LATENCY TRADEOFFS IN DISTRIBUTED STORAGE ACCESS

A Dissertation
Submitted to
the Temple University Graduate Board

In Partial Fulfillment
of the Requirements for the Degree of
DOCTOR OF PHILOSOPHY

---

by
Madhurima Ray
December 2019

Examining committee members:

Dr. Krishna Kant, Dissertation Advisory Chair, Department of Computer and Information Sciences
Dr Justin Shi, Department of Computer and Information Sciences
Dr Xubin He, Department of Computer and Information Sciences
Dr Sanjeev Trika, Intel Corporation

ABSTRACT

The performance of storage systems is central to handling the huge amount of data being generated from a variety of sources including scientific experiments, social media, crowdsourcing, and from an increasing variety of cyber-physical systems. The emerging high-speed storage technologies enable the ingestion of and access to such large volumes of data efficiently. However, the combination of high data volume requirements of new applications that largely generate unstructured and semistructured streams of data combined with the emerging high-speed storage technologies pose a number of new challenges, including the low latency handling of such data and ensuring that the network providing access to the data does not become the bottleneck.

The traditional relational model is not well suited for efficiently storing and retrieving unstructured and semi-structured data. An alternate mechanism, popularly known as Key-Value Store (KVS) has been investigated over the last decade to handle such data. A KVS store only needs a "key" to uniquely identify the data record, which may be of variable length and may or may not have further structure in the form of predefined fields. Most of the KVS in existence have been designed for hard-disk based storage (before the SSDs gain popularity) where avoiding random accesses is crucial for good performance. Unfortunately, as the modern solid-state drives become the norm as the data center storage, the HDD-based KV structures result in high read, write, and space amplifications which becomes detrimental to both the SSD's performance and endurance.

Also note that regardless of how the storage systems are deployed, access to large amounts of storage by many nodes must necessarily go over the network. At the same time, the emerging storage technologies such as Flash, 3D-crosspoint, phase change memory (PCM), etc. coupled with highly efficient access protocols such as NVMe are capable of ingesting and reading data at rates that challenge even the leading edge networking technologies such as 100Gb/sec Ethernet. At the same time, some of the higher-end storage technologies (e.g., Intel Optane storage based on 3-D crosspoint technology, PCM, etc.) coupled with lean protocols like NVMe are capable of providing storage access latencies in the 10-20μs range, which means that the additional latency due to network congestion could become significant. The purpose of this thesis is to addresses some of the aforementioned issues.

We propose a new hash-based and SSD-friendly key-value store (KVS) architecture called FlashKey which is especially designed for SSDs to provide low access latencies, low read and write amplification, and the ability to easily trade-off latencies for any sequential access, for example, range queries. Through detailed experimental evaluation of FlashKey against the two most popular KVs, namely, RocksDB and LevelDB, we demonstrate that even as an initial implementation we are able to achieve substantially better write amplification, average, and tail latency at a similar or better space amplification.

Next, we try to deal with network congestion by dynamically replicating the data items that are heavily used. The tradeoff here is between the latency and the replication or migration overhead. It is important to reverse the replication or migration as the congestion fades away since our observation tells that placing data and applications (that access the data) together in a consolidated fashion would significantly reduce the propagation delay and increase the network energy-saving opportunities which is required as the data center network nowadays are equipped with high-speed and power-hungry network infrastructures.

Finally, we designed a tradeoff between network consolidation and congestion. Here, we have traded off the latency to save power. During the quiet hours, we consolidate the traffic is fewer links and use different sleep modes for the unused links to save powers. However, as the traffic increases, we reactively start to spread out traffic to avoid congestion due to the upcoming traffic surge. There are numerous studies in the area of network energy management that uses similar approaches, however, most of them do energy management at a coarser time granularity (e.g. 24 hours or beyond). As opposed to that, our mechanism tries to steal all the small to medium time gaps in traffic and invoke network energy management without causing a significant increase in latency.

This dissertation is dedicated to...

... my loving and encouraging... parents Mr. Ansuman and Mrs. Sukla Ray, husband Mr. Sayon Ray and uncle Dr. Tarun K. Ray

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The emergence of data-intensive applications that demand low latency and technological improvement in both storage/network infrastructure and protocols demand new technologies for storing and managing data. In my thesis, I have tried to address some of these issues by exploring three key aspects: (a) a novel data storage system for efficiently storing and accessing large volumes of unstructured data being generated by emerging applications, (b) a coordinated management of network and storage to address the issues of network congestion arising from emerging storage technologies that can provide high throughput and to manage network energy consumption that is becoming a larger percentage of overall data center energy consumption, and (c) a proposal to implement end-to-end QoS for remote storage access. These issues are introduced in more detail in the following subsections.

1.1 FlashKey:A High-Performance Flash Friendly Key-Value Store

A number of emerging online applications including social media, e-commerce, scientific experiments, IoT monitoring of smart infrastructures, etc, generate streams of unstructured data that need to be stored and managed with very low latency. Key-value (KV) Stores are being used increasingly for this purpose because the traditional RDBMS are not well suited

for schemaless data. The current KV stores were largely designed for HDDs; we show in this thesis that by specializing the KVS design for emerging storage technologies such as Solid-State Disks (SSDs) we can achieve substantially lower latencies and longer device lifetimes.

The popular KVSes are largely designed after HDDs as backend, hence there are several drawbacks in their design. For example, the random IO performance of SSDs is comparable to their sequential performance unlike the HDDs where the sequential performance can be 2-3 orders of magnitude better than random performance. To hide the poor random access performance, the above mentioned KVSes perform a lot of merges and sorts of stored data to keep the data strictly sorted. The resulting reads and writes due to these merge/sort not only lower the performance but can also be very detrimental to endurance if the HDD is simply replaced by a SSD.

As SSDs become widely deployed in data centers, there is an opportunity to redesign the KV architecture to benefit from the SSDs' fast random performance. In our work (Ray et al., 2019), we have designed a flash friendly KV store called FlashKey, with the data structures and algorithms that exploit the good random IO performance by using hashing. It can also work with low-endurance SSDs because of significantly reduced write-amplification compared to the popular KVSes at the similar or better over-provisioning. The solution uses a hash-table structure to provide fast "Get" operations and a "logical band" based data organization that allows partial sequentialization of "Put" data writes to the SSD, since sequential writes still provide better performance and also improve the endurance of the SSD. Finally, FlashKey also includes a Log-Structured-Merge (LSM) tree structure for keys, without values, to support "Scan" operations.

Through a detailed experimental evaluation *on current SSDs*, we show that FlashKey, in spite of being an initial implementation, outperforms not only LevelDB, but also the RocksDB. In particular, we demonstrate the unique design of FlashKey optimizes SSD accesses and thus provides a much lower write amplification than both LevelDB and

RocksDB at comparable space amplification. It also provides a much lower average latency than both the KVSes under comparison. The improvement is not limited to only average latency, instead, we observe a lower 99.999 percentile read tail latency as well. The read amplification of FlashKey is comparable and, in fact, generally lower than that for LevelDB and RocksDB. Since FlashKey prioritizes high performance for individual record accesses over that for large key ranges, it is not as performant as the others when compared for the range queries. Nevertheless, it can handle range queries via the additional key-only LSM structure.

We show that for Get() and Put() operations, FlashKey achieves up to 85% improvement in average latency, 2x improvement in tail latencies, and 12x improvement in write amplification, at comparable or better space-amplification. The reasoning behind the improvements in Get() performance is that the hash lookup eliminates the need for tree-traversal on the disk. Finding the key index needs O(1) time with one storage access on average. Typically two storage accesses are required to complete a Get() even when the hash-table is on disk. Also, a strong primary hash function reduces collision and a secondary hash is used to avoid unnecessary disk lookup(s) upon primary hash collision. A typical Put(.) operation (without triggering GC) in FlashKey requires two writes to disk which provides the improvement in Put(). Finally, both Put() and Get() are well supported by an efficient garbage collection(GC) algorithm. We control write and read-amplification in FlashKey by laying out the data intelligently on disk in "logical bands". The logical bands allow for locally-sequential writes of data, and allow us to garbage collect the KV entries post updates/deletes in a smart manner using heuristics similar to those deployed inside the SSDs. Logical bands with the most invalid data-elements are garbage-collected first.

1.2 Adaptive Data Center Network Traffic Management for Distributed High Speed Storage

The need for accessing large amounts of storage allows the storage access to invariably go over the network. Furthermore, due to high IO rates of emerging storage technologies, the storage system can have enormous network bandwidth requirements. Also, since the emerging storage technologies and protocols can provide very low access latencies, a congested network on the path can contribute a significant portion of the end to end latency. Fortunately, the average IO or network traffic in a data center network is invariably very low and the real issue is not sustained high traffic but frequent bursts in traffic due to large data ingestion or output. In this thesis, we show that a well crafted dynamic migration and replication of data can deal with such bursts and minimize QoS violations.

To deal with those intermittent traffic bursts we design an incremental optimization (Ray et al., 2019) that dynamically grows by replicating or migrating the data chunks and then eventually shrink as the traffic burst fades away. The dynamic replication/migration scheme works at a medium time grain and based on the application access pattern determines whether to replicate or migrate a busy data chunk.

In addition to the data movement and copying explored here, dynamic per-flow changes to network routing can also be used to mitigate congestion and to consolidate traffic. We have studied this aspect in the following work (Biswas et al., 2019); therefore, we do not address it here. It is possible to integrate data movement/copying with intelligent routing, but due to the complexities involved, this is beyond the scope of this work. Also, in this work, we only consider traffic consolidation to facilitate energy management, rather than the energy management per se. We have addressed the latter in our work (Biswas et al., 2019).

The copy management discussed here could make use of the copies that are created by default for better resilience; however, our copying is limited to only to very hot chunks that would likely reside in the highest storage tier and extra copies are likely to be short-lived.

The dynamic data placement to achieve the congestion mitigation vs. traffic consolidation tradeoff requires that the storage be divided up into "chunks" of a suitable size and virtualized so that it is possible to move these chunks dynamically without any impact on the applications in terms of addressing or accessing the data they need. Achieving this tradeoff involves several challenges. First, if packet drops are to be avoided, an ongoing flow cannot use a different copy of the data chunk, instead, any migration is limited to only new flows. This smooth handoff scheme requires some time to make the change visible externally. Second, the migration/copy creation must be very lightweight so that the additional traffic generated does not worsen the congestion. Thus, we propose a proactive copy creation scheme with lazy synchronization across proactively created copies of the busy chunks. Third, the additional overhead of consistency management across chunk copies in relieving network congestion should not affect the overall performance. To overcome that, we use optimistic concurrency control (OCC), as in most of cases OCC works better than explicit locking under moderate data sharing. Fourth, the replication/migration used during congestion episodes should not result in substantial scattering of data throughout the network as it worsens low traffic performance and ability to do energy management. To address that, we try to reverse the process by applying consolidation at the low traffic period. Here, we have tried to reverse the process of replication/migration to take advantage of reduced latency, however, we have not applied any energy management techniques to save energy.

1.3 Coordinated Power Management in Data Center Networks

With the increasing size and energy footprint of data centers, their effective energy management is crucial. The major power consumers of the IT infrastructure in a data center are compute/memory subsystem, storage subsystem, and data center network (DCN). As computing and storage systems (electronic drives replacing the mechanical one) become more energy efficient, data movement, and hence data center networks consume an increasing percentage of the total energy consumption. The high-speed network consumes an increasing percentage of power as the network speeds and connectivity rise due to network upgrades. For example, the power consumption of a 10 Gb/sec Ethernet can be anywhere between 2-10 times the power consumption of 1 Gb/sec Ethernet. These upgrades are mostly not done for handling sustained high bandwidth demands. Instead, the deployment of higher-speed links is mostly motivated by the technological availability, tail latency considerations and ability to handle highly bursty workloads. Furthermore, infrastructure update increases the link speed which results in faster delivery and even lower network utilization. Hence, in the end, the average network utilization remains low. Finally, with the synchronous network links, the network power consumption remains constant whether the link is in use or not. Hence in this thesis, we concentrate on dynamically concentrating and spreading the traffic at a medium time grain, based on network utilization and opportunistically apply energy-saving methods to the unused network links to save energy without significantly increasing the latency.

We design (Sondur et al., 2017; Ray et al., 2018; Biswas et al., 2019) a semi-distributed intelligent routing mechanism to consolidate and spread the traffic based on network utilization and apply different sleep modes (L0s, L1, LPI (**L**ow **L**ower **I**dle)) to the unused links to save power. There are many existing works in the area of the data center network energy management but most of them propose a mechanism that works at a coarse time grain (24 hours or more) whereas our design tries to apply energy-saving techniques at a

medium time grains wherever possible. Traffic consolidation tried to concentrate the incoming traffic largely on fewest links and thus maximizes the opportunity for other links to sleep. However, flow path selection and consolidation to maximize sleep opportunities can be quite challenging in a large data center network because a workable scheme must simultaneously consider the availability of the server and other resources (CPU, memory, etc.), local bandwidth demands at the switches, and end-to-end congestion/delay on the network paths. Too much flow consolidation could end up with poor performance as it increases network latency.

In a small data center, this can be accomplished via a central controller that has global network and endpoint visibility, however, such a scheme does not scale even with a moderate size data center. In our work instead, we design a low-overhead semi-distributed scheme that involves three cooperating mechanisms : (a) an energy-aware distribution of requests to the endpoints by the user request assignment controller (RAC), (b) an intelligent assignment of a new flow at each switch by a local controller (LC), and (c) a lightweight global controller (GC) that monitors network traffic, build a global view of the network based on the information gathered from the LCs and provides hints to LC's and RAC for better placement of flows at endpoints. The key function of these controllers from an energy management perspective is to properly direct and consolidate network traffic to maximize "sleep" opportunities for the network interfaces. We show that the coordinated "hints" provided by the GC are vital to correct the myopic view of LCs and RAC for both avoiding congestion and maximizing network sleep opportunities. We studied simple mechanisms to coordinate the actions of these controllers and showed that such coordination can result in significant energy savings without needing VM migrations or moving an established flow. Also, where there are multiple agents involved in decision making, it is important that they coordinate with each other. Here GC is the one which dictates both LCs and the RAC, so there is no chance of conflict.

We explore these issues primarily in the context of the almost universal DCN topology in traditional data centers – the fat tree. The underlying mechanisms are general and can work for any regular network topology. To test it out we tried it on other DC infrastructure i.e. Hypercube network which is heavily deployed in the HPC environment and show that it too can save a significant amount of energy. Please note, HPC DC network consumes a significant portion of overall energy. We then compare the topologies (fat-tree and the hypercube) and show fat-tree can provide lower power consumption but at the cost of higher delay. There are further opportunities for further reduction of power without creating network congestion by more explicit congestion notification (ECN) that we are currently exploring.

We simulated the whole model in a popular networking simulation tool called ns3. It has all the network stack implemented and hence it is widely accepted in the network research community. However, it lacks many features that were needed to build our design. For example, different energy saving techniques, energy modes are not available, and the existing energy model is quite rudimentary. Hence, we enhanced the existing ns3 energy model framework of ns3 so that it can support the different sleep modes for each network port on a switch/router, endpoint, and backplane on switch/router in order to reduce the network energy consumption. The enhanced version can support two port-level power management mechanisms – unidirectional (L0-transmitter only) and bidirectional (L1-transmitter and receiver).

The basic serial link PHY supports two sleep states (low power), called L0s and L1, respectively which can be used for idle power management. The L0s power state is unidirectional, in that the transmitter for each direction of the link can independently decide to go into sleep mode when it has nothing to transmit, whereas the receiver side remains active. The L1 power state involves a handshake between transmitter and receiver, and thus allows both of them to go into sleep state when there is nothing to transmit. The L1 state can reduce idle power quite substantially. because it stops the transmission of syn-

chronization frames as well. However, the downside is that it requires re-synchronization and hence substantial delay. IEEE has defined standards for managing network devices to achieve energy efficiency. The Low Power Idle is defined in 802.3az for the Energy Efficient Ethernet (EEE) initiative. LPI can be thought of as an improved and link level version of L1. In our implementation, we support essentially L0s and LPI like mechanisms, which can be customized further as needed. There are new emerging energy management standards for 40 & 100 Gb/s Ethernet links such as 802.3bj (deep sleep mode) and 802.3bm (shallow sleep with fast wakeup). We can accommodate these standards in our implementation by a suitable change to parameters.

Although we have only implemented some basic energy management capabilities, they provide a launching pad for more sophisticated capabilities such as multiple sleep states and more sophisticated algorithms for deciding when to enter or exit a sleep state.

1.4 End-to-end QoS for distributed remote storage access

A related issue concerns the introduction of end to end QoS in remote storage services. Besides the emerging storage protocols (NVMe) for accessing new storage technologies efficiently, several high performance fabrics have been defined to provide low latency access to these technologies over the network such as persistent memory over fabric (Golander et al., 2017; Grun et al., 2018) and NVMe-OF (NVMe over fabric) (Black, 2018). Although PMoF/NVMe-OF assumed an RDMA capable transport (e.g., ROCE, iWARP or InfiniBand) in the past, low latency user-space TCP implementations such as (Inc., 2019) have also been defined recently.

The now-ubiquitous NVMe protocol allows for multiple queue-pairs along with weighted round-robin and priority handling of queues. The network also provides QoS classes, for example, the CoS (class of service) bits at the Ethernet level and DSCP (differentiated services code-point) at the IP level. However, currently, there is no general mechanism to

coordinate the network and storage QoS, although there has been some relevant work such as (Zhang et al., 2018). Such a coordination will require a well defined but extensible set of QoS treatments and end-to-end transmission of the corresponding QoS tags or "hints" so that each layer can provide the treatment using the locally available mechanisms. A limited form of hinting using NVMe directives, called multistreaming, was defined in (Kang et al., 2014) and has been used recently in Samsung SSDs (Kim et al., 2019). Currently, I am collaborating with my colleagues to comprehensively investigate the challenges in providing end-to-end QoS for remote storage access and explore possible directions to achieve a lightweight mechanism, that can be deployed in a sizable modern-day data center.

CHAPTER 2

FLASHKEY:A HIGH-PERFORMANCE FLASH FRIENDLY KEY-VALUE STORE

2.1 Introduction

In this chapter, we propose a new KV Store designed targeting the modern-day storage devices and data intensive applications. Key-value (KV) Stores are increasingly popular as they are well suited for storing and operating on streaming data generated by a variety of online applications including social media, e-commerce, scientific experiments, IoT monitoring of smart infrastructures, etc. Many KV store designs are currently in existence but are largely designed for hard disk drives (HDDs) as the backend. HDDs have very poor random read and write performance compared to their sequential performance (as shown in Table 2.1), and the KV systems therefore go to great lengths to sequentialize writes, and also to front the HDDs with memory or other caches to mitigate the poor random read performance. These schemes, unfortunately, have high read and write amplification penalties. As Solid-State Disks (SSDs) become widely deployed in Datacenters, there is an opportunity to redesign the KV architecture to benefit from the SSDs' fast random performance (Table 2.1).

In this paper, we present FlashKey - a flash friendly KV store, with data structures and algorithms that capitalize on good random IO performance by using hashing and can work with low-endurance TLC/QLC SSDs because of significantly reduced write-amplification. The solution uses a hash-table structure to provide fast "Get" operations and a "logical band" based data organization that allows partial sequentialization of "Put" data writes to the SSD, since sequential writes still provide better performance and also improve the SSD write-endurance. Finally, FlashKey also includes a Log-Structured-Merge (LSM) tree structure for keys, without values, to support "Scan" operations.

Table 2.1: Device IO Performance

Device (HDD:Segate SSD:Intel)	**Rand R/W (MB/s)**	**Seq R/W (MB/s))**	**Endurance DWPD(GB)**
HDD 10K Ent.	1.6	241	-
TLC DC P4610 1.6TB	2510/780	3200/2080	6712.329
QLC DS-P4420 7.68TB	1660/140	3200/1000	3068.493

Through a detailed experimental evaluation *on current SSDs*, we show that FlashKey, in spite of being an initial implementation, outperforms not only Google's original KV store called LevelDB, but also the state of the art and highly optimized RocksDB developed by Facebook. In particular, we demonstrate the following:

a) The unique design of FlashKey optimizes SSD accesses and thus provides a much lower write amplification than both LevelDB and RocksDB at comparable space amplification.

b) FlashKey provides a much lower average latency than both LevelDB and RocksDB. It also provides a much lower 99.999 percentile tail latency than LevelDB and RocksDB.

c) The read amplification of FlashKey is comparable and, in fact, generally lower than that for LevelDB and RocksDB.

d) FlashKey prioritizes high performance for individual record accesses over that for large key ranges. Nevertheless, it can handle range queries via additional key-only structures.

We show that FlashKey improves Get() and Put() latencies by up to 85%, and write amplification (WA) by up to 12x. The reasoning behind these improvements is as follows:

Get() performance : (1) Hash lookup eliminates the need for tree-traversal on the disk. Finding the key index needs O(1) time with one storage access on average. Typically two storage accesses are required to complete a Get() even when the hash-table is on disk. (2) A strong primary hash function reduces collision and a secondary hash is used to avoid unnecessary disk lookup(s) upon primary hash collision.

Put() performance : A typical Put(.) operation in FlashKey requires two writes to disk.

Garbage Collection (GC) performance, WA and RA: We control write and read-amplification in FlashKey by laying out the data intelligently on disk in "logical bands". The logical bands allow for locally-sequential writes of data, and allow us to garbage collect the KV entries post updates/deletes in a smart manner using heuristics similar to those deployed inside the SSDs. Logical bands with the most invalid data-elements are garbage-collected first.

The rest of the paper is organized as follows. Section 2.2 states the key value operations and the performance metrics used to evaluate any KV store. Sections 2.3 and 2.4 discuss the related work in the area of KV stores and the popular KV stores that are currently in use respectively. Section 2.5 discusses the architecture and algorithms of FlashKey. Section 2.6 evaluates the performance of FlashKey in comparison to LevelDB and RocksDB on key benchmarks. Finally, section 2.7 concludes with a summary and discussion of future work.

2.2 Key Value Operations and Metrics

The user-operations supported by all KV-stores are Put, Get, Delete, and (in-some-cases) Scan. These operations form the foundation for KV stores, and are sufficient for most applications. The Put(K,V) adds the Key:Value mapping, if it does not exist already, to the

associative array/dictionary managed by the store. The operation updates the value for the specified key if a mapping already exists. The Get(K) operation retrieves the associated value, or returns an error if none is found. The Delete(K) operation deletes a stored K:V mapping, or returns an error if no such mapping exists. The Scan(K,range) operation allows iterate over (ordered) keys, starting with specified keys K and reading all keys upto the specified range.

The primary metrics for evaluating any KV database are:

1. **Space Amplification (SA)**. SA is the ratio between the size consumed by the database on the disk to the actual size of the user data, including keys and values. Higher SA typically reduces the amount of background operations required to sequentialize writes (resulting in lower WA and higher net performance, at the expense of higher $/GB effective cost), and vice-versa.

2. **Write amplification (WA)**. This is the ratio of total bytes of data written by the KVS to the storage device, to the bytes of data written (Put(.) request) by the user. Lower WA corresponds to fewer writes to the disk, and hence better write performance, drive-endurance, and QoS.

3. **Read amplification (RA)**. This is measured as total bytes of data read from the disk divided by the bytes of data returned to the user for their Get queries. Lower RA results in fewer reads to the disk, and hence better read performance and QoS.

4. **Average Latency**. This measures the amount of time it takes on average for each Put or Get request to finish. Lower latencies provide better responsiveness to clients and their applications.

5. **Throughput**. This measures the amount of data that can be Put and Get per unit-time, from the KV store. Throughput measures sustained performance.

6. **Quality of Service (QoS)**. A problem with several storage solutions is that, while they provide good average performance, they suffer from a highly variable performance. Real world deployments, however, prefer a deterministic latency, especially for the reads.

Thus an important QoS metric for read operations is Read Tail Latency (RTL), which is measured as 99 percentile or even higher percentile of read latency.

2.3 Related Work

The existing works on KVS belong to four primary classes: in-memory KVS (Hu et al., 2015) (e.g., Memcached, Redis); Persistent-Memory based KVS (Wu et al., 2019; Eisenman et al., 2018; Kannan et al., 2018; Kaiyrakhmet et al., 2019)(e.g pmemkv *https://-github.com/pmem/pmemkv*); HDD or NAND flash-based KVS (e.g., LSM based KV); and all the other types of KVS (e.g., designed for SMR drives or Open-FTL) (Yao et al., 2017, 2019; Pitchumani et al., 2015; Wang et al., 2014; Zhang et al., 2017; Shen et al., 2017; Bjørling et al., 2017; Jin et al., 2017; Li et al., 2017; Marmol et al., 2015; Lee et al., 2016). Here, our discussion is limited to the Flash based KV stores for relevance.

Flash-based KV Store: In (Lu et al., 2016), the author used KV isolation similar to FlashKey, where the value is stored separately in a circular buffer -"vLog", indexed by the LSM tree. However, the vLog by design is inefficient for GC, as it uses FIFO for key selection and relocation during GC. Also, during GC, each key is validated by an LSM tree lookup which slows down the GC as a key lookup in (Lu et al., 2016) needs $\log(n)$ time, as compared to O(1) time in FlashKey.

