IMPROVING CLOUD LOAD-BALANCING:

FROM BIG DATA TO BIG WEB SERVICES

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Rohan Gandhi

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2015

Purdue University

West Lafayette, Indiana

TABLE OF CONTENTS

Page

LIST OF FIGURES v

ABSTRACT vii

1 Introduction 1

1.1 Online services composition 1

1.2 Load balancing in big data analytics frameworks 1

1.3 Load balancing traffic 2

2 Pikachu: How to Rebalance Load in Optimizing MapReduce On Heterogeneous Clusters 4

2.1 Introduction 4

2.2 Background 6

2.3 Impact of Heterogeneity 7

2.3.1 Setup 7

2.3.2 Impact of Heterogeneous Nodes 8

2.4 Dynamic Load Rebalancing 9

2.4.1 General Approach 9

2.4.2 Design Space 10

2.4.3 Design Refinement 12

2.5 Implementation 14

2.6 Evaluation 16

2.7 Related Work 19

3 Duet: Cloud Scale Load Balancing with Hardware and Software 21

3.1 Introduction 21

3.2 Background and Motivation 24

3.2.1 Ananta Software Load Balancer 25

Page

3.2.2 Limitations of Software Load Balancer 26
3.3 Duet: Core ideas 27
3.3.1 HMux 28
3.3.2 Partitioning 30
3.3.3 DUET: HMux + SMux 31
3.4 VIP Assignment Algorithm 33
3.4.1 VIP Assignment 35
3.4.2 VIP Migration 36
3.5 Practical Issues 39
3.5.1 Failure Recovery 39
3.5.2 Other Functionalities 40
3.6 Implementation 43
3.7 Testbed Experiments 45
3.7.1 HMux Capacity 45
3.7.2 HMux Failure Mitigation 47
3.7.3 VIP Migration 48
3.8 Evaluation 51
3.8.1 Simulation Setup 51
3.8.2 SMux Reduction 52
3.8.3 Latency vs. SMuxes 53
3.8.4 Duet vs. Random 54
3.8.5 Impact of Failure 55
3.8.6 VIP Migration 56
3.9 Discussion 59
3.10 Related Work 60
3.11 Conclusion 61
4 Future work 63
4.1 Reducing network bandwidth overhead of load balancer 63

Page

4.2 Cloud scale layer 7 load balancing 63
REFERENCES 65

LIST OF FIGURES

Figure Page

2.1 Four stages and their concurrency in MapReduce job execution. 6

2.2 Job completion time breakdown (normalized to total time) for Wordcount and Sort. 8

2.3 Ratio of progress rates of map and reduce tasks on fast and slow nodes. 11

2.4 Job execution time and breakdown of Wordcount under D_3 (P=2). . . 13

2.5 Shuffling progress and CPU utilization by each reducer on fast and slow nodes in D_3 (calculated P=2). 13

2.6 Shuffling progress and CPU utilization by each reducer on fast and slow nodes under D_3'. 15

2.7 Speedup of Tarazu and Pikachu over Hadoop, under Config-1. 17

2.8 Job completion time breakdown on fast and slow nodes (Normalized to Tarazu). 18

2.9 Speedup of Tarazu and Pikachu over Hadoop on the 60-node EC2 cluster on 2 configurations. 19

3.1 Performance of software Mux. 24

3.2 Storing VIP-DIP mapping on a switch. 28

3.3 **Duet architecture**: VIPs are partitioned across different HMuxes — VIP_1 and VIP_2 are assigned to HMux C_2 and A_6. Additionally, SMuxes act as backstop for all the VIPs. Every server (apart from SMuxes) runs host-agent that decapsulates the packets and forwards to the DIP. Links marked with solid lines carry VIP traffic, and links with dotted lines carry DIP traffic. 31

3.4 Memory deadlock problem during VIP migration. VIPs V1 and V2 both occupy 60% of switch memory each. The goal of migration is to migrate the VIPs from assignment in (a) to (b); Pikachu eliminates this problem by migrating VIPs through SMuxes, as shown in (c). 37

Figure Page

3.5 When the VIP assignment changes from ToR T_2 to T_3, only the links inside container-2 are affected. As a result, we can first select best ToR in a container based on the links within container, and then scan over all containers and remaining Core and Agg switches. 38

3.6 Load balancing in virtualized clusters. 39

3.7 Large fanout support. 41

3.8 Port-based load balancing. 43

3.9 Components in Pikachu implementation. 44

3.10 Our testbed. FatTree with two containers of two Agg and two ToR Switches each, connected by two Core switches. 45

3.11 HMux has higher capacity. 46

3.12 VIP availability during failure. 47

3.13 VIP availability during migration. 48

3.14 Latency breakdown. 50

3.15 Traffic and DIP distribution. 51

3.16 Number of SMuxes used in Duet and Ananta. 52

3.17 Latency (microseconds) vs. number of SMuxes in Ananta and Pikachu. 54

3.18 Number of SMuxes used by Duet and Random. 55

3.19 Impact of failures on max. link utilization. 55

3.20 Effectiveness of different migration algorithms. 57

ABSTRACT

Gandhi, Rohan PhD, Purdue University, May 2015. Improving Cloud Load-balancing: From Big Data to Big Web Services . Major Professor: Y. Charlie Hu.

With the rapid rise of cloud computing, scale-out online services running on large clusters are becoming the norm. The key function that enables scale-out services is the load balancer. Load balancer splits the computation and data in such online services, and is critical to the performance and reliability of the online services running in the cloud. My thesis focuses on the problems in designing load balancer for such high demanding online services.

First part of my thesis focuses on load balancing in large scale data processing in heterogeneous clusters that consist of the servers with different hardware. For power, cost, and pricing reasons, datacenters are evolving towards heterogeneous hardware. However, MapReduce implementations were originally designed for homogeneous clusters and performed poorly on heterogeneous clusters. In this work, we revisit the key design challenge in this important optimization for MapReduce on heterogeneous clusters and make three contributions. (1) We show that prior work estimates the target load distribution too early into MapReduce job execution, which results in the rebalanced load far from the optimal. (2) We articulate the delicate tradeoff between the estimation accuracy versus wasted work from delayed load adjustment, and propose a load rebalancing scheme that strikes a balance between the tradeoff. (3) We implement our design in the PIKACHU task scheduler, which outperforms Hadoop by up to 42%.

Second part of my thesis focuses on load balancing traffic and user requests received by the online services in the cloud. As the demand for cloud services grows, expensive and hard-to-scale dedicated hardware load balancers are being replaced

with software load balancers that scale using a distributed data plane that runs on commodity servers. Software load balancers offer low cost, high availability and high flexibility, but suffer high latency and low capacity per load balancer, making them less than ideal for applications that demand either high throughput, or low latency or both. In this work, we present DUET, which offers all the benefits of software load balancer, along with low latency and high availability at next to no cost. We do this by exploiting a hitherto overlooked resource in the data center networks the switches themselves. We show how to embed the load balancing functionality into existing hardware switches, thereby achieving organic scalability at no extra cost. For flexibility and high availability, DUET seamlessly integrates the switch-based load balancer with a small deployment of software load balancer. We enumerate and solve several architectural and algorithmic challenges involved in building such a hybrid load balancer. Our evaluation using prototype implementation and simulations shows that DUET provides 10x more capacity than a software load balancer, at a fraction of a cost, while reducing latency by a factor of 10 or more.

1. INTRODUCTION

With the rapid rise of the cloud computing, scale-out online services running in cloud (such as Amazon AWS, Microsoft Azure) are becoming the norm. Each online service consists of a set of servers that work together as a single entity. The key function that enables scale-out services is the load balancer. Load balancer splits the computation, traffic and data in such online services, and is critical to the performance and reliability of the online services running in the cloud. This thesis focuses on the problems in designing load balancer for high demanding online services.

1.1 Online services composition

Online services consist of a large number of components such as front-end servers, storage, big data analytics that interact with each other in implementing online service logic. Typically, each online service consist of two main pillars – (1) Big data analytics: that process large amount of data to capture the insights from the data and answer user requests. (2) Front-end servers: These servers receive the requests either from the users or from other servers within the same online service, and respond to those requests.

1.2 Load balancing in big data analytics frameworks

An important component of the online services is the big data analytics applications. These big data applications are powered by scale-out frameworks such as Hadoop implementing MapReduce that run on hundreds or thousands of the servers. The key goal of such frameworks is to load balance the data and computation across the servers to reduce the completion time of the big data analytics job. The first part

of my thesis focuses on how to load balance the data and computation to speed up the big data analytics jobs.

The key challenge in reducing the job completion time is on how to load balance the data across the servers so that all the servers finish their computation at the same time. This challenge is further compounded by the fact that (1) not all servers use the same hardware and (2) the heterogeneity in the hardware has complex relationship with the processing time.

In chapter 2, we first address the problem of balancing the computation to reduce the job completion time for the big data jobs. We first show the inefficiencies in terms of job completion time in using default Hadoop in heterogeneous cluster. Next, we highlight the key causes for such inefficiencies and characterize their individual impact. Lastly, we propose Pikachu that addresses these inefficiencies, and brings the job completion time within 95% of ideal job completion time.

1.3 Load balancing traffic

Another important component of the online services is the traffic load balancer. Services in the data center scale by running on multiple servers, each with an individual direct IP (DIP). The service exposes one or more virtual IP addresses (VIP) outside the service boundary. The load balancer receives the traffic destined for the VIP, splits it among the DIPs, and routes it to the individual DIPs. Because of the varsity of the load balancer, many cloud service providers (such as Amazon AWS, Microsoft Azure) provide load balancer as a service. A high performance load balancer is one of the most important components of a cloud service infrastructure. The load balancer thus touches every packet coming into the data center from the Internet, as well as a significant fraction of all intra-DC traffic. This traffic volume induces heavy load on both data plane and control plane of the load balancer [1]. The performance and reliability of the load balancer directly impact the latency, throughput and the availability of the cloud services hosted in the DC.

In chapter 3, we study the problem of load balancing in splitting the user traffic within online services running in the cloud. We first show the shortcomings of using state-of-the-art software load balancers – software load balancer incur high latency and high cost to the cloud provider. We address these shortcoming by designing Duet – a network switch based hardware load balancer backed by small number of software load balancer instances. The key design challenge in designing switch-based L4 load balancer is the limited memory available on individual switches. Duet uses scale-out design, where the load balancer VIP-to-DIP mapping is partitioned across switches backed by small number of software load balancer instances for high resiliency.

We show that Duet reduces latency incurred by the load balancer by almost 1000x and cost by 10x while providing the high reliability.

Lastly, chapter 5 proposes the future work to be conducted as a part of this thesis.

2. PIKACHU: HOW TO REBALANCE LOAD IN OPTIMIZING MAPREDUCE ON HETEROGENEOUS CLUSTERS

2.1 Introduction

For power, cost, and pricing reasons, datacenters have evolved towards heterogeneous hardware. For example, different hardware generations exist in Amazon EC2 [2] due to phased hardware upgrades over the years. Heterogeneity also arises due to other factors including special hardware such as GPUs, unequal creation of instances, and background load variation [3–5].

MapReduce, a high-level programming model for data-intensive applications [6], has been widely adopted in cloud datacenters such as Google, Yahoo, Microsoft, and Facebook [7–10], to power a significant portion of applications. However, the numerous MapReduce implementations have been designed and optimized for homogeneous clusters. A recent study [11] has shown that contemporary MapReduce implementations can perform extremely poorly on heterogeneous clusters.

The same study characterized how heterogeneous hardware, *i.e.,* mix of fast and slow nodes, adversely affects the performance of MapReduce frameworks into two primary effects. (1) Map-side effect: The built-in load balance of map tasks leads to faster nodes stealing tasks from slow nodes, which can greatly increase the network load which in turn can coincide with and slow down the subsequent network-intensive shuffle phase. (2) Reduce-side effect: MapReduce implementations assume homogeneous nodes and distribute the keys equally among reduce tasks. Such distribution leads to disparate progress on fast and slow nodes in heterogeneous clusters, and contributes to prolonged job completion time.

In [11], the authors proposed Tarazu, a suite of optimizations for heterogeneous clusters. For map-side effect, it adaptively allows task stealing from slow nodes and interleaving map tasks with shuffling on fast nodes. For reduce-side effect, it explores the natural solution, *i.e.,* rebalancing load between reducers running on fast and slow nodes. In particular, it estimates the target load split between fast and slow nodes, *i.e.,* key range partitions, right before the start of the reduce tasks, based on the relative progress rates of map tasks running on the fast and slow nodes so far. Evaluation results in [11] however show the simple load rebalancing scheme is only mildly effective, and can even degrade job performance from inaccurate key distribution estimation.

In this paper, we revisit the key design challenge in this important optimization for MapReduce on heterogeneous clusters: load rebalancing among reduce tasks to even out their completion time. We make three concrete contributions. First, we show that the relative progress rates of map tasks on fast and slow nodes often do not give a good indication of the relative progress rates of reduce tasks on heterogeneous nodes due to different resource requirement, and hence estimating the target reducer load distribution before reduce tasks start can result in the adjusted load being far from well-balanced.

Second, we explore the design space and articulate the tradeoff between the estimation accuracy versus wasted work from delayed load adjustment, and propose a load rebalancing scheme that strikes a balance between the two factors. We show an estimator that simply peeks into the initial relative progress rates of reduce tasks can still incur estimator error, because reducers on fast and slow nodes can have different room for increased resource utilization. Our final design captures this additional intricacy using observed reducer CPU utilization on fast and slow nodes to accurately estimate the target load split.

Finally, we implement our new load rebalancing scheme in the Pikachu task scheduler, and experimentally show it substantially outperforms Tarazu and Hadoop,

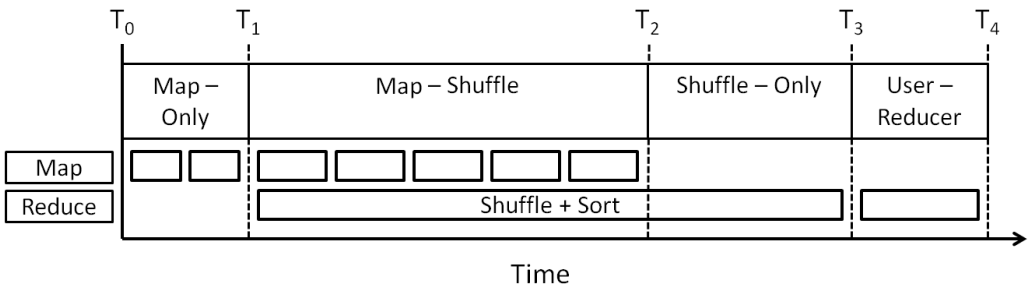


Fig. 2.1. Four stages and their concurrency in MapReduce job execution.

reducing the job completion time by up to 23% and 42%, respectively, for a diverse set of benchmarks and cluster configurations.

2.2 Background

The execution of a MapReduce job is broken down by the runtime system into many Map tasks and Reduce tasks (called reducers hereafter) running in parallel on different nodes of the cluster. A reducer consists of three types of subtasks: (1) shuffle, (2) sort, and (3) user-defined reduce function. Every node in the cluster has a fixed number of map and reduce slots, and the scheduler assigns a task whenever a slot frees up.

Four stages of MapReduce execution in Hadoop. To help illustrate of the impact of heterogeneous hardware on MapReduce performance, we divide the execution of a MapReduce job in Hadoop into four distinct stages in the time dimension, as shown in Figure 2.1. (1) **Map-Only**: In this stage, only map tasks are running across the nodes in the cluster; the reducer is yet to begin. (2) **Map-Shuffle**: This stage starts when the reduce tasks start to run (T_1 in Figure 2.1). The start time for reducers is configurable, but is typically set to be when the first wave of map tasks is finished, *i.e.,* at least one map task is finished on all nodes. In this stage, the reduce task continuously performs staggered shuffle and sort [1] (or simply shuffle-sort hereafter) to digest the output of each wave of map tasks. Effective, map tasks and

[1]They do not have to be strictly inter-leaved as each sort task can begin before the corresponding shuffle task is over, when sufficient amount of data has been shuffled.

shuffle-sort are running concurrently on all nodes. (3) **Shuffle-Only**: This stage begins when all map tasks are finished (time T_2) but the shuffle-sort phases of the reducers are yet to be finished. In this stage, only the shuffle and sort tasks are running concurrently. (4) **User-Reducer**: This stage begins when all the data have been shuffled and sorted, and only the user-defined reducer function executes. Finally, the job is said to be finished when the user-defined reducer function is finished on all the nodes.

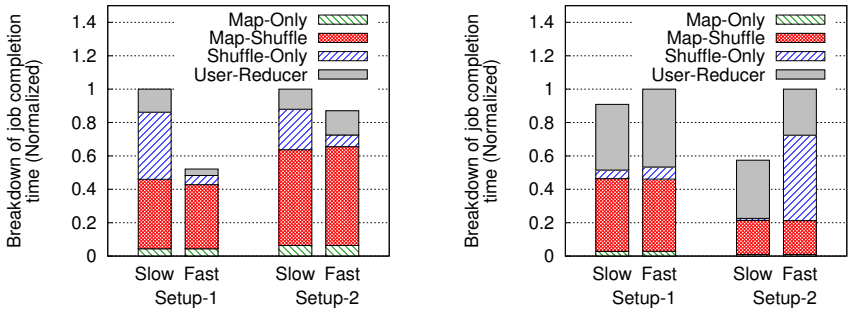
2.3 Impact of Heterogeneity

The scheduler of MapReduce implementations, *e.g.,* Hadoop, however, does not consider heterogeneity, which results in poor application performance on heterogeneous clusters. Using testbed measurement, we dissect the impact of hardware heterogeneity on the four stages of MapReduce execution.

