overhead.

We omit the presentation of results for writes as we observe similar behavior and performance to reads.

Figure 5.5 shows how the I/O path affects the performance in *Aquila*. *Cache-Hit* is the case where no I/O required and the total cost in this case is 2179 cycles. *DAX-PMEM* uses our optimized path for byte-addressable devices and *HOST-PMEM* uses direct I/O system calls to the host OS. In this case, *Aquila* achieves 7.77× lower latency. This happens because we remove system calls, and use a SIMD-optimized *memcpy*. *SPDK-NVMe* uses SPDK and bypasses the host OS for PCIe attached devices. *HOST-NVMe* uses direct I/O, similar to *HOST-PMEM*. In the case of *Aquila*, bypassing the host OS reduces the latency by 1.53×. In all cases, the remaining cost, excluding the I/O, remains the same. This shows that the way we choose to access storage devices in *Aquila* affects the total performance. Removing the interaction with host OS reduces the overheads by up to 7.77×.

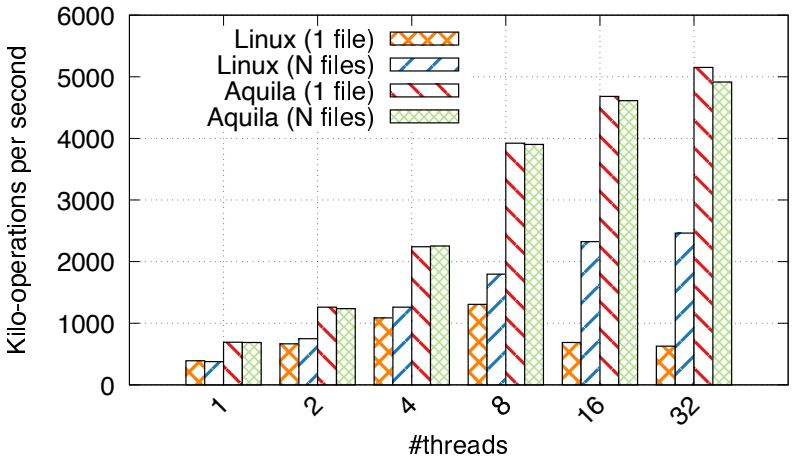


Figure 5.6: Linux vs. *Aquila* throughput (in ops/sec) using random reads for both a shared and a private file per thread with a dataset that fits in memory.

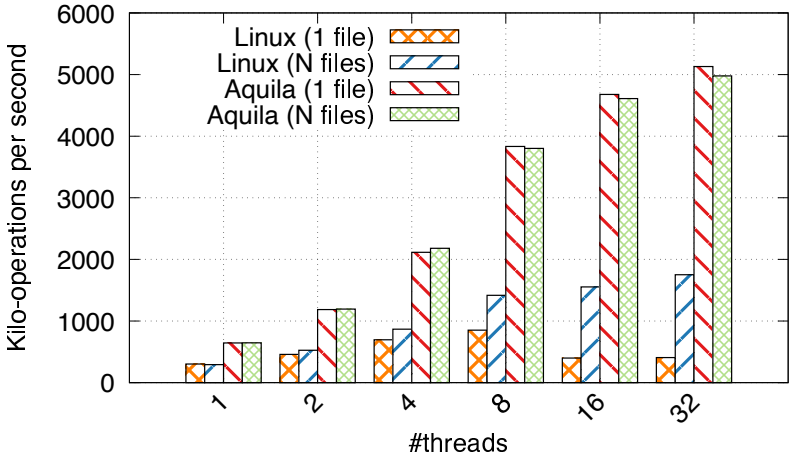


Figure 5.7: Linux vs. *Aquila* throughput (in ops/sec) using random reads for both a shared and a private file per thread with a dataset that does not fit in memory.

How does *Aquila* scale with the number of threads compared to Linux *mmap*?

Next, we examine how *Aquila* scales with an increasing number of cores. We distinguish two cases for accessing files with *memory-mapped I/O*: when all threads access a single

shared file and when each thread accesses a different file.

Figures 5.6 and 5.7 shows our results for a dataset ($100GB$) in two cases, one where it fits in memory ($100GB$ DRAM) and one where it does not fit in memory ($8GB$ DRAM). In both cases, as we increase the number of threads *Aquila* scales as follows: For a single shared file that fits in memory, *Aquila* achieves $1.81\times$ higher throughput with 1 thread and $8.37\times$ with 32 threads. In the case where the dataset does not fit in memory, the improvement is even more pronounced compared to Linux *mmap*. *Aquila* performs better by $2.17\times$ at 1 thread and by $12.92\times$ at 32 threads.

We use profiling to identify the reason for the large improvement over Linux *mmap* for a single shared file. We find that in Linux, a single lock that protects the radix tree of the cached pages is highly contented. In order to remove this bottleneck in *Aquila*, we remove this single lock with a lock-free hash table that keeps all cached pages. We see similar behaviour also for writes in Linux as this lock is also required to mark a page dirty. *Aquila* uses per-cpu red-black trees to store dirty pages and this also overcomes this limitation.

Using a separate, private, file per thread, we see lower, but still significant improvements: *Aquila* achieves higher throughput between $1.82\times$ (1 thread) and $1.99\times$ (32 threads) using the in-memory dataset and between $2.21\times$ (1 thread) and $2.84\times$ (32 threads) for the out-of-memory dataset.

In all these experiments, *Aquila* also provides better latency, both median and tail. Using a single thread and the dataset that does not fit in main memory, *Aquila* achieves $2.07\times$ lower median latency. Furthermore, *Aquila* has $13.6\times$ (p99) and $2.1\times$ (p99.9) lower tail latency.

Using 32 threads the improvements of *Aquila* in terms of latency are even higher. Using a shared file, *Aquila* achieves $8.52\times$ (median), $177\times$ (p99), and $213\times$ (p99.9) lower latency. For a separate file per thread *Aquila* achieves lower latency by $1.64\times$ (median), $42.64\times$ (p99), and $53.2\times$ (p99.9). Eliminating the shared contended lock in the case of the shared file provide huge improvements in tail latency for *Aquila*.

We see similar behaviour in writes compared to reads. For this reason, we omit the presentation of these results.

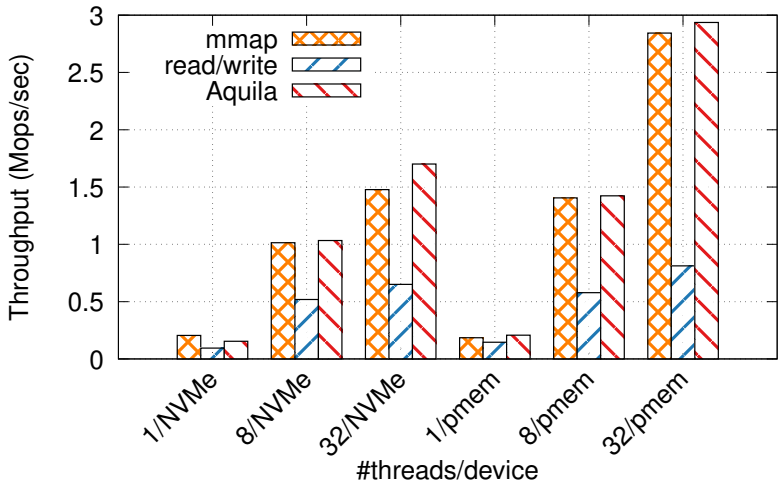


Figure 5.8: *mmap* vs. *read*/*write* vs. *Aquila* for RocksDB and a dataset that fits in memory.