In HashKV (Chan et al., 2018) the author separates the keys from its values based on the value-sizes. Small values are stored with keys in the LSM, and large values are stored in separate datastore indexed by the LSM tree. Such an approach would work well if items with small values are more popular than others, but otherwise, the additional search in the LSM may be detrimental to performance. In our case, all our lookups take on an average O(1) time independent of KV sizes or key access profile. HashKV also partitions the datastore into groups and uses key hashes to map the value across these fixed groups, whereas FlashKey allows more flexibility as it places KVs within arbitrary LBands.

In the paper (Debnath et al., 2010) the author caches the KV pairs in flash, indexed by an in-memory Cuckoo-hashed table for performance as the KVS is on HDD. An extended work (Debnath et al., 2011), optimizes the performance of hashing during a collision by linear chaining of the records mapped to the same index in flash. μDepot (Kourtis et al., 2019) uses a DRAM-based two-level hopscotch hashtable indexing that can be dynamically resized as the database grows. To restrict lookup(s) to a single IO, the index uses more bytes which helps to cover a large address space but increases DRAM footprint. In our work, the hashtable does not store the key which reduces the space requirement and update latency.

SILT (Lim et al., 2011) uses three different stores where a key is initially inserted into a write-optimized LogStore, indexed by in-memory hashtable, then eventually merged to one of the many on-flash HashStore, and finally ends up in an immutable SortedStore with even lower in-memory index. The key concern of SILT is to reduce the DRAM usage for indexing, perhaps at the cost of higher WA/SA, which is different from our objectives. SlimDB (Ren et al., 2017) picks up some of the design features of SILT and tries to optimize the index and filtering, compaction algorithms and reduces the memory footprint further. PebblesDB (Raju et al., 2017) uses Fragmented LSM (FLSM) to increase the write throughput ($\downarrow$ WA) at the cost of higher read latency and SA (Li et al., 2019) than RocksDB/LevelDB. The WA in FlashKey is lower than both the RocksDB and LevelDB with similar SA, and also the performance improvement comes at the cost of architecture change without sacrificing the read performance.

2.4 Current Key-Value Stores

A large number of KV databases are currently available where they are typically implemented in software (Minglani et al., 2017), to allow operation over volumes of disks, though on-disk implementations also exist (Devulapalli et al., 2008; Minglani et al., 2017).

Some of the popular KVSes - RocksDB (Facebook), LevelDB (Google), etc. designed based on *Log-structured merge (LSM) tree* (Sears and Ramakrishnan, 2012). Here, we will discuss the working principles of LSM trees and compare the performance of FlashKey against the LevelDB and RocksDB implementations in section 2.6. We selected LevelDB and RocksDB since they are widely accepted open-source solutions and are heavily deployed on the real systems.

2.4.1 Architecture of Log Structured Merge-Tree

LSM tree is designed with features that make it efficient on HDDs for providing indexed access to files with high insert, update, delete and scan operations (O'Neil et al., 1996). It is a data structure that converts small random writes to large sequential writes by batching them in memory, and further stores sorted files in multiple tree levels to provide O($\log N$) lookup time. The sequentialization makes LSM Tree better than B-Trees.

An LSM tree consists of multiple components $C_{-1}, C_0, .\ ..., C_k$ of exponentially increasing sizes. Component C_{-1} is in-memory and updated in-place, whereas the rest of the components are on-disk and updated out-of-place. For better Put(.) latencies, an insert to an LSM-tree gets added to a memory buffer at C_{-1}. While to ensure persistence, they are also appended to a Write-Ahead Log (WAL).

Once C_{-1} exceeds a certain size limit, a contiguous and sorted segment of entries is removed from C_{-1} and flushed onto the disk. This flush may cause level C_0 to exceed its size limit which then merged with C_1 using an algorithm similar to mergesort. This process of merging is called **"compaction"**, during which overlapping parts of C_0 and C_1 are merged, and the new merged file replaces the old files of C_1. Any component C_i (i$\geqslant$0), upon exceeding its size limit is compacted asynchronously with the adjacent C_{i+1}.

Compaction is a background process that removes overwritten and deleted data by the application. Compaction amplifies reads and writes, as while merging two overlapping files, all data has to be read from the files and compared, before writing them to the

final files. Depending on the rate and the extent of compaction, the system exhibits different trade-offs. Infrequent compaction increases the SA. Whereas frequent compaction increases both RA and WA significantly and may cause both read and write to stall as it uses more of both CPU and IO.

The data lookup in LSM-tree largely relies on in-memory caches for performance on cache-hits. Upon a cache miss, LSM tries to locate the data in C_{-1} first with a linear search starting from the most recent data to get the latest update if any. Upon missing out of C_{-1}, the read cascades through the following levels. Thus, in the best case, a lookup can be resolved at C_{-1}, however, in the worst case, it can travel through $C_0, .., C_k$ before resolving at C_k. Hence a single read operation can be amplified by reading a number of different components in the LSM tree.

2.4.2 LevelDB

LevelDB implements a LSM-tree as discussed in section 2.4.1, and primarily has two structures (C_{-1}) the in-memory sorted skiplist-based (mutable and immutable) "memtables" and ($C_i, \forall i \geqslant 0$) the "**S**orted **S**tring **T**able (SST)" files stored on disk at level L_0 through L_6. In LevelDB, an insert of a KV pair first goes to the "mutable" memtable. As multiple inserts exhaust the capacity of the memtable, it switches to a new memtable. The old memtable is marked as "immutable" and in the background flushed to the disk.

Each level L_0 to L_6 has a predefined size limit and every level L_i is 10x (*level size multiplier*) bigger than the adjacent level L_{i-1}. Each level has many SST files where each SST file stores KV pairs sorted by keys. With the exception of L_0, files within a level do not overlap in their key ranges. For L_0, overlap is allowed as the SST files in this level are essentially memtable(s) flushed from memory.

Compaction is triggered as any level L_{i-1} exceeds its size limit; during which LevelDB removes any key with deletion markers if there are no higher levels that contain a file whose range overlaps the current key. In the worst case, while compacting level L_{i-1} to

level L_i, LevelDB can read up to 10 files from level L_i, merge-sort and write back to level L_i. This can increase the WA up to 10 times. With level L_0 to L_6, new data propagates to higher-numbered levels through a series of compactions which can increase the WA to even higher. A garbage collection method is called at the end of every compaction to remove the obsolete files that are neither referenced from any level nor the output of any active compaction. Hence, in LevelDB, the SA always stays bounded, at the cost of high WA.

Cache-miss lookup takes logarithmic time as LevelDB searches the memtables and the SST files from L_0 to L_6 in order which increases the RA. For level L_0 (with overlap), a lookup might require to explore all the SST files (worst case) and for the other levels lookup only touches one SST file per level (no overlap), and does a binary-search within each SST. LevelDB also requires metadata lookup to find out which SST files at each level to look into which increases the RA further.

2.4.3 RocksDB

RocksDB, which extends LevelDB, is an embedded persistent key-value storage system. It has many additional features (Dong et al., 2017) beyond LevelDB, including multi-threaded compaction, multithreaded memtable insertion, reduced DB mutex holding, optimized compaction, etc. In addition to the LevelDB style compaction, RocksDB includes Universal compaction where instead of aggressively merging a smaller SST with a larger one, the system waits for several sorted runs of similar size before merging them into a big run. This new compaction method reduces WA at the cost of doubling SA at times. Multi-threaded compaction allows RocksDB to run parallel compactions across several non-overlapping regions of the KV store simultaneously as long as resources are available.

To reduce the SA, instead of statically setting the level target size as 10x of the prior level, RocksDB introduces a feature called dynamic level size adjustment where the size

target of levels is changed dynamically based on the size of the last level in a bottom-up manner. As the database size changes, it dynamically adjusts the size of each level to be $\frac{1}{10}$th the size of the data stored on the next higher-numbered level.

2.5 FlashKey

In this section, we define the architecture of FlashKey. Given the excellent random IO performance of Flash and other emerging NVRAM technologies, we believe that hashing is a viable approach for locating KV records. Our current FlashKey architecture, illustrated in figures 2.1 and 2.2, uses simple hashing, although more complex mechanisms such as cuckoo hashing for better space utilization or locality sensitive hashing (LSH) to have better range query performance - may also be useful. FlashKey uses (1) a hash table (HT) for storing per key metadata, (2) a flat file structure on disk to store KV pairs, in a set of *Logical bands* (LBands), (3) an optional LSM tree to store the unique set of keys, and (4) other system-level metadata structure. All of the above data structures are stored on disk, but the HT is generally cached in the host-DRAM. These are further described in the following subsections.

2.5.1 Data Structures and Layout

Hash Table (HT)

Each key is hashed using a *primary hash function* that maps a given key to an index into the hash table (HT). For each hash index, the HT stores the metadata (described further below) of up to two keys that map to this index. If more keys map to this hash index, the second HT entry becomes a pointer to a common overflow area where the metadata of all the keys mapping to this index is maintained in the form of a linked list. We henceforth call each HT entry as a "node" (as shown in Figure 2.1), in which the first two nodes are in the HT itself, and the rest are in the overflow area.

The HT stores metadata per KV pair that contains the address of the KV pair on the disk ("KV-addres") constitutes of LBand_seq_number and offset within that LBand, the length of the KV pair ("KV-length"), and a secondary hash value or Key Fingerprint ("Key-FP") to reduce disk lookup on collision as shown in Figure 2.1. We do not store the actual key in the HT entry. So when an input key is mapped to the HT entry, in the best case, a single lookup suffices to determine the potential value location on disk. In the worst-case, all the nodes in the corresponding linked list need to be read from the storage media, to locate the (potentially) matching record. However, by choosing a suitable (non-cryptographic) hash function and HT size, the overhead of hashing can be kept very low (Estébanez et al., 2014; Henke et al., 2008; Jenkins, 1997). In our experiments, we observe that with 36B HT entry size and a HT size equal to twice the maximum database size, the collision probabilities are under 4%.

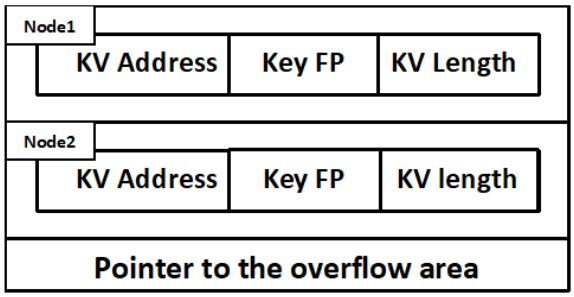


FIGURE 2.1: A Hash Table entry

As the database grows, the hash space is filled in with the incoming KV pairs. If the HT load factor (*No-of-keys-stored / capacity*) exceeds 90%, we allocate a larger HT and migrate all the entries from the old to the new HT. With suitable sizing of the database, such a move should be rare and can be done during periods of very low activity, although a live-migration type of approach can be used to further reduce the impact. The space for the old HT could then be reclaimed.

The HT is kept on the SSD, and cached in host DRAM for performance. FlashKey permits write-back caching of the HT in memory which allows the user to control the cache size. Zero DRAM consumption for HT places the entire table on disk - minimizing DRAM cost, but requiring additional IO for HT accesses. At the other extreme, the entire HT may be kept in DRAM, improving performance at the expense of DRAM cost and startup/recovery times. With suitable logging mechanisms (discussed later), the unflushed

dirty portions of HT can be recovered from its disk version and LBands in case of power loss. In our experiments, we use both in-memory and on-disk HT and compare the impact of the two bookend implementations of HT in sections 2.5.2 and 2.6.2. Depending on the requirement, the user can select an approach that balances these tradeoffs.

Logical Band (LBand)

The storage media inside most NAND flash-based SSDs is organized into multiple physical sections, which is the basic unit for defragmentation. We extend this idea in FlashKey to store key-value data, rather than disk-sector data. Accordingly, we divide the entire logical storage space into multiple logical bands (LBands) (Li and Trika, 2018).

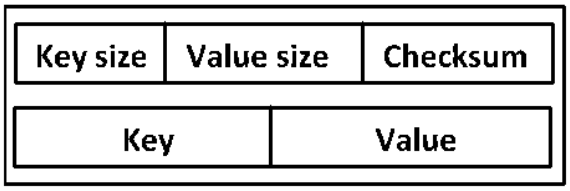


FIGURE 2.2: An LBand entry

Each LBand is designed to hold roughly a thousand KV pairs by default, and this can be reconfigured as needed. Each LBand stores LBand-metadata, multiple LBand entries, and optionally a LBand journal. Each LBand is in one of three states: empty, open, or closed. A linked list for empty bands called "EmptyLBandList" is used whenever FlashKey needs to open a new LBand to write new (or defragged) KV pairs. The selected band is moved from the empty state to open state. When an open band is filled up, it is moved to a closed state. The "ClosedLBandList" is used to identify bands that are filled and closed. At any given time there is only one LBand which can be in the "Open" state, into which the system writes the new/compacted KV pairs.

LBand Metadata Each open or closed LBand starts with band-metadata, which includes the sequence number of the LBand. The sequence number is used during system recovery. Each closed band includes metadata at the end of the band, which specifies the band's state (closed), and optionally a band-journal as discussed in 2.5.1.

LBand Entry Each entry in the LBand as shown in the Figure 2.2, stores the actual data for the KV pair. Each LBand entry also includes a 16 bytes header that has additional information about this KV pair, that specifies the length of the key and the value, and checksum for error detection purposes. This metadata associated with each entry allows FlashKey to support variable size key-value pairs.

Band Journal Band journal is an optional data structure to record the hashes and locations of the table entries, i.e., hash index and HT node information of all the key-value pairs stored in that respective LBand. It may be extended to include key-prefixes for the keys stored in the LBand. The band journal can be used to accelerate the garbage collection process and power loss recovery.

LSM Tree

Hashing intentionally destroys the locality and hence the key ordering information by randomly spreading the keys across the entire hash space. This makes Range-Scan(.) operations problematic. We allow the user to configure FlashKey to support Range-Scan(.). To support scan, FlashKey stores the set of unique keys in a separate LSM tree structure. Each time a new insert is requested, we insert only the key into the LSM tree, without any value (Trika et al., 2019). No subsequent (value) update for that key goes to the LSM tree. Hence the space required by the LSM tree is very low (2.7% for both 50GB and 100GB database) compared to the database size. In case of deletion, we delete the key from the LSM tree as well. During the range queries, we retrieve an iterator with the set of sorted keys within the given range from the LSM tree and iterate over it to get the values using the HT and LBand.

Other System Level Metadata Structures

FlashKey also maintains other system-level metadata structures including LBand invalidity table, a linked list for empty and closed bands, and statistical information. This information is kept in the host main memory during run time, and flushed to storage device periodically and when the FlashKey is closed. It is loaded/rebuilt during startup.

The LBand invalidity table records how much data is invalid in each LBand. We use this table to select which LBand should be selected for Garbage Collection/Compaction. The number of entries in this table equals the total number of LBand(s) in the system. Each entry contains the current amount of invalid bytes in the corresponding LBand. The table is updated after each PUT/DELETE operation. More specifically, if an existing key is being overwritten or deleted, the LBand index and the length of the stale KV pair can be found from the HT. Then, the corresponding invalidity table entry is selected, and the length of the stale KV pair is added into that entry. Storing the KV length in both HT entry along with the KV header at LBand, helps to reduce the number of disk reads as the KV is deleted or updated at the cost of a little higher storage space.

The statistical information includes the currently available free space, the amount of data that has been: (a) written to and read from the device, (b) written and read by the host, etc.

2.5.2 Algorithms

In this section, we describe the basic methods of FlashKey.

Data Retrieval using Get(.)

During the Get operation, FlashKey computes the HT entry index of the input key. Then it loads the HT entry from the storage device to the host main memory if required. It then computes a secondary hash based on the key and tries to match it to the key fingerprint stored in the node. If the secondary hash matches as well, then it uses the LBand index and

offset stored in that node to read the KV pair from the LBand. Next FlashKey compares the key read from the LBand with the key from the input and returns the value in case the two keys match and the value is not a delete token which signifies that the key has already been deleted. Otherwise, it searches through all the nodes at the HT index until a match is found or the nodeList ends, in which case it returns not found to the host.

Data Insertion using Put(.)

FlashKey tracks the location to be written in the currently open LBand, namely LBand_-Write_Position. During Put(.) operation, FlashKey checks if space is available to put the incoming KV pair in the currently open LBand. If space is insufficient, then it closes the current LBand and opens a new clean band. It triggers garbage collection to reclaim storage space by throwing away the invalid KV pairs in case the available clean bands are low in number ($<$ predefined Threshold). Once space is available, FlashKey writes the KV pair to LBand_Write_Position.

After the KV pair is written, FlashKey updates the corresponding hash-table entry. The logic it follows is similar to finding a key match in the Get operation. If no match is found, then the algorithm allocates a new node at the corresponding HT index entry and treat this as a new insert. The new node might be first or second, or any node in the overflow linked list. For any new insert, FlashKey also creates an entry with just the key in the LSM tree. On the other hand, in case a match is found, FlashKey updates the HT entry of the existing KV pair with the recent location of KV in the LBand.

Data deletion using Delete(.)

The delete operation is similar to the Put operation, except that instead of writing any value with the key, FlashKey creates a new entry where it drops a "DELETE token" for the deleted key at the current open LBand Write Position. For the entry created in the LBand, it also updates the corresponding HT entry. The deleted (marked) key is eventually

discarded during Garbage Collection. However, FlashKey removes the deleted (invalid) key from the LSM tree during the delete operation, so that the LSM tree only contains the valid keys.

Range Queries using Scan(.)

Given the starting key and the range for the scan operation, FlashKey gets the set of valid keys within that range from the LSM tree, iterates over those keys, and calls the Get(.) method to retrieve their values.

Garbage Collection

Garbage collection (GC) is triggered when the number of empty LBands falls below a certain threshold. During GC, FlashKey selects the source LBand with the most invalid KV pairs using the invalidity table. For the source LBand, it reads each KV and determines whether it is a valid key by matching its location with that pointed to by the HT. It then relocates the valid KV pairs to the current open LBand and updates the address of the KV pair at the corresponding HT node. For any valid KV pair, if the value at the LBand contains a delete token, FlashKey removes the entry for the KV pair from the LBand and the HT. When the relocation completes, FlashKey marks the source LBand as "empty" and pushes it to the empty band list. The band-journal mentioned in 2.5.1 may be used to optimize GC operations. For the sake of simplicity we use single GC thread, however, the performance can be enhanced further by having multiple of them, in which case all the GC threads and the foreground threads write to the HT and the open logical band. This requires locking the LBand entry at the corresponding HT index and the LBand-offset increment. Due to the small granularity of the lock, the lock has no major impact on the performance.

Clean Shutdown, Startup and Power Loss Recovery

During a clean (proper/not power-loss) database shutdown, all the dirty data in memory is written back to the storage device which includes (1) HT dirty entries, (2) LBand metadata for the open LBand, in part to identify where we left-off in the band, (3) LBand Journal for the open LBand, (4) LBand status table including the LBand invalidity information, (5) statistics, etc. A flush(.) command ensures that the data is non-volatile before the database shuts down.

During restart from a clean shutdown, FlashKey first loads the LBand status table to build the empty/closed LBand list. Then it reopens the last opened LBand and restores the write pointer to the appropriate offset. Next, it loads the open LBand metadata and journal. Finally, FlashKey loads the invalidity table and the statistics and gets ready for new host commands.

Restart/recovery from a dirty/improper shutdown depends on the system configuration. As stated earlier, the HT can be optionally cached in DRAM (write-back or write-through mode). A user can choose the right configuration to balance the DRAM consumption, the required number of disk reads and writes, and recovery time. Configuration selection also impacts the startup/recovery time after a system crash or power fails. To assist recovery, FlashKey also periodically writes its state-information to the storage device, including (1) dirty HT writes, (2) LBand invalidity table, (3) LBand Status table and (4) statistical information. We handle three different scenarios, based on where the HT and all metadata is placed, as follows.

On disk and write-through cached in memory: Under this configuration, both shutdown and restarts happen very fast. Even under sudden power loss, as everything is backed up on disk, minimal recovery is required for proper cleanup potentially of the last KV and a few metadata structures.

Fully in memory: This requires saving and loading the data structures during shutdown and restart respectively. Under sudden power loss condition, we must playback all non-empty LBand(s) in sequence number order to recover the HT and the other key data-structures. The band-journal information in each LBand may be used to accelerate the replay, since it contains the necessary information for replay, without requiring reading of the entire LBand from disk.

On disk and write-back cached in memory: We generally flush the dirty portions of the HT and other structures as we close an LBand. So to recover from the power loss, we restore the last saved KVS state, and then replay only the current open LBand. However, to reduce flush frequency, FlashKey can be configured to flush on every N^{th} LBand that is closed; in that case, we playback the last N LBand(s). This provides control over the time to recover from power loss conditions, since that depends on the flushing frequency.

2.6 Evaluation and Analysis

This section presents the experimental and analytic comparison of FlashKey to both LevelDB and RocksDB. In particular, section 2.6.1 details the system and workload configuration, and section 2.6.2 presents the experimental results. Section 2.6.3 provides an analytical model and an associated comparison of key performance metrics.

2.6.1 Experimental Setup

Benchmarks: In our evaluation, we use the popular benchmark called db_bench and compare it with our implementation in FlashKey. Even though both the benchmarking tools that come with LevelDB and RocksDB are named "db_bench", they use different key generation streams. Therefore, it is required to test against them separately, henceforth called *(db_bench_L)* and *(db_bench_R)* respectively. db_bench_L only works with LevelDB and FlashKey. db_bench_R only works with RocksDB and FlashKey. These evaluations illus-

trate the primary performance characteristic differences between FlashKey and the baseline KVSes.

Table 2.2: db_bench_R & db_bench_L Workloads

Workload 1	fillsequential	loads 50M KV
Workload 2	readrandom	50M random reads
Workload 3	overwrite	100M random writes
Workload 4	readwhilewriting(1:1)	1-R, 1-W, 50M reads/thread
Workload 5	readwhilewriting(3:1)	3-R, 1-W, 50M reads/thread

In section 2.6.2, we use the Yahoo! Cloud Server Benchmark (YCSB) (Cooper et al., 2010) for a system level evaluation. We use mapkeeper bindings to couple YCSB with FlashKey and LevelDB systems, and rocksjava binding with RocksDB. YCSB workload configurations are shown in Table 2.3.

KVS and System Configuration: To reduce variables that introduce significant variance, we disabled system and KVS features that can be equivalently added to all databases and systems. In particular, we turned off application based data caching, while leaving operating system level caching (page cache) in place. Also, except for a few cases, the tests are run using a single thread (multi-threading is generally turned off). The resulting configuration parameter information for RocksDB is shown in Table 2.4, and LevelDB configuration follows suite. The experiments were performed on an Intel7 pro NVMe SSD installed on a server running CentOS-7 with 8 Intel Core$^{(TM)}$ i7-7700 CPU executing at a rate of 3.6 GHz, and a 32 GB memory.

Table 2.3: YCSB Workload Configuration

W	**Ops Ratio**	**Operations**
A	50:50	Read/Update
B	95:5	Read/Update
C	100	Read
D	95:5	Read/Insert
E	95:5	RangeScan/Insert
F	50:50	Read/Rd-Mod-Wr

FlashKey Configuration: In FlashKey we use a simple modular hash function to generate index to HT, and a second 64-bit Jenkins (Jenkins, 1997) to generate the key fingerprints (the second hash that is included in the node). It is possible to use any low-overhead hash function that provides good collision performance. Hash-table size is set to 100M entries, which is 2x the number of KV pairs in the database. More sophisticated mechanisms, such as Cuckoo hashing, may allow for a smaller hash space, but this is not explored here. Our experimental results use hash-table in memory unless otherwise stated.

Table 2.4: RocksDB Configuration

Parameters	**Value**
All params except those below	default
block_restart_interval	1
write_buffer_size	128M
compression_type	None
level0_file_num_compaction_trigger	2
cache_size	0

Other tools and parameters: A significant part of our evaluation focuses on WA and RA metrics. To calculate these metrics, we use the nvme-cli *https://github.com/linux-nvme/nvme-cli* tool to read the SSD counters to measure the IO issued to the disk. Combined with workload information, the WA and RA metrics are then calculated as defined in section 2.2.

2.6.2 Experimental Results and Discussion

For our experiments with db_bench_L and db_bench_R, we use workload 1 of Table 2.2 to fill the database with KV pairs and run different benchmarks as stated in Table 2.2 above to measure the performance. We chose key size of 16B and value size of 1024B. Note that although KV's allow for rather large key-sizes, small key sizes are more efficient and popularly used in real applications (Lu et al., 2016).