2.3.1 Setup

Our heterogeneous cluster consists of 5 Xeon (slow) nodes and 2 Opteron (fast) nodes, all of which are connected to a 1Gbps switch. Each Opteron node has 8 cores and 16GB RAM, and each Xeon node has 2 cores and 2GB RAM. We run Hadoop Wordcount (CPU-intensive) and Sort (IO-intensive) benchmarks and analyze their job completion time. The total job size consists of 40GB input data, *i.e.,* 680 map tasks each with 64 MB data size.

We use two configurations in our experiments. Both have 8 and 2 map slots each on fast and slow nodes, proportional to their numbers of cores, as in Tarazu [11]. They differ in reduce slots per node. Config-1 uses 2 reduce slots on both fast and slow nodes, as in Tarazu, while Config-2 uses 4 and 1 reduce slots on fast and slow nodes, *i.e.,* proportional to their numbers of cores.



(a) Wordcount (b) Sort

Fig. 2.2. Job completion time breakdown (normalized to total time) for Wordcount and Sort.

2.3.2 Impact of Heterogeneous Nodes

Figure 2.2 shows the execution time and their breakdown into the four stages discussed in §2.2, of the two benchmarks on the fast and slow nodes, respectively, under the two configurations. We make the following observations.

(1) **Map-Only:** We observe the duration of Map-Only stage is short. For Wordcount, this stage ends when 1 wave of map tasks is over on the slow nodes, and 2 waves of map tasks are completed on the fast nodes.

(2) **Map-Shuffle:** The Map-Shuffle stage always finishes at almost the same time on the fast and slow nodes. This is due to the inherent load balancing feature of the task scheduler: whenever a Map slot is freed on a node, a new map task is scheduled. For example, in Config-1, each fast node processes far more map tasks (41%) than slow nodes (3.6%) for Wordcount. This imbalanced map task processing has two consequences. First, after the fast nodes finish map tasks on local data first (a locality feature of the Hadoop scheduler), they will execute remote map tasks (stealing data from the slow nodes). In Config-1, about 9% of the total map tasks (of the whole job) executed by a fast node in Wordcount are remote map tasks. Such remote map tasks generate extra network traffic from fetching data remotely. Second, since a fast node performed more map tasks, it will shuffle much more intermediate

data out to other nodes than slow nodes. In Figure 2.2(a) Config-1, each fast node in total shuffles out 7 times more data than each slow node.

(3) **Shuffle-Only:** Figure 2.2 shows the duration of the Shuffle-Only stage can vary significantly on fast and slow nodes. The gap results from the difference between shuffle-sort speeds on fast and slow nodes, which results in different total shuffle-sort durations – the shuffle-only stage is the leftover shuffle-sort beyond the time all map tasks are finished. Figure 2.2(a) shows the stage is 7.4 times shorter on fast nodes than on slow nodes for Wordcount, but completes at about the same on fast and slows for Sort, under Config-1.

(4) **User-Reducer:** Since the default scheduler equally partitions the key range across reducers, each reducer processes equal amount of data in the user-reducer phase. The execution time, however, can differ among different nodes due to the difference in their processing speed. Figure 2.2(a) shows under Config-1, this stage is 3.51 times slower on slow nodes than on fast nodes for Wordcount but finishes at about the same time for Sort, whereas under under Config-2, it finishes at about the same time for Wordcount, but is 1.26 times slower on fast nodes than on slow nodes.

(5) **Diversity of impact:** Overall, Figure 2.2 shows the impact of hardware heterogeneity on different stages differ for different applications under different configurations, suggesting it cannot be easily solved by any static map/reduce slot configuration.

2.4 Dynamic Load Rebalancing

We revisit the key design challenge in dynamic load rebalancing, a potentially effective technique to optimize MapReduce execution on heterogeneous clusters.

2.4.1 General Approach

The idea of load rebalancing is straight-forward: faster reducer gets more data; the task scheduler calculates the key range partition for fast and slow nodes that results in the reducers on them finishing at about the same time.

One can potentially derive an analytic model to capture the effects of all contributing factors to the reducer completion time on fast and slow nodes [11]. However, the extensive information needed in such a model are application and hardware specific, which requires extensive profiling and makes it infeasible to use in practice [11]. This motivates the practical approach of *dynamic load rebalancing*, *i.e.,* the task scheduler starts with the default even split policy, estimates the key range partitions for fast and slow nodes at runtime, and instructs the reduce tasks to carry their new workload accordingly.

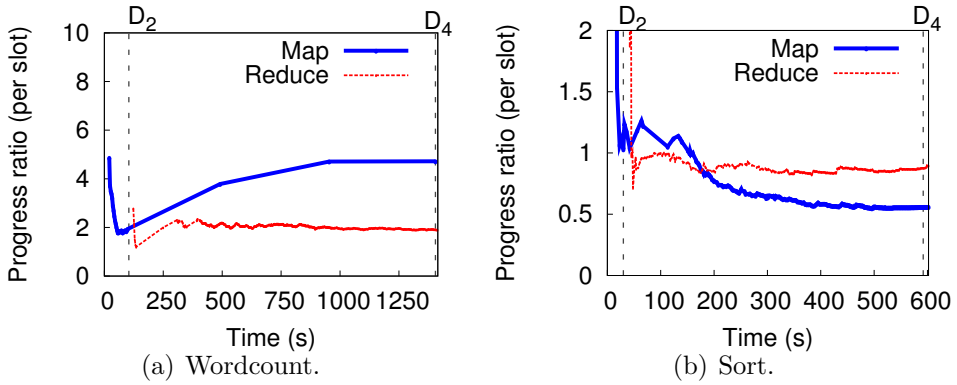
Dynamic load rebalancing faces two conflicting challenges. (1) The new load split estimate needs to be *accurate*, to maximally even out the reducer completion time on fast and slow nodes. (2) The new load split estimate needs to be calculated as *early* as possible, to minimize the wasted (and hence extra) data movement and processing. In particular, when the assignment of a bin changes from one reduce task to another, the data associated with the bin needs to be reshuffled to the newly assigned reduce task and re-processed thereafter. Conceptually, the two challenges are at odds with each other: the longer the task scheduler waits to estimate the new load split, the more information it can collect and estimate the split more accurately, but also the more wasted (and hence extra) data movement and processing due to the default even load split before rebalancing takes place.

2.4.2 Design Space

We define the target ratio of key partition sizes assigned to each reducer on a fast node to each reducer on a slow node as the *partition ratio* — P. The challenge is to calculate P accurately to balance the completion time of the reducers. We now explore the design space for when and how the task scheduler should attempt to estimate P.

D_1: At start of the Map-Only stage (T_0)[2]. At the beginning of job execution, since no information is available about the progress rates of map and reduce tasks, P

[2] T_0 to T_4 are marked in Figure 2.1.



(a) Wordcount. (b) Sort.

Fig. 2.3. Ratio of progress rates of map and reduce tasks on fast and slow nodes.

can only be set to the default value 1. This is the default even-split policy which is oblivious to cluster heterogeneity. [3]

D_2: At start of the Map-Shuffle stage (T_1). At T_1, since the reduce tasks have not started, P can only be estimated using the relative progress rates of map tasks (so far) on fast and slow nodes, *i.e.,* $P = \frac{S_{fa,m}}{S_{sl,m}}$, where $S_{fa,m}$ and $S_{sl,m}$ are the progress rates for map tasks on fast and slow nodes. This method is used in Tarazu [11].

The main advantage of this method is that, since shuffle-sort has not started, there is no need to reshuffle any data after the load rebalancing act. However, it can give a poor estimate of P. Map and reduce tasks are known to have very different resource requirements, *e.g.,* a map task is CPU-intensive in the first half and I/O-intensive in the second half, whereas shuffle-sort has interleaved network-intensive and CPU- and I/O-intensive phases. As a result, the relative speed of map tasks can be a poor approximation to the relative speed of shuffle-sort. Figure 2.3(b) shows for Sort, the ratio of map task progress rates at T_1 is 1.25 , which would be a poor approximation to the steady-state ratio of shuffle-sort progress rates 0.7.

D_3: during the Map-shuffle stage (between T_1 and T_2). Between T_1 and T_2, P can be estimated as $\frac{S_{fa}}{S_{sl}}$ where S_{fa} and S_{sl} denote the actual progress rates of Shuffle-Sort so far. The ratio, however, may not be a good approximation to the

[3] Although conceptually P can be set to a biased value based on the prior knowledge about the node heterogeneity, picking a suitable value is hard as the progress rate varies significantly for different phases and jobs on the same node.

ratio of progress rates of user-reducers on fast and slow nodes, which perform different operations from shuffle-sort subtasks.

D_3 achieves better estimate of P at the cost of the penalty associated with adjusting load in the middle of the Shuffle-Sort stage in two ways: (1) Re-Shuffling: the reducer on a fast node needs to shuffle in some data already shuffled to slow nodes; (2) Dropping data: the reducer on a slow node needs to drop some data shuffled in and sorted under even split.

To strike a balance between accuracy and penalty, the progress rates and their ratio of reducers on fast and slow nodes can be measured once they are observed to stabilize, typically after shuffling in one wave of map tasks.

D_4: during the Shuffle-Only stage (after T_2). Estimation of P can be further delayed till the shuffle-only or even user-reducer stage has started. At this point, the relative progress rates of these stages on fast and slow nodes can be measured accurately; but this design choice suffers a major disadvantage in terms of reshuffling costs as slow and fast nodes have fetched substantial amount of data, ranging from 30-100%. Thus, rebalancing load at this stage would result in too high data reshuffling penalty which is likely to erase the gain from rebalancing the data. We do not consider this option further.

2.4.3 Design Refinement

We implemented D_3 (details in §3.5, the calculated P=2 from Figure 2.3(a)) and reran Wordcount. The new execution time breakdowns, shown in Figure 2.4, show that the Shuffle-Only stage still finishes at different time on fast and slow nodes! To understand the reason, we plot the (total) map task completion rate and the rate map tasks are shuffled in by fast nodes and slow nodes for Wordcount in Figure 2.5. We make two observations. (1) The fast node is able to match the rate at which map tasks are completed, which shows that fast node is able to get enough CPU and network resources to fetch the map outputs. (2) The shuffle on slow nodes never

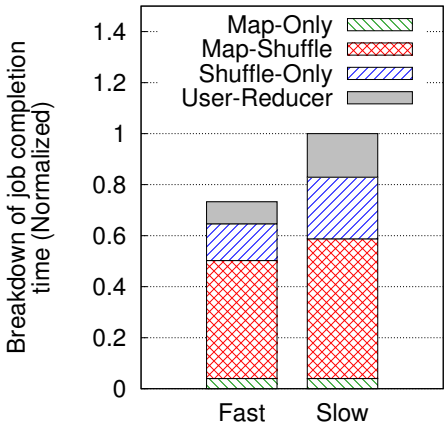


Fig. 2.4. Job execution time and breakdown of Wordcount under D_3 (P=2).

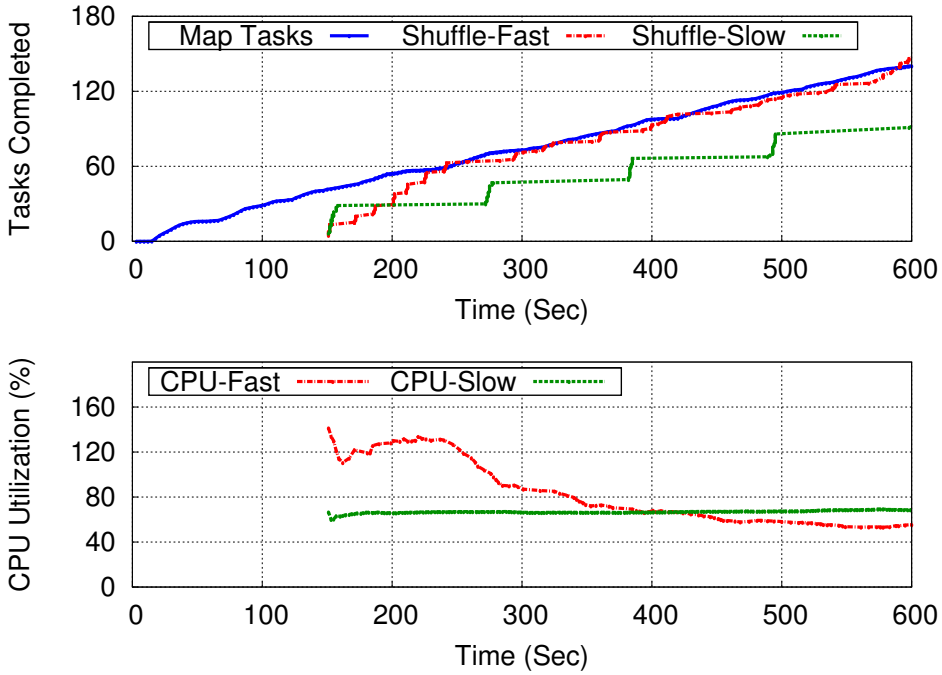


Fig. 2.5. Shuffling progress and CPU utilization by each reducer on fast and slow nodes in D_3 (calculated P=2).

catches up with the total number of map tasks completed, possibly due to lack of resources, *i.e.*, slow nodes are overloaded.

The CPU utilization shown in Figure 2.5 further confirms this explanation. We see the CPU utilization of the reducer on slow nodes is stable between 59-66%. Since the reducer on slow nodes always lags behind the map tasks completed, we can conclude

that 66% is the maximum CPU a reducer can get on slow nodes. In contrast, the reducer CPU utilization on fast nodes reaches 120% (the multi-threaded process uses multiple cores) at the start of reducers, then gradually decreases and stabilizes at 55%, at which moment it has caught up with map task completion. This suggests at the steady state, the reducer on fast nodes just needs 55% of CPU, but it can get as much as 120% of CPU if needed.

The above finding suggests D_3 needs to be adjusted to use the potential progress rate of the reducer on fast nodes, as opposed to the progress rate observed (so far). The partition ratio P is now calculated as

$$P = \frac{S_{fa}}{S_{sl}} * \frac{1}{E_{fa}} \tag{2.1}$$

where E_{fa} denotes the CPU efficiency (<1) of the reducer on fast nodes, defined as the ratio of the CPU utilization in the steady state (T_w in Figure 2.6 bottom) to the CPU utilization when shuffle (on fast nodes) has caught up with map tasks completed (T_s in Figure 2.6 top). In practice, we observe the steady state T_w on fast nodes is typically reached when 1 wave of map-tasks are completed after T_s. Note the CPU utilization on slow nodes is fairly stable. Figure 2.6 shows the CPU utilization and shuffle when the partition ratio is adjusted at time T_w using the refined scheme, denoted by D_3'. The calculated partition ratio was 4.34. It can be seen that the fast node regains CPU utilization and both slow and fast nodes shuffle data at the same rate.

Lastly, D_3' can be easily extended to more than two types of nodes. We skip the details due to page limit.

2.5 Implementation

We implemented our new load rebalancing scheme D_3' in Hadoop v0.20.203.0 [12] by adding ≈2KLOC. We name the new system PIKACHU. Partition ratio P is calculated at JobTracker based on Hadoop progress rates and CPU efficiency of the reducer processes, which are reported by TaskTracker on each node every 3 seconds.

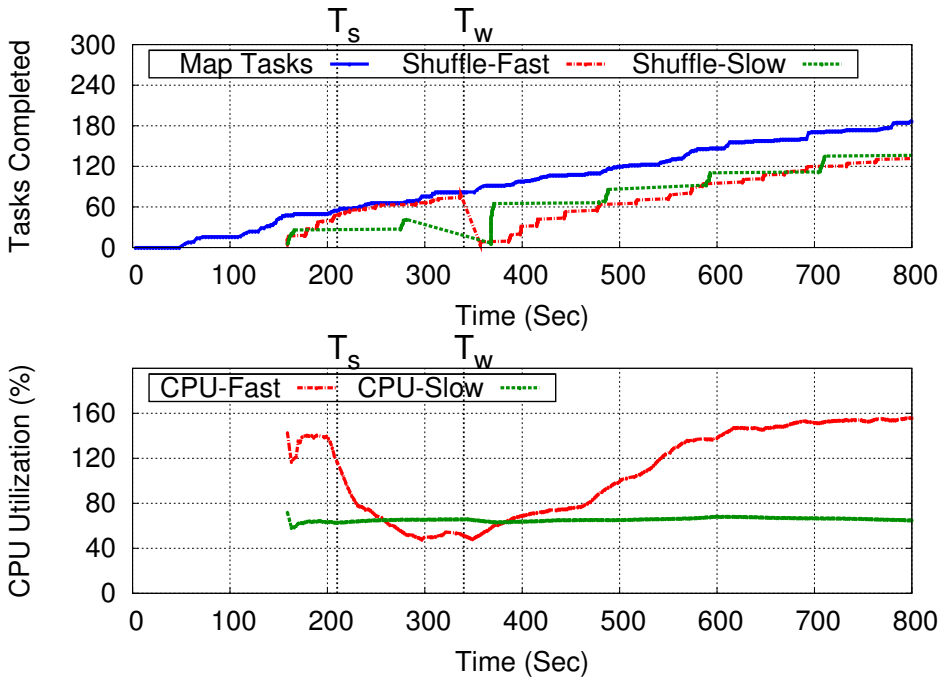


Fig. 2.6. Shuffling progress and CPU utilization by each reducer on fast and slow nodes under D_3'.