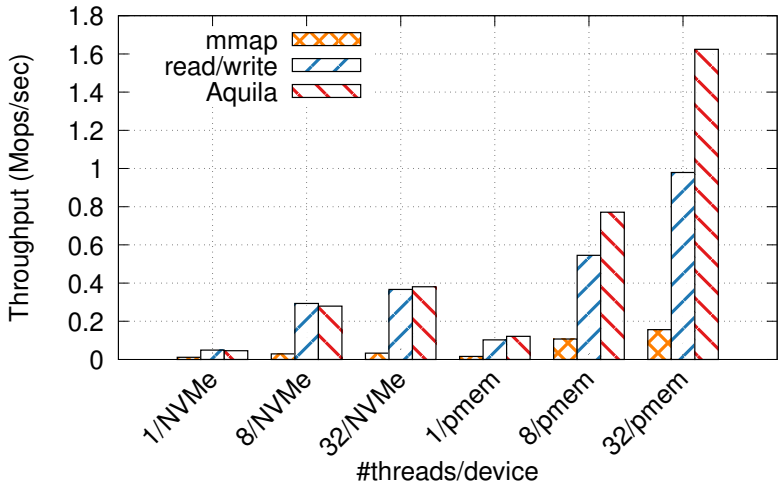


Figure 5.9: *mmap* vs. *read*/*write* vs. *Aquila* for RocksDB and a dataset that does not fit in memory.

How does *Aquila* perform compared to explicit *read*/*write* I/O calls for real applications?

In this section we examine how *Aquila*'s *memory-mapped I/O* compares to Linux explicit I/O for real applications.

RocksDB has been designed for explicit I/O. However, it provides an option to use *memory-mapped I/O* as well, to read data from SSTs. The developers of RocksDB state [51] that using *mmap* for an *in-memory* database with a *read-intensive* workload increases performance. However, they also state [50] that *mmap* sometimes causes problems when data does not fit in memory and is managed by a file system over a block device. In this section we show that *Aquila* outperforms direct I/O even in the case where the dataset is larger that the available DRAM cache.

We compare RocksDB with explicit I/O (direct I/O and an *LRU* cache of $8GB$), RocksDB with Linux *mmap* ($8GB$ page cache limited with *cgroups*), and RocksDB with *Aquila* ($8GB$ DRAM cache). For this evaluation we use YCSB with workload C (100% random reads). The value size is $1KB$ and the key size is about $30B$. Finally, we use two datasets, one of $8M$ records ($8GB$) that fits in the cache and a second dataset of $32M$ records ($32GB$) that is $4\times$ larger than the cache size.

Figure 5.8, shows this experiment using both NVMe and PMEM device and the dataset that fits in cache. We see that similar to what developers of RocksDB say, *mmap* is faster than *read*/*write* calls. Additionally, in this case *Aquila* is up to $1.15\times$ faster compared to Linux *mmap*.

Figure 5.9 shows our results for the dataset that does not fit in the cache, both for NVMe and PMEM. We see that Linux *mmap* performs poorly compared to other approaches. The main reason is that *mmap* prefetches $128KB$ for $1KB$ reads.

The PMEM device shows the potential of *Aquila* as storage devices become faster. In this case, *Aquila* results in higher RocksDB throughput by $1.18\times$ for 1 thread and by $1.65\times$ for 32 threads. With the NVMe device, *Aquila* and direct I/O have similar performance (between $0.96\times$ up to $1.06\times$) because throughput is limited by the device itself. Therefore, *Aquila* is able to improve upon explicit I/O performance, even for large reads, a case where explicit I/O performs best.

In all previous cases *Aquila* provides better median and tail latency. Using the dataset that fits in cache, *Aquila* achieves from $1.28\times$ to $1.39\times$ lower median latency compared to direct I/O and from $1.09\times$ to $1.27\times$ compared to *mmap* with the NVMe device. With PMEM, improvements are between $1.06\times$ and $1.21\times$ for medial latency and and between

1.01× and 1.15× for tail latency.

Using the dataset that does not fit in the cache, we compare *Aquila memory-mapped I/O* with Linux direct I/O, excluding Linux *mmap* as it performs poorly. For the NVMe device, medial latency improves similar to the in-memory dataset, between 0.98× and 1.12×. However, for the PMEM, *Aquila* achieves even lower medial latency compared to the out-of-memory dataset, between 1.24× and 1.28×.

We see larger improvements in all cases for tail (p99.9) latency. For in-memory datasets *Aquila* achieves 3.88× lower tail latency on average compared to Linux explicit I/O. For out-of-memory datasets, *Aquila* achieves 1.26× lower tail latency on average.

Finally, we do not provide an evaluation of write operations in RocksDB, generated by compactions. Compactions (and writes) in RocksDB take place in background threads and they are optimized to issue large (1-2MB) I/O requests. In this case the only bottleneck is the device itself, rather than the software stack.

How does *Aquila* perform compared to Linux *mmap* for real applications?

In this section, we use *Aquila* for Kreon, a persistent key-value store designed to use *memory-mapped I/O* in the common path. Kreon provides a custom *memory-mapped I/O* path, named *kmmap*, which improves several aspects of Linux *mmap*. We compare Kreon and *kmmap* with *Aquila*. We run all YCSB workloads using a single thread to show how overhead reduces in the single-thread path. We use a dataset of 16M records (16GB) with a 8GB cache.

Figure 5.10 shows our results. Using NVMe, *Aquila* achieves on average 1.02× higher throughput for all YCSB workloads. In this case the bottleneck is the NVMe device itself given the request size (4KB) and a single outstanding I/O. Latency improves significantly: *Aquila* achieves 1.29× lower median latency and 3.78× tail (p99.9) latency compared to *kmmap*.

With PMEM, where device throughput is not the dominating bottleneck, *Aquila* achieves on average 1.22× higher throughput. We also see significant improvements in terms of latency: *Aquila* achieves 1.43× lower median latency and 13.72× lower tail (p99.9) latency.

Finally, we run all YCSB workloads over Kreon using 16 threads and the PMEM device

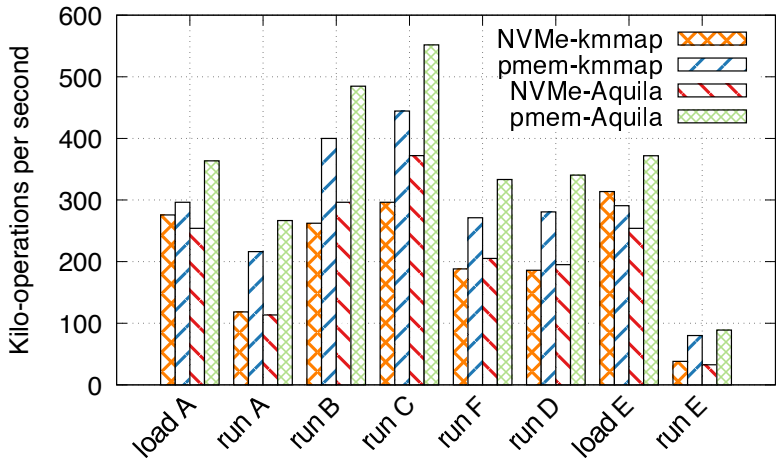


Figure 5.10: Kreon *kmmap* vs. *Aquila* for a dataset that does not fit in memory for NVMe and PMEM using a single thread.