Space Amplification Comparison

Space amplification (SA) significantly impacts KVS performance, WA, RA, and cost. An ideal comparison of KVS systems would evaluate these metrics with SA kept constant. Unfortunately, this is not practical with the baseline KVS solutions in the study as the SA for them varies widely within a run. We can only capture the final SA as the workload stops and there is no information provided for the intermediate SA (maximum SA) at any time within a run. In our comparison, we align SA of FlashKey to the final reported SA of the baseline KVS. We explain below (1) how the SA of the baseline KV system changes, (2) what is the impact of the change on WA, and (3) how FlashKey is different from them.

During database-load, all KVS systems use little to no overprovisioning, with space amplification between 1.02 and 1.12, as the KV pairs are loaded sequentially in the database.

Next, under intensive update traffic, SA increases due to out-of-place updates. As SA increases, LevelDB and RocksDB try to bound the space utilization through repetitive compactions. The frequent compaction hurts WA(↑), RA(↑), latency(↑), and throughput(↓). Both KVSes acquire more space to defer compaction as per their policies to limit the impact on performance and WA; this increases their SA during run-time to some unspecified (and not-reported) amount. LevelDB and RocksDB only report the database size (space-utilization) on disk for the KVS at the end of the test-runs, which is a snapshot post compaction. This results in significant under-reporting of their actual space utilization and hence calculation of the SA. In contrast, FlashKey allows the database to grow to only a fixed amount of additional LBand(s), and triggers GC as needed to keep the space-utilization bounded. In our experiments, we configure FlashKey space-utilization (and hence SA) to match the final-calculated SA of the baseline KVS per experiment. This is an extremely conservative comparison, therefore, against both LevelDB and RocksDB. We also note that since FlashKey tightly controls the space amplification at all times, it

can operate with a limited amount of storage space and low cost, whereas RocksDB exits with an error if they are unable to secure the necessary amount of space during the runs.

Write Amplification Comparison

The graphs in Figure 2.3 compare the improvement in write amplification (WA) with FlashKey over RocksDB and LevelDB when running under the db_bench_R and db_bench_L benchmarks respectively with their SA aligned. While comparing WA, we only consider workloads with writes, i.e., workloads 3, 4 and 5.

FIGURE 2.3: Comparison of WA of FlashKey against LevelDB and RocksDB using db_bench_L & db_bench_R respectively.

Figure 2.3(a) shows the WA improvements of FlashKey over LevelDB, and Figure 2.3(b) shows the same in comparison to RocksDB. In comparison to LevelDB at similar SA, FlashKey achieves WA improvement in the range of 4-12x on these workloads, with a maximum of 12x gain for workload 3. In comparison to RocksDB, FlashKey exhibits WA improvement of $\sim$7x, which remains consistent across all the workloads. The WA improvement that FlashKey achieves over RocksDB is less than the gain achieved in comparison to LevelDB, as RocksDB incurs comparatively lower WA than LevelDB at the cost of higher SA than that of the LevelDB.

As stated earlier, both LevelDB and RocksDB allow for greater SA during the run than the ending value that we equalized for a conservative comparison. RocksDB is more lax in

controlling SA than LevelDB, which allows it to do a bit better in WA. Even so, FlashKey achieves far better WA as compared to RocksDB.

Read Amplification Comparison

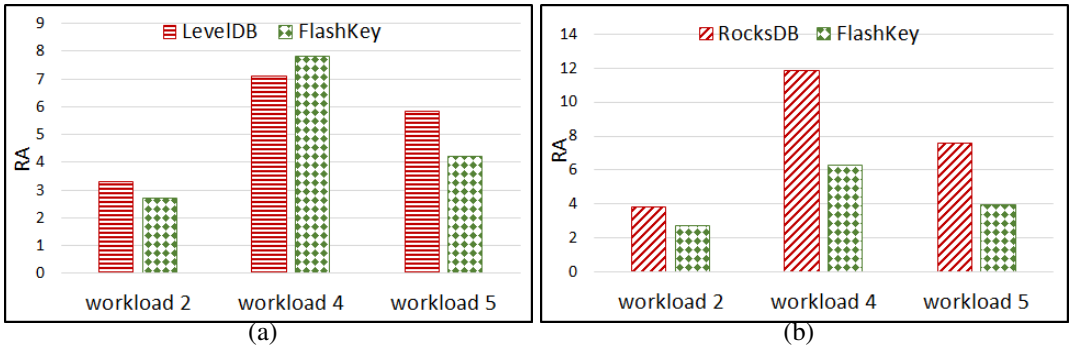


FIGURE 2.4: Comparison of RA of FlashKey against LevelDB and RocksDB using db_bench_L & db_bench_R respectively.

The graphs in Figure 2.4 show the change in RA achieved by using FlashKey against the other KV Stores. We compare the RA for workloads with random read requests, i.e., workloads 2, 4, and 5.

When tested against RocksDB, for all the workloads, FlashKey performs 1.5-2X better in terms of RA. However, compared to LevelDB, on workload 2 and 5, FlashKey improves the read-amplification by 1.2X and 1.4X respectively. However, for the workload 4, (i.e. 1:1 read-to-write ratio) the RA with FlashKey is somewhat greater (10%) than that of the LevelDB. Note that these two workloads (4 and 5) represent "read while writing" situation, and thus the read performance entirely depends on how eagerly the writes are done. LevelDB did only 1/2 as much writes as FlashKey during the entire read period thereby eking out better read performance than FlashKey. This is because of the strict compaction (with very high WA 18X) to keep the SA (1.24X) low, which helps random reads but at the cost of fewer writes, which itself saves on compaction overhead. The impact of fewer writes can be seen on the Average delay graphs of FlashKey over LevelDB

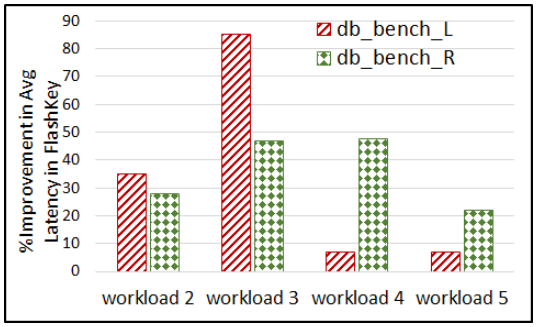


FIGURE 2.5: Comparison of Average Latency of FlashKey against LevelDB and RocksDB using db_bench_L & db_bench_R respectively

discussed in the following section. Note that, we still observe some improvement in RA with workload 5 as multithreading hides the impact of successive access to the same key.

Average Latency Comparison

Figure 2.5 compares the average latency for the workloads (2-5) at an IO queue-depth of 1. For all the workloads, FlashKey obtains improvements in average latency over both RocksDB and LevelDB. For LevelDB, the improvement is lower ($\sim$ 7%) for the read-write mix workloads (4 and 5) for reasons mentioned above. When tested with db_bench_-L, FlashKey achieves a maximum of 84% improvement with workload 3. The average latency for LevelDB is very high due to frequent compaction which also broadens the scope for having a higher latency gain when tested with the write-intensive workload. The latency improvement of FlashKey is due to its efficient O(1) architecture and algorithms that address the shortcomings of LevelDB.

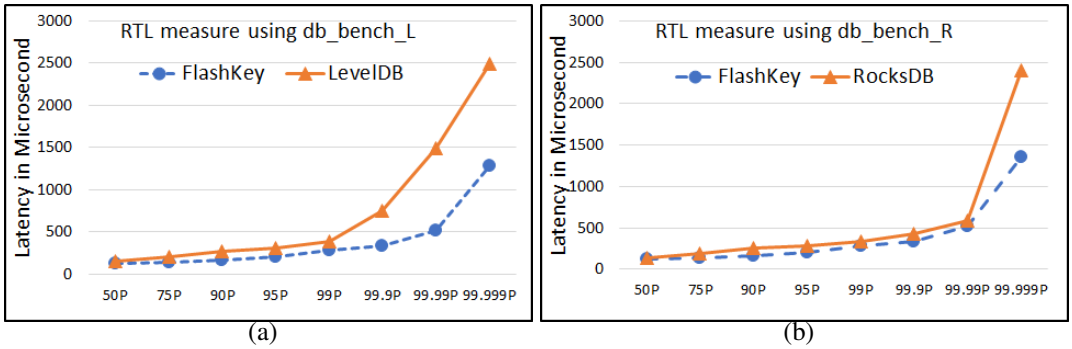
Overall, when compared to RocksDB, FlashKey achieves improvement in the range 22-50% in latency, with the highest improvement under workload 3. The gain over RockDB is lower as RocksDB improves latencies with reduced compaction at the cost of higher SA (SA in RocksDB was 1.7X compared to 1.24X of LevelDB). For workloads 4 and 5, FlashKey attains an average latency improvement over the RocksDB in the range (22-48)%.

Table 2.5: Percentage Improvement in Random Read QoS

Metric	LDB	RDB
RTL-50P	18.5%	10%
RTL-99.999P	48.43%	43.7%

Read QoS Comparison

Since FlashKey is designed to support fast Get requests on SSDs (via randomized hash-table lookups rather than LSM tree lookups), it shines on read tail latency (RTL) introduced earlier. Figure 2.6 illustrates RTL of FlashKey compared to both (a) LevelDB and (b) RocksDB. Lower numbers are better on the graph. Especially on "5 9s", aka 99.999P RTL, FlashKey is 2x better than both the baselines. Table 2.5 summarizes the RTL improvements.



(a) (b)

FIGURE 2.6: Comparison of Random Read QoS of FlashKey against (a) LevelDB and (b) RocksDB with workload 2 of db_bench_L & db_bench_R respectively.

Throughput Comparison

In this section, we compare the throughput of FlashKey, LevelDB, and RocksDB under the YCSB system benchmark, with configuration as detailed in section 2.6.1. For throughput analysis, we used the YCSB benchmark as its traffic generation is based on real cloud workloads. In contrast, db_bench traffic is rather artificial. For our experiments, we used a 50 GB database in which the keys are 23 bytes and values are 1000 bytes long. To keep the database size consistent, as recommended by the developers (Cooper et al., 2010),

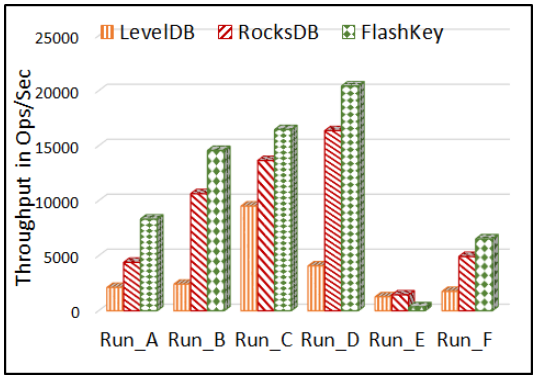


FIGURE 2.7: Throughput Comparison using YCSB

we execute the workloads of Table 2.3 in the following order: (1) Load 50M records in the database with workload A; (2) Run workload A, B, C, F, and D in order for 50M operations each; (3) Drop the database - i.e. completely delete it; (4) Load 50M records with workload E; (5) Run workload E for 1M operations (each scan cost 100 reads on average). Except for the record and the operation counts, all the other YCSB workload parameters are kept unchanged from their defaults.

YCSB runs atop a selected KVS. In our runs, we kept the KVS configurations unchanged from those detailed earlier for db_bench_L and db_bench_R. We previously discussed the difficulty of alignment of the SA of FlashKey with the SA of the baseline KV systems, and it is even more difficult to achieve this alignment with YCSB as some of the workloads dynamically insert new records, which increases the size of the database. We use a constant SA of 1.23x for FlashKey.

Figure 2.7 shows the performance in terms of *Throughput* of FlashKey compared to RocksDB and LevelDB when tested with YCSB workloads. For all the workloads except workload E, FlashKey's performance exceeds that of both RocksDB and LevelDB. Since our design is optimized for point insertions and lookups, and not for range queries, workload E (range-scan based workload) is as-expected slower when run on FlashKey. Range scan in FlashKey essentially does random reads and LSM traversal, plus HT lookups. Hence, for the workload E which constitutes of majorly range scan, FlashKey is not as

performant as either RocksDB or LevelDB due to not keeping the data in order. However, for the rest of the cases, we achieve better performance even without keeping the data in order.

Among the other workloads, while testing with workload A which is a 50:50 intermix of reads and updates, FlashKey achieves the maximum gain up to 90% against RocksDB over the others. In case of workload C which is purely read traffic, RocksDB comes closest to FlashKey, but is still 17% below the throughput of FlashKey.

Effect of moving the entire HT from memory to disk

As explained in section 2.5.1, the HT in FlashKey can be (1) entirely contained in the Host-DRAM, (2) on disk and write-back cached into the DRAM, or (3) on disk and write through cached in DRAM. Depending upon which location we choose for the HT, there are tradeoffs in DRAM cost, recovery time, access latency, WA, RA, etc. Here we compare the bookend configurations (1) and (3). Table 2.6 lists the average latency in microseconds (denoted as AL), WA and RA measurements for random read, random write, and read-write mix workloads (workloads 2, 3 and 5 respectively of Table 2.2). We conclude from the measurements that FlashKey is significantly more performant than RocksDB even when the hash-table is entirely on disk, with significantly better WA, latencies, and RA, especially on typical mixed read-write workloads. While a comparison against LevelDB is not shown in Table 2.6, note that LevelDB is slower than RocksDB, and our gains against LevelDB are even better.

2.6.3 Algorithmic Analysis of FlashKey

We define, the key size:K, value size:V, the maximum number of KV pairs in the system:N and the SSD capacity:C. FlashKey consumes a small amount of SSD capacity for HT and other system level information. The system level information is less than 1 MiB and is ignored in the following discussion. The HT size is $36N$ (bytes) since each HT entry

contains 36 bytes. As we use a hash function and hash-table size that largely avoids collisions, in most of the cases, we find our desired data with a high probability (95%+ in all our experiments) within the first two nodes.

The remaining SSD capacity is divided into multiple LBands where each LBand contains multiple LBand entries. Each entry contains a 16B LBand entry header and the actual KV pair. There is also a header associated with each LBand called LBand header (32B/LBand) which is different from the LBand Entry header. For simplicity, we skip this header and the LBand journal (optional structure) for now. At any given time, FlashKey does not use all the LBands' storage capacity for current KV pairs. Instead, a fraction of this capacity, henceforth denoted as OP, is reserved as over-provisioning for efficient compaction. OP is defined as OP = $\frac{\#Spare\_LBand(s)}{Total\_\#LBand(s) - Spare\_\#LBand(s)}$.

Table 2.6: FlashKey - In memory vs On disk HT comparison

Workload	Metric	RDB	Disk	Mem
2	AL	125	133	90
	RA	3.5	4.6	2.7
3	AL	346	250	183
	WA	10.2	3.8	1.5
5	AL	671	597	537
	WA	10.1	4.9	1.3
	RA	7.6	3.7	4

Now, among K, V, N, C, and OP, we have the following relationship: $C = (1 + OP)((16 + K + V)N + 36N)$. If the HT size is much smaller than SSD, we have $N = \frac{C}{(1+OP)(16+K+V)}$. The SA can be calculated as follows:

$SA = \frac{(1+OP)((16+K+V)N+36N))}{N(K+V)}$.

With low collision probability, for the Get(.) operation (both sequential and random), FlashKey reads the HT entry first, and then loads the KV pair from the corresponding LBand entry; hence the RA (not accounting for GC yet) is $RA = \frac{16+K+V+36}{K+V}$=$1 + \frac{52}{K+V}$.

Note that the RA (and also WA) are defined here in terms of bytes read (or written) rather than the IOs performed.

For the Put(.) operation, FlashKey writes the KV pair to the current open LBand first and then updates the HT entry. For the sequential put operations (with no valid data requiring relocation during GC), the WA is; $WA = \frac{16+K+V+36}{K+V}$=$1 + \frac{52}{K+V}$. Whereas for random Put(.) operations, there will be additional writes during the data relocation in GC and the corresponding WA is; $WA = \left(1 + \frac{52}{K+V}\right) X \left(\frac{1+OP}{1+OP+LW(z)}\right)$, where LW() is the Lambert W function (Desnoyers, 2012), and $z = -(1 + OP)X(e^{-(1+OP)})$.

For example, let us assume $C = 4$ TB, average key size $K = 20$ B, and average value size $V = 800$ B, and the minimum OP is 18%. Under this configuration, the maximum number of keys will be $N = \frac{4TB}{1.18(16+20+800)}$= 4.05 Billion. Since we need additional space for HT and other metadata, we set the maximum number of keys to 4 Billion. The HT size is set to $4X10^9X36X2 = 288$ GB, and the SA will be $SA = \frac{4TB}{4X10^9X(20+800)B}$=$1.22$.

RA under sequential and random Get(.) will become $1 + \frac{52}{820} = 1.1$, and WA under sequential Put(.) is also 1.1. The WA under the random Put(.) will be $1.1X3.46 = 3.81$, where 3.46 is the WA under 18% OP. These WA, SA, and RA measurements assume to have all the elements of FlashKey stored on the disk (worst case). A mixed random read and write workload introduces GC due to the random writes/updates. The GC requires extra reads and writes to the disk, and results in higher RA and WA. Please note that the above analysis provides only an estimate based on mathematical modeling; it does not incorporate the impact of sector-aligned I/O. This is different from the disk reads and writes which are issued in practice. As such, the mathematical analysis provides a lower bound, and the experimental results included above provide a better understanding of true behavior on real workloads.

2.7 Conclusions and Future Work

In this paper, we presented a new KVS system for SSDs called *FlashKey* that is based on hashing of keys, logical-banding of KV pairs, combined with an empty-value LSM-tree. We showed both experimental and analytical evaluation of FlashKey vs. popular LevelDB and RocksDB KV systems and demonstrated O(1) lookup of Get(.) requests while maintaining low RA and reducing WA by up to 12x. Average access and tail latencies are improved by up to 85% and 2x respectively. These improvements persist as database size increases, e.g., with 100M KV pairs and SA aligned at 1.57X, compared to RocksDB, FlashKey achieves 20-60% lower latency, $\sim 9X$ lower WA, and 1.5-2X lower RA for db_bench_R.

While FlashKey performs well compared to both RocksDB and LevelDB, there is scope for significant improvements. In particular, the memory needs of the hash table can be reduced significantly by hashing on a key range (by ignoring least significant few bits of the key) rather than on individual keys. The number of bits to ignore can be chosen to ensure that the space taken by the HT is no more than some specified fraction of the database size. Such a mechanism also helps with the range searches and will be the most useful for databases with small key-value sizes. The HT can also be cached partly in the memory based on a prediction of the most active regions of the key space. Finally, more sophisticated hashing mechanisms can be employed, e.g., Cuckoo hashing to increase HT space utilization, and locality sensitive hashing (LSH) to enhance sequential access performance. In the future, persistent memory can also be exploited to keep the HT.

CHAPTER 3

ADAPTIVE DATA CENTER NETWORK TRAFFIC MANAGEMENT FOR DISTRIBUTED HIGH SPEED STORAGE

3.1 Introduction

The purpose of this chapter is to design tenchiques to handle network congestion due to high performance storage systems. Storage systems i.e. storage technologies and protocols in data centers are undergoing a rapid transformation have already shattered the conventional wisdom that the networks are fast and the storage is slow (Pydipaty et al., 2018). For example, currently available high-end NVMe SSDs (e.g. Samsung MZPLL12THMLA) can put out up to 50 Gb/s for large sequential accesses such as streaming media (Zheng et al., 2019). Thus, a few SSDs in a storage server can overwhelm 100 Gb/s Ethernet. Faster technologies such as Intel Optane (*https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html*) not only consumes higher bandwidth, but also compete with end to end network latencies in larger networks.

Fortunately, the average IO or network traffic in a data center network is invariably very low and the real issue is not sustained high traffic but frequent bursts in traffic due to large data ingestion or output. We see this behavior even in our university cluster traf-

fic. Fig. 3.1 shows the traffic pattern of one logical unit (LUN) over 11 hours of a day which has several sharp peaks. Thus, a well crafted dynamic migration (or move) and replication (or copy) of data can deal with such bursts and minimize QoS violations. At the same time, consolidation of traffic during low periods is necessary to (a) avoid additional latency because of (unintended) scattering of data throughout the network as copies are created, migrated, or removed, and (b) effectively exploit the low power mode of the links. Unfortunately, traffic consolidation and congestion mitigation are conflicting goals thereby requiring careful data placement and movement techniques.

Achieving this tradeoff involves several challenges. First, if packet drops are to be avoided, an ongoing flow cannot use a different copy of the data chunk, instead, any migration is limited to only new flows. Second, the migration/copy creation must be very lightweight so that the additional traffic generated does not worsen the congestion. Third, the additional overhead of consistency management across chunk copies in relieving network congestion should not affect the overall performance. Fourth, the replication/migration used during congestion episodes should not result in substantial scattering of data throughout the network as it worsens low traffic performance and ability to do energy management.

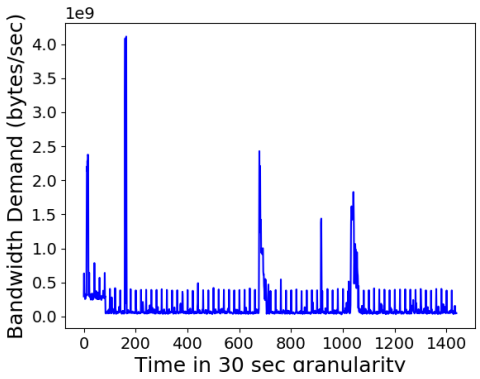


FIGURE 3.1: Access time-series based on Temple University cluster data traces

To address all these challenges, we have developed an incremental chunk management mechanism that adaptively combines chunk migration and replication, which is the main contribution of this paper. Extensive simulation shows that the combined chunk migration and replication mechanism provides 4x better performance than using them in isolation. For the simulation, we modify the existing NS3 simulator and show that in a disaggregated NVMe storage system under traffic bursts, our mechanism can provide up to 47%

improvement in latency with read-dominated traffic, and strategic data migration can provide up to 22% improvement in delays with write dominated traffic as compared to an optimized initial placement without any further changes. To the best of our knowledge, no work has addressed above mentioned challenges in achieving congestion vs. consolidation tradeoff in disaggregated NVMe based storage.

The rest of the paper is organized as follows. Section 3.2 explores the background and motivation behind the work. Sections 3.3 and 3.4 discuss the related work and the proposed chunk management methodology respectively. Section 3.5 discusses the optimal chunk placement and movement mechanisms, including both initial optimal placement and its incremental optimization. The experimental setup and results are covered in Section 3.6 and the paper is concluded in Section 3.7.

3.2 Motivation and Background

Storage systems are evolving rapidly from the slow spinning magnetic media to high-speed flash technologies (i.e., SSDs) and even higher speed technologies NVM technologies (e.g., PCM, MRAM, etc.), which can rival DRAM access latencies. Alongside there have been rapid developments in storage access protocols that are much leaner and lower in latency (e.g., the NVMe protocol that is rapidly becoming ubiquitous). This emerging high-speed storage will invariably be accessed over the data center network by many hosts regardless of how it is deployed (from fully centralized to fully distributed per server storage). With distributed applications deployed in VMs/containers on different hosts and accessing large amounts of data, remote storage access is a norm rather than an exception.

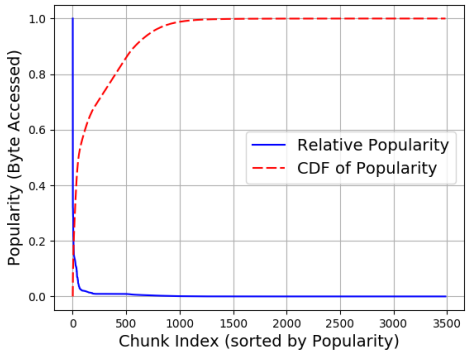


FIGURE 3.2: Chunk Popularity based on Temple University cluster data traces

The net result of these trends is that storage systems can drive tremendous throughputs, although this typically happens only for brief periods. Furthermore, with storage device latencies going into 10's of microseconds or less, the network latency becomes a significant part of the end to end latency and needs to be managed carefully, especially during congestion periods. Thus an intelligent management of network congestion during burst periods becomes important. Since the congestion management generally requires spreading out the data (to spread out the network traffic), we also need to consider the fact that scattered data becomes undesirable during low traffic periods (which are dominant) as it interferes with the ability to achieve low network latency and exploit power-saving techniques (e.g., sleep modes) for links with low utilization. Thus, an intelligent mechanism needs to manage data location (by copying or moving) in a way to both avoid the congestion and to avoid keeping it scattered throughout the network.

The dynamic data placement to achieve the congestion mitigation vs. traffic consolidation tradeoff requires that the storage be divided up into "chunks" of a suitable size and virtualized so that it is possible to move these chunks dynamically without any impact on the applications in terms of addressing or accessing the data they need. Storage virtualization is a very mature technology and is used extensively. It may be performed by the host, a virtualization appliance, or the switch. While each mechanism has its pros and cons in terms of network impact and delays, we do not delve into those details here.

Despite virtualization, chunk movement without allowing ongoing transactions to complete can lead to lost or out of order network packets.