We use virtual bins to dynamically change the load between fast nodes and slow nodes based on partition ratio P. Each map task output is partitioned into $10 \cdot N$ splits, where N is the total number of reducers. Initially each reducer is mapped with 10 virtual bins. Once the JobTracker determines the ratio P, it translates the ratio to the target number of virtual bins for reducers on fast and slow nodes. Let N_s and N_f be the total number of reducer slots on all slow nodes and all fast nodes, respectively. The numbers of virtual bins for a reducer on a slow node V_s and on a fast node V_f are calculated as

$$V_s = \frac{10 \cdot (N_s + N_f)}{N_s + P \cdot N_f}, V_f = \frac{10 \cdot (N_s + N_f) \cdot P}{N_s + P \cdot N_f} \tag{2.2}$$

Following this, the JobTracker assigns a new virtual bin mapping to each reducer. Upon receiving the new mapping, the reducers on fast nodes need to fetch the newly added virtual bins, while the reducers on slow nodes will drop the existing sorted data corresponding to the dropped virtual bins.

2.6 Evaluation

We also implemented Tarazu [11] in Hadoop (version 0.2.203.0). We compare job completion time under PIKACHU, Tarazu, and Hadoop. We also measure the overhead incurred in PIKACHU due to re-shuffling and re-sorting. We use five benchmark applications: *Wordcount, Sort, Multi-Wordcount, Inverted-index and Selfjoin* [11]. *Wordcount* counts the occurrences of every word. *Sort* sorts the given dataset. *Multi-Wordcount* counts all unique sets of 3 consecutive words. *Inverted-index* generates words-to-file indexing. *Self-join* generates association among *k+1* fields given the set of k-field association. *Sort* and *Selfjoin* are shuffle-intensive, whereas the other 3 applications are compute-intensive.

Performance on Local Cluster. Figure 2.7 shows the speedup (in terms of job completion time) achieved by PIKACHU and Tarazu against Hadoop for 5 different applications using Config-1. In addition to Hadoop, Tarazu and PIKACHU, we also measure the job completion time at the optimal partition ratio found using trial-and-

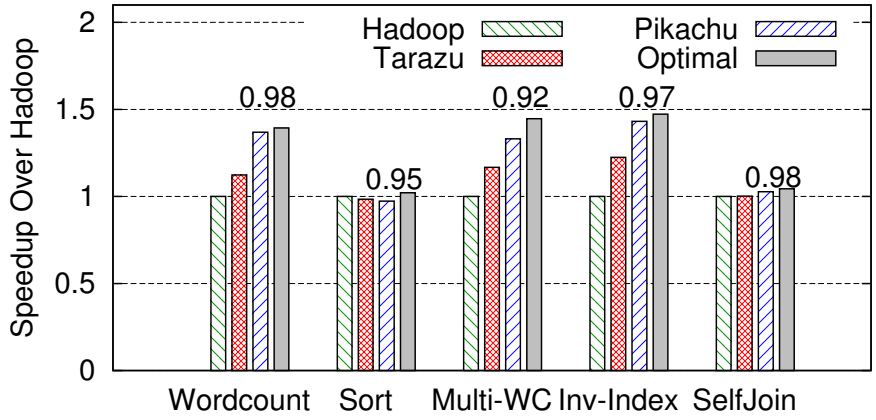


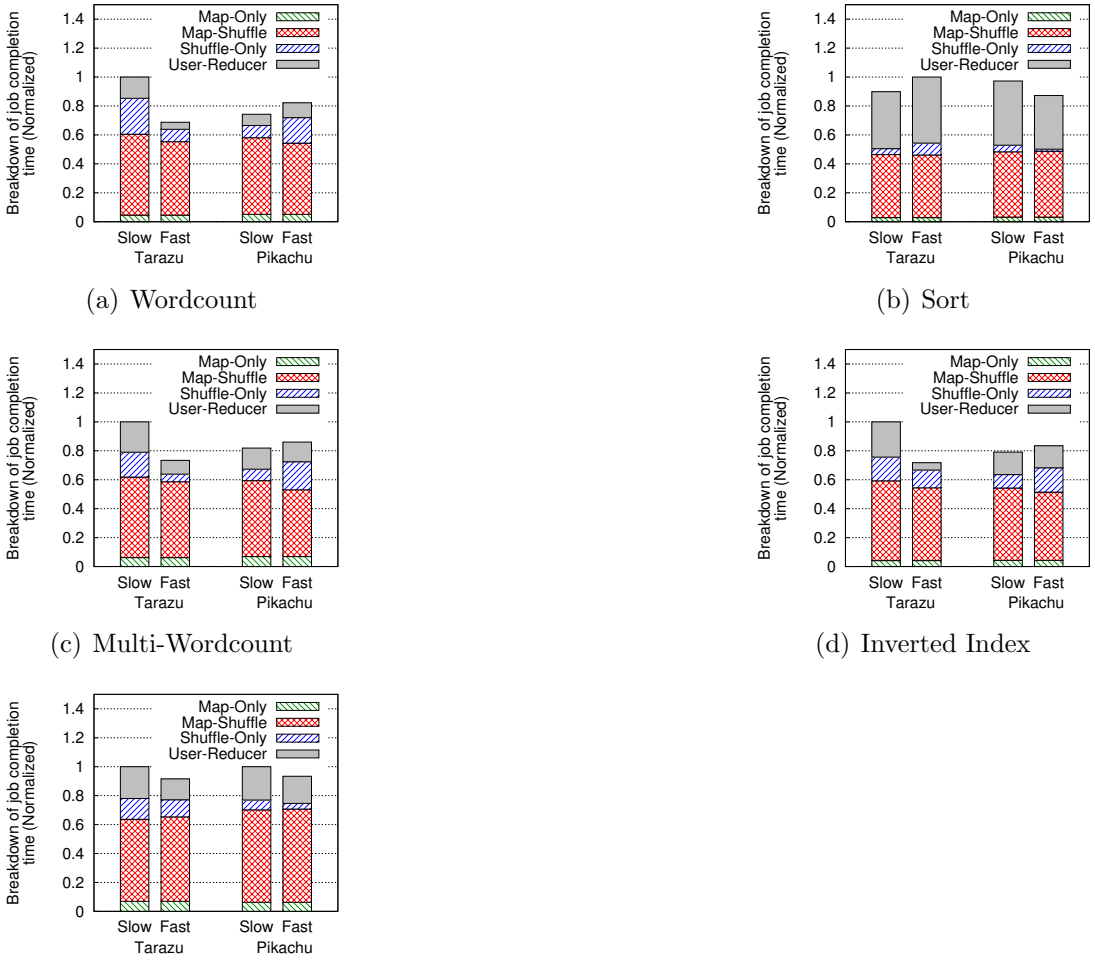
Fig. 2.7. Speedup of Tarazu and PIKACHU over Hadoop, under Config-1.

Table 2.1.
Partitioning ratio P and overhead under Tarazu, PIKACHU, and optimal partition for the five applications.

Observation	Wordcount			Sort			Multi-Wordcount			Inverted-Index			Self-join		
	T	P	O	T	P	O	T	P	O	T	P	O	T	P	O
Calculated P	2	4.5	4	1.1	0.67	0.9	2.37	3.7	3.5	2.44	3.4	3.3	1	1.1	1.2
Shuffle penalty	3.86%			4.13%			4.58%			4.75%			0.96%		

error method. The numbers above the bars denote the percentage of the optimal performance PIKACHU achieves. For Sort and Selfjoin applications, the initial configuration was close to optimal (the difference between the job completion time of Hadoop and Optimal was <4%) and there was little room for improvement. For the remaining applications, PIKACHU outperforms Hadoop by 33-42% and Tarazu by 14-22% because of better accuracy in calculating P. Furthermore, PIKACHU achieves 92-98% of the optimal job completion time, showing there is not much room to improve over PIKACHU.

Table 2.1 summarizes the partition ratios calculated by Tarazu (T), PIKACHU (P) and Optimal (O). The partition ratio calculated using PIKACHU is closer to Optimal compared to Tarazu. Table 2.1 also shows the overhead incurred by PIKACHU, mea-



(a) Wordcount

(b) Sort

(c) Multi-Wordcount

(d) Inverted Index

(e) Selfjoin

Fig. 2.8. Job completion time breakdown on fast and slow nodes (Normalized to Tarazu).

sured as the extra data shuffled by all the nodes in PIKACHU compared to Hadoop. We see PIKACHU incurs a low overhead 0.96-4.75% in re-shuffling and re-sorting.

Figure 2.8 shows the breakdown of the job completion time under PIKACHU normalized to the job completion time under Tarazu for all 5 applications on slow and fast nodes. It can be seen that in all 5 cases, the difference between the shuffle-only execution time, and more importantly the difference between the reducer task completion time, on the nodes are within 10% on PIKACHU and 31% on Tarazu.

Performance on EC2 Cluster. Finally, we compared PIKACHU with Tarazu and Hadoop on a 60-node heterogeneous cluster in EC2, consisting of 40 `m1.small` (slow)

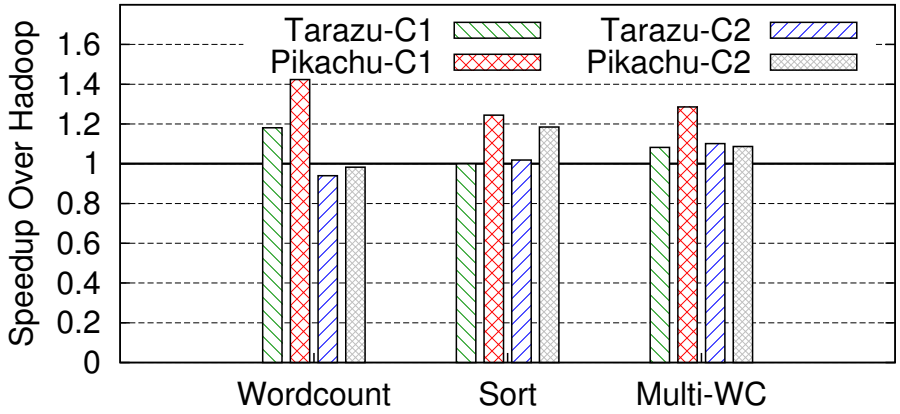


Fig. 2.9. Speedup of Tarazu and PIKACHU over Hadoop on the 60-node EC2 cluster on 2 configurations.

and 20 `m1.xlarge` (fast) nodes. We evaluated the performance using 3 applications, Wordcount, Sort and Multi-Wordcount under Config-1 and Config-2 for 180 GB data (2880 map tasks). Figure 2.9 shows the speedup achieved by Tarazu and PIKACHU over Hadoop for the 2 configurations.

In Config-1, Tarazu and PIKACHU outperform Hadoop by 0-18% and 25-42%, respectively. Config-2 was optimal configuration for Hadoop for Wordcount without much scope for improvement. Tarazu and PIKACHU performance was lower than Hadoop by 6% and 2%. For the other 2 applications, Tarazu and PIKACHU outperformed Hadoop by up to 10% and 18%, respectively.

2.7 Related Work

Many implementations, extensions, and domain specific libraries of MapReduce have been developed to support large-scale data processing [12–17]. None of them explicitly study optimizing MapReduce execution on heterogeneous hardware. LATE [3] was one of the first work to show the shortcomings of MapReduce on heterogeneous clusters. However, it focused on straggler detection and mitigation. Mantri [8] further explores the causes of stragglers/outliers. Such designs treat the symptoms of heterogeneity, *i.e.,* stragglers, as opposed to the root cause, and speculatively re-execute tasks on fast nodes, wasting utilization of slow nodes.

Lee *et al.* also considered heterogeneity in the MapReduce scheduler [18, 19] and proposed a fair scheduler [19] for a multi-tenant heterogeneous cluster. This work is orthogonal to ours as it improves the performance of multiple jobs rather than a single job. Finally, Tarazu [11] has already been discussed previously.

3. DUET: CLOUD SCALE LOAD BALANCING WITH HARDWARE AND SOFTWARE

3.1 Introduction

A high performance load balancer is one of the most important components of a cloud service infrastructure. Services in the data center scale by running on multiple servers, each with an individual direct IP (DIP). The service exposes one or more virtual IP addresses (VIP) outside the service boundary. The load balancer receives the traffic destined for the VIP, splits it among the DIPs, and routes it to the individual DIPs using IP-in-IP encapsulation.

The load balancer thus touches every packet coming into the data center from the Internet, as well as a significant fraction of all intra-DC traffic. This traffic volume induces heavy load on both data plane and control plane of the load balancer [1]. The performance and reliability of the load balancer directly impact the latency, throughput and the availability of the cloud services hosted in the DC.

Traditional load balancers are dedicated hardware middleboxes [20, 21] that are very expensive. In contrast, Ananta [1] is a software load balancer that runs on commodity servers. Ananta consists of a central controller, and several software Muxes (SMux) that provide a distributed data plane. Each SMux maintains all VIP-to-DIP mappings, and implements traffic splitting and encapsulation functionality in software. The Ananta architecture is flexible, highly scalable and ensures high availability.

However, software load balancers have two fundamental limitations, both of which stem from the fact that they process the packets in software. First, processing packets in software limits *capacity*. Experiments show that the CPU on individual Ananta SMux becomes a bottleneck once the incoming traffic exceeds 300K packets per sec-

ond. While the aggregate capacity of software load balancer can be scaled out by adding more servers, doing so raises cost. For example, handing 15Tbps traffic (typical for a mid-sized DC) requires over 4000 SMuxes, costing over USD 10 million.

Second, processing packets in software incurs high, and highly variable *latency*. An Ananta SMux, handling as little as 100K packets per second can add anywhere from 200μsec to 1ms of latency. Applications such as algorithmic stock trading and high performance distributed memory caches demand ultra-low (a few microseconds) latency within the data center. For such applications, the latency inflation by the software load balancer is not acceptable.

In this paper, we propose DUET, which addresses these two drawbacks of software load balancers. DUET uses *existing* switch hardware in data centers to build a high performance, in-situ, organically scalable hardware load balancer and *seamlessly* combines it with a small deployment of software load balancer for enhanced availability and flexibility.

DUET is based on two key ideas. The first idea is to build a load balancer from existing switches in the data center network. The key insight is that the two core functions needed to implement a load balancer – traffic splitting and packet encapsulation – have long been available in commodity switches deployed in data center networks. Traffic splitting is supported using ECMP, while packet encapsulation is supported using tunneling. However, it is only recently that the switch manufacturers have made available APIs that provide detailed, fine-grained control over the data structures (ECMP table and tunneling table) that control these two functions. We re-purpose unused entries in these tables to maintain a database of VIP-to-DIP mappings, thereby enabling the switch to act as a Mux in addition to its normal forwarding function. This gives us an in-situ, hardware Mux (HMux) – without new hardware. Since splitting and encapsulation are handled in the data plane, the switch-based load balancer incurs low latency (microseconds) and high capacity (500 Gbps).

While HMuxes offer high capacity, low latency and low cost, the architecture is less flexible than software load balancers. Specifically, handling certain cases of switch

failures is challenging (§3.5.1). Thus, our second idea is to integrate the HMuxes with a small deployment of SMuxes, to get the best of both worlds. We make the integration seamless using simple routing mechanisms. In the combined design, most of the traffic is handled by the switch-based hardware load balancer, while software load balancer acts as a backstop, to ensure high availability and provide flexibility.

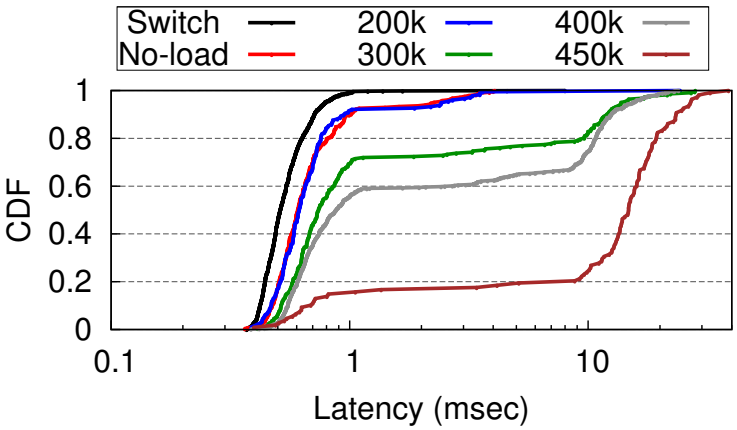
Compared to dedicated hardware load balancers, or pure software load balancers (Ananta), DUET is highly cost effective. It load-balances most of the traffic using *existing* switches (HMuxes), and needs only a small deployment of software load balancer as a backstop. Because most of the traffic is handled by the HMuxes, DUET has significantly lower latency than software load balancers. At the same time, use of software load balancer enables DUET to inherit high availability and flexibility of the software load balancer.