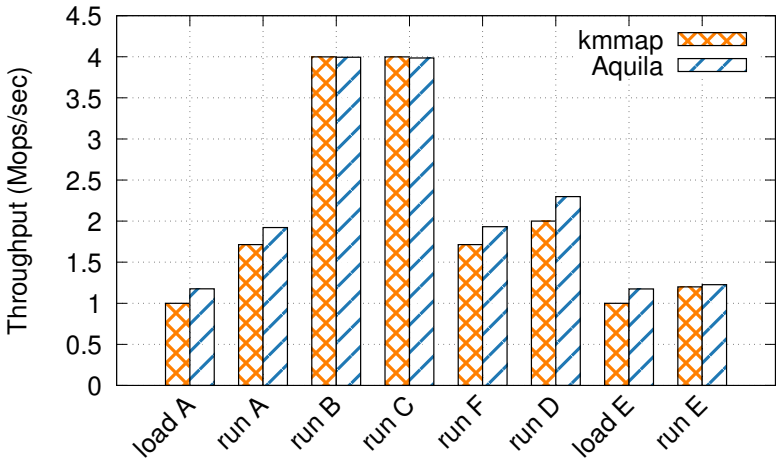


Figure 5.11: Linux *kmmap* vs. *Aquila* for a dataset that fits in memory, a PMEM device and 16 threads.

to show the scalability of *Aquila* over an already optimized *memory-mapped I/O* path. Figure 5.11 shows our results. *Aquila* achieves 1.08× higher throughput, 1.05× lower median latency, and 1.22× lower tail (p99.9) latency, on average for all YCSB workloads. We have to note here that Kreon *memory-mapped I/O* [106] is optimized for this specific case and

outperforms Linux *mmap* for all cases, both in terms of throughput and latency.

5.5 Summary

This chapter discusses how *memory-mapped I/O* can address overhead concerns as fast storage devices start to dominate I/O in modern servers. *Memory-mapped I/O* has important advantages: it can remove system call overheads, eliminate cache lookups even for user space caches and hit operations, and offer synchronous I/O capabilities. We observe that the main operations of *mmap* occur at different frequencies: virtual memory management, page table accesses, and device access are the common path operations. We propose a design that moves the common path operations of *memory-mapped I/O* to the processor mode normally used for the OS of VM guests (non-root ring-0), thus avoiding significant overheads. *Aquila* still allows all functionality and flexibility of *mmap* by using the host OS (running in root ring-0 mode) for the operations outside of the common path. We implement *Aquila* using Dune, and we evaluate it with micro-benchmarks and two persistent key-value stores and we observe significant performance gains both in terms of throughput and latency.

This chapter shows that the single-thread performance of storage cache management with memory-mapped I/O is higher than a user-space cache and read/write system calls. In the case of next-generation fast storage devices, the overhead of memory-mapped I/O can be comparable to the device access latency itself. We take a holistic approach to remove protection domain switches in the common path and significantly improve performance. On the other hand, our approach requires application modifications, and system calls have higher overheads than Linux. We target data center-oriented applications where common path operations commonly do not require any system calls.

Chapter 6

Related Work

The related work of this dissertation falls in two different research areas. These are: (a) *memory-mapped I/O* and (b) persistent key-value stores. Next, we provide more details for both areas.

6.1 Memory-Mapped I/O

We categorize related work of *memory-mapped I/O* in four areas: (a) replacing *read*/*write* system calls with *mmap* for persistence, (b) providing scalable address spaces, (c) extending virtual address spaces beyond physical memory limits, and (d) dataplane operating systems.

6.1.1 Using *memory-mapped I/O* in data-intensive applications

Both MonetDB [19] and MongoDB [30] (with *MMAP_v1* storage engine) are popular databases that use *mmap* to access data. When data fits in memory, *mmap* performs very well. It allows the application to access data at memory speed and removes the need for user-space cache lookups. Facebook's RocksDB [49], a persistent key-value store, provides both *read*/*write* and *mmap* APIs to access files. The developers of RocksDB state [51] that using *mmap* for an *in-memory* database with a *read-intensive* workload increases performance. However, they also state [50] that *mmap* sometimes causes problems when data does not fit in memory and is managed by a file system over a block device.

Tucana [105] and Kreon [106] are write-optimized persistent key-value stores that are designed to use *memory-mapped I/O* for persistence. In Tucana we show that for a write-intensive workload the *memory-mapped I/O* results in excessive and unpredictable traffic to the devices, which results in freezes and increases tail-latency. Kreon [106] provides a custom *memory-mapped I/O* path inside the Linux kernel that improves write-intensive workloads and reduces the latency variability of Linux *mmap*. In this work, we address scalability issues and also present results for *memory-mapped I/O* with workloads beyond key-value stores.

DI-MMAP [48, 47], removes the swapper from the critical path and implements a custom (FIFO based) eviction policy using a fixed-size memory buffer for all *mmap* calls. This approach provides significant improvement compared to the default Linux *mmap* for HPC applications. We evaluate *FastMap* using more data-intensive applications, representative of data analytics and data serving workloads. In particular, our work assumes higher levels of I/O concurrency, and addresses scalability concerns with higher core counts. Additionally, in *Aquila*, we take a more fundamental approach, by placing the application non-root ring-0 to further reduce protection costs.

FlashMap [59] combines memory (page tables), storage (file system), and device-level (FTL) indirections and checks in a single layer. *FastMap* and *Aquila* provide specific optimizations only for the memory level and results in significant improvements in a file system and device agnostic manner.

2B-SSD [8] leverages SSD internal DRAM and the byte addressability of the PCIe interconnect to provide a dual, byte and block-addressable SSD device. It provides optimized write-ahead logging (WAL) over 2B-SSD for popular databases and results in significant improvements. FlatFlash [1] moves this approach further and provides a unified memory-storage hierarchy that results in even larger performance improvements. Both of these works move a large part of their design inside the device. Both *FastMap* and *Aquila* work in a device-agnostic manner and provides specific optimizations in the operating system layer.

UMap [108] is a user-space memory-mapped I/O framework which adapts different policies to application characteristics and storage features. Handling page faults in user-

space (using *userfaultfd* [75]) introduces additional overheads that are not acceptable in the case of fast storage devices. On the other hand, techniques proposed by *FastMap* can also be used in user-space memory-mapped I/O frameworks and provide better scalability in the page-fault path.

Similar to [31], *FastMap* introduces a read-ahead mechanism to amortize the cost of pre-faulting and improve sequential I/O accesses. However, our main focus is to reduce synchronization overheads in the common *memory-mapped I/O* path and enhance scalability on multicore servers. A scalable I/O path enables us to maintain high device queue depth, an essential property for efficient use of fast storage devices.