This mobility limitation tends to make the problem of data movement/copy management more challenging, as discussed later in the paper. It is also worth noting that the storage access pattern is generally highly skewed, which means that a small fraction of chunks will account for a large percentage of accesses. This is illustrated in Fig. 3.2 for our university traffic by plotting the distribution of chunk popularity. It turns out that approximately 11% of the LBA (logical block addresses) are responsible for 80% of the

traffic. In fact, the popularity distribution often turns out to be similar to Zipf. Obviously, only the highly popular chunks need to be moved or copied to deal with congestion; however, since highly popular chunks are likely to be accessed from multiple hosts, managing their mobility becomes quite challenging.

In addition to the data movement and copying explored here, dynamic per-flow changes to network routing can also be used to mitigate congestion and to consolidate traffic. We have studied this aspect in our earlier work (Murugan et al., 2012); therefore, we do not address it here. It is possible to integrate data movement/copying with intelligent routing, but this is beyond the scope of this paper. Also, while one of the objectives of traffic consolidation during low traffic periods is better network energy management, this paper is not focused on that aspect either.

The copy management discussed in this paper could make use of the copies normally created for better resilience; however, our copying is limited only to very hot chunks that would likely reside in the highest storage tier and extra copies are likely to be short-lived.

3.3 Related Work

Several works focus on performance aware data placement and data replication from different viewpoints. In particular, references (Yang et al., 2017; Wang et al., 2017) concentrate on placement in a cache tiering environment to achieve performance in terms of in-storage access latency, but the *network bottleneck* issue is overlooked. References (Xiong et al., 2008; Zamanian et al., 2015) focus on a distributed file system and database respectively and propose replication with *static workload estimate*. However, the authors do not consider the overhead due to *consistency control* inside the workload estimates.

The authors in (Ananthanarayanan et al., 2011; Abad et al., 2011; Xie et al., 2010; Zheng et al., 2014; Jin et al., 2013; Meng et al., 2010; Li et al., 2014) focus on the locality-aware data placement in a distributed storage system. References (Ananthanarayanan

et al., 2011; Abad et al., 2011; Xie et al., 2010) mainly consider distributed big data applications (e.g, Hadoop and distributed database). In (Ananthanarayanan et al., 2011), the author uses the correlation between the number of accesses and concurrent accesses to find popularity. In (Abad et al., 2011) the author works on the same basis (chunk's popularity), but the popularity is determined by correlation analysis between access frequency and age of the file. The work in (Xie et al., 2010) additionally considers the host's capability for hosting the replicas while making the placement decisions. References (Zheng et al., 2014; Jin et al., 2013; Meng et al., 2010; Li et al., 2014) discuss network-aware end-point consolidation, with references (Meng et al., 2010) and (Li et al., 2014) place VMs with more mutual communication close to one another to reduce delay. In (Zheng et al., 2014), authors have extended the work of (Wang et al., 2012) by considering VM utilization and correlation analysis for both the VMs and flows in a coordinated manner. Scheduling the data access can also improve the performance (Zaharia et al., 2010), but in a large distributed storage system this would introduce *additional overhead* to manage the coordination.

Locality aware data replication/placement has also been studied in Wide Area Network (Sivasubramanian et al., 2005; Karlsson and Karamanolis, 2004). GlobeDB (Sivasubramanian et al., 2005) uses clustering of data chunks based on the read and write amount conducted by each server and place each chunk cluster to any single server based on a cost function, which considers three metrices: read, write and consistency traffic. Reference (Karlsson and Karamanolis, 2004) studies different heuristics to solve the NP-hard chunk placement problem.

Contrary to the above approaches, in this paper, we propose (a) a dynamic data-chunk placement scheme that balances both congestion and consolidation depending on change in traffic bursts, (b) adapts to the traffic behavior (read/write) by using either replication or migration to mitigate congestion; (c) incrementally reacts to any perturbation in traffic; and (d) finally addresses all the overheads due to replication, migration, consistency control.

3.4 Proposed Methodology

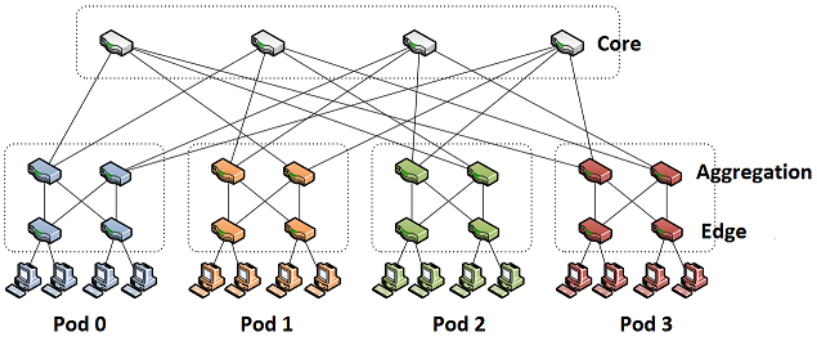


FIGURE 3.3: An illustration of fat-tree network

We assume a traditional data center network architecture organized as a "fat tree" illustrated in Fig. 3.3. An order k fat-tree has k pods, each pod contains k switches arranged in two layers (aggregation and access) with $k/2$ switches each. The figure shows a rather small fat-tree of order $k = 4$. The leaf nodes connect to top-rack (ToR) switches, and each rack contains some number of compute and storage servers. All links are bi-directional and generally have higher bandwidth at higher levels of the hierarchy. Nevertheless, higher level links are more likely to be congested if the applications and the storage used by them are scattered throughout the network.

We assume that all storage is virtualized with the chunk size of 256KB. Since this paper is only concerned with storage traffic, we assume separate logical channels carrying the storage traffic. Such channels can be realized using MPLS at IP level or through the use of CoS (class of service) feature in data center Ethernet.

Although any storage chunk can reside on any storage server, it is desirable to keep the chunks close to the applications that access them the most while still consolidating the traffic under normal conditions. This can be done initially by solving an optimization problem, which we discuss in section 3.5. Solving such an optimization problem requires knowledge of various access types and intensities, which may be available based on histor-

ical behavior, however, the access patterns are likely to change over time. Thus in addition to the initial placement, we also need a mechanism for continuous monitoring of the entire network and occasional incremental optimization that moves or changes the copies of a certain number of chunks. We expect the need for incremental optimization to arise only occasionally and likely involve only a small fraction of all the chunks.

Our monitoring architecture is similar to one proposed in (Biswas et al., 2019) and involves a local controller (LC) residing at each switch and a global controller (GC) to coordinate among the LCs. The GC can run on the centralized management box, which usually has a logically separate communication path to all the switches. The well known SDN architecture provides a suitable paradigm for this (Kobayashi et al., 2014), where GC can be thought of as a controller interacting with individual switches with additional monitoring capabilities.

The prime job of an LC is to collect utilization of its switch and that of the incoming/outgoing links periodically, and it can do this without much overhead. A GC occasionally communicates with the LCs in the system to build the holistic picture of the network nodes and links. The LCs at ToR switches (rack level) track the utilization of all the chunks hosted by that rack. For scalability, the LCs hide the server level details inside the rack and only expose a consolidated view of the endpoints externally to the GC. The GC acts upon the overall rack-level statistics whereas the LC controls any decision internal to the rack which is essential to keep the GC overhead under check.

Given this architecture, it is possible to make decisions about when a chunk should have a copy made, moved, or a copy deactivated. The overhead of such dynamic replication includes (a) additional traffic at the time of replication, and (b) the synchronization traffic at the time of any updates. Other than the GC and the LC, the Virtualization is responsible for all the mapping and the translation, where the mapping can be influenced by the GC or the LCs. But we would not discuss the API related details here.

To address these, we assume a mechanism that minimizes replication traffic during high traffic episodes by using either a *reactive* or a *proactive* approach based on the dominant traffic type as learned from the history of active chunks. The *reactive* approach determines the number of extra copies needed for heavily used chunks during traffic bursts and creates those copies inline. In contrast, the *proactive* approach creates, in advance of the traffic burst and based on historical data, a small number of copies of the chunks that are known to be hit heavily. These copies are retained during low traffic periods but not in fully synchronized state. Instead, they are synchronized opportunistically in the background using *lazy synchronization*. The key advantage of the proactive scheme is substantially less copy synchronization traffic during burst periods; however, this comes at the cost of higher storage consumption. With either scheme, once the burst is gone, the redundant copies are deactivated.

3.5 Optimal Chunk Placement and Movement

3.5.1 Initial Placement of Chunks

We assume that the applications and the chunks used by them are placed optimally initially to minimize network traffic and adjusted incrementally as the traffic or applications change.

For initial placement, we formulate an optimization problem where x_{un} is a decision variable which is 1 if either the application or the chunk u are assigned to node n and 0 otherwise. Suppose r_i and r_o are the cost of accessing a chunk that is within and outside the node respectively, i.e. $r_o > r_i$. To linearize the problem, we also introduce an auxiliary decision variable e_{uv}, which is 1 if and only if the application u and chunk v are placed on different nodes. Let w_{uv} denote the weight (or relative intensity) of accessing chunk-v

from application-u and I_u the IO demand by the application or chunk u. We could then define our *Traffic-aware Application-Chunk Placement (TACP)* as follows:

$$\text{Min} \sum_{(u \in \mathcal{A} \wedge v \in \mathcal{C})} w_{uv} e_{uv} r_o + w_{uv}(1 - e_{uv}) r_i \text{ s.t.} \tag{3.1}$$

$$\sum_{n=1}^{\mathcal{N}} x_{un} = 1 \ \ \forall u \in \mathcal{A} \vee \mathcal{C} \tag{3.2}$$

$$\sum_{u \in (\mathcal{A} \vee \mathcal{C})} I_u x_{un} \leqslant \mathcal{L} \ \ \forall n \tag{3.3}$$

$$w_{uv} = 0 \ \ \{u, v\} \in \mathcal{A} \vee \{u, v\} \in \mathcal{C} \tag{3.4}$$

$$e_{uv} \geqslant x_{un} - x_{vn}, \ e_{uv} \geqslant x_{vn} - x_{un}, \tag{3.5}$$

$$e_{uv} \geqslant x_{un} + x_{vn} - 1 \ \forall u, \forall v, \forall n \tag{3.6}$$

$$e_{uv} \in \{0, 1\}, \ x_{uv} \in \{0, 1\} \ \ \forall u, \forall v \tag{3.7}$$

Here constraint in eqn(2) states that every application and chunk copy is assigned to some node. Equation(3) states that the overall IO rate of the node (denoted $\mathcal{L}$) is respected. The constraint in eqn(4) states that there is no IO between two applications or two data chunks. Constraints in (5)-(6) ensure that if e_{uv} = 1, then u and v are placed on different nodes.

The above formulation can be used at multiple levels in a real network where a *node* could represent pod in entire fat-tree, rack within a pod, and servers within a rack. During the initial placement problem, the TACP is solved twice to determine in which *pods* and then which *nodes* inside that pod the application/chunks are assigned.

Theorem 1. *The problem TACP is NP-hard.*

Proof. This can be proved easily by a reduction from the Minimum K-cut problem (MKP) in a graph representing flows between nodes. We skip the detailed proof for brevity. □

Because of the NP-hard nature of the TACP problem, we propose the following heuristics to solve it efficiently. We solve the problem in two stages. First, we assign apps/chunk-copies to a set of virtual nodes, each of which corresponds to a real node and has a specified IO capacity (i.e., every node is assumed to have the same IO capacity). Next, we map the virtual nodes to the real nodes of the network.

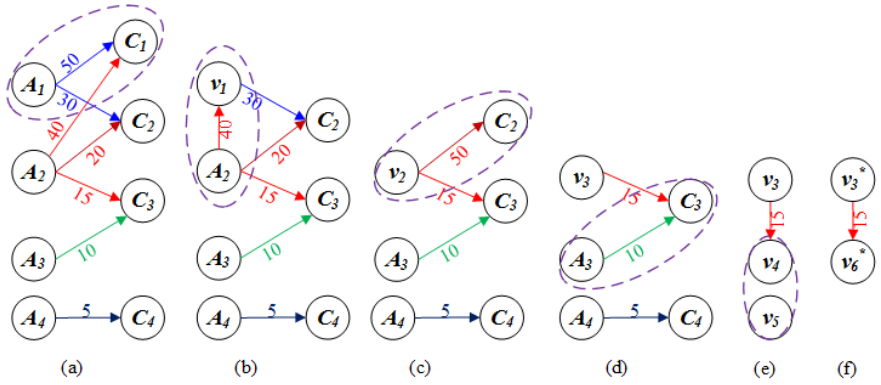


FIGURE 3.4: Assigning apps/chunk-copies to the virtual nodes

In the first step, we construct an affinity graph Fig. 3.4(a) with application and data nodes (A_i's and C_j's) and weighted edges showing traffic demand from A_i's to C_j's. We then greedily choose the edge with the highest weight (i.e. $A_1 \rightarrow C_1$ in Fig. 3.4(a)) and merge its vertices to create a new composite vertex, and direct all incoming and outgoing vertices from/to the merged node to the composite vertex. Fig. 3.4(b) shows this with composite vertex v_1. The composite node corresponds to application A_1 and chunk C_1 being placed on the same node. The composite vertex is attempted to be merged further so long as the co-location does not exceed the capacity of a virtual node. E.g., in Fig. 3.4(d) merging v_3 and C_3 would exceed the virtual node capacity, and thus at this point, the algorithm looks to a distinct merger (to create another virtual node). The end result is two virtual nodes v_3 and v_4 that cannot be merged further.

At this point, there may be some disconnected components that are merged/packed into at most N virtual nodes using the typical bin-packing solution (Garey and Johnson, 1990). Thus v_3^* and v_6^* become two virtual nodes after the first step.

In the second step, the virtual nodes are assigned to the pods. For this, we start with the n virtual nodes from step 1, and if $n < N$, expand them to N virtual nodes by inserting $N-n$ dummy nodes. We then recursively apply Kernighan-Lin (K-L) graph bi-partitioning algorithm that minimizes the cut weights across two partitions. The K-L algorithm divides these N v-nodes into two partitions each one with a size of $N/2$ v-nodes (assuming N is even). The K-L algorithm is again applied to these two partitions if they have non-zero virtual nodes. This process is repeated until each partition has more k virtual nodes. These partitions can now be thought of individual pods, where the virtual nodes are assigned from left-to-right based on their app/chunk loads (from maximum to minimum).

Theorem 2. *Assuming N is even, the K-L algorithm will be called at most $\left(\frac{N}{k} - 1\right)$ times.*

Proof. The K-L algorithm divides the virtual nodes into two equal-size partitions. Thus if we start with N virtual nodes, then from the formulas of G.P. series it is easy to verify that after $\left(\frac{N}{k} - 1\right)$ steps we will get the partitions with the size of k. □

Henceforth the term **"node"** explicitly refers to a **"rack"**, since all our strategies beyond this point are based on the rack-level estimates.

3.5.2 Incremental Placement Optimization

Since the application behavior and hence the network traffic will vary over time, it is essential to monitor the traffic constantly and incrementally make adjustments to the position or number of copies of highly used chunks.[1]

To facilitate chunk migration/replication, we assume that we initially deploy k copies (e.g., $k = 3$) for every chunk across the storage servers. These copies include the initial

[1] It is also possible to occasionally re-run the initial optimization but this may not be desirable if it leads to substantial data movement.

number of copies needed in the initial optimization (the active copies) and a few extra copies (inactive copies). The additional copies, if any, are placed according to some fair distribution rule, for which we use a simple hash-based approach described below. The purpose of the extra copies is to allow for inexpensive copy activation when required by the high demand without creating substantial additional network traffic. All inactive copies are synchronized opportunistically (to minimize activation cost) but active copies are kept entirely consistent.

In addition, the GC maintain a bitmap $BM(c)$ for each chunk c where each entry is primarily k bits long (we use $k = 3$). Thus if $BM(c, i)$, $i \in 0..k-1$ is 1(0), then a predefined hash function $h_i(c)$ for this position provides the location for the ith active(inactive) copy of chunk c. The GC also stores the activation ordering of any chunk using $k-1$ additional bits which it uses during copy deactivation (thus each BM entry costs $2k-1$ bits for any k). The LCs do not require access to this bitmap. The mechanism will, however, require communication between the GC and the virtualization engine during the time of migration/replication, but detailed consideration of the overheads of this interaction is beyond the scope of this paper.

We assume that the episodes involving the arrival of large traffic bursts and their subsequent dissipation are infrequent, as observed from several traffic traces. Therefore, the frequency of update of BM is expected to be relatively low. Nevertheless, in very large data centers, the maintenance of a centralized data structure such as BM may be undesirable. This aspect will be explored in our future research.

Copy Activation and Deactivation

For both replication and migration of chunks, we use the following mechanism based on the traffic monitoring (in bytes/sec) by the LC. For replication, the LC watches the outgoing traffic of the node (this is primarily the read traffic), and for migration, the incoming traffic (this is primarily the write traffic). When this traffic overshoots a certain high cutoff

threshold τ_h (shown as 75% in Table 3.1), the LC reports the event to the GC along with the top P highly used chunk numbers with their respective utilizations.

Suppose that the current utilization of the i-th chunk is U_i. Then we have the following situations:

Replication: Suppose that chunk i has n active replicas with evenly distributed load among them. Then, creating another copy (replica) of that chunk will reduce their estimated load to approximately $\frac{n}{n+1}U_i$.

Migration: Migrating the chunk from node m to p will reduce load U_i from node m and increase U_i at destination node p. For any chunk with multiple active copies in the system, the incremental optimization migrates only the copy from the node that has raised the event, while keeping the locations of the remaining active copies being unchanged.

Based on this approximation the GC selects chunk from the list in descending order of their utilization at the nodes and activates the replicas at some other node that (a) has an inactive replica of that chunk, (b) can accept a certain amount of additional load (i.e. $\frac{n}{n+1}U_i$ in case if replicating the i-th chunk) and (c) has the least load (preferably preoccupied) satisfying (a) and (b). The GC continues to activate replicas until the expected load of the congested node goes below the normal limit of τ_{n2}. In case there is no node that satisfies these conditions, that chunk is not replicated or migrated and GC moves to the next chunk in the sorted list.

When the traffic surge fades away, our scheme needs to deactivate some copies activated during congestion so that the overhead of maintaining the consistency among the replicas is avoided. A migration also involves a copy creation initially followed with new traffic directed to the copy. Only when the ongoing traffic dies down, the original copy is deactivated. Deactivating extra copies when the traffic subsides can be rather tricky. Deactivating it too early after the traffic burst subsides could hurt in terms of delay since there still might a built-up backlog that must be cleared.

There could be several policies for copy deactivation. Each of them has its advantages and downsides. It may be difficult to find a single copy with low utilization under our traffic forwarding mechanism. So infrequently, the LC updates the GC with the set of Q chunks having low utilization. From the list, GC selects only the chunks with multiple active copies. It then deactivates the last created copy of the chunk and shifts its load equally to other active copies. (Note that no deactivation will happen if the remaining copies cannot handle the extra load).

Concurrency Control

Since we use multiple copies as a way of relieving congestion, we must address the multi-copy consistency issues as well for chunks that are shared across applications and thus could get access requests for them asynchronously from multiple applications. As usual, this can be achieved in two ways: (a) via locking or pessimistic concurrency control (PCC) or (b) via optimistic concurrency control (OCC) which rolls back any conflicting transactions (Kung and Robinson, 1981). It is well known that the OCC works better in environments with low contention for the resources. Assuming that the chunk sharing is not prevalent, the contention is expected to be low, and hence we choose to implement the OCC. In OCC, the resource access by a transaction must still be monitored; however, the requester can proceed without locking. When the transaction is ready to commit, a check is made if any other transactions have committed since this transaction started. If so, the transaction is rolled back and may retry after some delay to avoid the conflict. The transaction also checks for conflict upon arrival and backs off if another transaction has already started. Conflicts are still possible since there is a small gap between determining that there is no conflict and actually recording that the transaction has started.

It is important to note that our concurrency control operates at the storage level and is only concerned with individual read and write operations. Additional concurrency control

may be applied at a higher level (e.g., for database transactions running on top of the storage system).

3.6 Experimental Evaluation

3.6.1 Experimental Setup

To comprehensively evaluate the network congestion control mechanism we have enhanced the popular NS3 network simulation package to include the modeling of chunk based storage access, concurrency control, and traffic control via local controllers (LC) and global controller (GC). We built a small fat tree infrastructure for $k = 4$ with 100 Gb/s network links in the core and the aggregate layer and 10 Gb/s at the access or edge level.

For the storage, we consider device characteristics similar to those for currently available low-latency NVMe SSDs. However, we do not consider or simulate the complexities of the storage systems here – assuming that the storage device utilization is kept reasonably low, the effect of queuing for storage devices is not important to our study. For routing, we use the Equal-Cost Multi-Path Routing (ECMP) with a point to point connections. Also to avoid acting upon intermittent short-lived spikes, we perform *exponential smoothing* of the congestion metric. Some of the key parameters for the experimental setup are given in Table 3.1.

Table 3.1: Simulation Configuration Values

Parameters	**Values**	**Parameters**	**Values**
Fat Tree Size (k)	4	Mid Cut-off (τ_{n2})	55%
Zipf dist.(α)	0.8	Normal Cut-off (τ_{n1})	45%
#application instances	47	Default chunk size	256K
#unique data chunks	1500	SSD Read Latency	$25\mu s$
#servers per rack	14	SSD Write Latency	$100\mu s$
High Cut-off (τ_h)	75%	Smoothing factor (β)	0.85

Application and Data Chunk Model

In our system, we have M application instances and N data chunks. We define the data chunk access intensities following a Zipf distribution over the N ($N > M$) unique chunks with the decay factor $\alpha = 0.8$. We then randomly map the applications to the chunks. Based on the mapping between an application to the set of chunks, we determine the application frequencies. Without any sharing on the chunks across applications, the application set should also follow a Zipf distribution. Additionally, we share each chunk across a small set of applications.

Copy Activation Overhead

As stated earlier we can either create copies proactively or reactively. Under *proactive control*, we decide m alternative locations per chunk using m hash functions and create copies in advance, which are then synchronized in the background. In *reactive control*, we create copies as needed (i.e., at the time of congestion). Additionally, under replication, as we keep multiple copies of chunks active and share them across a small number of applications, we use Optimistic Concurrency Control (OCC) model to keep the copies consistent.

Chunk size and Metadata Overhead of our Scheme

The suitable chunk size is configured by the virtualization layer and it involves balancing the overhead of metadata maintenance (which prefers large sizes) vs. the overhead of moving chunks and controlling false data sharing (which prefers small chunks). For our mechanism, smaller chunks are preferred, and the chosen size of 256KB is unlikely to be burdensome in most situations because of the highly skewed nature of storage accesses. Note that our mechanism only needs to store the chunk id, offset, and three active locations of that chunk which results in a few bytes (32 bytes) per chunk. This has a miniscule storage overhead (0.0125%).

As described in section 3.5.2, for the bitmap GC uses only 5 bits per chunk (for 3 extra copies), hence for a petabyte of storage with 256K chunk size, the bitmap will require 2GB of memory space. GC also stores the node level utilization for all the nodes to decide the best-fit location for the chunk while replication or migration. LCs do keep track of the chunks inside a single node which constitutes of chunk number, server number, and utilization. We make the assumption that at most 10% of the data is active at any point in time for which LC requires an in-memory monitoring structure. Given the chunk usage are roughly balanced across the pods in the data center (but there could be local imbalance inside a pod) each LC will require 0.5GB to 1.5GB of memory for monitoring.

Application Traffic Generation Model

We have largely used synthetic traffic for our evaluation as it allows for an easy change in the traffic parameters. We generate traffic based on the statistical characteristics of real block device access traces collected from both Temple University Cluster[2] and (Verma et al., 2010). Analysis of the trace file in section 3.1 shows several periods of high activity, which we term as traffic *"burst"*, and develop our traffic generation model around these traces.

In our model, network flow duration follows Pareto distribution with shape=3.5 and mean as 3.6 ms. The bandwidth of each flow follows a uniform distribution with mean 1Gbps and a range from 500 Mbps to 1.5Gbps. To avoid the remote communication latency we keep the bisection bandwidth of the network topology adequate. We measure performance as – *the average and the maximum observed latency faced by any application issuing remote storage access request.*

[2] https://drive.google.com/file/d/1gSCToOCk4oigxLd8jKO3LRLlMWoblr9C

3.6.2 Experimental Results and Discussion

Optimized vs. Random Initial Placement

We start with a comparison of our optimized initial placement (*OIP*) against a uniform random placement (*URP*) of both the applications and the chunks. Under 20% utilization, the *OIP* only occupies 8 out of 16 available nodes (racks). In contrast, the *URP* involves all nodes as expected. We use the same traffic for both the cases and compare the performance of both the placements.

Fig. 3.5 shows the comparison of average latency under changing read fraction (95% to 1%). With bidirectional traffic, the network link gets congested at the extreme points where the traffic is either read or write dominated. At 95% read or 99% write traffic, we achieve the maximum reduction (approximately 95%) in average latency with *OIP*. Also *OIP* does not have any packet delivery issues whereas *URP* fails to deliver approximately 1% of the packets by the end of the simulation due to long delays. As long as the resources are available, *OIP* tries to consolidate the applications with their associated data chunks and accommodates them in the same pod (perhaps on the same node) to reduce packet propagation delay. Although mixed read/write traffic results in lower congestion due to bidirectional links, *OIP* can still achieve 18% reduction in average latency at 50% read traffic.