To design PIKACHU , we addressed two main challenges. First, individual switches in the data center do not have enough memory to hold the entire VIP-to-DIP mapping database. Thus, we need to partition the mappings among the switches. We devise a simple greedy algorithm to do this, that attempts to minimize the "leftover" traffic (which is perforce handled by the software load balancer), while taking into account constraints on switch memory and demands of various traffic flows.

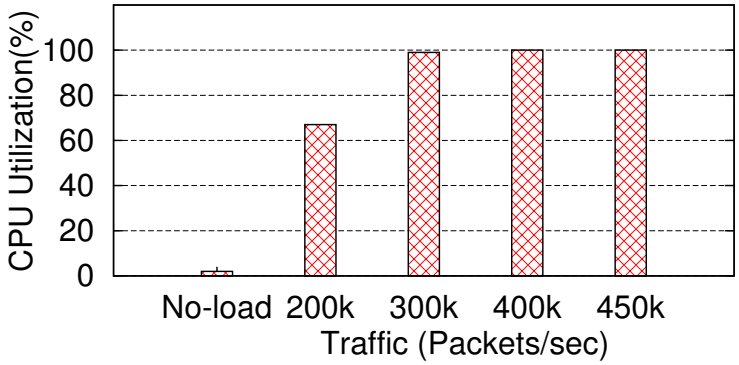
The second challenge is that this mapping must be regularly updated as conditions change. For example, VIPs or DIPs are added or removed by customers, switches and links fail and recover etc. We devise a migration scheme that avoids memory deadlocks and minimizes unnecessary VIP movement.

We evaluate DUET using a testbed implementation as well as extensive, large-scale simulations. Our results show that DUET provides 10x more capacity than the pure software load balancer, at a fraction of the SMux cost, while also reducing the latency inflation by 10x or more. Additionally, we show that PIKACHU quickly adapts to the network dynamics in the data center including failures.

In summary, the paper makes the following three contributions. First, We characterize the conditions, design challenges, and design principles for moving load bal-



(a) End-to-end latency



(b) CPU Utilization

Fig. 3.1. Performance of software Mux.

ancing functionality directly into hardware switches which offer significantly lower latency and higher capacity than software servers. Second, we present the design and implementation of a switch-based load balancer. To the best of our knowledge, this is the first such design. Third, we show how to seamlessly combine the switch-based load balancer with software load balancer to achieve high availability and flexibility. Again, to the best of our knowledge, this is the first "hybrid" load balancer design.

3.2 Background and Motivation

We provide background on load balancing functionality in DCs, briefly describe a software-only load balancer architecture (Ananta), and point out its shortcomings.

A DC typically hosts multiple services. Each service is a set of servers that work together as a single entity. Each server in the set has a unique direct IP (DIP) address. Each service exposes one or more virtual IP (VIP) outside the service boundary. The load balancer forwards the traffic destined to a VIP to one of DIPs for that VIP. Even services within the same DC use VIPs to communicate with each other, since the indirection provided by VIPs offers several benefits. For example, individual servers can be maintained or upgraded without affecting dependent services. Management of firewall rules and ACLs is simplified by expressing them only in terms of VIPs, instead of DIPs, which are far more numerous and are subject to churn.

The key to the efficient functioning of the indirection architecture is the load balancer. A typical DC supports thousands of services [1, 22], each of which has at least one VIP and many DIPs associated with it. All incoming Internet traffic to these services and most inter-service traffic go through the load balancer. As in [1], we observe that almost 70% of the total VIP traffic is generated within DC, and the rest is from the Internet. The load balancer design must not only scale to handle this workload but also minimize the processing latency. This is because to fulfill a single user request, multiple back-end services often need to communicate with each other — traversing the load balancer multiple times. Any extra delay imposed by the load balancer could have a negative impact on end-to-end user experience. Besides that, the load balancer design must also ensure high service availability in face of failures of VIPs, DIPs or network devices.

3.2.1 Ananta Software Load Balancer

We first briefly describe the Ananta [1] software load balancer. Ananta uses a three-tier architecture, consisting of ECMP on the routers, several software Muxes (SMuxes) that run on commodity servers, and are deployed throughout the DC, and a host agent (HA) that runs on each server.

Each SMux stores the VIP to DIP mappings for all the VIPs configured in the DC. Using BGP, every SMux announces itself to be the next hop for every VIP. Incoming packets for a VIP are directed to one of the SMuxes using ECMP. The SMux selects a DIP for the VIP, and encapsulates the packet, setting the destination address of the outer IP header to the chosen DIP. At the DIP, the HA decapsulates the incoming packet, rewrites the destination address and port, and sends it to server. The HA also intercepts outgoing packets, and rewrites their IP source addresses from the DIP to the VIP, and forwards the direct server return (DSR).

Ananta can support essentially an unlimited number of VIPs and DIPs, because it stores this mapping in the large main memory on commodity servers. While a single SMux in Ananta has limited capacity (due to software processing), Ananta can still scale to handle large volumes of traffic. First, Ananta deploys numerous SMuxs, and relies on ECMP to split the incoming traffic among them. Second, DSR ensures that only the incoming or the VIP traffic goes through the load balancer. Ananta also includes a mechanism called fast path to enhance scalability. Fast path allows all inter-service traffic to directly use DIPs, instead of using VIPs. However, this negates the benefits of the VIP indirection. For example, if fast path is enabled, service ACLs have to be expressed in terms of DIPs.

In summary, implementing parts of load balancing functionality in software allows Ananta to be highly scalable and flexible. However, processing packets in software is also the Achilles heel for Ananta, because it adds latency, and limits the throughput, as we discuss next.

3.2.2 Limitations of Software Load Balancer

Figure 3.1(a) shows the CDF of the RTTs for the VIP traffic load-balanced by a production Ananta SMux as traffic to the VIP varies between 0 and 450K packets/sec. Even at zero load the SMux adds a median latency of 196μsec. The latency variance is also significant, with the 90^{th} percentile being 1ms. The median RTT (without load

balancer) in our production DCs is 381μsec, so the inflation in latency is significant for the intra-DC traffic, which accounts for 70% of the total VIP traffic. (For the remaining traffic from the Internet, it is a lesser problem due to larger WAN latencies). The high latency inflation and high latency variability result from processing the packets in software. We also see that the added latency and the variance get much worse at higher load.

The results also illustrate that an individual SMux instance has low capacity. Beyond 300K packets/sec, the CPU utilization reaches 100% (Figure 3.1(b)). Thus, for the hardware SKU used in our DCs, each SMux can handle only up to 300K packets/sec, which translates to 3.6 Gbps for 1,500-byte packets. At this rate, supporting 15 Tbps VIP traffic for a mid-sized (40K servers) DC would require over 4K SMuxes, or 10% of the DC size; which is unacceptable[1].

3.3 Duet: Core ideas

In the previous section, we saw that while software load balancers are flexible and scalable, they suffer from low throughput and high latency. In this paper, we propose a new design called Duet that offers scalability, high throughput and low latency, at a small fraction of the software load balancer's cost.

Duet is based on two novel ideas. First, we leverage idle resources of modern, commodity data center switches to construct a hardware load balancer. We call this design Hardware Mux (HMux). HMux offers microsecond latency, and high capacity, without the need for *any* additional hardware. However, the HMux design suffers from certain shortcomings. Thus, our second idea is to combine the HMux with Ananta-like software Mux (SMux). The combined system is called Duet in which the SMux acts as a backstop for the HMux.

[1]Newer technologies such as direct-packet IO and RDMA may help match packet processing capacity of the SMux to that of the NIC (10 Gbps), but they may not match packet processing capacity of the switch (600 Gbps+) as we explain in § 3.3.1.

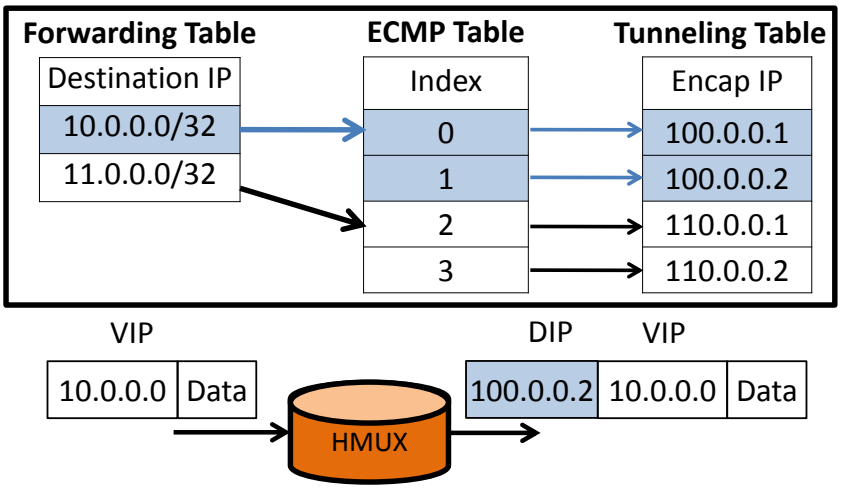


Fig. 3.2. Storing VIP-DIP mapping on a switch.

We now describe the design of HMux. To simplify the description, we will assume that the DC is not virtualized, *i.e.,* one DIP corresponds to one server. The changes required to support VMs are described in §3.5.2.

3.3.1 HMux

Ananta's SMux implements two key functions to load balance traffic: (1) for each VIP, split traffic equally among its DIPs, and (2) use IP-in-IP encapsulation to route the VIP traffic to the corresponding DIPs. Both of these functions have long been available on commodity switches, *i.e.,* traffic splitting is supported using ECMP and IP-in-IP encapsulation is supported using tunneling. However, major switch vendors have only recently started to provide the APIs for fine-grained control over ECMP and tunneling functionality.

Our key insight is that by carefully programming the ECMP and tunneling tables using these new APIs, we can make a commodity switch act as a hardware Mux (HMux), in addition to its normal functionality. In fact, this can be easily done on most of the switches used in our DCs today.

Figure 3.2 shows the HMux design. A packet arriving at a switch goes through a processing pipeline. We focus on three tables used in the pipeline. The packet matches one entry in the host forwarding table which then points to multiple ECMP table entries. These ECMP table entries correspond to multiple next hops for the packet[2]. The actual next hop for the packet is selected by using the hash of the IP 5-tuple to index into the ECMP table. The tunneling table enables IP-in-IP encapsulation by storing the information needed to prepare the outer IP header for a given packet.

To construct HMux, we link the ECMP and tunneling functionalities. Consider a packet destined for VIP 10.0.0.0 that arrives at the HMux. There are two DIPs (100.0.0.1 and 100.0.0.2) for this VIP. The host forwarding table indicates that the first two entries in the ECMP table pertain to this VIP. The ECMP entries indicate that packets should be encapsulated, and point to appropriate entries in the tunneling table. The switch encapsulates the packet using IP-in-IP encapsulation, and the destination address in the outer IP header is set to the DIP address specified in the tunneling table entry. The packet is then forwarded to the appropriate interface.

Thus, at the expense of some entries in the host forwarding, ECMP and tunneling tables, we can build a load balancer using commodity switches. In fact, if all the VIP-to-DIP mappings are stored on every top-of-rack (ToR) switch as well as every access switch, this HMux design can provide load balancing functionality to all intra-DC and inter-DC traffic. However, the amount of space available in the three tables is limited, raising two distinct issues.

Number of VIPs: The first problem is the size of the host forwarding table. The switches in our DC have 16K entries in the host table. The host table is mostly empty, because it is used only for routing within a rack. But even the 16K entries may not be enough to hold all VIPs in a large DC. One way to address this problem is by using longest prefix match (LPM) forwarding table. However, LPM table is

[2]The information is split between ECMP group table and ECMP table; we omit such details due to lack of space.

heavily used for routing within and across DCs, and is not available to be used for load balancing. We support higher number of VIPs using SMuxes as explained in §3.3.3.

Number of DIPs: The second problem concerns the sizes of the ECMP and tunneling tables. ECMP table typically holds 4K entries, and is mostly empty (see § 3.9). The tunneling table typically holds 512 entries. In our DC, few applications use tunneling, so these entries are mostly free as well. The number of DIPs an individual HMux can support is the minimum of the number of free entries in the ECMP and the tunneling tables (see Figure 3.2). Thus, an individual HMux can support at most 512 DIPs. This is orders of magnitude smaller than the total number of DIPs. We address this challenge next.

3.3.2 Partitioning

We address the problem of limited size of ECMP and tunneling tables using two mechanisms: (1) We divide the VIP-to-DIP mapping across multiple switches. Every switch stores only a small subset of all the VIPs, but stores all the DIPs for those VIPs. This way of partitioning ensures all the traffic for a particular VIP arrives at a single switch and the traffic is then equally split among the DIPs for that VIP. (2) Using BGP, we announce the VIPs that are assigned to the switches, so that other switches can route the VIP packets to the switch where the VIP is assigned.

Figure 3.3 illustrates this approach. VIP_1 has two DIPs (D_1 and D_2), whereas VIP_2 has one (D_3). We assign VIP_1 and VIP_2 to switches C_2 and A_6 respectively, and flood the routing information in the network. Thus, when a source S_1 sends a packet to VIP_1, it is routed to switch C_2, which then encapsulates the packet with either D_1 or D_2, and forwards the packet.

Another key benefit of partitioning is that it achieves *organic scalability* of HMuxes — when more servers are added in the DC and hence traffic demand increases, more

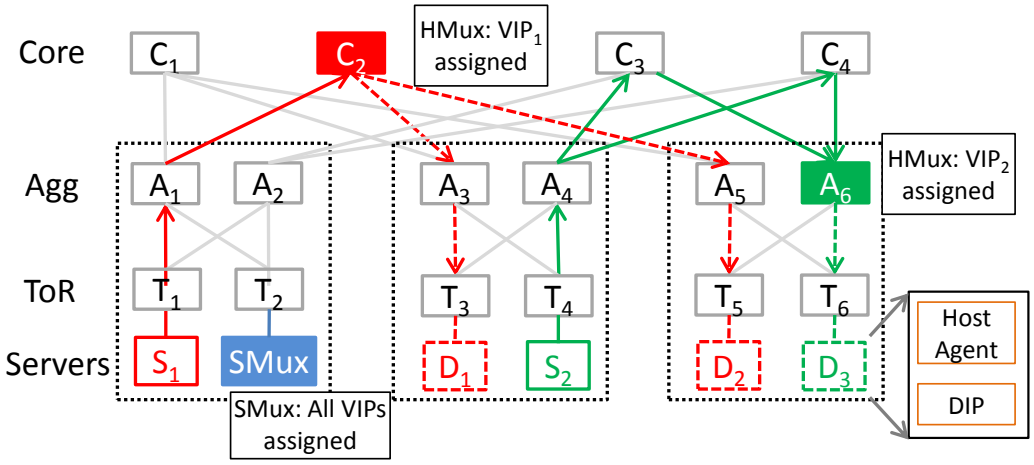


Fig. 3.3. **Duet architecture**: VIPs are partitioned across different HMuxes — VIP_1 and VIP_2 are assigned to HMux C_2 and A_6. Additionally, SMuxes act as backstop for all the VIPs. Every server (apart from SMuxes) runs host-agent that decapsulates the packets and forwards to the DIP. Links marked with solid lines carry VIP traffic, and links with dotted lines carry DIP traffic.

switches will also be added and hence the aggregate capacity of HMuxes will also increase proportionally.

3.3.3 DUET: HMux + SMux

While partitioning helps increase the number of DIPs HMux can support, that number still remains limited. The HMux design also lacks the flexibility of SMux, because VIPs are partitioned and "pinned" to specific HMuxes. This makes it challenging to achieve high VIP availability during network failures. Although replicating VIP across a few switches may help improve failure resilience, it is still hard to achieve the high availability of Ananta because Ananta stores the complete VIP-DIP mappings on a large number of SMuxes.

This motivates us to architect Pikachu — a new load balancer design to fuse the flexibility of SMux and the high capacity and low latency of HMux.

Design

DUET's goal is to maximize VIP traffic handled using HMux, while using SMux as a backstop. Thus, besides an HMux on each switch, DUET also deploys a small number of SMuxes on commodity servers (figure 3.3). The VIPs are partitioned among HMuxes as described earlier. In addition, each SMux announces all the VIPs. The routing protocol preferentially routes VIP traffic to HMux, ensuring that VIP traffic is primarily handled by HMux – thereby providing high capacity and low latency. In case of HMux failure, traffic is automatically diverted to SMux, thereby achieving high availability. To ensure that existing connections do not break as a VIP migrates from HMux to SMux or between HMuxes, all HMuxes and SMuxes use the same hash function to select DIPs for a given VIP.

The preferential routing to HMux can be achieved in several ways. In our current implementation, SMux announces the VIPs in aggregate prefixes, while HMux announces /32 routes to individual VIPs. Longest prefix matching (LPM) prefers /32 routes over aggregate prefix routes, and thus directs incoming VIP traffic to appropriate HMux, unless that HMux is unavailable.