Byte-addressable persistent memory DIMMs, attached in memory slots, can be accessed similarly to DRAM with the processor *load*/*store* instructions. Linux provides Direct Access (DAX), a mechanism that supports direct mapping of persistent memory to user address space. File systems that provide a DAX mmap [139, 140, 45, 34, 136] bypass I/O caching in DRAM. On the other hand, other works [65] have shown that DRAM caching benefits applications when the working set fits in DRAM and can hide higher persistent memory latency compared to DRAM (by up to $\sim 3\times$). Accordingly, *FastMap* uses DRAM caching and supports both block-based flash storage and byte-addressable persistent memory. *FastMap* will benefit all DAX mmap file systems that need to provide DRAM caching for *memory-mapped I/O*, as *FastMap* is file system agnostic. Additionally, *Aquila* can leverage DAX file systems to provide strict POSIX semantics to user applications and optimize the *memory-mapped I/O* path through the separation of protection and common path operations.

6.1.2 Providing a scalable virtual address space

Bonsai [32] shows that anonymous memory mappings, i.e. not backed by a file or device, suffer from scalability issues. This type of memory mapping is mainly used for user memory allocations, e.g. *malloc*. The scalability bottleneck in this case is due to a contended read-write lock, named *mmap_sem*, that protects access to a red-black tree that keeps VMAs (valid virtual address spaces ranges). In the case of page faults, this lock is

acquired as read lock. In the case of *mmap*/*munmap* this lock is acquired as write lock. Even in the read lock case, NUMA traffic in multicores limits scalability. *Bonsai* proposes the use of *RCU*-based binary tree to provide lock-free lookups, resulting in a system scaling up to 80 cores. *Bonsai* removes the bottleneck from concurrent page faults, but still serializes *mmap*/*munmap* operations even in non-overlapping address ranges.

In Linux, shared mappings backed by a file or device have a different path in the kernel, thus requiring a different design to achieve scalability. There are other locks (see Section 4.2.1) that cause scalability issues and *mmap_sem* does not result in any performance degradation. As we see from our evaluation of *FastMap*, using 80 cores the time spent in *mmap_sem* is 37.4% of the total execution time; therefore, *Bonsai* is complementary to our work and will also benefit our approach.

Furthermore, authors in [77] propose an alternative approach to provide scalable address space operations, by introducing scalable range locks to accelerate non-conflicting virtual address space operations in Linux.

RadixVM [33] addresses the problem of serialization of *mmap*/*munmap* in non overlapping address space ranges. This work is done in the *SV6* kernel and can also benefit from *FastMap* in a similar way to *Bonsai*. In *Aquila* we use the radix tree from RadixVM to provide scalable virtual address range management.

The authors in [21] propose techniques to scale Linux for a set of kernel-intensive applications, but do not tackle the scalability limitations of *memory-mapped I/O*. In *pedsort* authors modify the application to use one process per core for concurrency and avoid the contention over the shared address space. In this chapter we solve this issue at the kernel level, thus providing benefits to all user applications.

6.1.3 Extending the virtual address space over storage

The authors in [121] claim that by using *mmap* a user can effectively extend the main memory with fast storage devices. They propose an optimized page reclamation procedure with a new page recycling method to reduce context switches. This makes it possible to use extended vector I/O – a parallel page I/O method. In our work, we implement a

custom per-core mechanism for managing free pages. We also preallocate a memory pool that removes the performance bottlenecks identified in [121]. Additionally, we address scalability issues with *memory-mapped I/O*, whereas the work in [121] examines setups with up to eight cores, where the Linux kernel scales well.

FlashVM [117] uses a dedicated flash device for swapping virtual memory pages and provides flash-specific optimizations for this purpose. SSDAlloc [7] implements a hybrid DRAM/flash memory manager and a runtime library that allows applications to use flash for memory allocations in a transparent manner. SSDAlloc proposes the use of $16-32\times$ more flash than DRAM compared to FlashVM and to handle this increase they introduce a log-structured object store. Instead, *FastMap* and *Aquila* target the storage I/O path and reduces the overhead of *memory-mapped I/O*. Therefore, our work is not a replacement for swap nor does it provide specific optimizations to extend the process memory address space over SSDs.

NVMalloc [133] enables client applications in supercomputers to allocate and manipulate memory regions from a distributed block-addressable SSD store (over FUSE [89]). It exploits the *memory-mapped I/O* interface to access local or remote NVM resources in a seamless fashion for volatile memory allocations. NVMalloc uses Linux *mmap*. Consequently, it can also benefit from *FastMap* at large thread counts combined with fast storage devices.

SSD-Assisted Hybrid Memory [103] augments DRAM with SSD storage as an efficient cache (in object granularity) for Memcached [135]. Authors claim that managing a cache in a page granularity incurs additional overheads. As we provide an application agnostic approach, we stick to managing memory at page granularity and we optimize scalability in the common path.

6.1.4 Dataplane operating systems

Exokernel [46] proposes to separate the OS dataplane and the OS control plane to provide higher throughput and lower latency with specialized library OSes. Systems, such as IX [15], ZygOS [112], Shinjuku [71], MICA [86], and Chronos [73] optimize the network

path, while *Aquila* targets the storage path. Although storage and networking share similarities, they also require to address different challenges, e.g. related to virtual memory management.

Arrakis [110], in addition to networking targets storage I/O. Arrakis uses SR-IOV [79] to provide multiple virtual PCIe devices and handle protection and multiplexing in the I/O controller. *Aquila* takes a more holistic approach that includes the user-space storage cache and improves virtual memory management and device access in *memory-mapped I/O*.

Similar to IX [15], ZygOS [112], and Shinjuku [71] *Aquila* uses Dune [14] to have access to privileged hardware features. All of these systems target networking while *Aquila* provides a way to access fast storage devices with the minimum required overhead. Finally, ReFlex [76] provides an optimized way to access remove flash storage. It uses Dune to closely integrate networking and storage processing and achieves low latency and high throughput in the storage server. In *Aquila* we assume that the server has access to fast local storage devices. Approaches similar to ReFlex can be used to extend our work for fast remote storage.

6.2 Persistent Key-Value Stores

6.2.1 LSM-Tree based key-value stores taxonomy and optimizations

We identify the following three dimensions in the design space of LSM-Tree key-value stores that affect I/O amplification and CPU efficiency: (1) size and placement of SSTs, (2) logical level organization, and (3) value location. In Table 6.1, we present a taxonomy of existing systems based on these dimensions. These systems, to some extend, have tried to take advantage of device properties and improve performance.

Size and placement of SSTs: In an LSM-Tree key-value store levels are physically organized on the device in units named Sorted String Tables (SSTs). Their size is typically large (order of MBs) because *large SSTs* guarantee maximum device throughput, eliminating the effects of the I/O pattern (sequential or random) and metadata I/Os, as metadata are small

and fit in memory. Equivalent, *small and sequentially* placed SSTs on the device [118] can achieve the same properties of maximum device throughput. This results in high I/O amplification but improves read performance at the expense of maintaining more metadata. Emerging device technologies allow using *small SSTs with random placement* which introduces randomness but has the potential to reduce I/O amplification [106]. This approach is suitable for devices where random I/O throughput degrades gracefully compared to sequential I/O throughput.

Logical level organization: Keys in each level are logically organized either *fully* or *partially*. Full organization keeps the key space in fully sorted, non-overlapping SSTs. Full organization is usually done with leveling compaction [102, 49, 134, 9, 40, 141, 53, 81, 91, 27]. However, B-tree indexes have also been used to either optimize reads and scans [118] or reduce amplification [106]. Partial organization introduced in [66] maintains the key space in overlapping units, e.g. in the form of tiering compaction [137, 39, 113, 96, 72] which reduces merge amplification at the cost of reduced read and scan performance.