The *OIP* will no longer be optimal under traffic bursts, and incremental optimization becomes necessary. It replicates or migrates busy chunks depending upon which direction of a bidirectional link is congested. Both operations require activation of additional copies of chunks at different nodes. In the case of replication, all future requests are evenly distributed among all active copies. In contrast, in the case of migration, all new requests go to the new copy, and when old requests die down, the old copy is deactivated.

Impact of Incremental Optimization

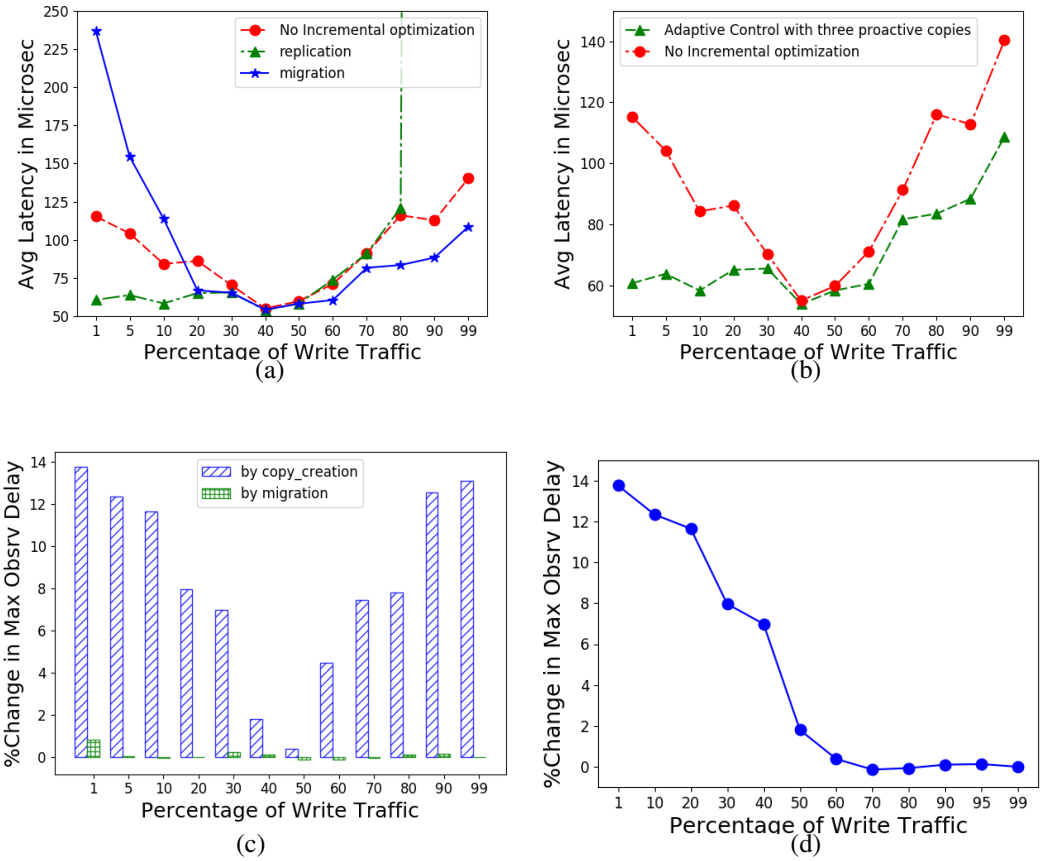


FIGURE 3.6: (a,b) Average latency and (c,d) maximum observed latency reduction for various chunk management mechanisms

Figs. 3.6 shows the effect of varying percentage of read on average delay under 20% utilization. At that utilization, our *OIP* only uses half the number of nodes, which increases the utilization of those resources to higher than 20%. The rest of the nodes are not at all utilized. We assess the performance of incremental optimization by comparing average latency for the requests under two cases; in one, we only use the *OIP* (denoted as "no opt") whereas in the other we use incremental optimization over

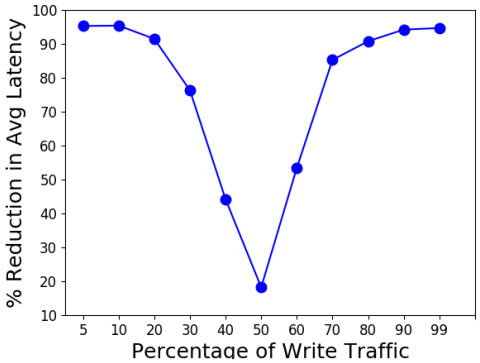


FIGURE 3.5: Avg. Latency for Random vs optimized placement

the *OIP* to change the number or location of active copies to accommodate the change in demand.

Change in average delay with changing read fraction The following scenario can be treated as the best case for the incremental optimization where we create inactive copies *proactively* and perform *lazy synchronization*. We create inactive copies of working set on all the nodes. Hence during congestion, the traffic that we need to send to activate those inactive copies is mostly very small. Also, assuming copies everywhere (*"copies everywhere"*) gives us the flexibility to activate the replica on any node. Here in the current experiment independent of the ongoing traffic characteristics (read/write), we either only migrate or replicate.

Fig. 3.6(a) shows the average latency with and without the incremental optimization following the *OIP*. With read dominated traffic, the incremental optimization (which creates copies of the busy chunks) achieves approximately 47% improvement over *no opt*. This improvement results from the enhanced parallelism due to the extra copies. However, with write dominated traffic, extra copies cost additional synchronization traffic which actually increases the delay significantly as compared to *no opt*. Instead, a migration under write-heavy traffic achieves a 22% latency reduction. Obviously, migration is not helpful under read-dominated traffic. It follows that we need an adaptive mechanism that can track the congestion and the nature of the traffic (e.g., read fraction) and accordingly decides whether to copy or move chunks.

So we introduce adaptive control based on the read fraction. The system decides whether we copy or move the busy chunks. Fig. 3.6(b) compares the average latency of this mechanism against *no opt*. When the traffic is mostly read; the system tries to reduce the congestion by replicating but when the traffic largely write dominated, the system reduces the consistency control overhead by migrating the busy chunks. Please note

when there is not much traffic ongoing in either direction (read% 50-60%), neither copy nor migration is beneficial as the congestion in both the directions becomes insignificant.

Change in maximum observed latency with changing read fraction Fig. 3.6(c) depicts the change in maximum observed latency that supports our findings from incremental optimization (Fig 3.6(a)). The maximum observed latency for any flow decreases by 14% compared to *no-opt* as we replicate due to parallel request service. However, when we migrate, the maximum observed latency remains identical to that for *no opt*. So by migrating, we do not lose anything in terms of maximum observed latency, whereas copy creation always reduces the maximum observed latency. Fig. 3.6(d) confirms our findings from Fig. 3.6(b) when tested under adaptive control and compared against *no opt*. It shows improvement in maximum observed latency is achieved through replication under read dominated traffic.

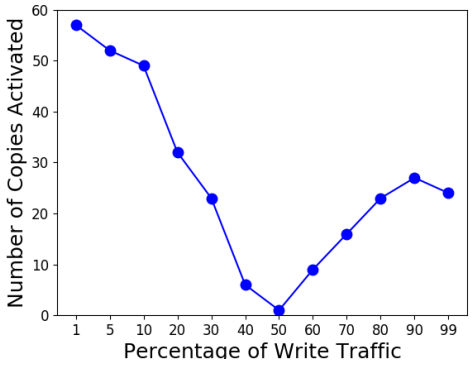


FIGURE 3.7: No. of copies activated with adaptive control

Number of copies activated vs. read fraction under different policies Fig. 3.7 shows the number of additional copies that are activated (and eventually deactivated with replication) over time. Across all the cases, in the worst case, we only activate a maximum of 50 additional copies out of all 1500 chunks. However, copy deactivation in our system follows a conservative approach. This sluggish deactivation might not have any impact on read dominated traffic, but it could increase concurrency control traffic when we replicate under write-heavy traffic.

Due to the performance advantage of the adaptive mechanism over simple migration/replication, we use the *adaptive mechanism* for the rest of our experiments.

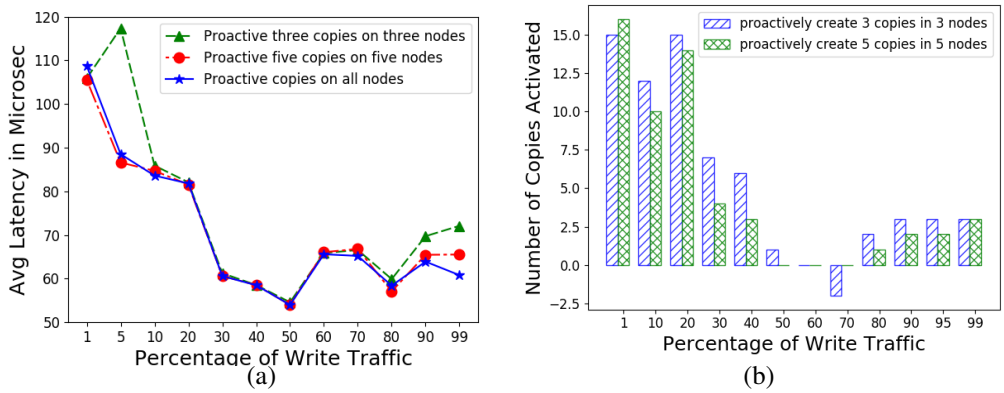


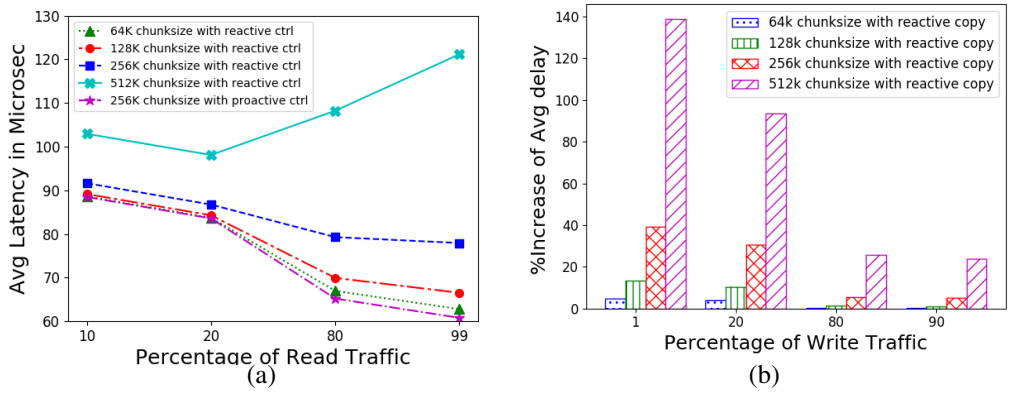
FIGURE 3.8: Comparing (a) Average delay and (b) number of copy activated under limited copy approach over copy everywhere

From the perspective of efficient copying/migration, we would like to have a copy of each chunk available at each node. However, this would require an enormous amount of extra storage. Hence to keep the number of copies limited, we tested our system with three (*three copies*) and five (*five copies*) proactively created copies of all the chunks. We compare the performance of these cases against *copies everywhere*. Fig. 3.8(a) shows that *three copies* are enough to achieve significant performance improvement. With mostly read traffic, no significant performance improvement is observed beyond *three copies*. However, with write traffic, increasing the number of inactive copies can provide better location choices for migration, but at the cost of increased consistency control overhead.

Many-a-times the system fails to find a new destination for a busy chunk due to IO rate limitations at nodes hosting the inactive copies. Note that we determine the locations of the inactive copies in advance, and in a highly dynamic system, it is very likely that the action taken for one busy chunk can influence the migration/replication decision of another busy chunk. So if we now compare the number of inactive copies that are successfully activated during the bursts Fig. 3.8(b), fewer chunks are activated with the limited number of copies than for *copies everywhere*. For example with 90% read traffic, we activate 34 total copies with *three copies*, 35 with *five copies* and 49 with *copies everywhere*.

We do see a few exceptions; e.g., at 40% read, *three copies* creates two more copies than *copies everywhere*, yet the latency is 2% less in the latter case. This suggests that

as the system fails to find a destination for a busy chunk, it chooses a less important chunk. Creating copies of such chunks requires more migrations/replications to bring the congestion down.



(a) (b)

FIGURE 3.9: (a) Average delay with proactive and reactive control and (b) Increase in delay during bursts over proactive approach

Since keeping *three copies* is enough to handle congestion, we use this configuration for further experiments.

Impact on Average delay under reactive copy creation with varying chunk size

Figs. 3.9 show the impact of increasing chunk size on average latency with inline copy creation (as opposed to our proactive method). In Fig. 3.9(a) we show the average latency by varying the chunk size from 64k to 512k. We compare the performance against our proactive approach with *three copies (default chunk size)*.

As stated above, our default chunk size is 256k. We perform the test with four different read proportions consisting of either read or write-heavy traffic. With read dominated traffic, the delay increases with the chunk size. At 99% read, compared to the proactive approach, the reactive method with a chunk size of 512k increases overall latency by 99%. This latency increase is due to the copy creation traffic that puts additional load on read path itself. For the same reason, with the write-heavy traffic, the increase in

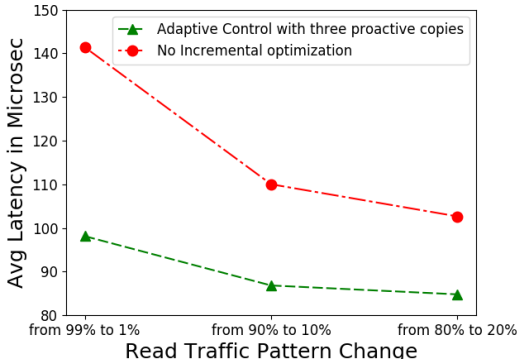


FIGURE 3.10: Avg latency vs. read fraction

delay due to reactive copy creation decreases significantly. With 90% write traffic, we see an increase of 0.2%, 1%, 5%, 23% in delay with chunk size 64, 128, 256, and 512k respectively. Fig. 3.9(b) zooms into the bursts period, and the plot shows the increase in delay over proactive copy which confirms the same. Here due to bigger chunk size, the delay increases by 140% at 99% read fraction, which is more than the increase in overall delay.

Thus our findings from Fig. 3.8 and Figs. 3.9 indicate that with write-heavy traffic, one can afford to have reactive copies as long as the chunk size is small to moderate. With limited copies, activation may fail due to the shortage of resources at those nodes. Reactive copies will reduce the concurrency control traffic and might provide better locations for migration. In contrast, with read oriented traffic, it is beneficial to create copies proactively.

Adapting to Traffic type and Intensity Changes

Finally, we tested our system under not just change in traffic intensity, but traffic where both the intensity and the read-write ratio changes. Fig. 3.10 shows the change in average delay when both traffic type and intensity changes. We use three cases where traffic type changes from (1) 99% to 1% (2) 90% to 10% and (3) 80% to 20% read, with traffic intensity increases as before. We use *three copies* and compare the performance against *no opt*. Please note that in all our previous experiments, the read fraction was constant

throughout a simulation and only varied across different runs. However, in this experiment, the read fraction of the traffic changes within a single run to test the adaptability. We found that our mechanism adapts nicely by both migrating and replicating chunks based on the change in traffic intensity and type. Also, we achieve 30% improvement over no optimization as we move from completely read to write heavy traffic.

3.7 Conclusions and Future Work

In this paper, we discussed an adaptive mechanism to decide whether to migrate, activate an inactive copy, or reduce the number of active copies to handle changes in the network traffic due to variations in network traffic generated by high-speed storage devices that are becoming the norm in the data center. Through extensive simulations, we show that it is possible to achieve near 47% improvement in average delay with purely read traffic. However, for chunks that are mostly written, the best strategy is to migrate them out of the congested node and place them at some low utilized node.

In the future, we would like to study the scalability of the interactions between the GC, LC, and virtualization engine (VE) in a moderate to large-size data center environment. Assuming switch-based virtualization, the LC and VE functionalities can be integrated into a single module which avoids the overheads associated with separate entities. Furthermore, the GC can use a federated model whereby each pod has a lower-level GC that communicates with its peers to keep an overall network view consistent across all the GCs. However, this straight forward design still has many challenges that are to be addressed to achieve performance at scale, which we would like to explore.

In addition, we would like to integrate data migration/replication with intelligent routing through flow consolidation on network links to design a storage and network-aware consolidation vs congestion trade-off mechanism.

CHAPTER 4

COORDINATED POWER MANAGEMENT IN DATA CENTER NETWORKS

4.1 Introduction and Motivation

In this chapter, we are going to discuss a holistic power management of the data center network (DCN) via coordination among distinct controllers with different functionality and scope. This area of research is currently relevant since, with increasing size and energy footprint of data centers, their effective energy management is crucial. The major power consumers of the IT infrastructure in a data center are compute/memory subsystem, storage subsystem, and data center network (DCN) (Greenberg et al., 2008). In this paper, our focus is the DCN, which consumes an increasing percentage of power as the network speeds and connectivity rise due to network upgrades. For example, the power consumption of a 10 Gb/sec Ethernet can be anywhere between 2-10 times the power consumption of 1 Gb/sec Ethernet, depending on the number of ports and the technology used (Sohan et al., 2010). The power consumption goes up with the number of powered-up ports even when they are idle since the underlying link Phy is synchronous and constantly consumes power to keep transmit and receive sides in sync. Furthermore, the increasing speeds generally result in much lower network utilization, since the deployment of higher-speed links

is mostly motivated by the technological availability, latency considerations, and ability to handle highly bursty workloads, and much less by sustained high bandwidth demands. This is particularly true in HPC data centers with dense interconnects (e.g., hypercube, toroid) (Murugan et al., 2012) and HPC workloads that may go through multiple steps of collective communication (high network traffic) and parallel computation (very little network traffic). Such situations make network energy management techniques even more important. Another type of data center infrastructure is a multi-tenant colocation data center or a "colo" center. Such a center rents out servers and associated networks and storage infrastructure to multiple tenants. The colo operator is mainly responsible for power, cooling, security, hardware maintenance, etc. Therefore, it is crucial to improve both the energy efficiency and sustainability of such data centers (Guo et al., 2018).

Our underlying model of the data center is one that serves *external* requests coming in through a user request assignment controller (RAC). Each such request lands on a server, and in turn could generate *internal* flows across various servers using the DCN. For example, an HTTP request may initiate some application and database traffic across the servers. Since our focus is on DCN, we avoid the issues of application deployments and the details of application-level interactions. Instead, we only consider source and destination assignments for individual flows.

As in any energy management context, there are three potential ways in which energy consumption of DCNs can be reduced: (a) Shape the workload at the source by techniques like batching of requests or proper admission control policies, (b) Reduce the link speeds commensurate with the loading – a kind of dynamic voltage/frequency switching (DVFS) control, and (c) Assign the source, destination and network path for each request so that the traffic is largely concentrated on fewest links and thus maximizes the opportunity for other links to sleep.

Workload shaping (Erickson et al., 2013) is a well-studied subject and we do not consider it here, although it could help in more effective use of sleep modes that we do explore

here. Link level DVFS is also thoroughly explored, but it is practical only at a very large time-scale and coarse granularity level. This is because most networks provide only a few speeds (e.g., 40, 10 and 1 Gb/sec) and the switching essentially amounts to PHY switching and can be very slow. Although IEEE introduced the Rapid PHY Selection (RPS) mechanism to allow for dynamic speed changes, such a mechanism is still too slow and may cause packet drops and other problems (Blanquicet et al., 2007). Therefore our focus is primarily on (c) and we use the link sleep mechanisms defined at PHY and higher layers, as detailed in section 4.4. In particular, we use the Ethernet specific sleep mechanism called Low Power Idle (LPI) in addition to the generic PHY level mechanisms.

The energy-efficient Ethernet standard approved by IEEE in 2010, improves the Ethernet energy proportionality by defining a link sleep mode known as Low Power Idle (LPI). Although the standard defined the low-level mechanisms for switching to low power mode, the EEE standard does not define the strategy for deciding when to enter and leave the low-power mode. The right configuration of EEE is critical for maximum energy saving and low-performance overhead (Sergio et al., 2012) and the best setting depends on the distribution of the traffic on the network. Currently, all these parameters are set by the vendor at default values without any knowledge of the workload. This often results in LPI not working well and perhaps the cause of the general perception that LPI doesn't work and should be turned off. Previous studies show that energy savings depend on the traffic pattern and network load (Christensen et al., 2010; Reviriego et al., 2010). Some of the recent research has tried to understand the impact of EEE for different types of applications such as MapReduce (e Silva and Carpenter, 2017), video streaming, scientific computing, etc. For example, Map-Reduce has a very specific traffic pattern during different phases that one could exploit for network energy management without introducing substantial latency. In our work, the routing mechanism dynamically consolidates the traffic to enhance possibilities for using low power mode for as many links as possible with a loose coordination mechanism.

Flow path selection and consolidation to maximize sleep opportunities can be quite challenging in a large data center network because a workable scheme must simultaneously consider the availability of endpoint (or server) resources (CPU, memory, etc.), local bandwidth demands at the switches, and end-to-end blocking/delay on the network paths. In a small data center, this can be accomplished via a central controller that has global network and endpoint visibility, however, such a scheme does not scale. In this paper instead, we design a low-overhead semi-distributed scheme that involves three cooperating mechanisms: (a) an energy-aware distribution of requests to the endpoints by the user request assignment controller (RAC), (b) an intelligent assignment of a new flow at each switch by a local controller (LC), and (c) a lightweight global controller (GC) that monitors network traffic and provides hints to LC's and RAC for better placement of flows at endpoints.

The most prevalent network topology in data centers is "fat-tree", and this is likely to remain so in the foreseeable future. Therefore, we largely use this topology; however, our underlying mechanisms are general and can work for any regular network topology. In particular, we also study the hypercube topology, which is most relevant in the context of HPC data centers. However, good mechanisms to handle the energy savings vs. congestion avoidance trade-off do depend on the specifics of the topology. Also, while this paper considers Ethernet-based interconnect, the precise MAC layer is unimportant; what matters is the available energy management capabilities. Thus our results can be applied to the highly popular Infiniband based data centers as well, assuming similar sleep mechanisms.

The rest of the paper is organized as follows. Section 4.2 provides a broad overview of our scheme along with the key contributions. Section 4.3 reviews the related works on network energy management. Section 4.4 discusses power management mechanisms for a network interface including the low power idle (LPI) mechanism for Ethernet. Section 4.6 discusses the enhancement of the NS3 frameworks. Section 4.5 discusses various controllers, their interaction, and how they are implemented. Section 4.7 presents details

on our simulation-based evaluation of the mechanisms as well as the simulation results. Finally, section 4.8 concludes the discussion and lays out future work.

4.2 Overview and Contributions

An intelligent choice of flow paths through the DCN and their potential reorganization requires global visibility into the network that is not present in traditional networks. Although a software-defined network (SDN) provides the same kind of visibility, still there is a matter of cost and other deployment issues associated with it. So our proposal evolves from the existing management layer, residing inside switches. We assume that the management layer on each switch monitors the bandwidth usage on all the interfaces. Periodically this information is passed to a global network coordinator i.e. Global Controller (GC) that runs on some server. GC can use this information for flow placement and/or reshuffling; however, doing so will make our solution unscalable. Thus in our approach, GC plays only an advisory role and provides hints to the local controllers. Following the instructions, the local controller can undertake some measurements to alleviate network congestion in the event of approaching congestion. Those measurements are for example - the probability of choosing a path, changing the priority of the outgoing links, etc. Those have been discussed in Section 4.5, and Section 4.7. The whole intent of our work is to keep the GC both lightweight and non-critical – the LCs can continue to function uninterrupted but with degraded performance even if the GC fails and is restarted. There is a question of contradiction between the course of actions, taken by the LC and GC. But this not likely to occur since the LC only follows the recommendation provided by the GC, which is invoked infrequently. LC is only responsible for 1)sending the bandwidth information and 2)changing the local parameters. The above two actions are responsible for both energy efficient and congestion free communication.

Properly directing the incoming request to the appropriate server and choice of destination server for the flow is crucial for ensuring consolidation of the workload at the hosts without creating any overload. From the perspective of DCN, traffic consolidation is an important aspect of energy management. But aggressive consolidation of requests at the host level may lead to congestion at the network links if the consolidation decision being made independently without considering the network capabilities. Likewise, greedily packing flows into the network links can also create network congestion. Thus, the network switches must ensure that consolidation on outgoing links does not lead to network congestion at any time. We accomplish the goal of energy management without creating congestion by having the GC provide hints both to the RAC and the LCs. By doing that GC ensures necessary adaptation of LCs and RAC to the perturbation in a lightweight and non-critical manner.