The number of SMuxes needed depends on several factors including the VIP traffic that cannot be assigned to HMux due to switch memory or link bandwidth limits (§3.4), the VIP traffic that failovers to SMux due to HMux failure (§3.5.1), and the VIP traffic that is temporarily assigned to SMux during VIP migration (§3.4.2). We estimate it based on historical traffic and failure data in DC.

Benefits

The key benefits of DUET are summarized below.

Low cost: DUET does not require any additional hardware – it uses idle resources on existing switches to provide load balancing functionality. DUET also requires far fewer SMuxes than Ananta, since SMuxes are used only as a backstop for HMuxes, and hence carry far less traffic.

High capacity and low latency: this is because VIP traffic is primarily handled by HMux on switch.

High availability: by using SMux as a backstop during failures, DUET enjoys the same high availability as Ananta.

High limit on number of VIPs: If the number of VIPs exceeds the capacity of the host forwarding table (16K), the additional VIPs can be hosted on SMux. Traffic data (Figure 3.15) in our production DCs shows that VIP traffic distribution is highly skewed – most of the traffic is destined for a small number of "elephant" VIPs which can be handled by HMux. The remaining traffic to "mice" VIPs can be handled by SMux.

These benefits can only be realized through careful VIP-switch assignment. The assignment must take into account both memory and bandwidth constraints on individual switches, as well as different traffic load of different VIPs. The assignment must dynamically adapt to changes in traffic patterns and network failures. In the next two sections, we describe how DUET solves these problems, as well as provides other load balancing functions.

3.4 VIP Assignment Algorithm

We formalize the VIP-switch assignment problem using the notations listed in Table 3.1.

Input: The input to the algorithm includes the list of VIPs (V), the DIPs for each individual VIP v (d_v), and the traffic volume for each VIP. The latter is obtained from network monitoring. The input also includes the network topology, consisting of a set of switches (S) and a set of links (E). The switches and links constitute the two types of resources (R) in the assignment. Each resource instance has a fixed capacity C_i, *i.e.,* the link bandwidth for a link, and memory capacity that includes residual ECMP and tunneling table capacity available for PIKACHU on a switch. To absorb

Notation	**Explanation**
V	Set of VIPs
d_v	Set of DIPs for the v-th VIP
S, E	Set of switches and links respectively
R	Set of resources (switches and links)
C_i	Capacity of i-th resource
$t_{i,s,v}$	v-th VIP's traffic on i-th link, when it is assigned to s-th switch
$L_{i,s,v}$	load (additional utilization) on i-th resource if v-th VIP is assigned to s-th switch
$U_{i,s,v}$	Cumulative utilization of i-th resource if v-th VIP is assigned to s-th switch
$U_{i,v}$	Cumulative utilization of i-th resource after v VIPs have been assigned
$MRU_{s,v}$	Max. Resource Utilization (MRU) after v-th VIP is assigned to s-th switch

Table 3.1.
Notations used in VIP assignment algorithm.

the potential transient congestion during VIP migration and network failures, we set the capacity of a link to be 80% of its bandwidth.

Objective: Find the VIP-switch assignment that maximizes the VIP traffic handled by HMux. As explained earlier, this will improve latency and reduce cost by cutting the number of SMux needed. We do not attempt to minimize the extra network propagation delay due to indirection because the propagation delay contributes only less than 30μsec of the 381μsec RTT in our DC.

Constraints: Any VIP-switch assignment should not exceed the capacity of any of the resources.

The VIP assignment problem is a variant of multi-dimensional bin-packing problem [23], where the resources are the bins, and the VIPs are the objects. Multi-dimensional bin-packing problems are NP-hard [23]. DUET approximates it with a greedy algorithm, which works quite well in our simulations based on real topology and traffic load of a large production network.

3.4.1 VIP Assignment

We define the notion of maximum resource utilization (MRU). We have two types of resource – switches and links. MRU represents the maximum utilization across all switches and links.

Algorithm sketch: We sort a given set of VIPs in decreasing traffic volume, and attempt to assign them one by one (*i.e.,* VIPs with most traffic are assigned first). To assign a given VIP, we consider all switches as possible candidates to host the VIP. Typically, assigning a VIP to different switches will result in different MRU. We pick the assignment that results in the smallest MRU, breaking ties at random. If the smallest MRU exceeds 100%, *i.e.,* no assignment can accommodate the load of the VIP, the algorithm terminates. The remaining VIPs are not assigned to any switch – their traffic will be handled by the SMuxes. We now describe the process of calculating MRU.

Calculating MRU: We calculate the additional utilization (load) on every resource for each potential assignment. If the v-th VIP is assigned to the s-th switch, the extra utilization on the i-th link is $L_{i,s,v} = \frac{t_{i,s,v}}{C_i}$ where traffic $t_{i,s,v}$ is calculated based on the topology and routing information as the source/DIP locations and traffic load are known for every VIP. Similarly, the extra switch memory utilization is calculated as $L_{s,s,v} = \frac{|d_v|}{C_s}$, *i.e.,* the number of DIPs for that VIP over the switch memory capacity.

The cumulative resource utilization when the v-th VIP is assigned to the s-th switch is simply the sum of the resource utilization from previously assigned (v-1) VIPs and the additional utilization due to the v-th VIP:

$$U_{i,s,v} = U_{i,v-1} + L_{i,s,v} \tag{3.1}$$

The MRU is calculated as:

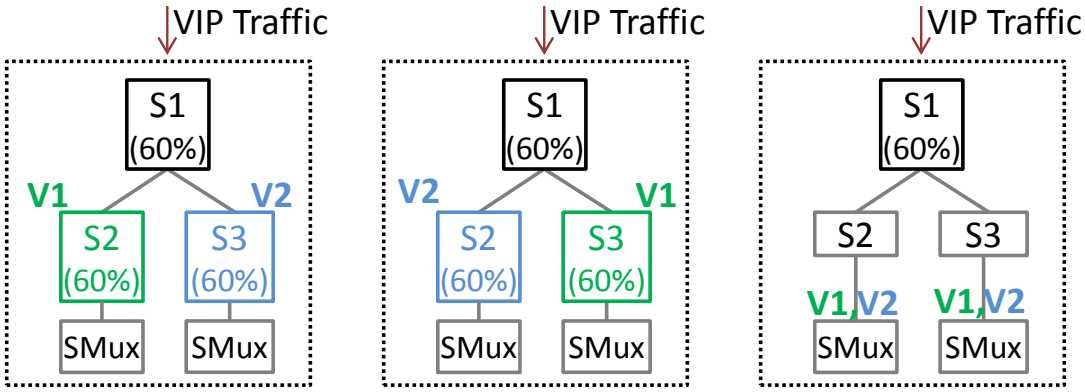
$$MRU_{s,v} = max(U_{i,s,v}), \forall i \in R \tag{3.2}$$

3.4.2 VIP Migration

Due to traffic dynamics, network failures, as well as VIP addition and removal, a VIP assignment calculated before may become out-of-date. From time to time, Duet needs to re-calculate the VIP assignment to see if it can handle more VIP traffic through HMux and/or reduce the MRU. If so, it will migrate VIPs from the old assignment to the new one.

There are two challenges here: (1) how to calculate the new assignment that can quickly adapt to network and traffic dynamics without causing too much VIP reshuffling, which may lead to transient congestion and latency inflation. (2) how to migrate from the current assignment to new one.

A simple approach would be to calculate the new assignment from scratch using new inputs (*i.e.,* new traffic, new VIPs etc.), and then migrate the VIPs whose assignment has changed between the current assignment and the new one. To prevent



(a) Initial (b) Final (c) Through SMux

Fig. 3.4. Memory deadlock problem during VIP migration. VIPs V1 and V2 both occupy 60% of switch memory each. The goal of migration is to migrate the VIPs from assignment in (a) to (b); PIKACHU eliminates this problem by migrating VIPs through SMuxes, as shown in (c).

routing black holes during VIP migration, we would use make-before-break — *i.e.*, a VIP would be announced from the new switch before it is withdrawn from the old switch. This simple approach is called *Non-sticky*.

The *Non-sticky* approach suffers from two problems. First, it may lead to transitional memory deadlock. Figure 3.4 shows a simple example where initially VIP V1 and VIP V2 are assigned to switches S_2 and S_3, respectively, but swap positions in the new assignment. Further, either VIP takes 60% of the switch memory. Because of limited free memory, there is no way to swap the VIPs under the make-before-break approach. When there are a large number of VIPs to migrate, finding a feasible migration plan becomes very challenging. Second, even if there was no such deadlock, calculating a new assignment from scratch may result in a lot of VIP reshuffling, for potentially small gains.

DUET circumvents transitional memory deadlocks by using SMux as a stepping stone. We first withdraw the VIPs that need to be moved from their currently assigned switches and let their traffic hit the SMux[3]. We then announce the VIPs from

[3]Recall that SMux announces all VIPs to serve as a backstop (§3.3.3)

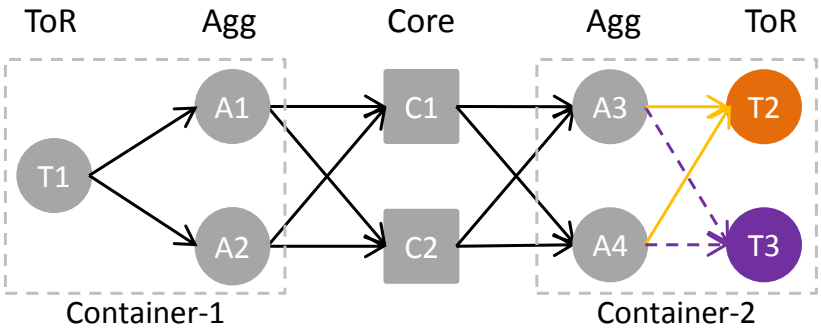


Fig. 3.5. When the VIP assignment changes from ToR T_2 to T_3, only the links inside container-2 are affected. As a result, we can first select best ToR in a container based on the links within container, and then scan over all containers and remaining Core and Agg switches.

their newly assigned switches, and let the traffic move to the new switches. This is illustrated in Figure 3.4(c) where both VIP's (V1 and V2) traffic is handled by SMux during migration.

Because SMux is used as a stepping stone, we want to avoid unnecessary VIP reshuffling to limit the amount of VIP traffic that is handled by SMux during migration. Hence, we devise a *Sticky* version of the greedy VIP assignment algorithm that takes the current assignment into account. A VIP is moved only if doing so results in significant reduction in MRU. Let us say that VIP v was assigned to switch s_c in the current assignment, and the MRU would be the lowest if it is assigned to switch s_n in the new assignment. We assign v to s_n only if $(MRU_{s_c,v} - MRU_{s_n,v})$ is greater than a threshold. Else we leave v at s_c.

Complexity: It is important for DUET to calculate the new assignment quickly in order to promptly adapt to network dynamics. Since all $L_{i,s,v}$ can be pre-computed, the complexity to find the minimum MRU (Equation 3.2) for VIP-switch assignment is $O(|V| \cdot |S| \cdot |E|)$.

This complexity can be further reduced by leveraging the hierarchy and symmetry in the data center network topology. The key observation is that assigning a VIP to different ToR switches inside a container will only affect the resource utilization inside the same container (shown in Figure 3.5). Therefore, when assigning a VIP,

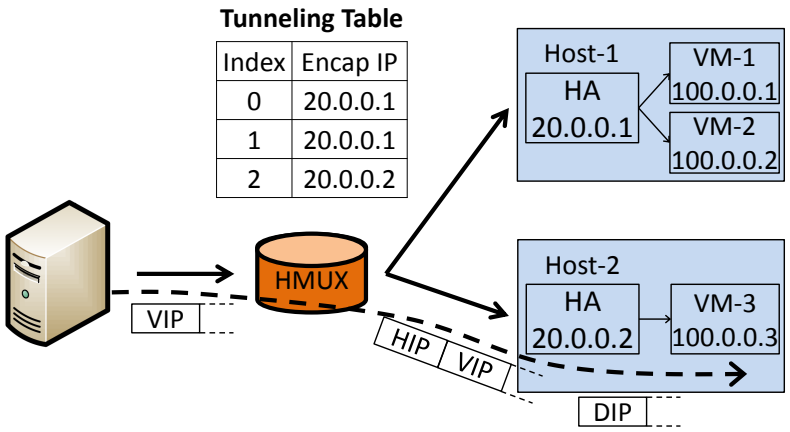


Fig. 3.6. Load balancing in virtualized clusters.

we only need to consider one ToR switch with the lowest MRU inside each container. Because ToR switches constitute a majority of the switches in the data center, this will significantly reduce the computation complexity to $O(|V| \cdot ((|S_{core}| + |S_{agg}| + |C|) \cdot |E| + |S_{tor}| \cdot |E_c|))$. Here C and E_c denote the containers and links inside a container. S_{core}, S_{agg} and S_{tor} are the Core, Aggregation and ToR switches respectively.

3.5 Practical Issues

We now describe how Pikachu handles important practical issues such as failures and configuration changes.

3.5.1 Failure Recovery

A critical requirement for load balancer is to maintain high availability even during failures. Pikachu achieves this primarily by using SMuxes as a backstop.

HMux (switch) failure: The failure of an HMux is detected by neighboring switches. The routing entries for the VIPs assigned to the failed HMux are removed from all other switches via BGP withdraw messages. After routing convergence, packets for these VIPs are forwarded to SMuxes, since SMuxes announce all VIPs.

All HMux and SMux use the same hash function to select DIPs for a given VIP, so existing connections are not broken, although they may suffer some packet drops and/or reorderings during convergence time (<40ms, see §3.7.2). Because in our production DCs we rarely encounter failures that are more severe than three switch failures or single container failures at a time, we provision sufficient number of SMuxes to handle the failover VIP traffic from HMuxes due to those failures.

SMux failure: SMux failure has no impact on VIPs assigned to HMux, and has only a small impact on VIPs that are assigned only to SMuxes. Switches detect SMux failure through BGP, and use ECMP to direct traffic to other SMuxes. Existing connections are not broken, although they may suffer packet drops and/or reorderings during convergence.

Link failure: If a link failure isolates a switch, it is handled as a switch failure. Otherwise, it has no impact on availability, although it may cause VIP traffic to re-route.

DIP failure: The DUET controller monitors DIP health and removes failed DIP from the set of DIPs for the corresponding VIP. Existing connections to the failed DIP are necessarily terminated. Existing connections to other DIPs for the corresponding VIP are still maintained using resilient hashing [24].

3.5.2 Other Functionalities

VIP addition: A new VIP is first added to SMuxes, and then the migration algorithm decides the right destination.

VIP removal: When a VIP assigned to an HMux is to be withdrawn, the controller removes it both from that HMux and from all SMuxes. VIPs assigned to *only* SMuxes need to be removed only from SMuxes. BGP withdraw messages remove the corresponding routing entries from all switches.

DIP addition: The key issue is to ensure that existing connections are not remapped if DIPs are added to a VIP. For VIPs assigned to SMuxes, this is easily

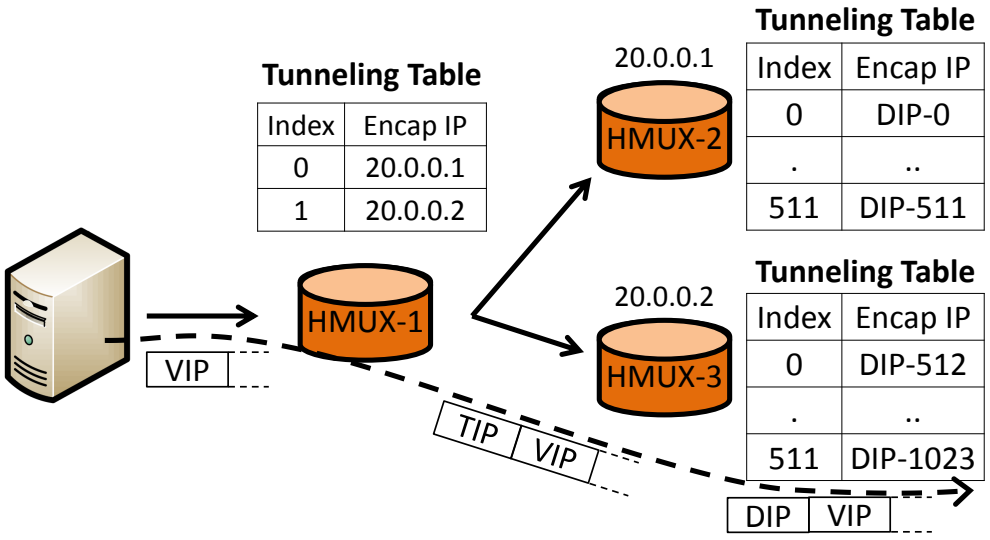


Fig. 3.7. Large fanout support.

achieved, since SMuxes maintain detailed connection state to ensure that existing connections continue to go to the right DIPs. However, HMuxes can only use a hash function to map VIPs to DIPs (Figure 3.2). Resilient hashing only ensures correct mapping in case of DIP *removal* – not DIP *addition*. Thus, to add a DIP to a VIP that is assigned to an HMux, we first remove the VIP from the HMux, causing SMuxes to take it over, as described earlier. We then add the new DIP, and eventually move the VIP back to an appropriate HMux.

DIP removal: DIP removal is handled in a manner similar to DIP failure.