Value location: Finally, values can be placed either *in-place* with keys or in a separate *value log*. Typically, values are stored in-place because this results in optimal scan behavior at the expense of increasing amplification due to value movement during merge operations. Previous work has proposed techniques [27, 81, 91, 105, 106] that store values in a log, reducing amplification significantly, relying on modern devices to alleviate the impact on scan performance.

In addition to the basic three dimensions, key-value stores employ a set of various techniques to optimize for different aspects of system operation. These include tail latency or targeted optimizations for specific workloads (e.g updates). Next, we present a set of representative techniques.

bLSM [118], that targets HDDs, uses a B-tree index per level and bloom filters to enable efficient look-up operations. *Kreon* shares the idea of a B-tree index per level but keeps an index only for the metadata and it does not fully rewrite levels during spills trading I/O randomness for CPU efficiency. bLSM also introduces gear scheduling to bound write latency.

key-value Stores	SST size, placement	Organization	Value placement
LSM[102], RocksDB[49],Locs[134], Dostoevsky[40], Triad[9], Mutant[141], bLSM[118], Silk[10], cLSM[53], VT-tree [119]	Large	Full	In-place
Atlas[81], WiscKey[91], HashKV[27]	Large	Full	Log
Jungle[2]	Large	Partial	Log
LSM-trie[137], Monkey[39], SifrDB[96], Novelsm[72], PebblesDB[113]	Large	Partial	In-place
Kreon[106]	Small	Full	Log
B$^{\epsilon}$–Tree[17]	Small	Full	In-place

Table 6.1: Taxonomy of the main approaches to design key-value stores in three dimensions.

Gear scheduling is a progress-based compaction scheduler that throttles compactions in the lower levels of the LSM-Tree. This scheduler is able to prioritize compactions taking place in the higher levels of the LSM-Tree, close to the in memory component improving tail latency of client applications. Silk [10] similar to bLSM tackles the same problem by introducing progress based compaction. In addition to bLSM, Silk performs compactions during off peak periods to reduce the probability of heavy compactions during bursty client activity. With this approach it trades increased I/O amplification, since it does not merge adjacent levels that necessarily grow by a constant factor f [11, 102], for bounding tail latency. FD-tree [85] is an LSM-Tree for flash devices, that replaces bloom filters, saving their corresponding memory budget, with fractional cascading [28] to speed up look up operations. VT-tree [119] tries to reduce I/O amplification merging only the actual overlapping parts between SSTs. In particular, instead of blindly performing a merge sort operation between a set of SSTs of L_i and L_{i+1} it identifies which parts are non-overlapping and it performs merge sort only for the overlapping parts. These techniques are orthogonal to *Kreon* and it can benefit from them.

LSM-trie [137] identifies as the main source of I/O amplification during the compaction process between two levels L_i and L_{i+1} the amount of data read and written from L_{i+1}. This is because L_{i+1} is f times larger than L_i and the compaction process always needs to read and write L_i to merge it with L_{i+1}. To reduce the data read and written from L_{i+1} it divides it

into sub-levels and place keys in SSTs according to their key hash. Then it uses bits of the hash to select a part of the sublevel of L_{i+1} that it merge sorts with the SST of L_i. This optimization trades range queries in favor of reduced amplification. *Kreon* shares the same goal of reducing I/O amplification by using a value log and small SSTs but also supports range queries.

Atlas [81] is a key-value store that aims to improve data-serving density and data replica space efficiency. To achieve these, Atlas employs a LSM–based approach and separates keys from values to avoid moving values during compactions. Similarly, WiscKey [91] proposes the separation of keys and values to reduce write amplification. It stores values in a data log and keeps a LSM index for the keys. Furthermore, it implements a prefetching mechanism for speeding up range queries because values are written randomly on the device. Jungle [2] is an LSM-Tree key-value store optimized for updates. It uses a value log and organizes its levels using tiering. In particular, each level is organized as a forest of B-trees with overlapping key ranges. This organization reduces I/O amplification [11] but hurts look up performance. This is because each look up should check in the worst case all B-trees within a level. However, this trade off is a good fit for Jungle since it targets update heavy workloads, where the number of distinct keys grow slow.

PebblesDB [113] identifies as the main problem of write amplification in the LSM-Tree the repeated merges of files at each level during compaction. To fix this, it keeps overlapping sorted files at each level instead of non-overlapping. However, this approach adds overhead in the read path since multiple files need to be checked instead of a single. To improve this, PebblesDB introduces guards which act as a coarse grain index per level inspired by skip lists. *Kreon* shares the idea of using an index per level with the difference that in *Kreon* case is full. Furthermore, it uses *memory-mapped I/O*, keeps both keys and values on a separate log, and executes spill operations only on pointers to keys and values.

6.2.2 Other write optimized data structures

TokuDB [125] implements at its core a B^{ϵ}–Tree structure. It keeps a global B-tree index in which it associates a small buffer per B-tree node. Buffers are relatively small so it keeps

them unsorted and scans them during look-up queries. When a buffer fills it is spilled to its N children, where N is the fan out of the B-tree. Tucana [105] uses a B^{ϵ}–Tree which buffers keys only at the last level of the tree and relies on a ratio of memory/data to operate efficiently. *Kreon* keeps a buffer per level in order to achieve better batching and is able to server larger datasets with smaller memory/data ratio.

KVell [84] is an efficient log structured key-value store designed for fast storage devices. It uses a value log and a single level B-tree index in which it stores metadata (pointers) to the actual key-value pairs. Furthermore, it uses asynchronous IO (io_uring [70]) and batching to perform efficient I/O with the devices. *Kreon* shares the same efficiency goals for fast storage devices with KVell but it uses *memory-mapped I/O*. This eliminates the need to constantly perform look-up operations about the position of a device block in DRAM even in the case of hits. This can save up to 30% [58] CPU overhead when the dataset fits in memory. Furthermore, *Kreon* uses an LSM-Tree structure which allows its index to grow efficiently beyond DRAM limits without the known performance issues of a B-tree index structure [56].

Chapter 7

Future Work

This dissertation provides solutions to the main pathologies of memory-mapped I/O for data-intensive applications over fast storage devices. Next, we describe some directions for future work.

7.1 Huge Pages in Memory-Mapped I/O

In this dissertation, we use regular ($4KB$) pages. Other works [80, 104] proposed mechanisms that use huge pages ($2MB$) for anonymous mappings. These include a dynamic mechanism to use both regular and huge pages. The main advantage of using huge pages in anonymous mappings is that it reduces page faults by a large factor. Additionally, it reduces TLB pressure, as TLBs have a relatively small number of entries compared to the number of regular page mappings in a modern server with $100s$ of GB of DRAM.

Furthermore, huge pages can produce large ($2MB$) I/Os to devices. With sequential access patterns, this can further improve the performance combined with the lower number of page faults and less TLB misses. An initial exploration shows [92] that the use of huge pages in memory-mapped I/O for sequential accesses results in significant performance benefits with lower CPU consumption. On the other hand, using huge pages for random accesses can increase I/O amplification and reduce performance. This leads to the need for dynamic approaches (i.e. to use both regular and huge pages) in memory-mapped I/O, based on access patterns.