It is important to note that the energy management can be handled –simultaneously if needed – at multiple time granularities. For example, built-in hardware mechanisms often operate transparently at fine time scales (e.g., C1-C6 CPU states), whereas sophisticated traffic consolidation schemes based on solving the optimization problems and reconfiguration can operate at long time granularities (10's more minutes or more). Most techniques explored in the literature fall in the latter category (Mahadevan et al., 2011). In contrast, our primary interest is in medium grain mechanisms that can be implemented in software but must be relatively lightweight and non-disruptive.

We emphasize on the placement of the new incoming flows intelligently along the network to avoid rerouting of any ongoing flow. Since rerouting of an ongoing flow can cause variable packet delay, packet reordering, packet drops, etc. due to its critical hand-off process. Additionally, it can also increase energy consumption under different circumstances. By extensive simulations, we show that our scheme can significantly *reduce the overall network power consumption (by up to ∼40% compared to the DCN without any power saving scheme). This is especially true during low traffic periods which become*

increasingly prevalent in DCNs as the link speeds increase, as discussed in Section 4.1. The power savings comes at the cost of increased latencies, but that increase becomes significant only at high network utilization, where it is not even clear whether we can save any energy by applying the aforementioned techniques.

We have used the network simulator NS3 for the implementation of our techniques. While working with the simulators, we find out that NS3 currently has little to offer in terms of energy management. It merely defines some parameters on energy consumption. To remove this deficiency, we have implemented an energy model for NS3 based on currently available network device energy management features in both inside-the-box fabrics (e.g., PCI-E) and outside-the-box fabrics (e.g., Ethernet). We have enhanced different NS3 packages to capture the real time matrices like link utilization, idle time, sleep time, active time, etc. at the net-devices. Besides, we have enhanced the simple deterministic routing model provided by NS3 into a hint based probabilistic routing model. In this paper, we describe these implementations and demonstrate the test results for the two common data center interconnect networks, fat-tree, and hypercube.

4.3 Related Works

Low Power Idle: Mostowfi (Mostowfi, 2015) studied EEE LPI operations that put the device into two states - deep sleep and fast wake. Thaenchaikun et al (Thaenchaikun et al., 2015) proposed an energy-saving model using a control plane that utilizes an energy-aware routing protocol. They showed that a combined strategy for routing protocol and energy-aware path augmenting solution can provide good energy savings. Nedevschi et al. (Nedevschi et al., 2008) proposed a buffer and burst scheme that shapes traffic to alternate active and idle periods, thereby providing increased opportunities to sleep. The authors also proposed a scheme where the rate of operation of network links is dynamically adapted to the arrival rate of packets. Abts, Dennis and Marty (Abts et al., 2010) discussed

link adaptation to dynamically reduce the link speed for less energy consumption using a central controller. They do so using adaptive link rate (ALR), which is a predecessor to LPI mechanism for EEE that attempts to switch PHY(s) at run time (as opposed to simply the initialization time). It turns out that even with Rapid PHY Selection (RPS) it takes time to change the speed of the link and there is a huge overhead of changing the link speed.

Traffic Consolidation: The better outcome through LPI is possible with minimization of the number of active links and switches, in certain cases. The works in (Al-Fares et al., 2010)(Wang et al., 2012) have tried to implement different intelligent mechanisms to achieve the goal of traffic consolidation. All the solutions consist of a centralized optimizer, which tracks the traffic statistics and redirects the traffic according to the optimization solution. For example, Wang (Wang et al., 2012) discusses the correlation of peak workloads among different flows while consolidating. In our work, the centralized optimizer uses topology statistics for both the traffic merging and request aggregation in a coordinated manner to avoid the conflicts in the agenda.

Endpoint/VM Consolidation: To save power at the endpoints, VM consolidation has also been considered as an efficient method in the literature. Most of them (Verma et al., 2008),(Liu et al., 2009) have considered mainly VM placement and migration as optimization problems to reduce the server energy consumption without any non-local network side considerations. Thus, the server energy savings may be accompanied by increased packet drop and jitter.

Request Assignment Controller:

In (Gao et al., 2017), the authors present an energy-aware request assignment controller that distributes the tasks on VMs based on their speed and energy efficiency. In their model, the migration decisions are based on the vCPU units demanded by an application and the available capacity of the host and of the other servers in the cluster. Authors in (Paya et al., 2015) schedule the tasks on VMs that are normalized based on system and application-level resource management. Their scheduling and migration of VMs are based

on the vCPU units demanded by an application and the available capacity of the host and of the other servers in the cluster.

Coordinated Energy Management: A coordination between VM consolidator and the network is essential to address the increased packet drop and jitter problem, as we have explored in this paper. In our work, we perform the endpoint consolidation by aggregating the external request to the target servers. There are several previous works on server-network coordination like (Zheng et al., 2014)(Jin et al., 2013)(Meng et al., 2010)(Li et al., 2014). The primary concern of the (Meng et al., 2010) and (Li et al., 2014) is to place VMs close to each other, those having more mutual communication. That reduces the path length, in other words, the delay and the bandwidth requirement at the higher layer of the data center. (Zheng et al., 2014) has extended the work of (Wang et al., 2012), by considering VM utilization and correlation analysis for both the VMs and flows - in a coordinated way. In (Jin et al., 2013), the optimization problem of VM placement and flow routing has been represented as a unified optimization problem and tries to solve it. Doing all these approaches, both the probability of network congestion and energy consumption of the overall system (includes both endpoints and DCN) can be reduced significantly.

Since VM migration can be expensive both in terms of latency and network traffic (Verma et al., 2008)(Travostino et al., 2006), we do not depend mainly on VM migration in our work. Instead, our main goal is to provide hints to the local controller and the Request Assignment Controller to work in sync. Besides, we work on the minimal assumptions: the affinity of VMs or the correlation of loads between mutual flows and VMs are unknown in our case. In reality, these metrics would be very uncertain, so VM consolidation/migration and flow routing based on that statistic might not bring expected results.

Network Energy Model Simulator: The existing work on energy modeling in NS3 is quite limited. Hu (Wu et al., 2011) presented the first framework for incorporating the energy model in NS3. Their study computes topology specific energy consumption via a framework shown in Fig 4.1 that does not link the network device to the energy model for

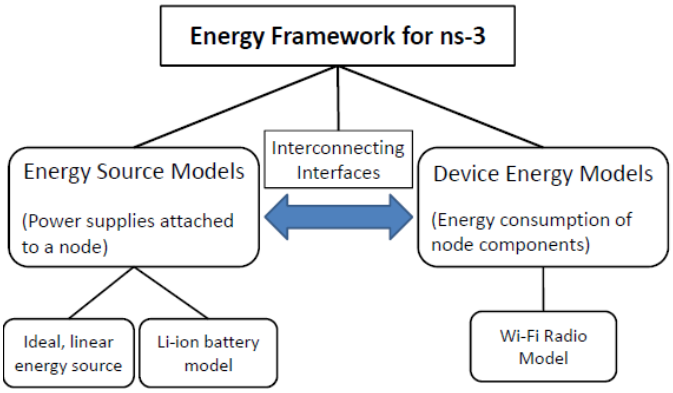


FIGURE 4.1: Existing NS3 energy model.

capturing the work done by the device, i.e. the actual work of packet transmission. But per packet transmission energy is not realistic for synchronous links. Tapparello (Tapparello et al., 2014) used the same model from Hu to design an energy harvesting framework. Our model is built over these basic frameworks while enhancing the energy metrics captured and granularity of data collected. We believe that these granular data and enhanced energy metrics will be of interest for future research studies and aid in understanding the underlying mechanism of device energy consumption.

4.4 Ethernet Power Management

4.4.1 Operating States of the Network Links

Nearly all of the network links are currently based on the serial links that use differential signaling because such technology can easily handle noise and does not suffer from cross-talk and clock skew issues. The links are then built using one or more such serial interfaces called "lanes". This has led to the notion of repurposable Phy layer, i.e., the same basic Phy module that can support very different higher layer technologies including PCI-Express, Ethernet, Infiniband, Fiber Channel, etc. At a very low level, all the links possess three operating states:

L0: This is the normal operational state with the highest power consumption, say, P_{L0}. P_{L0} does not have much dependence on the utilization since "filler" Idle messages

are constantly exchanged whenever the link is idle even for very short periods. In other words, the link is always 100% busy.

L0s: This is a sleep state with entry and exit penalty typically in 10's to 100's ns range. In L0s, the power level P_{L0s} is around 40-50% of P_{L0} depending on the design (Christensen et al., 2010). There is usually a trade-off between transition latency and sleep power. L0s control applies independently to both sides of a bidirectional link. The link is kept "trained" during L0s and thus a part of the interface must stay awake.

L1: This is a much higher latency non-operational state with exit latencies in 10's of μs range, but generally with very low power consumption (e.g., 10% of P_{L0}) (Christensen et al., 2010). This state requires a handshake between transmit and receive sides and link retraining upon exit from low power mode. If either side refuses to go into L1, L1 will not be entered. The training symbols are not exchanged (like L0s) during L1 and thus link retraining is required on wake up, which makes the exit latency quite high.

The L0s and L1 states are defined at the PHY level but may not be exposed at a higher level. In case of Ethernet, IEEE has defined an enhanced version of L1 called Low Power Idle (LPI) in the 802.3az-2010 (Christensen et al., 2010) standard. Instead of transmitting continuous idle signals LPI sends periodic refresh signals to maintain the synchronization. Wake up from LPI involves a significant exit latency since the transmitter needs to wake up the receiver before transmitting anything. There are also newly emerging energy management standards for 40 & 100 Gb/s Ethernet links, for example, 802.3bj (Mostowfi and Shafie, 2016) (deep sleep mode) and 802.3bm (shallow sleep with fast wakeup). The deep sleep mode is essentially the same as LPI described here, but the shallow sleep allows for faster wakeup.

4.4.2 LPI Models

In this paper, we describe two types of LPI models: Transmitter Only Sleep (TOS) model and (ii) Transmitter Receiver Sleep (TRS) model as described below. Basically, TOS is

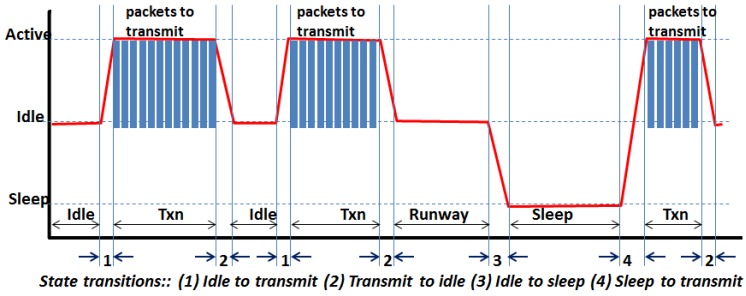


FIGURE 4.2: TOS packet transmit model

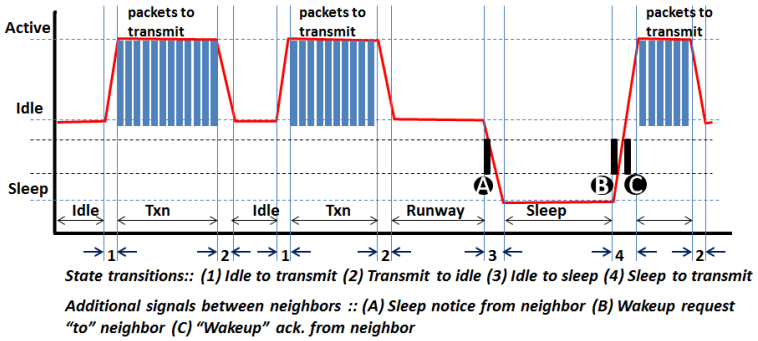


FIGURE 4.3: TRS packet transmit model

based on the availability of hardware level L0s sleep state and TRS is based on handshaking based L1 state, as stated earlier.

TOS Model: The TOS model is mainly driven by the hardware, hence it can apply energy management method at fine time grain for example gaps between successive packets on a link. Low power consumption of L0s state is offset by the very low exit latency to move from sleep to active state and resume transmission. The packet-transmit diagram for TOS is given in Fig 4.2. During active transmission periods, the device is in active state S_{active} and hence consumes active power. During the idle (no transmission) period, the link steps down to sleep state after some "runway" interval, as discussed by Kant in (Kant, 2011). We have followed the same "runway" concept both in case of TOS and TRS, to avoid unnecessary delay introduced due to transition to sleep states. A dynamic runway to adapt to the network state is left for future studies.

TRS Model: Compared to the TOS, the TRS model works at a coarser time grain and uses L1 sleep state. The runway for TRS is kept larger than TOS because of its higher exit latency. In the TRS model, each device is aware of its neighboring devices' energy state. Before switching to a deep sleep state, an idle device has to inform its neighbor of its transition to sleep state. Likewise, if the neighbor of a device is in a sleep state, the device has to wake up the sleeping neighbor, before forwarding the network traffic. This adds a level of management burden on both the transmit and the neighboring receive devices. This overhead involves - additional queue depth to buffer packets until the neighbor wakes up and acknowledges its active status, and an inter-device handshaking through packets exchange (sleep packet and wake up packet). The TRS state transition model and the packet transmission modes with the overlay of inter-device wake-up packets are shown in Fig 4.3. We study this model along with its effect on latency, queue depth and opportunities to extend the sleep state. This mechanism can save more energy than TOS when the link utilization is low, as we have discussed in the results.

4.5 Controllers and Coordination

The energy efficiency achieved by applying LPI can best be attained by the mean of different controllers and the coordination. In our design, the network operation and traffic management are controlled by three interconnected controllers: Global Controller (GC), Local Controller (LC), and the topology-aware Request Assignment Controller (RAC). We have further divided the functionality of GC into two entities, namely, Global Traffic Monitor (GTM) and the Global Traffic Consolidator (GTC).

When multiple controllers are involved, it is crucial to build a consistent strategy that needs to be followed by the controllers to avoid any contradicting actions (Raghavendra et al., 2008). GC is responsible for network monitoring and providing hints to LCs and RAC for traffic consolidation and external request placements, respectively. The main

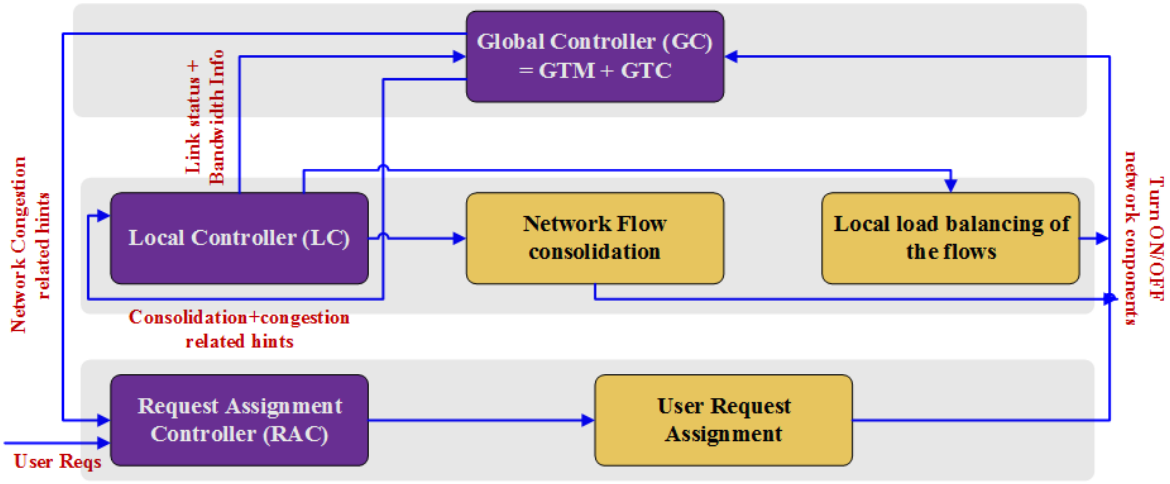


FIGURE 4.4: Overall Diagram

functionality of GTC is to build *consistent strategies* for the LCs in order to maximize sleep opportunities and to avoid network congestion. Migrating an ongoing flow from one path to another can be disruptive and may reorder packets; therefore, any migration should be done very infrequently. This requires that the GTM is reasonably active in collecting up to date information so that GTM can provide timely hints to LC's and RAC about flow/request placements for enough flow consolidation and at the same time avoid the network congestion.

Each LC periodically collects low-level network statistics (For example, link utilization, number of active flows, etc.) and sends it to GTM. GTM builds a global view of the network based on that accumulated intelligence. Depending on the global state of the network, the GTM sends back very little hints that can help LCs to make routing decisions. By adopting this simple information exchange mechanism, it is possible to keep the GTM scalable even with reasonably large networks.

LCs are responsible for the placement of a new incoming flow and can only do local consolidation of the flows. For example, in the fat-tree network shown in Fig. 4.5, an edge switch has two links towards the aggregation switch. From a power management perspective, it is better to concentrate traffic on one of them (if possible), so that the other link can have the longer idle duration and hence a better chance to save power by going into

the sleep mode. Similar consolidation principle can be applied to other interconnection typologies like hypercube, as shown in Fig. 4.6. It is clear that if each LC greedily focuses traffic on one of the links, this may result in congestion or suboptimal traffic distribution at higher levels. The purpose of GTM is to provide hints to LCs to avoid this situation. However, if this hinting is inadequate and congestion or poor traffic distribution does occur further up in the network, GTC can handle it by migrating some of the flows in such a way that they can still reach their destination. The GTC must ensure that it (1) does not move too many flows within the network, and (2) a flow that has already been moved from its original path, is not moved yet again in its lifetime. The later restriction is adequate in most commercial workload environments serving customer queries since the likelihood of having many extremely long-lasting flows is negligible. However, in HPC environments with long-running workflows, this aspect may need to be studied further.

Besides providing hints to the LC, at the same time GTM provides these topology aware hints to the RAC. Doing so, the LC and RAC can work collaboratively. These hints help the RAC to place an external request in such a way so that LCs get the opportunity to consolidate the traffic without much degradation in the quality of services. In the following, we discuss the optimization achieved by a coordination between these various controllers.

4.5.1 Local Controller

As mentioned before LCs work at the switching node level. LC operates based on the probability factor assigned to the outgoing links, and a *customized routing table* which records all the currently active flows. When any packet comes to a switch, LC first matches the flow id and the destination associated with the packet. If it finds any matches with any existing entry then it simply forwards the packet to the path associated with the entry. If not, the packet is forwarded based on the probability factor assigned by the GTC.

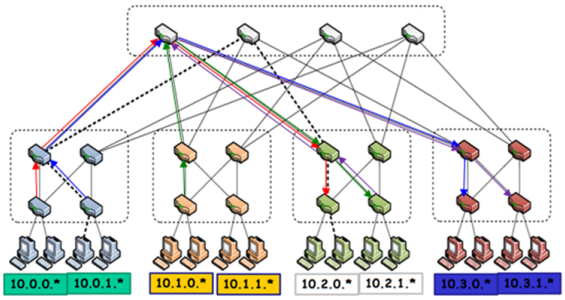


FIGURE 4.5: Fat-tree network illustrated

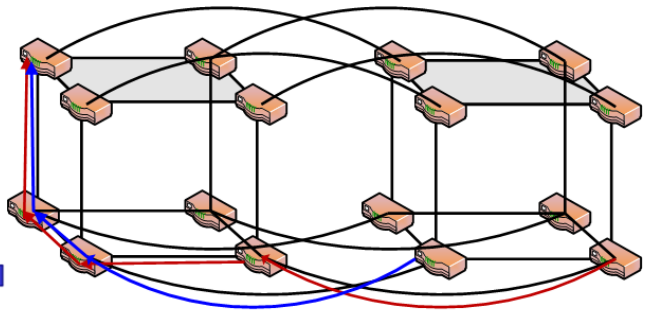


FIGURE 4.6: Hypercube network illustrated

In case the LC at a switch is not able to place a flow in its first attempt (due to congestion at some links), it simply drops the flow. We call such incidents "flow blocking". The amount of flow blocking is used as an indicator to balance our twin objectives of flow consolidation and congestion avoidance through load balancing. Notice that instead of dropping these flows, the LCs can try these flows through different links. However, we ignore this case for simplicity.

4.5.2 Global Controller

As stated earlier GC gives hints to LCs to consolidate the traffic over a small number of links, where others mostly remain idle and or more likely to have gaps between packets transmission and can easily move to sleep mode. GC does this by assigning a *probability factor*, which the LCs refer to while forwarding the flows. Notice that in a k-ary fat tree each edge and aggregate switch has $\frac{k}{2}$ candidate links for forwarding the traffic over other Pods. The candidate links are given ranks or priorities that are consistent throughout the network. Without any loss of generality, we assume that the candidate links can be ranked from left-to-right, i.e. the flows are mostly consolidated to the leftmost links of the switches.

For example in Fig. 4.5, the LCs assign the flows at the leftmost links which ensure better flow consolidation and thus energy savings. The probability factor determines which fraction of the incoming traffic is forwarded to each one of the $\frac{k}{2}$ candidate links.

We now explain calculation of the probability factor for the candidate links for k = 4, which can be generalized for any other k. There are two candidate links at the i-th switch, which are denoted from left-to-right as $\{l_{i1}, l_{i2}\}$ and their probability factors are $\{P_1, P_2\}$ respectively. The key idea is to ensure that the flows are mostly consolidated at the leftmost links at the time of low-traffic hours, whereas they are gradually spread across the network at the high-traffic hours to avoid unnecessary flow blocking. We define the overall network *utilization factor* as the ratio of cumulative bandwidth of all the inter-rack flows, and the total capacity of the edge-level links. This utilization factor is used as a nob to tune the values of P_1 and P_2 to consolidate or spread the traffic over the network. The overall scheme can be described by 4 states, as described below.

1. State1 (S1): The system stays in this state at the lull hours. This happens when there is no flow blocking in the last N windows (each window consists of n number of flows) and the utilization is less than a threshold τ_1. In this case, P_1 is incremented by Δ_i (whereas P_2 is decreased by the same amount) and N becomes 1. At this state P_1 is quickly incremented so that the flows are more consolidated at the leftmost links.

2. State2 (S2): The network switches from S1 to S2 if the utilization grows from τ_1 to τ_2 ($\tau_2 > \tau_2$), without experiencing any flow blocking in the last N windows. In this case, P_1 is incremented by Δ_i and N is incremented by a factor of α ($\alpha > 1$). Thus, in this state the probability of forwarding the flows to the leftmost links increases, but at a slower rate.

3. State3 (S3): The system transitions from S2 to S3, when the utilization grows beyond τ_2. In this state, the probability factors remain unchanged to avoid overburdening the leftmost links further.

4. State4 (S4): The system goes to S4 whenever it experiences any flow blocking. In this case, P_1 is decremented by Δ_d, whereas N is decremented by a factor of β

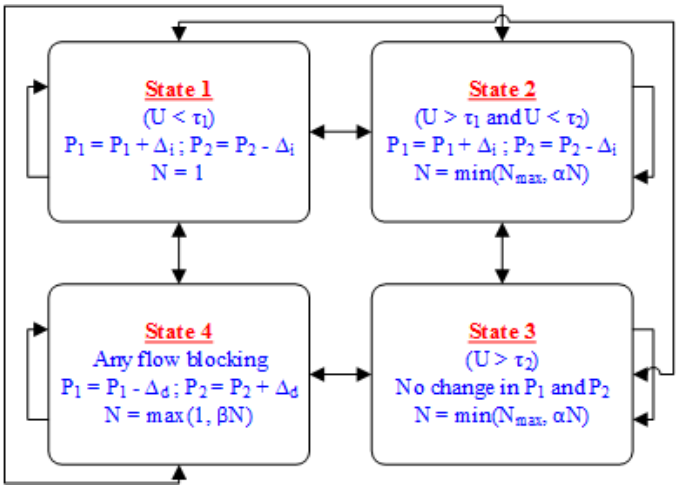


FIGURE 4.7: Proposed state diagram for assigning P_1 and P_2 to the links. U denotes the utilization factor.

($\beta < 1$). Thus if there are more flow blocking in the successive windows, P_1 is decremented quickly to shift more flows to the rightmost links. The entire state diagram is depicted in Fig. 4.7, where $N_{\max}$ is assumed to be the maximum value of N beyond which N is not increased.

We adjust the τ_1 value by adopting a simple learning approach. When there is no flow blocking for $N_{\max}$ windows with an utilization of U_{NB}, then the value of τ_1 is set to $U_{NB}(1 - \eta\Delta_d)$, where $\eta = 0, 1, 2, ..$ is a predefined constant. The objective of this is to keep τ_1 low enough to avoid flow blocking, and at the same time keep it sufficiently high for effective consolidation.

In addition to that, the GTC also provides hints to the request assignment controller to assign the user requests for better energy efficiency as mentioned in section 4.5.3.

In general for a switch with $n + 1$ possible links, the probability factors are calculated as follows. Assume that the probability factors for the $n + 1$ links are $P_1, P_2, \ldots, P_{n+1}$ respectively from left to right. Thus if P_1 is incremented (or decremented) by some amount, then probability factors for all the other corresponding links are calculated by solving the

above expressions:

$$P_2 + P_3 + \ldots + P_{n+1} = 1 - P_1$$
$$P_i = \zeta^{i-1}(1 - P_1) \quad \forall\, 2 \leqslant i \leqslant n+1, 0 \leqslant \zeta \leqslant 1 \tag{4.1}$$

where ζ is a system constant. The idea is that the probability factors of the links decrease exponentially from left to right. Thus the traffic is mostly consolidated at the left-most links, which provides enough opportunity for the links towards the right to sleep.