Virtualized clusters: In virtualized clusters, the HMux would have to encapsulate the packet twice – outer header carries the IP of the host (native) machine, while inner header carries IP of the VM hosting the DIP. However, today's switches cannot encapsulate a single packet twice. So, we use HA in tandem with HMux, as shown in Figure 3.6. The HMux encapsulates the packet with the IP of the host machine (HIP) that is hosting the DIP. The HA on the DIP decapsulates the packet and forwards it to the right DIP based on the VIP. If a host has multiple DIPs, the ECMP and tunneling table on the HMux holds multiple entries for that HIP (HIP

20.0.0.1 in Figure 3.6) to ensure equal splitting. At the host, the HA selects the DIP by hashing the 5-tuple.

Heterogeneity among servers: When the DIPs for a given VIP have different processing power, we can proportionally split the traffic using WCMP (Weighted Cost Multi-Path) where faster DIPs are assigned larger weights. WCMP can be easily implemented on commodity switches.

VIPs with large fanout: Typically the capacity of the tunneling table on a single-chip switch is 512. To support a VIP that has more than 512 DIPs, we use indirection, as shown in Figure 3.7. We divide the DIPs into multiple partitions, each with at most 512 entries. We assign a single transient IP (TIP) for each partition. As a VIP, a TIP is a routable IP, and is assigned to a switch. When assigning a VIP to an HMux, we store the TIPs (as opposed to DIPs) in the tunneling table (Figure 3.7). When a packet for such a VIP is received at the HMux, the HMux encapsulates the packet with one of the TIPs and forwards it to the switch to which the TIP is assigned. That switch decapsulates the TIP header and re-encapsulates the packet with one of the DIPs, and forwards it. The latency inflation is negligible, as commodity switches are capable of decapsulating and re-encapsulating a packet at line rate. This allows us to support up to $512 * 512 = 262,144$ DIPs for a single VIP, albeit with small extra propagation delay[4].

Port-based load balancing: A VIP can have one set of DIPs for the HTTP port and another for the FTP port. Pikachu supports this using the tunneling table and ACL rules. ACL (Access Control) Rules are similar to OpenFlow rules, but currently support a wider range of fields. We store the DIPs for different destination ports at different indices in the tunneling table (Figure 3.8). The ACL rules, match on the IP destination and destination port fields, and the action is forwarding the packet to the corresponding tunneling table entry. Typically the number of ACL rules supported is larger than the tunneling table size, so it is not a bottleneck.

[4]The VIP assignment algorithm also needs some changes to handle TIPs. We omit details due to lack of space.

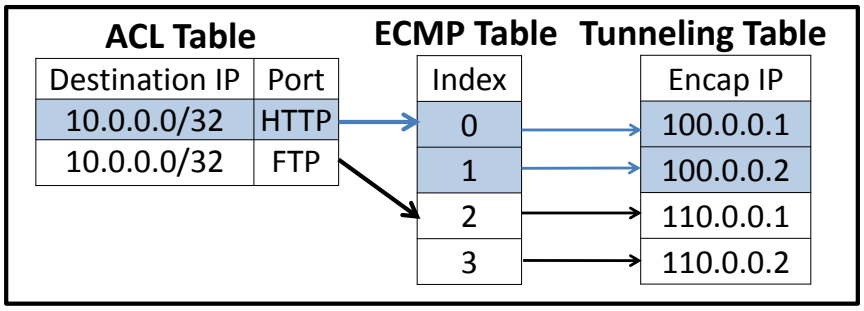


Fig. 3.8. Port-based load balancing.

SNAT: Source NAT (SNAT) support is needed for DIPs to establish outgoing connections[5]. Ananta supports SNAT by maintaining state on SMuxes [1]. However, as discussed earlier, switches cannot maintain such connection state. Instead, DUET supports SNAT by sharing the hash function used by HMux with the host agent (HA). Like Ananta, DUET assigns disjoint port ranges to the DIPs, but unlike Ananta, the HA on the DIP does not randomly choose an unused port number. Instead, it selects a port such that the hash of the 5-tuple would correctly match the ECMP table entry on HMux. The HA can do this easily since it knows the hash function used by HMux. Note that the HA needs to do this *only* during establishment (*i.e.,* first packet) of outgoing connections. If an HA runs out of available ports, it receives another set from the PIKACHU controller.

3.6 Implementation

In this section, we briefly discuss the implementation of the key components in PIKACHU : (1) PIKACHU Controller, (2) Host Agent, and (3) Switch Agent, and (4) SMux, as shown in Figure 3.9.

PIKACHU **Controller:** The controller is the heart of PIKACHU. It performs three key functions: (1) Datacenter monitoring: It gathers the topology and traffic information from the underlying network. Additionally, it receives the VIP health

[5]Outgoing packets on established connections use DSR.

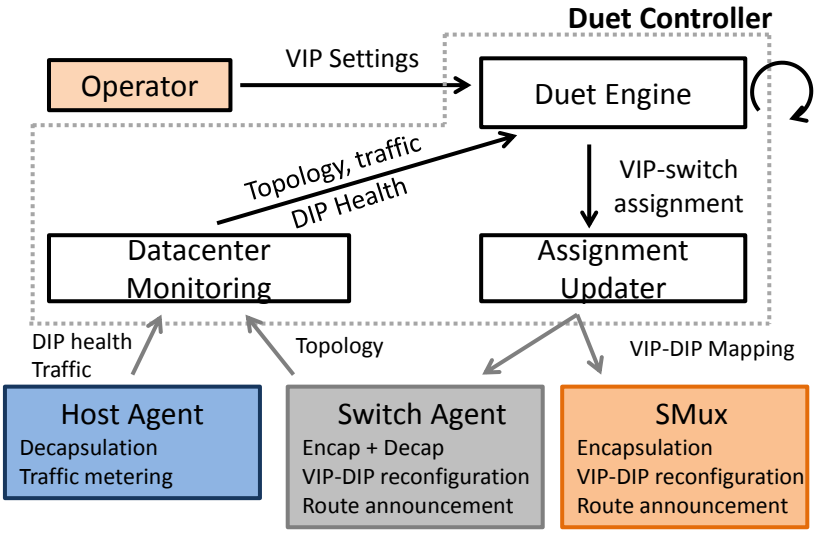


Fig. 3.9. Components in Pikachu implementation.

status periodically from the host agents. (2) Pikachu Engine: It receives the VIP-to-DIP mapping from the network operator and the topology and traffic information from the DC-monitoring module, and performs the VIP-switch assignment as described in § 3.4. (3) Assignment Updater: It takes the VIP-switch assignment from the Pikachu engine and translates it into rules based on the switch agent interface. All these modules communicate with each other using RESTful APIs.

Switch Agent: The switch agent runs on every switch. It uses vendor-specific APIs to program the ECMP and tunneling tables, and provides RESTful APIs which are used by the assignment updater to add/remove VIP-DIP mapping. On every VIP change, the switch agent fires touting updates over BGP.

Host Agent and SMux: The host agent and SMux implementation are the same as in Ananta. The host agent primarily performs packet decapsulation, SNAT and DIP health monitoring. Additionally, the host agents perform traffic metering and report the statistics to the Pikachu controller.

Same as in Ananta, we run a BGP speaker along side of each SMux to advertise all the VIPs assigned to the SMux.

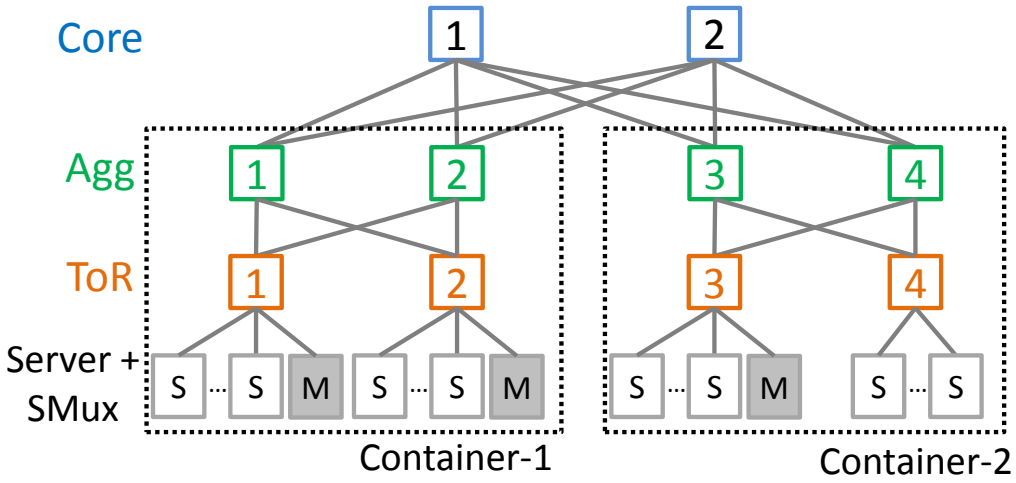


Fig. 3.10. Our testbed. FatTree with two containers of two Agg and two ToR Switches each, connected by two Core switches.

In total, the controller code consists of 4200 LOC written in C#, and the switch agent code has about 300 LOC in Python.

3.7 Testbed Experiments

Our testbed (Figure 3.10) consists of 10 Broadcom-based switches and 60 servers. Of the 60 servers, 34 act as DIPs and the others are used to generate traffic. Each of ToRs 1, 2 and 3 is also connected to a server acting as SMux.

Our testbed experiments show: (1) HMuxes provide higher capacity, (2) Pikachu achieves high availability during HMux failure as the VIP traffic seamlessly falls back to SMuxes, and (3) VIP migration is fast, and Duet maintains high availability during VIP migration.

3.7.1 HMux Capacity

If the load balancer instances have low capacity, packet queues will start building up, and traffic will experience high latency. This experiment illustrates that individual

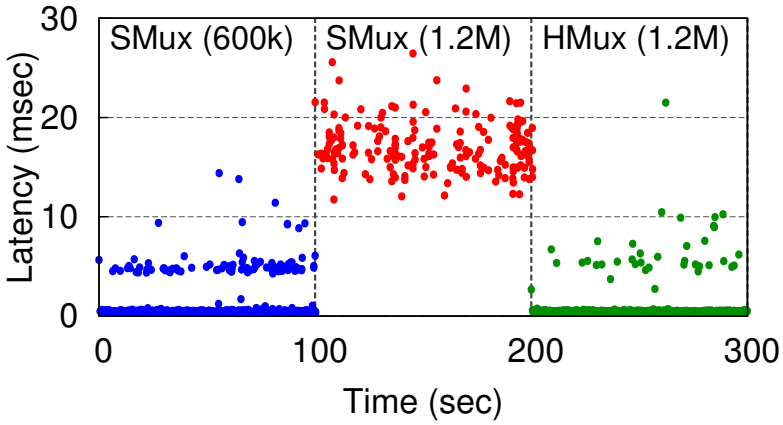


Fig. 3.11. HMux has higher capacity.

HMuxes instances (*i.e.,* a switch) have significantly higher capacity than individual SMux instances.

The experiment uses 11 VIPs, each with 2 DIPs. We send UDP traffic to 10 of the VIPs, leaving the 11th VIP unloaded.

The experiment has three steps. (1) All 11 VIPs are assigned to the SMuxes, and we generate a total traffic of 600K packets per second to the 10 VIPs (60K per VIP). Since each VIP is announced from every SMux, the traffic is split evenly between all SMuxes, and each SMux is handling 200K packets per second. (2) At time 100 sec, we increase the traffic to 1.2M packets per second, so each SMux is handling 400K packets per second. (3) Finally, at time 200 sec, we switch all VIPs to a *single* HMux hosted on ToR 1.

The metric of interest is the latency to the *unloaded* VIP, measured using pings sent every 3ms. We measure the latency to the unloaded VIP so that the latency *only* reflects the delay suffered at the SMux or HMux – the VIP or the DIP itself is not the bottleneck. The results shown in Figure 3.11.

We see that until time 100 sec, the latency is mostly below 1ms, with a few outliers. This is because each SMux is handling only 200K packets per second, which is well within its capacity (300K packets per second – see §3.2), and thus there is no significant queue buildup. At time 100, the latency jumps up – now each SMux is

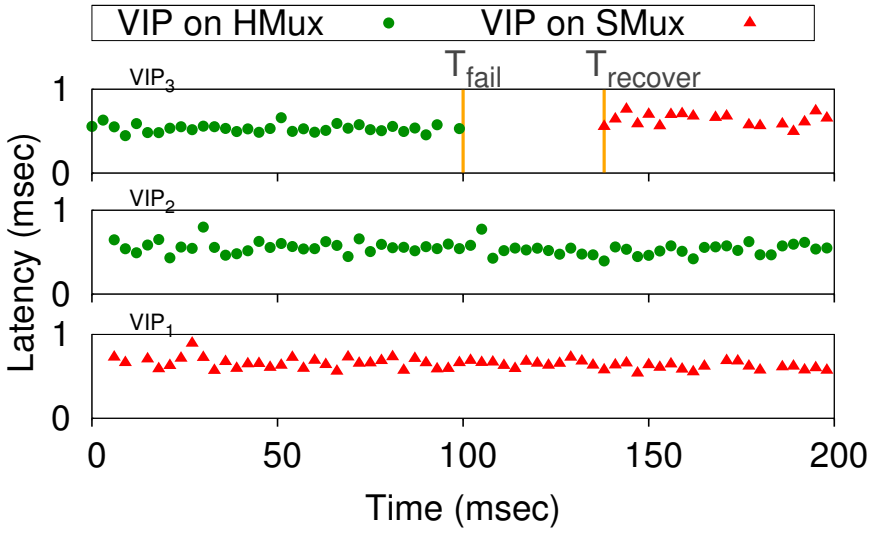


Fig. 3.12. VIP availability during failure.

handling 400K packets per second, which is well beyond its ability. Finally, at time 200 sec, when all VIPs are on a single HMux, the latency goes down to 1ms again. This shows that *a single* HMux instance has higher capacity than at least 3 SMux instances.

In fact, since HMux processes all packets in the *data plane* of the switch, it can handle packets at line rate, and no queue buildup will occur till we exceed the link capacity (10Gbps in this experiment).

3.7.2 HMux Failure Mitigation

One of the most important benefits of using the SMux as a backstop is automatic failure mitigation, as described in §3.5. In this experiment, we investigate the delay involved in failing over from an HMux to an SMux. This delay is important because *during* failover, traffic to the VIP gets disrupted.

We assign 7 VIPs across HMuxes and the remaining 3 to the SMuxes. We fail one switch at 100 msec. We measure the impact of HMux failure on VIP availability by monitoring the ping latency to all 10 VIPs every 3ms.

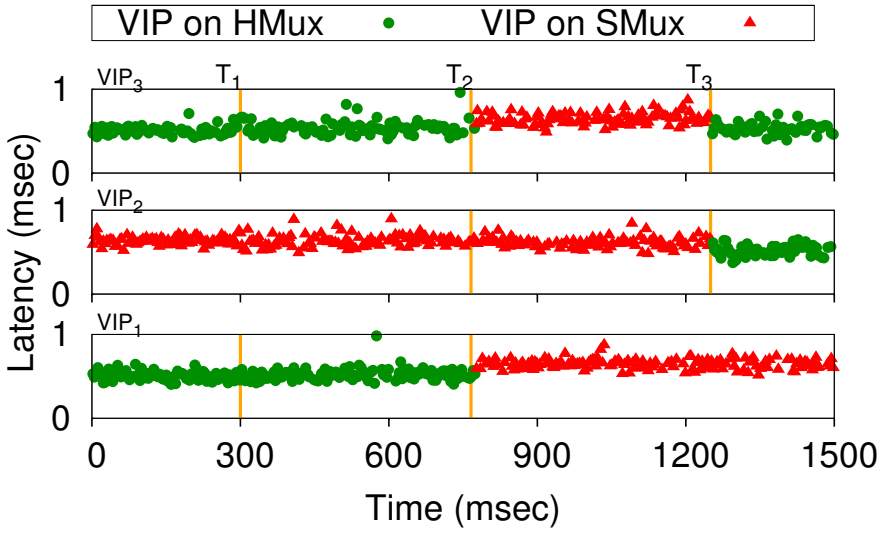


Fig. 3.13. VIP availability during migration.

Figure 3.12 shows the ping latency for three VIPs: (1) One on the failed HMux (VIP_3), (2) One on a healthy HMux (VIP_2), and (3) One on an SMux (VIP_1), respectively.

We make three observations: (1) The traffic to VIP_3 falls over to SMux within 38 msec after HMux failure. The delay reflects the time it takes for other switches to detect the failure, and for the routing to converge. The VIP was not available during this period, *i.e.,* there is no response to pings. (2) After 38 msec, pings to VIP_3 are successful again. (3) The VIPs assigned to other HMuxes and SMuxes are not affected; their latency is unchanged during HMux failure. These observations demonstrate the effectiveness of using SMux as a backstop in the PIKACHU design.

3.7.3 VIP Migration

Recall that we also use SMux as a backstop during VIP migration. We now investigate the delays involved in this process. This delay is important because it places a lower bound on how quickly PIKACHU can react to network conditions.