7.2 Memory-Mapped I/O and Persistent Memory

Byte-addressable NVM devices (i.e. persistent memory – PMEM) attached to memory DIMMs provide even lower access latency and higher throughput than block-addressable flash-based storage devices [65]. Additionally, the user can access PMEM using processor load and store instructions rather than DMA. Therefore, the user can map PMEM directly to its address space, bypassing DRAM. DAX [87] is a mechanism that allows using PMEM for storage. DAX does not require page faults and eviction operations in the common path. Only the first access to each page results in a page fault because it does not use DRAM as a storage I/O cache. At the same time, PMEM has higher access latency and lower throughput than DRAM [65]. In the case where the workload exhibits high spatial locality, there can be benefits from DRAM caching. Understanding these tradeoffs and their implications for data transfer appears to be a fairly complex and very intresting direction for future work.

7.3 Physical Memory Extension

In this dissertation, we use persistent key-value stores for the evaluation of our proposed mechanisms. In Chapter 4, we also evaluate memory-mapped I/O in terms of increasing the virtual address space for user processes over fast storage devices. We translate all volatile memory allocations (i.e. *malloc*/*free*) over a memory-mapped file or device. This approach makes the available virtual memory size of a process relative to the device size and not the physical DRAM size. This dissertation shows that Linux memory-mapped I/O is not suitable for this purpose as it introduces significant slowdown. Future work should consider further techniques and tradeoffs with respect to locality and memory use of heaps in modern data-intensive applications.

7.4 Beyond Persistent Key-Value Stores

In this dissertation, we focus on persistent key-value stores. The intuition behind this is that they are an important building block for a wide range of applications. On the other

hand, there is a large number of applications that use different ways to persist their data. These include databases (i.e. SQL and NoSQL), graph procecssing, and various persistent data structures. Today, it is common to design such systems with a user-space cache and system calls to access persistent data. We propose the use of memory-mapped I/O to manage storage caching, and this approach removes the software cost entirely in the case of hits.

However, using memory-mapped I/O requires to redesign certain aspects of the application I/O path. Our contributions can still provide benefits to these systems, similar to persistent key-value stores. Priorities (Chapter 3) can affect the eviction/writeback path based on user needs. Our scalable memory-mapped I/O path (Chapter 4) can provide significant performance gains in multi-threaded applications that are typical today. Finally, removing protection domain switches in the case of misses (Chapter 5) can increase the single-thread performance, and these benefits are more pronounced in low latency storage devices.

7.5 Dependence on Storage Devices

In this dissertation, we use local flash-based storage devices either connected to SATA or PCIe bus. We use the Linux I/O path to issue I/Os and check for their completion. In the last part of our work (Chapter 5), we also emulate byte-addressable storage devices with DRAM and access them with memory copies. This approach results in minimal overheads but spends precious CPU cycles for data movement.

Our contributions are applicable to different types of storage devices and ways to move data. Future work includes the evaluation of different mechanisms to access byte-addressable storage devices. These include memory copies from the processor, DMAs between different memory areas, and custom hardware mechanisms. Furthermore, the use of disaggregated NVM accessible over fast networks (i.e. RDMA) can be also considered. The general principle from our contributions is that the faster the underlying storage is, the more pronounced the benefits from our work are.

Chapter 8

Conclusions

In this dissertation, we propose the use of memory-mapped I/O to manage storage caches and remove software overheads for hits. We show that the use of memory-mapped I/O to access data over fast storage devices provides significant improvements compared to read/write system calls and user-space caching that is typical today.

Memory-mapped I/O removes the need for cache lookups in the common path as they are handled by hardware through the virtual memory mappings. On the other hand, Linux memory-mapped I/O has significant issues for data-intensive applications. These issues include: (1) the lack of control for evictions, which results in unpredictable performance under heavy I/O, (2) scalability issues with increasing the number of application threads, and (3) the high page fault cost that happen during misses.

We address these issues as follows. First, we design a persistent key-value store that uses memory-mapped I/O to access storage devices. We demonstrate the advantages and drawbacks of memory-mapped I/O. Second, we propose a priority-based mechanism that handles evictions in memory-mapped I/O, based on application needs. To show the applicability of this mechanism, we build an efficient memory-mapped I/O persistent key-value store that makes use of page priorities and shows significant improvements in both throughput and tail latency. Third, we remove all centralized contention points in memory-mapped I/O path and provide a scalable path for fast storage devices. Finally, we separate protection and common path operations in memory-mapped I/O to reduce protection costs. We leverage CPU virtualization extensions to reduce the overhead of page

faults and maintain the protection semantics of the OS.

Overall, applications designed to use memory-mapped I/O to access storage devices can benefit from our techniques with minimal modifications. Our results show that using memory-mapped I/O provides significant performance improvements with lower CPU consumption.

Bibliography

[1] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'19, pages 971–985, New York, NY, USA, 2019. Association for Computing Machinery.

[2] Jung-Sang Ahn, Mohiuddin Abdul Qader, Woon-Hak Kang, Hieu Nguyen, Guogen Zhang, and Sami Ben-Romdhane. Jungle: Towards dynamically adjustable key-value store by combining lsm-tree and copy-on-write b+-tree. In *Proceedings of the 11th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'19, page 9, USA, 2019. USENIX Association.

[3] Nadav Amit. Optimizing the TLB shootdown algorithm with page access tracking. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 27–39, Santa Clara, CA, July 2017. USENIX Association.

[4] Nadav Amit, Amy Tai, and Michael Wei. Don't Shoot down TLB Shootdowns! In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[5] Apache. Hbase. `https://hbase.apache.org/`. Accessed: January 10, 2021.

[6] Jens Axboe. FIO: Flexible I/O Tester. `https://github.com/axboe/fio`. Accessed: January 10, 2021.

[7] Anirudh Badam and Vivek S. Pai. Ssdalloc: Hybrid ssd/ram memory management made easy. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 211–224, USA, 2011. USENIX Association.

[8] Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2b-ssd: The case for dual, byte- and block-addressable solid-state drives. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA'18, pages 425–438. IEEE Press, 2018.

[9] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. Triad: Creating synergies between memory, disk and log in log structured key-value stores. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 363–375, Berkeley, CA, USA, 2017. USENIX Association.

[10] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. Silk: Preventing latency spikes in log-structured merge key-value stores. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 753–766, USA, 2019. USENIX Association.

[11] Nikos Batsaras, Giorgos Saloustros, Anastasios Papagiannis, Panagiota Fatourou, and Angelos Bilas. VAT: asymptotic cost analysis for multi-level key-value stores. *CoRR*, abs/2003.00103, 2020.

[12] R. Bayer and M. Schkolnick. *Concurrency of Operations on B-Trees*, pages 129–139. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[13] Rudolf Bayer and Edward McCreight. *Organization and maintenance of large ordered indexes*. Springer, 2002.

[14] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Pre-*

sented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), pages 335–348, Hollywood, CA, 2012. USENIX.

[15] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association.

[16] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 423–436, USA, 2010. USENIX Association.

[17] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. An introduction to b^ϵ-trees and write-optimization. *;Login: The USENIX magazine*, 40(5):22–28, October 2015.

[18] Philip Bohannon, Peter Mcllroy, and Rajeev Rastogi. Main-memory index structures with fixed-size partial keys. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 163–174, New York, NY, USA, 2001. ACM.