4.5.3 Request Assignment Controller

Based on the hints provided by GC, the RAC assigns the incoming requests to the servers. It also takes into account the host capacity (I/O per second, number of VMs, etc) while assigning the request. There are multiple policies for the RAC (Ray et al., 2018). Some of the policies can be taken independently by the RAC, whereas others need network information provided by the GC. The detailed evaluations of different policies of RAC are beyond the scope of this paper, instead, we mainly evaluated our performances to show the interactions between GC and LC.

4.5.4 Scalability of Control Mechanism

The amount of information exchanged between LCs and GC increases both with the size of the data center and the dynamism of the traffic in terms of congestion episodes. It is important to note that each switch handles the congestion locally in its outgoing links; only the congestion in the down-links requires exchange of hints between LCs and GC. However, the congestion in the downlink means that a lot of data headed to the corresponding rack. If this happens frequently or persistently, it means that there is a deficiency in the hints are provided to RAC by GC, rather than a scalability issue. In this regard, we note that the congestion notification to the GC is an event-driven process, rather than a periodic one. In the fat-tree topology, as the size of network (i.e., the parameter k) increases, so

does the number of alternative links, which is likely to reduce congestion episodes. Nevertheless, as the network size increases, the GC must keep track of more information and provide hints to more switches.

Suppose that at some point in time, the fraction X links have exceeded the higher threshold. The corresponding LCs will then continue sending the congestion notification until the congestion reaches down to the lower threshold. From the definition of Fat-tree, in a K-array Fat-tree, we have $K^2/4$ core switches, each connecting to all K pods. This gives $K^3/4$ downlinks from core switches. Each of the K pods has $K/2$ aggregate switches, each with $K/2$ downlinks, or a total of $K^2/4$ downlinks per pod from aggregate switches. The number of edge switches is the same as aggregate switches, each connecting to $K/2$ "hosts", which yields another $K^3/4$ downlinks. Thus we have a total of $3.K^3/4$ downlinks.

In real data centers, the "host" of a pure fat-tree structure can be considered as a rack connected via "top-of-the-rack" (ToR) switch to the edge switch on one side and the rack servers on the other. Thus a pure fat-tree structure beyond the rack level has $N = K^3/4$ racks and $3N$ downlinks. A standard rack can typically hold 42 1U servers, and as many as 96 blade servers. A rack typically also has two uplinks to load balancing and reliability, but let's ignore that for simplicity. We also do not consider congestion on the links going to individual servers in order to keep the focus on the fat-tree which lives above the rack.

Now to get some insight into the request handling capability required by the GC, we need to choose the fraction of congested links, say X, realistically. In a pathological scenario, it is possible that $X = 1$, i.e., every link is congested. This is completely unreasonable since real data center networks are designed with enough capacity to make packet drops and congestion a very infrequent occurrence. We assume that it is adequate to be prepared to handle, on the average, one congested link in every downlink set of a switch, i.e., $X = 2/K$. (Note that this assumption correctly accounts for the fact that congestion becomes less likely in larger network due to more alternate links). Thus, the required

request handling capability required by the GC, say R, is given by:

$$R \leqslant (3.K^3/4) * X = 3.K^2/2 = 6N/K \tag{4.2}$$

Now consider a case of $K = 32$, which yields $N = K^3/4 = 8K$. With 40 servers per-rack, this corresponds to a very large data center with 320K servers. For this, $R = 1.5K$. Suppose that the GC recomputes hints every 100 ms and send them out to individual switches to mitigate congestion. Such responsiveness should suffice for most interactive applications. Now, receiving, processing, and responding to 1.5K congestion notification requests in 100 ms should be possible for high-end server with 16 cores. The networking requirements are also very modest – 15K packets/sec ingress and egress, each perhaps only 100-200 bytes in size and should not stress the minimum 10 Gbps capability that one would expect to have here. As to the storage requirements, maintaining $3N$ or 24K different congestion entries is quite small and could well be mostly residing in processor caches (L1-L3).

A truly large data center may require breaking up GC into per-pod GCs coordinated by a global GC, but we do not consider such extreme cases here.

4.6 Implementation of Enhanced Energy Modeling Framework in NS3

To support idle power management and to study the trade-off between energy management and performance; we enhanced the NS3 package in several areas. We discuss about those enhancement in great detail in a separate work (Sondur et al., 2017). We modeled the port level and backplane level energy management techniques by overloading a few classes as shown in Fig. 4.18. In this UML diagram, we show the enhancements to the default NS3 libraries. We extended the base classes of the net-device, node, energy model and energy source to capture the component state model, additional timing metrics, energy metrics and packet delay.

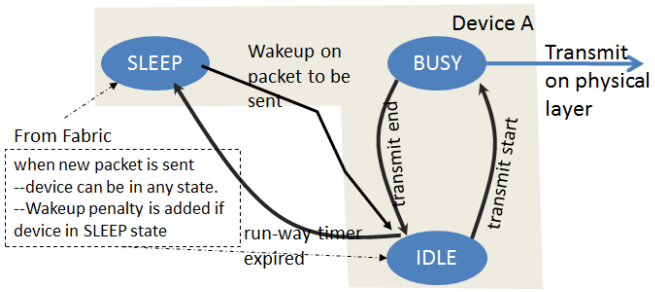


FIGURE 4.8: Point-to-point Net Device state model

The NS3 network devices implemented two states: BUSY and IDLE. Since we have a concept of "runway", to simulate our idea, we have enhanced the net devices by including low power reactive (LPR) state. The duration of this state can be set as user-defined parameter. It is reasonable to assume that the power consumption during entry to and exit from LPR is the same as IDLE power. During the LPR state, the power is the same as the active power. Furthermore, the additional features of our extended version are explained in sections 4.6.1 to 4.6.5.

4.6.1 Network Devices (NS3::PointToPointNetDevice)

Net device tracks the device queue, and implements MAC & PHY layer logic for P2P connections, in a network topology. We overload the methods Receive, Send, TransmitStart and TransmitComplete to support our TOS/$L0s$ and TRS/$L1$ mechanism. When no packet activity occurs for a predetermined runway period, the port transitions to SLEEP, thereby conserving energy. Any packet transmission during SLEEP state incurs an additional penalty to wake up the port. Therefore, whenever the device is in SLEEP state (deep or shallow), and first packet arrives, the packet is scheduled at time $t_0 + t_{wakeUp}$; where t_0 = current time and t_{wakeUp}=wake up latency. This additional time is accounted for transmission time. The state transition diagram of net device in shown in figure 4.8.

The following attributes[1] support the $L0s$ idle power management:

- t_{wakeUp} *wakeup penalty* (exit penalty for waking up the sleeping device)

[1] *italicized attributes* are preset values and **bold attributes** are computed metrics

- t_{runway} *runway monitor timer*
- **node backplane state** (active or sleep)
- **network device state** (enhanced state model)
- $\mathbf{t_{sleep\ |\ idle\ |\ busy}}$ sleep, idle & busy state duration
- **sleep count** to track sleep opportunities
- **queue metrics** observed during packet events

The queue length observation provides additional metric to understand the service time, average queue length, device utilization, etc. A further extension of this class to emulate $L1$ power level management and inter-device handshake is shown in Fig. 4.18.

4.6.2 Energy Models (NS3:SimpleDeviceEnergyModel)

The energy model "models" the behavior of the device w.r.t packet transmission and device states. Our enhanced energy model includes both the active current and the leakage current. We designed our enhEnergyModel to represent the energy consumption behavior of the network device (enhPointToPointNetDevice). The leakage current was used to model the behavior of the SLEEP state. Besides we provided flexibility to supplement the supply voltage (as active voltage) and another low power state voltage (to be used during L0s state).

4.6.3 Energy consumption of backplane/fabric (NS3:node)

In the basic switch fabrics, all the incoming and outgoing net-devices are connected through the device backplane as shown in Fig 4.10. The switch backplane consumes more power compared to the ports, connected to it. An NS3 node can be a basic representation of the switch backplane. We derived our extension emNode from basic NS3 node. We designed the energy management model on the node fabric by implementing a fabric state model and monitoring logic shown in Figure 4.9. The node fabric periodically checks the activity

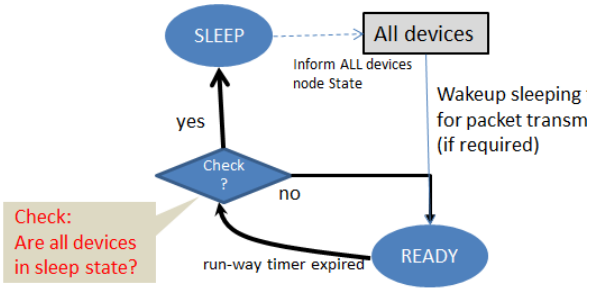


FIGURE 4.9: Fabric state model

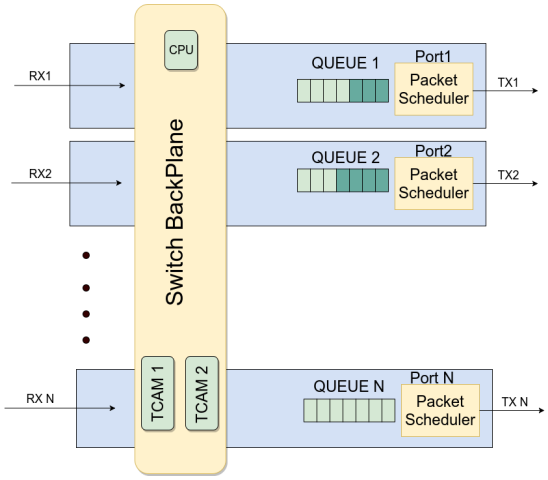


FIGURE 4.10: Switch Fabrics

of its connected network devices. Based on the inactivity interval, the fabrics can switch to low power (SLEEP) state. While in SLEEP state, the penalty for processing the first packet would be $t_0 + t_{wakeup}$, where t_{wakeup} is the wakeup penalty and in the order of 100s of μs(C6). Because of the high cost of fabric exit latency, the fabric wake time suppresses the net device wake up time. The attributes and corresponding metrics captured in the node backplane are as below:

- t_{wakeUp} wake up penalty (i.e., exit penalty)
- t_{runway} backplane run monitor timer
- $\mathbf{t_{sleep\ |\ busy}}$ sleep & busy state durations
- **backplane state** (as given by backplane state model)
- **count** of sleep states to track sleep opportunities
- $\mathbf{E_{sleep\ |\ busy}}$ sleep & active energy consumed by the backplane

4.6.4 Energy Source (NS3:BasicEnergySource)

BasicEnergySource is responsible to calculate the power consumption to the entire node. Since we have linked device models directly to the network device, we need to modify

the energy consumption calculation of this module (The *UpdateEnergySource* and *CalculateRemainingEnergy* methods). We define the overall node power consumption as:

$$Power(backplane) + \sum_{n=1}^{TotalPorts} Power(port_n)$$

Our energy source contains an active state voltage and low power state voltage[2] required to calculate energy consumption at various device states.

4.6.5 Enhanced Energy Efficient Routing Mechanism (NS3:Ipv4GlobalRouting)

We have enhanced the default routing module of NS3 by introducing deterministic and probabilistic routing. The base change for both types of routing is at the packet attributes. We include flow id attribute with each packet, to differentiate the packets from different flows. Besides the default routing table structure, we derived one dummy routing table structure. This includes additional attributes like putting the default routing table intact. The custom routing table includes information like bandwidth usage, flow path, current system constant, etc. Like LC (discussed earlier), this enhanced module sends periodically the dummy/custom routing table information to the GC. This frequency of information exchange can be varied by tuning user-defined parameter. The implementation of the GC is done on *GlobalRouteManagerImpl* class, which by default is the controller of all the *Ipv4GlobalRouting* instances.

The goal of both the deterministic and probabilistic routing is consolidation. While consolidation GC should put the network congestion into consideration. The decision (assign probability/priority) is based on the bandwidth usage data sent by the derived *Ipv4GlobalRouting* instances. In case of probabilistic routing, GC method of *GlobalRouteManagerImpl* class only assign one *system constant* value to the *Ipv4GlobalRouting* instances. The task of the LC would be to find out the probability distribution from that

[2] defined as per sleep state

system constant value (as discussed in section 4.5). The interactions of different enhanced NS3-modules is shown in 4.9.

4.7 Evaluation of Coordinated Power Management Technique

4.7.1 Experimental Setup

We used our developed simulator to show the effect of traffic consolidation on different network performance metrics. Unless otherwise mentioned, we assume a fat-tree topology with k = 4 that accommodates 16 host machines. We assume that the link bandwidths of all levels (i.e. edge, aggregate and core levels) are identical and equal to 10Gb/s. We study the following network parameters for comparison purposes: average end-to-end latency, power consumption and flow blocking percentage. We compare the advantage of power savings by adopting the TOS and TRS, considering both probabilistic and deterministic routing.

We have used synthetic data load to test the new functionality. Flow duration in the network follows Pareto distribution with shape equal to 3.5 and mean of 36 ms, ranging from 22ms to 375ms. The bandwidth of each flow follows a uniform distribution with mean 100 Mbps and within a range 50 Mbps to 150Mbps. To support the varying level of burst traffic, we implemented a two-state Markov Modulated Poisson Process (MMPP) for a flow. In this model, the two states have different (Poisson) flow generation rates based on the given burst-rate and average rate parameters.

The parameters used during our experiment are listed in Table 4.1. Active power for 10 Gbps NIC card is considered as 14.85 W (Chelsio N210 10GbE Server Adapter [3]). According to the IEEE 802.3 standard, the deep sleep and shallow sleep state power consumption is 10% and 40% of the active power consumption respectively (Christensen et al., 2010). So, the sleep state power of TOS and TRS is 5.94W and 1.485W respectively. While applying LPI, the values of Wake up time of 10 Gbps link is $4.8\mu s$ (Christensen

[3] http://www.ieee802.org/802_tutorials/05-July/Tutorial July Nordman.pdf

Table 4.1: Simulation Configuration (parameters used in experiment)

Parameters	**Values**
Active Power	14.85W
Idle State Power	14.85W (Sync Link)
TOS Sleep State Power	5.94W
TOS Sleep to Active Penalty	$0.1\mu s$
TOS Runway	$5\mu s$
TRS Sleep State Power	1.85W
TRS Sleep to Active Penalty	$4.8\mu s$
TRS Runway	$23\mu s$
Backplane Active Power	60W
Backplane Sleep Power	12W
Backplane Sleep to Active Penalty	$200\mu s$
Backplane Runway	$500\mu s$

et al., 2010). The wake-up time for TOS is $0.1\mu s$, since this is a hardware mechanism and does not depend on the transmitter and receiver message exchange. So the penalty is negligible compared to the TRS.

4.7.2 Comparison between different power saving strategies

In our model we have used broadly four different power saving techniques;

(a) No power management at all,

(b) Transmit only sleep, henceforth called TOS for each port

(c) Transmit-Receive sleep, henceforth called TRS for each port and

(d) Additional switch backplane power saving mechanism with port based TOS and TRS discussed earlier.

We first compare the probabilistic routing proposed in section 4.5 and the NS3 provided ECMP routing. In case of NS3 ECMP routing, each packet is routed uniformly randomly over one of the shortest paths to the destination. So, ECMP does not provide

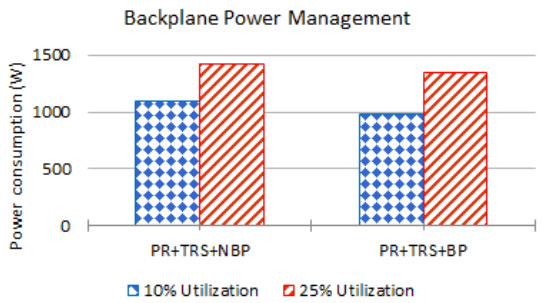


FIGURE 4.12: Power with BP management

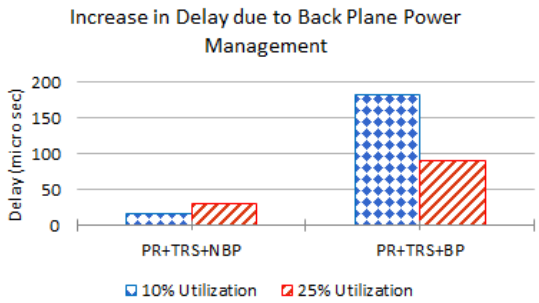


FIGURE 4.13: Delay with BP management

good opportunities for power savings. For the same reason, the delay is smaller as compared to the consolidated routing (less queue depth). The result of the comparison is shown in Fig 4.11. Here the baseline condition for comparison is the NS3 default routing with no power management.

It is clear that energy management techniques like TOS and TRS can save a significant percentage of power, regardless of routing techniques (probabilistic routing and ECMP). The best power saving can be achieved by probabilistic routing with TRS. The power savings with PR + TRS as compared to the baseline is 40%, although they come at the cost of increased delay. The PR + TOS mechanism also provides great power savings, but with a smaller increase in delay (16 μs). While the delay increase is significant in percentage terms, it is important to keep in mind that the end to end network delays are in 100 of μs range, and the impact of an additional 16 μs delay on the performance depends on the nature of the application. While some HPC applications can be very latency-sensitive, most are unlikely to be affected by a small increase in networking delays. The additional delays are a result

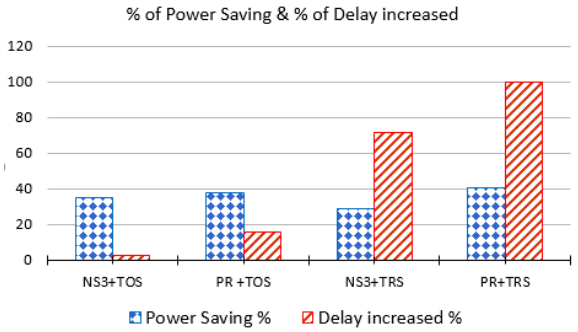


FIGURE 4.11: Comparison of power and delay with different energy management techniques

of additional queuing delay due to traffic consolidation and are unavoidable. Also, TOS provides better power and performance trade-off than TRS in case of ECMP. The percentage of power-saving and delay increase with ns3 ECMP routing and TOS (ns3+TOS) is 35.1 % and 2.8%, whereas with TRS the values are 29 % and 71.9% respectively. The reason behind higher energy consumption and higher delay in TRS is the frequent transition between active and deep sleep states. TOS does not show the same trend, because of smaller wake-up delay and no association of request-acknowledgment message exchange mechanism (Section 4.4).

4.7.3 Effects of backplane management

Fig 4.12-4.13 show effect of backplane (BP) management, on power consumption and network delay. From Fig 4.12 we can observe that with 10% utilization (10 %), TRS+PR can reduce the power consumption by 10.3% using the backplane power management. However, the delay due to BP increases significantly as shown in Fig. 4.13, especially in case of low utilization. This is because the backplane gets much opportunity to switch back to sleep state, while causing more flow delay. In case of higher utilized network, the black plane gets less opportunity to switch into a deep sleep state. So, the delay impact is less severe. From Fig. 4.13 we can observe that with backplane management, the average delay in case of 25% utilization($117.111\mu s$) is less than the average delay associated at 10% utilization ($181.811\mu s$).

4.7.4 Comparison between probabilistic and deterministic routing

We next compare the probabilistic routing (PR) with a deterministic routing(DR) scheme. DR is similar to greedy approach proposed by the Elastic Tree (Heller et al., 2010), except that the priority of the links is changed dynamically. The GC provides congestion hints to the LC and based on that the priority is changed. In DR routing scheme, traffic is forwarded to the highest priority links up to some threshold. When this threshold exceeds,

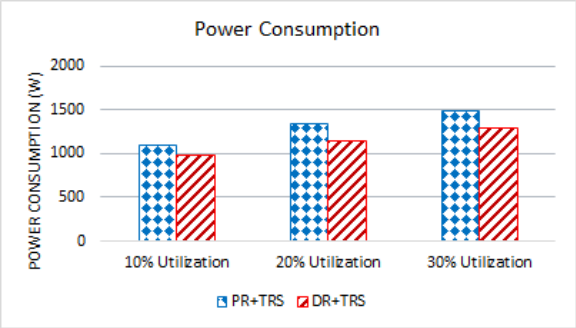


FIGURE 4.14: Power consumption with PR and DR

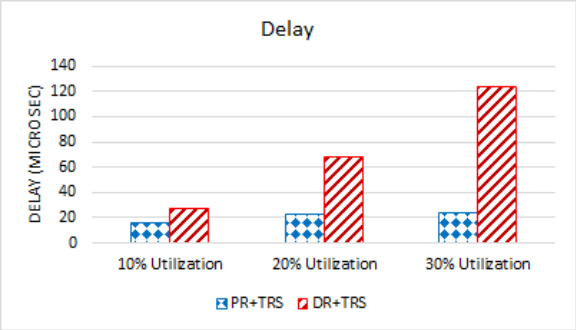


FIGURE 4.15: Delay with PR and DR

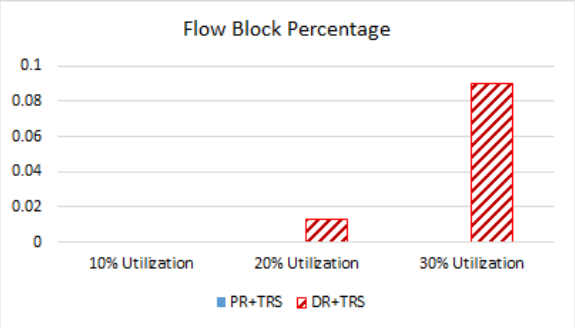


FIGURE 4.16: Flow-Block in PR vs DR

the remaining new flows are forwarded to the next highest priority link, and so on. In our experiment, we assume the maximum threshold of DR to be 0.8.

Fig. 4.14-4.16 show the comparison between PR and DR. From Fig. 4.14 we can observe that the power consumption is reduced significantly in case of DR. This is because of the more aggressive consolidation of the traffic in case of DR. This comes at the cost of higher end-to-end delay and flow blocking as shown in Fig. 4.15-4.16. From Fig. 4.15 we can observe that, when the network load is low (10%), the difference in average delay is not so significant. At the lower utilization, there is no flow blocking (drop) as well (see Fig. 4.16). However, as the network utilization become moderate (25%) or high (40%), both the delay and flow blocking start increasing in case of DR. For example in 25% and 40% utilization, the percentage of flow drop becomes 0.013% and 0.09% of the total flows, respectively. The delay is also higher significantly compared to PR. In a moderate load (20%), the average delay has increased by almost 203%. The delay increase could be

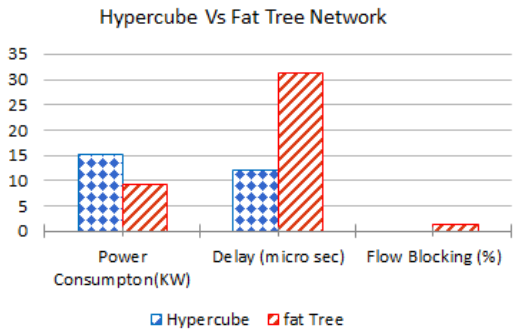


FIGURE 4.17: Comparison of Fat tree and Hyper-cube

problematic at high utilization, but packet drop surely is a problem in data center environments.

4.7.5 Comparison between fat-tree and hypercube network topology

Although fat-tree is the most popular data center network topology, other topologies are also used, particularly in HPC data centers. These include hypercube, toroid, and related topologies. Here we compare hypercube and fat-tree with respect to our probabilistic routing (PR) mechanism. To make the hypercube and fat tree infrastructures comparable, we equate the bisection bandwidth of the two while selecting the k for fat-tree and the dimension d for the hypercube. The bisection bandwidth for fat tree is $(k^3)/8$. The bisection bandwidth for d-dimensional hypercube is 2^{d-1}. Therefore,

$$(k^3/8) = 2^{d-1} \Rightarrow k = \sqrt[3]{2^{d+2}} \tag{4.3}$$

Based on the above equation, we create a 7-dimension hypercube topology with one host per switch, which results in an interconnection network of 128 hosts. We also use k = 8 for fat-tree to accommodate 128 hosts.

Fig. 4.17 compares and power, delay, and blocking probability for the two networks. It is clear that hypercube yields higher power than the fat-tree but also has correspondingly lower delay. It also provides better congestion management and flow blocking. Hypercube consumes 72% more power than the fat tree topology, but with a significantly lower

average delay. The average delay with the fat-tree is almost 263% higher, and the flow blocking is 4.6%. With detailed observation, we found that for fat-tree topology, the congestion is dominant in the link from the core to the aggregate switch, which is the key reason for higher flow blocking.