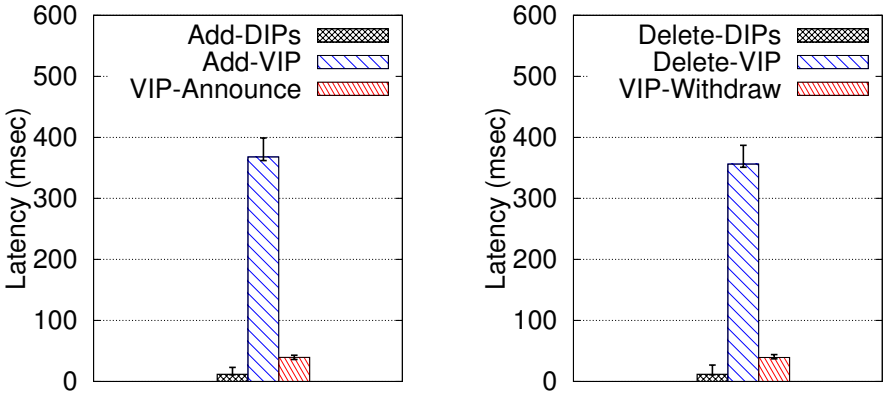
In this experiment, we assign 7 VIPs to the HMuxes and the remaining 3 VIPs to the SMuxes. We migrate a VIP from HMux-to-SMux (VIP_1), SMux-to-HMux (VIP_2), and HMux-to-HMux through SMux (VIP_3) at different times. We measure the VIP availability by monitoring the ping latency (every 3ms) to these VIPs, and we also measure the migration delay.

Figure 3.13 shows the ping latency. At time T_1, the controller starts the first wave of migration by sending the migrate command (migrate to SMuxes) to the corresponding switch agents for VIP_1 and VIP_3. It takes about 450ms for the migration to finish (time T_2), at which time, the controller sends another migrate command (migrate back to HMux) to VIP_2 and VIP_3, which takes about 400ms to take effect (time T_3). We see that that all three VIPs remain fully available during the migration process. The VIPs see a very slight increase in latency when they are on SMux, due to software processing of packets on SMux.

Note that unlike the failure scenario discussed earlier, during the migration process, there is no "failure detection" involved. This is why we see no ping packet loss in Figure 3.13.

Figure 3.14 shows the three components of the migration delay: (1) latency to add/delete a VIP as measured from the time the controller sends the command to the time other switches receive the BGP update for the operation, (2) latency to add/delete DIPs as measured similarly as the VIPs, (3) latency for the BGP update (routing convergence), measured as the time from the VIP is changed in the FIB on one switch till the routing is updated in the remaining switches, *i.e.,* BGP update time on those switches.

Almost all (80-90%) of the migration delay is due to the latency of adding/removing the VIP to/from the FIB. This is because our implementation of the switch agent is not fully optimized – improving it is part of our future work.



(a) Add (b) Delete

Fig. 3.14. Latency breakdown.

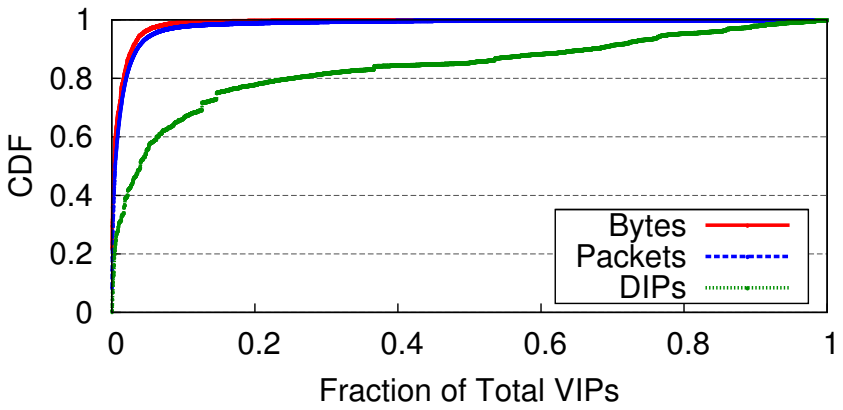


Fig. 3.15. Traffic and DIP distribution.

3.8 Evaluation

In this section, we use large-scale simulations to show that: (1) Pikachu needs far fewer SMuxes than Ananta to load balance the same amount of VIP traffic; (2) Despite using fewer SMuxes (and hence being cheaper), Pikachu incurs low latency on load balanced traffic; (3) The VIP assignment algorithm is effective; (4) Network component failures do not cause significant congestion, even though Pikachu's VIP assignment algorithm is oblivious to network component failures; (5) The migration algorithm is effective.

3.8.1 Simulation Setup

Network: Our simulated network closely resembles that of a production datacenter, with a FatTree topology connecting 50k servers connected to 1600 ToRs located in 40 containers. Each container has 40 ToRs and 4 Agg switches, and the 40 containers are connected with 40 Core switches. The link and switch memory capacity were set with values observed in production datacenters: routing table and tunneling table sizes set to 16k and 512, respectively, and the link capacity set to 10Gbps between ToR and Agg switches, and 40 Gbps between Agg and Core switches.

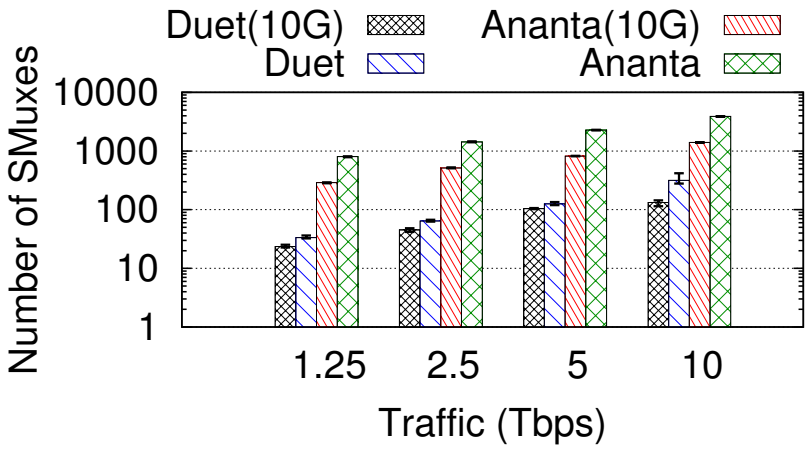


Fig. 3.16. Number of SMuxes used in Duet and Ananta.

Workload: We run the simulations using the traffic trace collected from one of our production datacenters. The trace consists of 30K VIPs, and the number of DIPs and the traffic distribution across the VIPs are shown in Figure 3.15. We divide the 3-hour trace into 10-minute intervals, and calculate the VIP assignment in each interval, based on the traffic demand matrix (the number of bytes sent and received between all sources and destinations), the topology and the forwarding tables.

3.8.2 SMux Reduction

We first compare the number of SMuxes needed in Pikachu and Ananta to load-balance same amount of traffic in the datacenter.

We calculate the number of SMuxes needed by Ananta such that no SMux receives traffic exceeding its capacity. We consider two SMux capacities: 3.6Gbps, as observed on the production SMuxes (§3.2), and 10Gbps, assuming the CPU will not be a bottleneck.

The number of SMuxes needed for Duet depends on the capacity of SMux, the traffic generated by VIPs that could not be assigned to HMuxes, and specifics of failure model, and migration probabilities (§3.3.3). In this experiment, we assign the VIPs to HMuxes using the algorithm described in §3.4, which tries to assign as many

VIPs to HMuxes as it can, subject to switch memory and link bandwidth constraints. We have specified the memory and bandwidth details earlier.

Based on failure scenarios in [25,26], we provision the number of SMuxes to handle the maximum traffic under either (1) entire container failure, or (2) three random switch failures. For example, if an entire container fails, the total traffic T to all the VIPs assigned to the switches inside need to fail over to SMuxes. Thus the number of SMuxes needed is $\frac{T}{C_{smux}}$ where C_{smux} is SMux capacity.

We ignore migration – it is covered in §3.8.6.

Figure 3.16 shows that PIKACHU requires far fewer SMuxes compared to Ananta at all traffic rates. *Note the log scale on Y axis.* For all the traffic rates, PIKACHU was able to assign 16k VIPs to the HMuxes (routing table limit). Overall, compared to Ananta, DUET requires 12-24x times fewer SMuxes when the SMux capacity is 3.6 Gbps and 8-12x times fewer SMuxes when the SMux capacity is 10Gbps, across different traffic loads.

We note that for all traffic scenarios, majority of the SMuxes needed by DUET were needed to handle failure. The fraction of SMuxes needed to handle the traffic to the VIPs that could not be assigned to the HMux is small. This shows that the VIP assignment algorithm does a good job of "packing" VIPs into HMuxes.

3.8.3 Latency vs. SMuxes

Another way to look at the trade-off described in §3.8.2 is to hold the traffic volume constant, and see how many SMuxes Ananta needs to provide the same latency as DUET. This is shown in figure 3.17.

We hold the traffic at 10Tbps, and vary the number of SMuxes for Ananta from 2000 to 15,000. The black line shows median latency for Ananta. The red dot represents DUET. DUET used 230 SMuxes, and achieved median latency of 474 μsec.

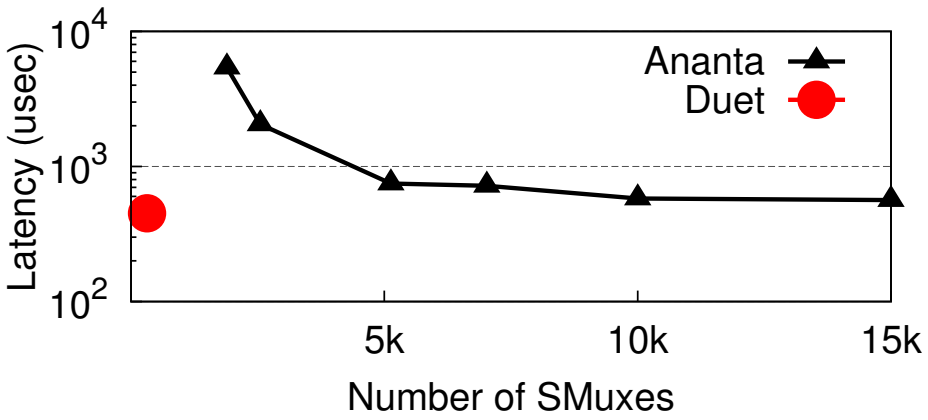


Fig. 3.17. Latency (microseconds) vs. number of SMuxes in Ananta and PIKACHU.

We see that if Ananta were to use the same number of SMuxes as DUET (230), the median latency would be many times higher (over 6 ms). On the other hand, Ananta needs 15,000 SMuxes to achieve latency comparable to DUET.

The absolute latency numbers may appear small – however, recall that median DC RTTs are of the order of 381 μsec[6], and in many cases, to satisfy a single user request, an application like Search traverses load balancer multiple times. Any time lost in the network is *wasted* time – which could have otherwise been used by the application to improve user experience [27–29].

3.8.4 Duet vs. Random

To understand the impact of assigning VIPs based on the maximum resource utilization, we compare the performance of PIKACHU in terms of the number of SMuxes against a random strategy (Random) that selects the *first feasible* switch that does not violate the link or switch memory capacity. This assignment algorithm can be viewed as a variant of FFD (First Fit Decreasing) as the VIPs are assigned in the sorted order of decreasing traffic volume.

[6]Newer technologies such a RDMA lower this to 2-5 μsec!

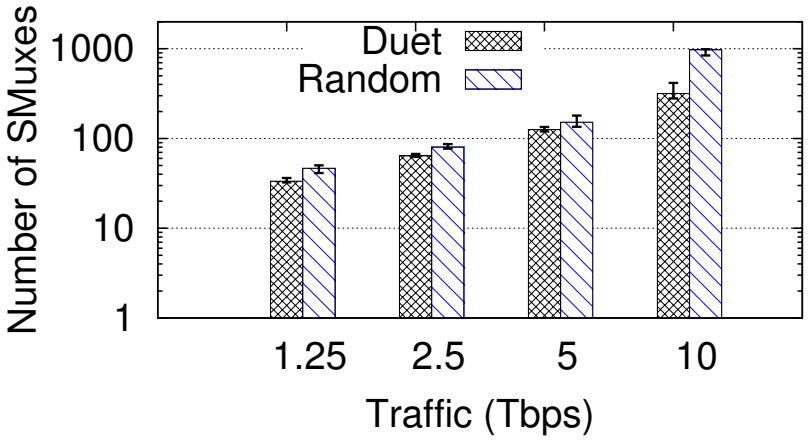


Fig. 3.18. Number of SMuxes used by Duet and Random.

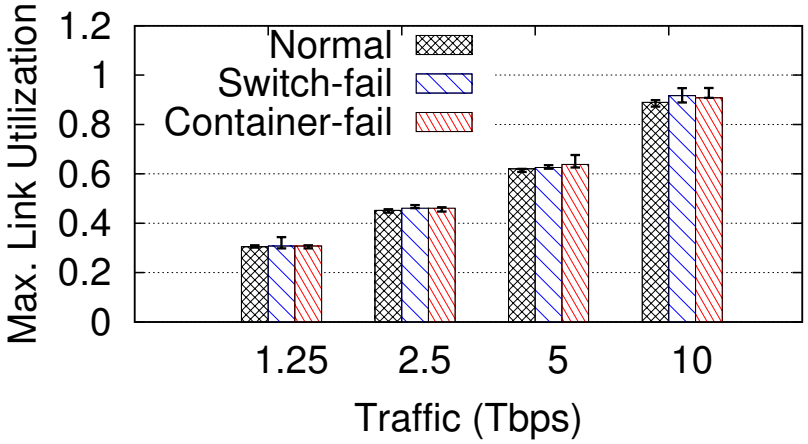


Fig. 3.19. Impact of failures on max. link utilization.

Figure 3.18 shows the total number of SMuxes needed by Pikachu and Random (note the log scale). We see that Random results in 120%–307% more SMuxes compared to Pikachu as the traffic load varies from 1.25 to 10 Tbps. This shows that by taking resource utilization into account, Duet ensures that only a small fraction of VIPs traffic is left to be handled by the SMuxes.

3.8.5 Impact of Failure

Microbenchmark results in §3.7.2 showed that Duet can handle HMux failures well – the VIPs fall back to SMux, and the disruption to the VIP traffic is minimal.

In §3.8.2, we considered the number of SMuxes DUET needs to cope with failures. We now consider the bigger picture – what impact does failures of several switches, or even a container have on overall traffic?

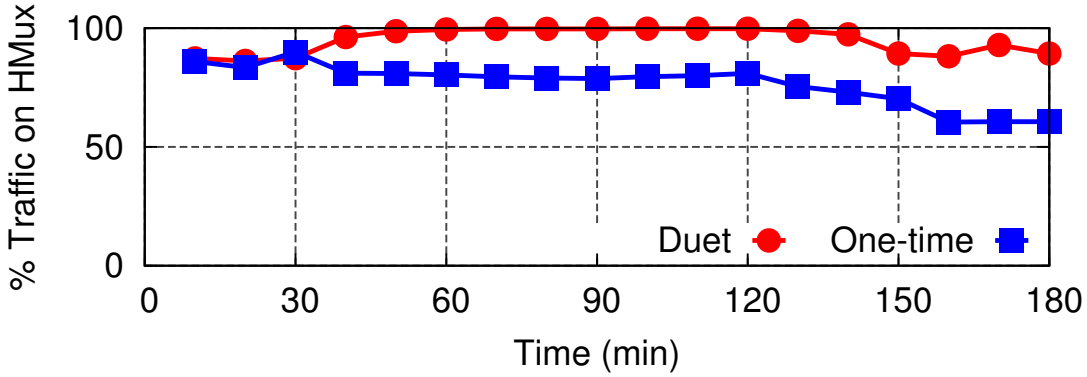
We consider the same failure model as was used in §3.8.2 – a container or up to 3 switches can fail simultaneously. We evaluate failure resilience of PIKACHU by measuring the maximum link utilization under these two scenarios: failure of a randomly selected container, or 3 randomly selected switches.

A random switch failure affects link traffic load in two ways. It causes the traffic of the VIPs assigned to the failed switch to be shifted to the backstop SMuxes, and other through traffic to be shifted to the alternative path. A container failure affects the traffic in more complicated ways: it not only causes all the switches inside to be disconnected, but also makes all the traffic with sources and destinations (DIPs) inside to disappear.

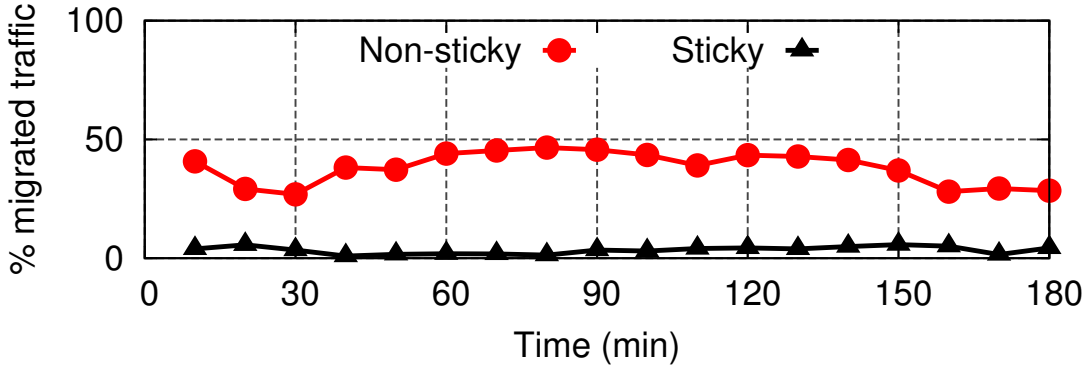
Figure 3.19 shows the measured maximum link utilization during the two failure scenarios in the 10 experiments. We see that as expected, link failures can result in transient congestion. However, the utilization increase of any link in the network is no more than 16%, and hence is comfortably absorbed by the 20% bandwidth reservation made in the VIP assignment algorithm. Interestingly, the single container failure (with 44 switches inside) often results in less congestion than 3-switch failure. This can be explained by two reasons: (1) any traffic with source and sinks (DIPs) inside the container has disappeared, and (2) all the rest traffic which have sources or sinks outside the container are not shifted to other paths as their paths do not go through any switch inside the container.