[19] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, December 2008.

[20] Jeff Bonwick and Bill Moore. Zfs: The last word in file systems, 2007.

[21] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, USA, 2010. USENIX Association.

[22] Gerth Stolting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 546–554, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[23] R. Burns and W. Hineman. A bit-parallel search algorithm for allocating free space. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2001. Proceedings. Ninth International Symposium on*, pages 302–310, 2001.

[24] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. Numa-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 157–166, New York, NY, USA, 2013. ACM.

[25] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.

[26] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SIAM International Conference on Data Mining*, 2004.

[27] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. Hashkv: Enabling efficient updates in kv storage via hashing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 1007–1019, Berkeley, CA, USA, 2018. USENIX Association.

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

[29] Shimin Chen, Phillip B Gibbons, and Todd C Mowry. Improving index performance through prefetching. *ACM SIGMOD Record*, 30(2):235–246, 2001.

[30] Kristina Chodorow. *MongoDB: The Definitive Guide*. O'Reilly Media, second edition, 5 2013.

[31] Jungsik Choi, Jiwon Kim, and Hwansoo Han. Efficient memory mapped file i/o for in-memory file systems. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, 2017. USENIX Association.

[32] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using rcu balanced trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 199–210, New York, NY, USA, 2012. ACM.

[33] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Radixvm: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 211–224, New York, NY, USA, 2013. ACM.

[34] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through Byte-Addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP'09, pages 133–146, New York, NY, USA, 2009. Association for Computing Machinery.

[35] Brian F. Cooper. Core workloads. `https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads`, 2018.

[36] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[37] 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.

[38] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'15, pages 631–644, New York, NY, USA, 2015. Association for Computing Machinery.

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

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

[41] 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. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[42] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017.

[43] Laurent Dufour. Speculative page faults (Linux 4.14 patch). `https://lkml.org/lkml/2017/10/9/180`, 2017.

[44] Laurent Dufour. Speculative page faults. `https://lwn.net/Articles/786105/`, 2019.

[45] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory.

In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[46] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP'95, pages 251–266, New York, NY, USA, 1995. Association for Computing Machinery.

[47] B. V. Essen, H. Hsieh, S. Ames, and M. Gokhale. Di-mmap: A high performance memory-map runtime for data-intensive applications. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 731–735, Nov 2012.

[48] Brian Essen, Henry Hsieh, Sasha Ames, Roger Pearce, and Maya Gokhale. Di-mmap–a scalable memory-map runtime for out-of-core data-intensive applications. *Cluster Computing*, 18(1):15–28, March 2015.

[49] Facebook. RocksDB. `https://rocksdb.org/`. Accessed: January 10, 2021.

[50] Facebook. RocksDB IO. `https://github.com/facebook/rocksdb/wiki/IO`. Accessed: January 10, 2021.

[51] Facebook. RocksDB Tuning Guide. `https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide`. Accessed: January 10, 2021.

[52] Facebook. Rocksdb performance benchmarks. `https://github.com/facebook/rocksdb/wiki/Performance-Benchmarks`, 2015.

[53] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 32:1–32:14, New York, NY, USA, 2015. ACM.

[54] Google. Leveldb. `http://leveldb.org/`. Accessed: January 10, 2021.

[55] Google. Leveldb benchmarks. `https://leveldb.googlecode.com/svn/trunk/doc/benchmark.html`, 2015.

[56] Goetz Graefe. Write-optimized b-trees. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 672–683. VLDB Endowment, 2004.

[57] Brendan Gregg. The flame graph. *Queue*, 14(2):10:91–10:110, March 2016.

[58] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. Oltp through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD'08, pages 981–992, New York, NY, USA, 2008. ACM.

[59] Jian Huang, Anirudh Badam, Moinuddin K. Qureshi, and Karsten Schwan. Unified address translation for memory-mapped ssds with flashmap. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 580–591, New York, NY, USA, 2015. Association for Computing Machinery.

[60] IBM. Power ISA. Version 2.06 Revision B.

[61] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, Martin Kersten, et al. Monetdb: Two decades of research in column-oriented database architectures. *A Quarterly Bulletin of the IEEE Computer Society Technical Committee on Database Engineering*, 35(1):40–45, 2012.

[62] Intel. OPTANE DC Persistent Memory. `https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html`. Accessed: January 10, 2021.

[63] Intel. OPTANE SSD DC P4800X SERIES. `https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/optane-dc-p4800x-series.html`. Accessed: January 10, 2021.

[64] Intel. Virtualization Technology Specification for the Intel Itanium Architecture (VT-i). Accessed: January 10, 2021.

[65] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen

Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[66] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and Rama Kanneganti. Incremental organization for data recording and warehousing. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 16–25, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[67] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Betrfs: A right-optimized write-optimized file system. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 301–315, Santa Clara, CA, February 2015. USENIX Association.

[68] jemalloc. `http://jemalloc.net/`. Accessed: January 10, 2021.

[69] Bob Jenkins. A hash function for hash Table lookup. `http://www.burtleburtle.net/bob/hash/doobs.html`, 2009.

[70] Jens Axboe. Efficient IO with io_uring. `https://kernel.dk/io_uring.pdf`. Accessed: January 10, 2021.

[71] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, Boston, MA, February 2019. USENIX Association.

[72] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning lsms for nonvolatile memory with novelsm. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 993–1005, Berkeley, CA, USA, 2018. USENIX Association.

[73] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings*

of the Third ACM Symposium on Cloud Computing, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[74] Linux kernel. cgroups. `https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt`. Accessed: January 10, 2021.

[75] Linux kernel. Userfaultfd. `https://www.kernel.org/doc/Documentation/vm/userfaultfd.txt`. Accessed: January 10, 2021.

[76] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote Flash ≈ Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'17, pages 345–359, New York, NY, USA, 2017. Association for Computing Machinery.

[77] Alex Kogan, Dave Dice, and Shady Issa. Scalable range locks for scalable address spaces and beyond. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys'20, New York, NY, USA, 2020. Association for Computing Machinery.

[78] Bradley Kuszmaul. A comparison of fractal trees to log-structured merge (lsm) trees. *White Paper*, 2014.

[79] Patrick Kutch. PCI-SIG SR-IOV primer: An introduction to SR-IOV technology, 2011. Intel application note, 321211-002.

[80] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 705–721, Savannah, GA, November 2016. USENIX Association.

[81] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. Atlas: Baidu's key-value storage system for cloud data. In *MSST*, pages 1–14. IEEE Computer Society, 2015.

[82] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[83] Leslie Lamport. Concurrent reading and writing. *Commun. ACM*, 20(11):806–811, November 1977.

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

[85] Y. Li, B. He, Q. Luo, and K. Yi. Tree indexing on flash disks. In *2009 IEEE 25th International Conference on Data Engineering*, pages 1303–1306, March 2009.

[86] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast in-Memory Key-Value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 429–444, USA, 2014. USENIX Association.

[87] Linux. Direct Access for files. `https://www.kernel.org/doc/Documentation/filesystems/dax.txt`. Accessed: January 10, 2021.

[88] Linux. KVM - Nested VMX. `https://www.kernel.org/doc/Documentation/virtual/kvm/nested-vmx.txt`. Accessed: January 10, 2021.