4.8 Conclusions

In this paper, we extend the basic energy management capabilities of networking devices in the context of the popular NS3 network simulation tool. We also propose a coordinated energy management strategy using local (switch-level) and global controllers. We show that a significant amount of power reduction can be achieved by adopting an intelligent, lightweight, hints-based coordination scheme across the controllers. We apply the mechanism to two popular network topologies - the fat-tree and the hypercube and show fat-tree can provide lower power consumption but at the cost of higher delay. There are further opportunities for further reduction of power without creating network congestion by more explicit congestion notification that we are currently exploring.

4.9 Interaction Between Different Enhanced Modules

Please refer to Fig 4.18

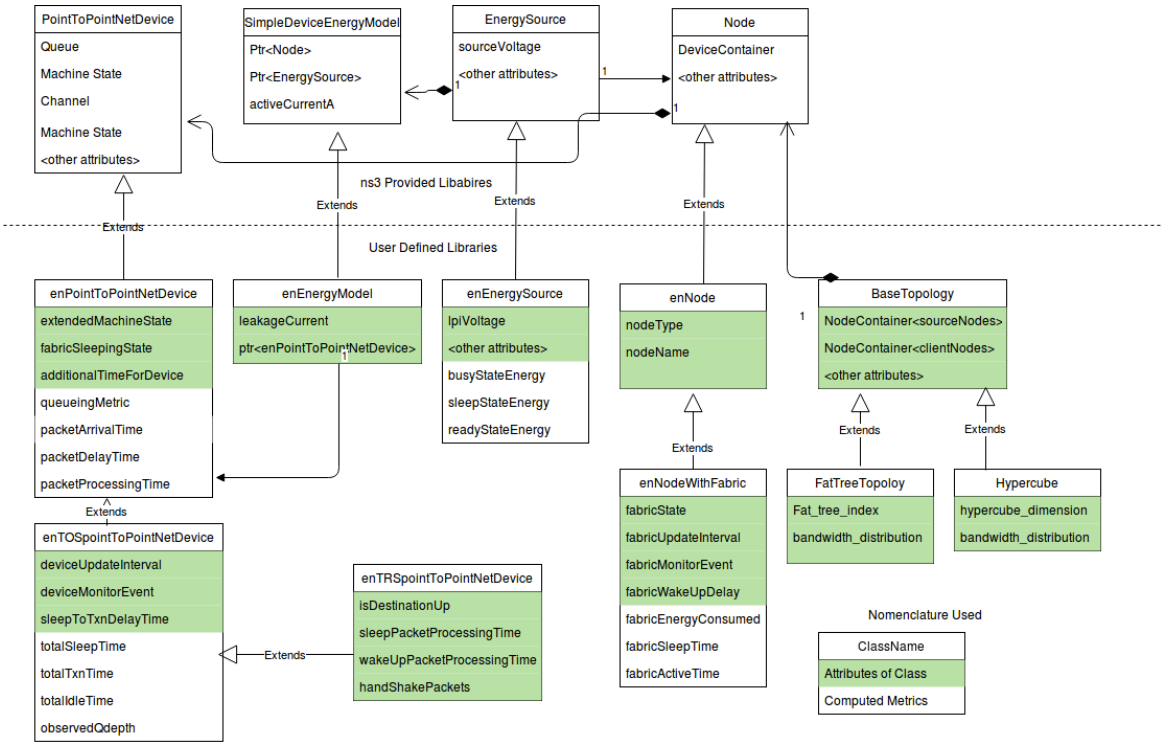


FIGURE 4.18: Interactions of different modules in NS3

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

In this dissertation, we presented opportunities and challenges in developing high-performance data access for modern applications and remote storage systems infrastructure.

We focus on developing a novel KV system designed to leverage the behavior of SSDs to achieve better performance for the application and yet a longer life for the SSDs. It adopts a faster hashing-based architecture for key-indexing and a ”logical band” based sequential datastore to take advantage of SSDs' sequential write speed and to save SSDs' limited endurance. As the hashing destroys the locality across the keys thus to support range query we use a key-only LSM tree structure.

We showed both experimental and analytical evaluation of FlashKey vs. popular LevelDB and RocksDB KV systems and demonstrated O(1) lookup of Get(.) requests while maintaining low RA and reducing WA by up to 12x. Average access and tail latencies are improved by up to 85% and 2x respectively. These improvements persist as database size increases, e.g., with 100M KV pairs and SA aligned at 1.57X, compared to RocksDB, FlashKey achieves 20-60% lower latency, $\sim 9X$ lower WA, and 1.5-2X lower RA for db_bench_R.

We noted many research challenges along the way, which led us to explore new directions. For example, we compare the performance of the FlashKey for two bookend configurations, in one case the HT is stored in-memory whereas in the other the HT is on-disk and for the in-memory HT implementation, we observe the memory requirement of the hash table grows with the database size. Although we achieve better performance than RocksDB or LevelDB even after placing the hashtable on disk, still the performance is poor as compared to the other extreme when the hashtable is in memory. So as an enhancement, we decide to reduce the size of the hashtable significantly by hashing on a key range (by ignoring least significant few bits of the key) rather than on individual keys. Reducing the hashtable size will automatically reduce the memory footprint of the hashtable. We estimate the number of bits to ignore from the key statically such that the space occupied by the HT in memory should not be more than some smaller fraction of the database size. Additionally, sophisticated hashing mechanisms for e.g., Cuckoo hashing can be employed as an extension to the above approach to increase HT space utilization.

For the other bookend implementation, i.e. on-disk HT, we plan to design an in-memory caching policy based on a prediction of the most active regions of the keyspace. Here instead of identifying a single hot key, we try to select a busy key-range, which helps to reduce the unnecessary IOs during caching.

Next, we design a dynamic mechanism to decide whether to migrate or activate an inactive copy or get rid of additional active copies to tackle the network traffic variation inflicted by the emerging high-speed storage devices attached to the data center network. Through extensive simulations, we show that it is possible to achieve near 47% improvement in average delay with purely read traffic. In contrast, the best strategy for the chunks that are majorly written is to migrate them from the congested part of the network to a comparatively lightly loaded section of the network.

Our straight forward design still has many challenges that are to be addressed to achieve performance at scale, which we would like to explore. As an enhancement to

the above-mentioned design, currently, we are integrating the data migration/replication with intelligent routing (through flow consolidation on network links) to design a storage and network-aware consolidation vs congestion trade-off mechanism.

Lastly, we have studied the data center network energy management that involves coordination among three types of controllers: local controller at each switch, a global network controller, and an energy aware user request assignment controller. We studied lightweight mechanisms to coordinate the actions of these controllers through infrequent hinting and showed that such coordination can result in significant (up to 40% in Fat tree with LPI) energy savings without needing VM migrations or shuffling of established flows. As a data center network, we majorly used the fat tree topology, however, we also consider the HPC DC networks (i.e. hypercube) as it consumes a significant portion of the overall energy consumption. While comparing against both the topology we find out that the fat-tree can provide low power consumption but at the cost of higher delay.

We extensively simulated our mechanism in the popular NS3 network simulator and enhanced the basic energy management capabilities for networking devices which was otherwise very rudimentary. Although we have only implemented some basic energy management capabilities, they provide a launching pad for more sophisticated capabilities such as multiple sleep states and more sophisticated algorithms for deciding when to enter or exit a sleep state. In addition to that, we have also implemented different controller structures(e.g. LC, GC) and some simple coordination mechanisms between them.

Another important aspect concerns the end-to-end QoS for storage-intensive applications. A data center environment hosts many different applications with different QoS needs. The NVMe protocol provides several queue-pairs (admin, urgent, high, medium, low, etc.) with different priorities and applies weighted round-robin scheduling among them (except the admin and urgent) to support different QoS classes for local storage access. The network also provides support for QoS at different levels of the network stack (CoS bit at the Layer2 and DSCP at Layer3) for network transfers. (The data center Eth-

ernet (White, 2012) supports additional features as well but is not available universally.) Unfortunately, there is no general recipe for building coordination across the network and the storage to have a consistent end-to-end QoS treatment for remote storage access. Such a coordination will require (1) a well defined but extensible set of QoS classes, and (2) a set of end-to-end transmission tags or "hints" which can be interpreted consistently by each layer along the path. The goal of this project is to design such a mechanism that is lightweight, scalable and can minimize QoS violations for the requests.

BIBLIOGRAPHY

Abad, C. L., Lu, Y., and Campbell, R. H. (2011), "DARE: Adaptive data replication for efficient cluster scheduling," *IEEE ICCC*, pp. 159–168.

Abts et al. (2010), "Energy Proportional Datacenter Networks," *SIGARCH Comput. Archit. News*, 38, 338–347.

Al-Fares et al. (2010), "Hedera: Dynamic Flow Scheduling for Data Center Networks," in *USENIX NSDI*, pp. 19–19.

Ananthanarayanan et al. (2011), "Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters," *EuroSys '11*, pp. 287–300.

Biswas, J., Ray, M., Sondur, S., Pal, A., and Kant, K. (2019), "Coordinated Power Management in Data Center Networks," *SUSCOM*, 22, 1 – 12.

Bjørling, M., González, J., and Bonnet, P. (2017), "LightNVM: The Linux Open-Channel {SSD} Subsystem," in *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pp. 359–374.

Black, D. L. (2018), "NVMe-over-Fabrics: Enabling Next Generation Infrastructure," *Presentation at Flash Memory Summit, Santa Clara, CA*.

Blanquicet et al. (2007), "An initial performance evaluation of rapid PHY selection (RPS) for energy efficient Ethernet," in *IEEE LCN*, pp. 223–225.

Chan, H. H. W., Li, Y., Lee, P. P. C., and Xu, Y. (2018), "HashKV: Enabling Efficient Updates in KV Storage via Hashing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 1007–1019, Boston, MA, USENIX Association.

Christensen, K., Reviriego, P., Nordman, B., Bennett, M., Mostowfi, M., and Maestro, J. A. (2010), "IEEE 802.3 az: the road to energy efficient ethernet," *IEEE Communications Magazine*, 48.

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. (2010), "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pp. 143–154, New York, NY, USA, ACM.

Debnath, B., Sengupta, S., and Li, J. (2010), "FlashStore: high throughput persistent key-value store," *Proceedings of the VLDB Endowment*, 3, 1414–1425.

Debnath, B., Sengupta, S., and Li, J. (2011), "SkimpyStash: RAM space skimpy key-value store on flash-based storage," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pp. 25–36, ACM.

Desnoyers, P. (2012), "Analytic Modeling of SSD Write Performance," in *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR '12, pp. 12:1–12:10, New York, NY, USA, ACM.

Devulapalli, A., Dalessandro, D., and Wyckoff, P. (2008), "Data Structure Consistency Using Atomic Operations in Storage Devices," in *2008 Fifth IEEE International Workshop on Storage Network Architecture and Parallel I/Os*, pp. 65–73.

Dong, S., Callaghan, M., Galanis, L., Borthakur, D., Savor, T., and Strum, M. (2017), "Optimizing Space Amplification in RocksDB," in *CIDR*.

e Silva, R. F. and Carpenter, P. M. (2017), "Energy Efficient Ethernet on MapReduce Clusters: Packet Coalescing To Improve 10GbE Links," *IEEE/ACM Transactions on Networking*.

Eisenman, A., Gardner, D., AbdelRahman, I., Axboe, J., Dong, S., Hazelwood, K., Petersen, C., Cidon, A., and Katti, S. (2018), "Reducing DRAM footprint with NVM in Facebook," in *Proceedings of the Thirteenth EuroSys Conference*, p. 42, ACM.

Erickson et al. (2013), "Using Network Knowledge to Improve Workload Performance in Virtualized Data Centers," Ph.D. thesis, Stanford University.

Estébanez, C., Saez, Y., Recio, G., and Isasi, P. (2014), "Performance of the most common non-cryptographic hash functions," *Software: Practice and Experience*, 44, 681–698.

Gao et al. (2017), "Energy-aware Load Balancing in Heterogeneous Cloud Data Centers," in *ACM ICMSS*, pp. 80–84.

Garey, M. R. and Johnson, D. S. (1990), *Computers and Intractability; A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., New York, NY, USA.

Golander, A., Manole, S., and Korman, Y. (2017), "Persistent memory over fabric (PMoF)," *Proc. of 10th ACM International Systems and Storage Conference (SYSTOR '17)*.

Greenberg et al. (2008), "The Cost of a Cloud: Research Problems in Data Center Networks," *SIGCOMM Comput. Commun. Rev.*, 39, 68–73.

Grun, P., Bates, S., and Davis, R. (2018), "Persistent Memory over Fabrics," *SNIA Persistent Memory summit, San Jose, CA*.

Guo, Y., Pan, M., Gong, Y., and Fang, Y. (2018), "Dynamic Multi-Tenant Coordination for Sustainable Colocation Data Centers," *IEEE Transactions on Cloud Computing*.

Heller et al. (2010), "ElasticTree: Saving Energy in Data Center Networks," in *USENIX Conference on Networked Systems Design and Implementation*.

Henke, C., Schmoll, C., and Zseby, T. (2008), "Empirical Evaluation of Hash Functions for Multipoint Measurements," *SIGCOMM Comput. Commun. Rev.*, 38, 39–50.

Hu, X., Wang, X., Li, Y., Zhou, L., Luo, Y., Ding, C., Jiang, S., and Wang, Z. (2015), "LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pp. 57–69, Berkeley, CA, USA, USENIX Association.

Inc., S. (2019), "TCPDirect Delivers Lowest Possible Latency between the Application and the Network," http://forms.solarflare.com/Media/Default/PDFs/Software/SF-117079-AN-D_Solarflare_TCPDirect_White_Paper.pdf.

Jenkins, R. J. (ONLINE, 1995- 1997), "Hash Functions for Hash Table Lookup," .

Jin, H. et al. (2013), "Joint Host-Network Optimization for Energy-Efficient Data Center Networking," in *IEEE IPDPS*, pp. 623–634.

Jin, Y., Tseng, H., Papakonstantinou, Y., and Swanson, S. (2017), "KAML: A Flexible, High-Performance Key-Value SSD," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 373–384.

Kaiyrakhmet, O., Lee, S., Nam, B., Noh, S. H., and Choi, Y.-r. (2019), "SLM-DB: Single-Level Key-Value Store with Persistent Memory," in *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pp. 191–205.

Kang, J.-U., Hyun, J., and Maeng, H. e. (2014), "The multi-streamed solid-state drive," *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*.

Kannan, S., Bhat, N., Gavrilovska, A., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. (2018), "Redesigning LSMs for nonvolatile memory with NoveLSM," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pp. 993–1005.

Kant, K. (2011), "Multistate power management of communications links," in *Proc. of COMSNET*.

Karlsson, M. and Karamanolis, C. (2004), "Choosing replica placement heuristics for wide-area systems," in *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*

Kim, T., Hong, D., and et.al., S. H. (2019), "Fully Automatic Stream Management for Multi-Streamed SSDs Using Program Contexts," *17th USENIX Conference on File and Storage Technologies (FAST 19)*.

Kobayashi, M., Seetharaman, S., Parulkar, G., Appenzeller, G., Little, J., Van Reijendam, J., Weissmann, P., and McKeown, N. (2014), "Maturing of OpenFlow and software-defined networking through deployments," *Computer Networks*, 61, 151–175.

Kourtis, K., Ioannou, N., and Koltsidas, I. (2019), "Reaping the performance of fast NVM storage with uDepot," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pp. 1–15, Boston, MA, USENIX Association.

Kung, H. T. and Robinson, J. T. (1981), "On Optimistic Methods for Concurrency Control," *ACM Trans. Database Syst.*, 6, 213–226.

Lee, S., Liu, M., Jun, S., Xu, S., Kim, J., et al. (2016), "Application-managed flash," in *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, pp. 339–353.

Li, D. et al. (2014), "Joint power optimization through VM placement and flow scheduling in data centers," in *IEEE IPCCC*, pp. 1–8.

Li, J., Chen, Z., Chen, Z., Xiao, N., Liu, F., and Chen, W. (2017), "KV-FTL: A Novel Key-Value-Based FTL Scheme for Large Scale SSDs," in *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 106–114, IEEE.

Li, P. and Trika, S. N. (2018), "Techniques to Manage Key-Value Storage at a Memory or Storage Device," US Patent App. 15/279,279.

Li, Y., Tian, C., Guo, F., Li, C., and Xu, Y. (2019), "ElasticBF: elastic bloom filter with hotness awareness for boosting read performance in large key-value stores," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pp. 739–752.

Lim, H., Fan, B., Andersen, D. G., and Kaminsky, M. (2011), "SILT: A Memory-efficient, High-performance Key-value Store," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 1–13, New York, NY, USA, ACM.

Liu et al. (2009), "GreenCloud: A New Architecture for Green Data Center," in *ACM ICAC-INDST*, pp. 29–38.

Lu, L., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2016), "WiscKey: Separating Keys from Values in SSD-conscious Storage," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pp. 133–148, Santa Clara, CA, USENIX Association.

Mahadevan, P. et al. (2011), "On energy efficiency for enterprise and data center networks," *IEEE Communications Magazine*, 49, 94–100.

Marmol, L., Sundararaman, S., Talagala, N., and Rangaswami, R. (2015), "{NVMKV}: A Scalable, Lightweight, FTL-aware Key-Value Store," in *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, pp. 207–219.

Meng et al. (2010), "Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement," in *IEEE INFOCOM*, pp. 1154–1162.

Minglani, M., Diehl, J., Cao, X., Li, B., Park, D., Lilja, D. J., and Du, D. H. C. (2017), "Kinetic Action: Performance Analysis of Integrated Key-Value Storage Devices vs. LevelDB Servers," in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 501–510.

Mostowfi, M. (2015), "A simulation study of energy-efficient ethernet with two modes of low-power operation," *IEEE Communications Letters*, 19, 1702–1705.

Mostowfi, M. and Shafie, K. (2016), "An analytical model for the power consumption of Dual-Mode EEE," *Electronics Letters*, 52, 1308–1310.

Murugan, M., Du, D. H., and Kant, K. (2012), "On the Interconnect Energy Efficiency of High End Computing Systems," *Sustainable Computing: Informatics and Systems*.

Nedevschi et al. (2008), "Reducing Network Energy Consumption via Sleeping and Rate-adaptation," in *USENIX NSDI*, pp. 323–336.

O'Neil, P., Cheng, E., Gawlick, D., and O'Neil, E. (1996), "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, 33, 351–385.

Paya et al. (2015), "Energy-aware load balancing and application scaling for the cloud ecosystem," *IEEE Transactions on Cloud Computing*.

Pitchumani, R., Hughes, J., and Miller, E. L. (2015), "SMRDB: key-value data store for shingled magnetic recording disks," in *Proceedings of the 8th ACM International Systems and Storage Conference*, p. 18, ACM.

Pydipaty, R., George, J., Saha, A. K., and Dutta, D. (2018), "The Effect of Non Volatile Memory on a Distributed Storage System," in *IEEE HiPCW*, pp. 11–17.

Raghavendra, R., Ranganathan, P., Talwar, V., Wang, Z., and Zhu, X. (2008), "No "Power" Struggles: Coordinated Multi-level Power Management for the Data Center," *SIGARCH Comput. Archit. News*.

Raju, P., Kadekodi, R., Chidambaram, V., and Abraham, I. (2017), "PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China.

Ray et al. (2018), "Opportunistic Power Savings with Coordinated Control in Data Center Networks," in *ACM ICDCN*, pp. 48:1–48:10.

Ray, M., Biswas, J., Pal, A., and Kant, K. (2019), "Adaptive Data Center Network Traffic Management for Distributed High Speed Storage," *LCN*.

Ray, M., Li, P., Kant, K., and Trika, S. (2019), "FlashKey:A High-Performance Flash Friendly Key-Value Store," In publication.

Ren, K., Zheng, Q., Arulraj, J., and Gibson, G. (2017), "SlimDB: A space-efficient key-value storage engine for semi-sorted data," *Proceedings of the VLDB Endowment*, 10, 2037–2048.

Reviriego, P., Hernadez, J., Larrabeiti, D., and Maestro, J. A. (2010), "Burst Transmission for Energy-Efficient Ethernet," *IEEE Internet Computing*.

Sears, R. and Ramakrishnan, R. (2012), "bLSM: A General Purpose Log Structured Merge Tree," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pp. 217–228, New York, NY, USA, ACM.

Sergio et al. (2012), "Optimal configuration of Energy-Efficient Ethernet," *Computer Networks*.

Shen, Z., Chen, F., Jia, Y., and Shao, Z. (2017), "DIDACache: A deep integration of device and application for flash based key-value caching," in *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pp. 391–405.

Sivasubramanian, S., Alonso, G., Pierre, G., and van Steen, M. (2005), "GlobeDB: Autonomic Data Replication for Web Applications," in *Proceedings of the 14th International Conference on World Wide Web*.

Sohan, R., Rice, A., Andrew, W. M., and Mansley, K. (2010), "Characterizing 10 Gbps network interface energy consumption," in *Local Computer Networks (LCN), 2010 IEEE 35th Conference on*, pp. 268–271, IEEE.

Sondur, S., Ray, M., Biswas, J., and Kant, K. (2017), "Implementing data center network energy management capabilities in NS3," in *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*, pp. 1–8.

Tapparello, C., Ayatollahi, H., and Heinzelman, W. (2014), "Energy Harvesting Framework for Network Simulator 3 (Ns-3)," in *Proceedings of the 2Nd International Workshop on Energy Neutral Sensing Systems*, ENSsys '14, pp. 37–42, New York, NY, USA, ACM.

Thaenchaikun, C. et al. (2015), "Augmenting the energy-saving impact of IEEE 802.3az via the control plane," in *IEEE ICCW*, pp. 2843–2849.

Travostino et al. (2006), "Seamless Live Migration of Virtual Machines over the MAN/WAN," *Future Gener. Comput. Syst.*, 22, 901–907.

Trika, S. N., Park, D., Li, P., Corrado, F. R., and Dickinson, R. A. (2019), "Data management system employing a hash-based and tree-based key-value data structure," US Patent App. 15/856,686.

Verma et al. (2008), "pMapper: power and migration cost aware application placement in virtualized systems," in *ACM/IFIP/USENIX International Conference on Middleware*, pp. 243–264.

Verma, A., Koller, R., Useche, L., and Rangaswami, R. (2010), "SRCMap: Energy Proportional Storage Using Dynamic Consolidation," in *8th USENIX FAST*, pp. 20–20, Berkeley, CA, USA.

Wang, P., Sun, G., Jiang, S., Ouyang, J., Lin, S., Zhang, C., and Cong, J. (2014), "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*, p. 16, ACM.

Wang, T., Wang, J., Nguyen, S. N., Yang, Z., Mi, N., and Sheng, B. (2017), "EA2S2: An Efficient Application-Aware Storage System for Big Data Processing in Heterogeneous Clusters," in *2017 26th (ICCCN)*, pp. 1–9.

Wang, X. et al. (2012), "CARPO: Correlation-aware power optimization in data center networks," in *IEEE INFOCOM*, pp. 1125–1133.

White, J. (2012), "Technical Overview of Data Center Networks," *SNIA Education*.

Wu, H., Nabar, S., and Poovendran, R. (2011), "An energy framework for the network simulator 3," in *SimuTools*.

Wu, K., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. (2019), "Towards an Unwritten Contract of Intel Optane SSD," in *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19). USENIX Association, Renton, WA*.

Xie et al. (2010), "Improving MapReduce performance through data placement in heterogeneous Hadoop clusters," *IPDPSW 2010 IEEE International Symposium on*, 9, 29–42.

Xiong, J., Li, J., Tang, R., and Hu, Y. (2008), "Improving data availability for a cluster file system through replication," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–8.

Yang, Z. et al. (2017), "AutoTiering: Automatic data placement manager in multi-tier all-flash datacenter," in *2017 IEEE 36th (IPCCC)*, pp. 1–8.

Yao, T., Wan, J., Huang, P., He, X., Wu, F., and Xie, C. (2017), "Building efficient key-value stores via a lightweight compaction tree," *ACM Transactions on Storage (TOS)*, 13, 29.

Yao, T., Wan, J., Huang, P., Zhang, Y., Liu, Z., Xie, C., and He, X. (2019), "GearDB: A GC-free Key-Value Store on HM-SMR Drives with Gear Compaction," in *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pp. 159–171.

Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., and Stoica, I. (2010), "Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling," in *ACM EuroSys*, pp. 265–278.

Zamanian, E., Binnig, C., and Salama, A. (2015), "Locality-aware Partitioning in Parallel Database Systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 17–30.

Zhang, J., Lu, Y., Shu, J., and Qin, X. (2017), "FlashKV: Accelerating KV performance with open-channel SSDs," *ACM Transactions on Embedded Computing Systems (TECS)*, 16, 139.

Zhang, J., Kwon, M., and Gouk, D. e. (2018), "Flashshare: punching through server storage stack from kernel to firmware for ultra-low latency SSDs," *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, p. 477– 492.

Zheng, K. et al. (2014), "Joint power optimization of data center network and servers with correlation analysis," in *IEEE INFOCOM*, pp. 2598–2606.

Zheng, S., Hoseinzadeh, M., and Swanson, S. (2019), "Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks," in *17th USENIX (FAST 19)*.