3.8.6 VIP Migration

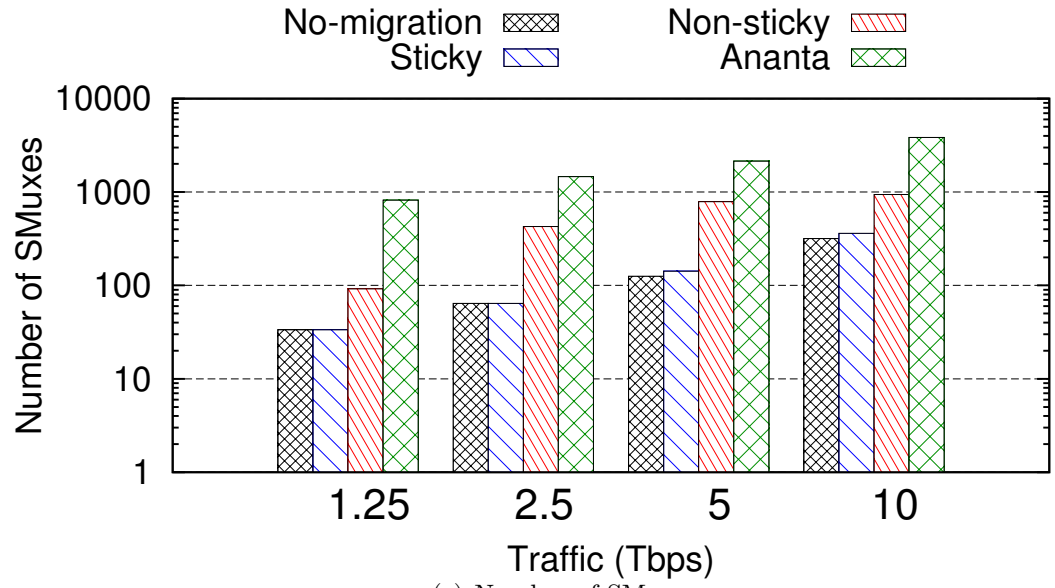
In this section, we evaluate the effectiveness of PIKACHU 's VIP migration algorithm, *Sticky* (§3.4.2). We set the threshold to be $\delta = 0.05$, *i.e.,* a VIP will migrate to a new assignment only if doing so reduces the MRU by 5%.



(a) Traffic load-balanced by HMux



(b) Traffic shuffled during migration



(c) Number of SMux

Fig. 3.20. Effectiveness of different migration algorithms.

We compare *Sticky* with *Non-sticky*, which calculates the new assignment from scratch based on current traffic matrix (§3.4.1), but migrates all the VIPs at the same time through SMuxes to avoid the memory deadlock problem. We evaluate these two schemes by re-running the 3-hour traffic trace, where we reassign and migrate the VIPs for *Sticky* and *Non-sticky* every 10 minutes. The total VIP traffic varies between 6.2 to 7.1 Tbps in this trace.

Effectiveness: We first compare the portion of total traffic that are handled by the HMuxes under the two assignment schemes – the larger the portion, the more effective the assignment algorithm. Here, we also compare *Sticky* and *Non-sticky* against *One-time* algorithm, which assigns the VIPs at time 0 sec, and never change it. Figure 3.20(a) shows the results over the duration of the trace. First, as expected, while the portion of traffic handled by HMuxes started out the same, the initial assignment which is used by *One-time* throughout the trace, gradually loses its effectiveness, and results in only 60-89% (average 75.2%) of the total being handled by HMuxes. In contrast, *Sticky* and *Non-sticky* handle 86-99.9% (average 95.3%) of the traffic in HMuxes, from continuously adapting to the traffic dynamics. Second, even though *Sticky* only migrates VIPs that reduce the MRU by at least 5%, it is as effective as *Non-sticky* in maximizing the traffic assigned to HMuxes. In particular, it handles 86-99.7% traffic (average 95.1%) in HMuxes, which is almost identical to the 87-99.9% traffic (average 95.67%) handled by HMuxes under *Non-sticky*.

Traffic shuffled: Next, we compare the fraction of the total VIP traffic migrated under *Sticky* and *Non-sticky*– the less traffic are migrated, the fewer SMuxes need to be reserved as stepping stone. Figure 3.20(b) shows that migration using *Non-sticky* results in reshuffling almost 25-46% (average 37.4%) of the total VIP traffic each time throughout the trace duration, compared to only 0.7-4.4% (average 3.5%) under *Sticky*. Such a drastic reduction in the traffic shuffled under *Sticky* is attributed to its simple filtering scheme: a VIP is only migrated if it improves the MRU by 5%.

Number of SMuxes: Figure 3.20(c) shows the number of SMuxes needed by *Sticky* and *Non-sticky*. Additionally, we also calculate the SMuxes needed without

migration (marked as No-migration) as well as number of SMuxes needed in Ananta considering the SMux capacity to 3.6Gbps. The number of SMuxes needed in *Sticky* and *Non-sticky* is calculated as maximum of SMuxes needed for VIP traffic, failure and transition traffic. It can be seen that, *Non-sticky* always requires more SMuxes compared to No-migration and *Sticky*, showing that *Sticky* does not increase the number of SMuxes to handle the traffic during migration.

3.9 Discussion

Why are there empty entries in switch tables? PIKACHU uses empty entries in the host table, ECMP table, and tunneling table in switches to implement HMux. Several reasons contribute to the abundance of such free resources in our production datacenter. The host table of ToR switches has only a few dozen entries for the hosts within each rack, and that of the rest of the switches is mostly empty. The ECMP table of switches is mostly empty because of the hierarchical DC network topology, where each switch has a small number of outgoing links among which all outgoing traffic is split via ECMP. The tunneling table is mostly free since few online services use encapsulation other than load balancing itself. We acknowledge that other DCs may have a different setup, but we believe that our design will be applicable in common cases.

VIP assignment: While the greedy VIP assignment algorithm described in §3.4 works well in our scenarios, we believe that it can be improved. The VIP assignment problem resembles bin packing problem, which has many sophisticated solutions. We plan to study them in future. Also, while we consider VIPs in order of traffic, other orderings are possible (*e.g.,* consider VIPs with latency sensitive traffic first).

Failover and Migration: DUET relies on SMuxes to simplify failover and migration. As hinted in §3.3.3, it may be possible to handle failover and migration by replicating VIP entries in multiple HMuxes. We continue to investigate this approach,

although our initial exploration shows that the resulting design is far more complex than our current design.

3.10 Related Work

To the best of our knowledge, DUET is a novel approach to building a performant, low-cost, organically scalable load balancer. We are not aware of any load balancing architecture that fuses switch-based load balancer with the software load balancers. However, there has been much work on load balancers, and we briefly review it here.

Load balancer: Traditional hardware load balancers [20, 21] are expensive and typically only provide 1+1 availability. DUET is much more cost effective, and provides enhanced availability by using SMuxes as a backstop. Importantly, compared to traditional load balancers, PIKACHU gives us control over very important vantage point in our cloud infrastructure.

We have already discussed Ananta [1] software load balancer extensively. Other software-based load balancers [30–32] are also available, but they lack the scalability and availability of Ananta, as shown in [1]. Embrane [33] promises scalability, but suffers from the same fundamental limitations of the software load balancer.

OpenFlow based load balancer: Two recent proposals focus on using OpenFlow switches for load balancing. In [34], authors present a preliminary design for a load balancing architecture using OpenFlow switches. They focus on minimizing the number of wildcard rules. The paper, perhaps because it is a preliminary design, ignores many key issues such as handling switch failures. Plug-n-Serve [35] is another preliminary design that uses OpenFlow switches to load balance web servers deployed in unstructured, enterprise networks. DUET is very different from these approaches. DUET uses a combined hardware and software approach. DUET does not rely on OpenFlow support. DUET is designed for data center networks, and pays careful attention to handling numerous practical issues including various types of failures and VIP migration to adapt to network dynamics.

Partitioning OpenFlow rules: Researchers have also proposed using OpenFlow switches for a variety of other purposes. For example, DIFANE [36] uses some switches in the data center to cache rules, and act as authoritative switches. While a load balancing architecture can be built on top of DIFANE, the focus of the paper is very different from DUET. In vCRIB [37] authors propose to offload some of the traffic management rules from host agent to ToR switches, as well as to other host agents. Their goal is to ensure resource-aware and traffic-aware placement of rules. While vCRIB also faces problems such as managing network dynamics (*e.g.,* VM migration), their main focus is quite different than DUET.

SDN architecture and middleboxes: Similar to PIKACHU , researchers have leveraged SDN architecture in the context of middleboxes to achieve policy enforcement and verification [38, 39], which is again a different goal than PIKACHU .

Improving single server performance: Researchers have substantially improved packet processing capabilities on commodity servers [40, 41], which could potentially improve SMux performance. But, these improvements are unlikely to bridge the differences in packet processing capabilities between HMux and SMux for the load balancer workload.

Lastly, several algorithms for calculating flow hashes (*e.g.,* resilient hashing [24], cuckoo-hashing [40]) offer a wide variety of trade-offs. We do not review them here, although DUET can leverage any advances in this field.

3.11 Conclusion

PIKACHU is a new distributed hybrid load balancer designed to provide high capacity, low latency, high availability, and high flexibility at low cost. The PIKACHU design was motivated by two key observations: (1) software load balancers offer high availability and high flexibility but suffer high latency and low capacity per load balancer, and (2) commodity switches have ample spare resources and now also support programmability needed to implement load balancing functionality. The PIKACHU

architecture seamlessly integrates the switch-based load balancer design with a small deployment of software load balancer. We evaluate DUET using a prototype implementation and extensive simulations using traces from our production DC. Our evaluation shows that DUET provides 10x more capacity than a software load balancer, at a fraction of its cost, while reducing the latency by over 10x, and can quickly adapt to network dynamics including failures.

Acknowledgements

We thank the members from Microsoft Azure team, especially Chao Zhang, for their help in shaping PIKACHU. We also thank the reviewers and our shepherd Ali Ghodsi for their helpful feedback.

4. FUTURE WORK

In this chapter, we list the avenues that we shall pursue in the future work.

4.1 Reducing network bandwidth overhead of load balancer

Although Duet enables low cost and high performance cloud scale load balancer, it also incurs high bandwidth usage of the DC network because of the intrinsic nature of traffic redirection. First, even if the traffic source and the DIPs that handle the traffic are in the same ToR, the traffic first has to be routed to the Muxes, which may be faraway and elongate the path traveled by the traffic. Second, in both Ananta and Duet, the Muxes select DIPs for a VIP by hashing the five-tuple of IP headers, and hence are oblivious to DIP locations. As a result, even if the Mux and some DIPs are located nearby the source, the traffic can be routed to faraway DIPs in the DC, again traversing longer paths. Lastly, these designs do not leverage the server location flexibility in placing the DIPs closer to the sources to shorten the path. Such traffic overhead leads to high bandwidth usage of the DC network which not only requires the DC operator to provision high network bandwidth which is costly, but also makes the network more prone to transient congestion which affect latency-sensitive services.

In the future work, we plan to design load balancer architecture and algorithms to reduce the network balandwidth overheads while providing the low cost and high performance benefits. We plan to complete the design by June 2015.

4.2 Cloud scale layer 7 load balancing

Layer 4 (L4) load balancing (LB) is a basic load balancing mechanism supported in today's public cloud. As online services grow complex, they require content based

switching, *i.e.,* the traffic on the same VIP is split across different sets of servers based on the HTTP content, which cannot be done by an L4 LB. L7 load balancing enables such fine-grained content-based switching, and is a vital building block of many online services [30].

First, only a subset of the cloud providers provide L7 LB as a service. These cloud providers implement scale-out L7 LB in software running on commodity servers, where each server handles subset of the user requests. These L7 LB designs use proxy-like mechanism, where for a client request, each L7 LB instance first establishes a TCP connection with the client to get the HTTP request, then selects the server and starts a new connection with the server. This design choice exhibits limited availability where an LB instance failure results in user-visible connection resets of all connections handled by the failed LB instance. As a result, client flows break which significantly hampers user experience and online service revenue.

In the future work, we plan to design scalable, highly available L7 LB that cloud providers (or 3^{rd} party vendors) can use to provide L7 LB as a service. We plan to complete the design by November 2015.

REFERENCES

REFERENCES

[1] P. Patel *et al.*, "Ananta: Cloud scale load balancing," in *SIGCOMM*, 2013.

[2] "Amazon ec2," aws.amazon.com/ec2/.

[3] M. Zaharia *et al.*, "Improving mapreduce performance in heterogeneous environments," in *OSDI'08*.

[4] B. Farley *et al.*, "More for your money: exploiting performance heterogeneity in public clouds," in *SoCC '12*.

[5] C. Reiss *et al.*, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *SoCC '12*.

[6] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *OSDI'04*.

[7] G. Ananthanarayanan *et al.*, "Why let resources idle? aggressive cloning of jobs with dolly," in *HotCloud'12*.

[8] ——, "Reining in the outliers in map-reduce clusters using mantri," in *OSDI'10*.

[9] M. Zaharia *et al.*, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *EuroSys '10*.

[10] E. Bortnikov *et al.*, "Predicting execution bottlenecks in map-reduce clusters," in *HotCloud'12*.

[11] F. Ahmad *et al.*, "Tarazu: optimizing mapreduce on heterogeneous clusters," in *ASPLOS '12*.

[12] "Hadoop," http://lucene.apache.org/hadoop.

[13] "Facebook hive," http://hadoop.apache.org/hive.

[14] C. Olston *et al.*, "Pig latin: a not-so-foreign language for data processing," in *SIGMOD '08*.

[15] "Apache mahout: Scalable machine learning and data mining," http://mahout.apache.org.

[16] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, "Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language," in *OSDI'08*.

[17] "X-rime: Hadoop based large scale social network analysis." http://xrime.sourceforge.net/.

[18] G. Lee *et al.*, "Topology-aware resource allocation for data-intensive workloads," in *APSys '10*.

[19] ——, "Heterogeneity-aware resource allocation and scheduling in the cloud," in *HotCloud'11*.

[20] "A10 networks ax series," http://www.a10networks.com.

[21] "F5 load balancer," http://www.f5.com.

[22] P. Bodík, I. Menache, M. Chowdhury, P. Mani, D. A. Maltz, and I. Stoica, "Surviving failures in bandwidth-constrained datacenters," in *SIGCOMM, 2012*.

[23] C. Chekuri and S. Khanna, "On multi-dimensional packing problems," in *SODA, 1999*.

[24] "Broadcom smart hashing," http://www.broadcom.com/collateral/wp/StrataXGSSmartSw
WP200-R.pdf.

[25] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," in *ACM SIGCOMM CCR, 2011*.

[26] X. Wu, D. Turner, C.-C. Chen, D. A. Maltz, X. Yang, L. Yuan, and M. Zhang, "Netpilot: automating datacenter network failure mitigation," *ACM SIGCOMM CCR, 2012*.

[27] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *SIGCOMM*, 2010.

[28] J. Hamilton, "The cost of latency," http://perspectives.mvdirona.com/2009/10/31/TheCost

[29] L. Ravindranath, J. padhye, R. Mahajan, and H. Balakrishnan, "Timecard: Controlling User-Perceieved Delays in Server-based Mobile Applications," in *SOSP*, 2013.

[30] "Ha proxy load balancer," http://haproxy.1wt.eu.

[31] "Loadbalancer.org virtual appliance," http://www.load-balancer.org.

[32] "Netscalar vpx virtual appliance," http://www.citrix.com.

[33] "Embrane," http://www.embrane.com.

[34] R. Wang, D. Butnariu, and J. Rexford, "Openflow-based server load balancing gone wild," in *Usenix HotICE*, 2011.

[35] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-serve: Load-balancing web traffic using openflow," *ACM SIGCOMM Demo*, 2009.

[36] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with difane," in *SIGCOMM*, 2010.

[37] M. Moshref, M. Yu, A. Sharma, and R. Govindan, "Scalable rule management for data centers," in *NSDI*, 2013.

[38] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "Simple-fying middlebox policy enforcement using sdn," in *SIGCOMM, 2013*.

[39] S. Fayazbakhsh, V. Sekar, M. Yu, and J. Mogul, "Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions," *Proc. HotSDN*, 2013.

[40] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, high performance ethernet forwarding with cuckooswitch," in *CoNext, 2013*.

[41] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks: Exploiting parallelism to scale software routers," in *SOSP, 2009*.