[89] Linux FUSE (Filesystem in Userspace). `https://github.com/libfuse/libfuse`. Accessed: January 10, 2021.

[90] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer, and Babak Falsafi. Scale-out processors. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 500–511, Washington, DC, USA, 2012. IEEE Computer Society.

[91] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, Santa Clara, CA, February 2016. USENIX Association.

[92] Ioannis Malliotakis, Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. Hugemap: Optimizing memory-mapped i/o with huge pages for fast storage. In *Workshop on Challenges and Opportunities of HPC Storage Systems 2020 (CHAOSS'20)*, 2020.

[93] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196. ACM, 2012.

[94] W. Mauerer. *Professional Linux Kernel Architecture*. Wrox professional guides. Wiley, 2008.

[95] Paul E McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-copy update. In *AUUG Conference Proceedings*, page 175. AUUG, Inc., 2001.

[96] Fei Mei, Qiang Cao, Hong Jiang, and Jingjun Li. Sifrdb: A unified solution for write-optimized key-value stores in large datacenter. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 477–489, New York, NY, USA, 2018. ACM.

[97] Dutch T. Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield. Parallax: Virtual disks for virtual machines. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 41–54, New York, NY, USA, 2008. ACM.

[98] Microsoft. Run Hyper-V in a Virtual Machine with Nested Virtualization. `https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/user-guide/nested-virtualization`. Accessed: January 10, 2021.

[99] MonetDB. `https://www.monetdb.org/Home`. Accessed: January 10, 2021.

[100] Null block device driver. `https://www.kernel.org/doc/Documentation/block/null_blk.txt`. Accessed: January 10, 2021.

[101] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley db. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association.

[102] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.*, 33(4):351–385, June 1996.

[103] X. Ouyang, N. S. Islam, R. Rajachandrasekar, J. Jose, M. Luo, H. Wang, and D. K. Panda. Ssd-assisted hybrid memory to accelerate memcached over high performance networks. In *2012 41st International Conference on Parallel Processing*, pages 470–479, Sep. 2012.

[104] Ashish Panwar, Sorav Bansal, and K. Gopinath. Hawkeye: Efficient fine-grained os support for huge pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 347–360, New York, NY, USA, 2019. Association for Computing Machinery.

[105] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 537–550, Denver, CO, 2016. USENIX Association.

[106] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. An efficient memory-mapped key-value store for flash storage. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 490–502, New York, NY, USA, 2018. ACM.

[107] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. Optimizing memory-mapped i/o for fast storage devices. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020.

[108] I. Peng, M. McFadden, E. Green, K. Iwabuchi, K. Wu, D. Li, R. Pearce, and M. Gokhale. UMap: Enabling Application-driven Optimizations for Page Management. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 71–78, 2019.

[109] Persistent Memory Development Kit (PMDK). `https://pmem.io/pmdk/`. Accessed: January 10, 2021.

[110] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 1–16, USA, 2014. USENIX Association.

[111] pmem.io: Persistent Memory Programming. `http://pmem.io/`. Accessed: January 10, 2021.

[112] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP'17, pages 325–341, New York, NY, USA, 2017. Association for Computing Machinery.

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

[114] Jinglei Ren. YCSB-C. `https://github.com/basicthinker/YCSB-C`, 2016.

[115] Ohad Rodeh. B-trees, shadowing, and clones. *Trans. Storage*, 3(4):2:1–2:27, February 2008.

[116] Allen Samuels. The consequences of infinite storage bandwidth. `https://goo.gl/Xfo7Lu`, 2018.

[117] Mohit Saxena and Michael M. Swift. Flashvm: Virtual memory management on flash. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'10, page 14, USA, 2010. USENIX Association.

[118] Russell Sears and Raghu Ramakrishnan. blsm: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 217–228, New York, NY, USA, 2012. ACM.

[119] Pradeep J. Shetty, Richard P. Spillane, Ravikant R. Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building workload-independent storage with vt-trees. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 17–30, San Jose, CA, 2013. USENIX.

[120] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, New York, NY, USA, 2013. ACM.

[121] Nae Young Song, Yongseok Son, Hyuck Han, and Heon Young Yeom. Efficient memory-mapped i/o on fast storage device. *ACM Trans. Storage*, 12(4):19:1–19:27, May 2016.

[122] SPDK – BlobFS. `https://spdk.io/doc/blobfs.html`. Accessed: January 10, 2021.

[123] SPDK – Blobstore. `https://spdk.io/doc/blob.html`. Accessed: January 10, 2021.

[124] Storage Performance Development Kit (SPDK). `https://spdk.io/`. Accessed: January 10, 2021.

[125] INC TOKUTEK. Tokudb: Mysql performance, mariadb performance, 2013.

[126] TPC-C. `http://www.tpc.org/tpcc/`. Accessed: January 10, 2021.

[127] TPC-H. `http://www.tpc.org/tpch/`. Accessed: January 10, 2021.

[128] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the*

Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP'13, pages 18–32, New York, NY, USA, 2013. Association for Computing Machinery.

[129] R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C.M. Martins, A.V. Anderson, S.M. Bennett, A. Kagi, F.H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48–56, May 2005.

[130] Prashant Varanasi and Gernot Heiser. Hardware-Supported Virtualization on ARM. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, New York, NY, USA, 2011. Association for Computing Machinery.

[131] VMware. Running Nested VMs. `https://communities.vmware.com/docs/DOC-8970`. Accessed: January 10, 2021.

[132] Inc. VMware. Vmware virtual san 6.1 product datasheet. `https://www.vmware.com/files/pdf/products/vsan/VMware_Virtual_SAN_Datasheet.pdf`. Accessed: January 10, 2021.

[133] C. Wang, S. S. Vazhkudai, X. Ma, F. Meng, Y. Kim, and C. Engelmann. Nvmalloc: Exposing an aggregate ssd store as a memory partition in extreme-scale machines. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 957–968, May 2012.

[134] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An efficient design and implementation of lsm-tree based key-value store on open-channel ssd. In *Proceedings of the Ninth European Conference on Computer Systems*, page 16. ACM, 2014.

[135] Open Source Software. Memcached was originally developed by Brad Fitzpatrick for LiveJournal in 2003. Memcached. `http://memcached.org/`. Accessed: January 10, 2021.

[136] Xiaojian Wu, Sheng Qiu, and A. L. Narasimha Reddy. SCMFS: A File System for Storage Class Memory and Its Extensions. *ACM Trans. Storage*, 9(3), August 2013.

[137] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 71–82, Santa Clara, CA, July 2015. USENIX Association.

[138] Xen. Nested Virtualization in Xen. `https://wiki.xenproject.org/wiki/Nested_Virtualization_in_Xen`. Accessed: January 10, 2021.

[139] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.

[140] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP'17, pages 478–496, New York, NY, USA, 2017. Association for Computing Machinery.

[141] Hobin Yoon, Juncheng Yang, Sveinn Fannar Kristjansson, Steinn E. Sigurdarson, Ymir Vigfusson, and Ada Gavrilovska. Mutant: Balancing storage cost and latency in lsm-tree data stores. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 162–173, New York, NY, USA, 2018. ACM.

[142] Erez Zadok, Ion Badulescu, and Alex Shender. Extending file systems using stackable templates. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 5–5, Berkeley, CA, USA, 1999. USENIX Association.