PIKACHU: How to Rebalance Load in Optimizing MapReduce On Heterogeneous Clusters

Rohan Gandhi, Di Xie, Y. Charlie Hu
Purdue University

Abstract

For power, cost, and pricing reasons, datacenters are evolving towards heterogeneous hardware. However, MapReduce implementations, which power a representative class of datacenter applications, were originally designed for homogeneous clusters and performed poorly on heterogeneous clusters. The natural solution, rebalancing load among the reducers running on heterogeneous nodes has been explored in Tarazu, but shown to be only mildly effective.

In this paper, we revisit the key design challenge in this important optimization for MapReduce on heterogeneous clusters and make three contributions. (1) We show that Tarazu 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% and Tarazu by up to 23%.

1 Introduction

For power, cost, and pricing reasons, datacenters have evolved towards heterogeneous hardware. For example, different hardware generations exist in Amazon EC2 [1] 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 [18, 11, 15].

MapReduce, a high-level programming model for data-intensive applications [10], has been widely adopted in cloud datacenters such as Google, Yahoo, Microsoft, and Facebook [7, 8, 17, 9], to power a significant portion of applications. However, the numerous MapReduce implementations have been designed and optimized for homogeneous clusters. A recent study [6] 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 [6], 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 [6] 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.

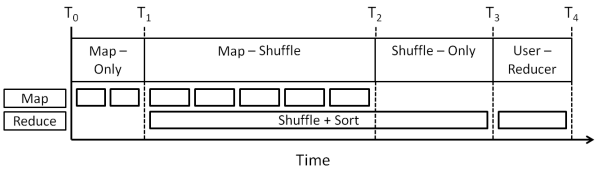


Figure 1: Four stages and their concurrency in MapReduce job execution.

Finally, we implement our new load rebalancing scheme in the PIKACHU task scheduler, and experimentally show it substantially outperforms Tarazu and Hadoop, reducing the job completion time by up to 23% and 42%, respectively, for a diverse set of benchmarks and cluster configurations.

2 Background

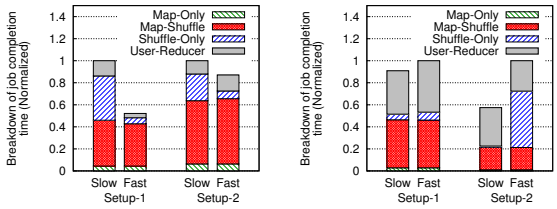
The execution of a MapReduce job is broken down by the runtime system into many Map tasks and Reduce tasks (called reducers hereafter) running in parallel on different nodes of the cluster. A reducer consists of three types of subtasks: (1) shuffle, (2) sort, and (3) user-defined reduce function. Every 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 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 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 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.

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.

[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.



(a) Wordcount (b) Sort

Figure 2: Job completion time breakdown (normalized to total time) for Wordcount and Sort.

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 [6]. 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.

3.2 Impact of Heterogeneous Nodes

Figure 2 shows the execution time and their breakdown into the four stages discussed in §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(a) Config-1, each fast node in total shuffles out 7 times more data than each slow node.

(3) **Shuffle-Only:** Figure 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(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(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 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.

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.

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 [6]. 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 [6]. 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.

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

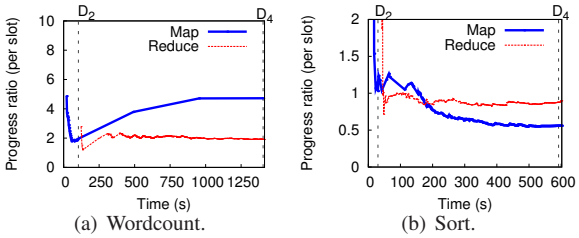
D_2: At start of the Map-Shuffle stage (T_1). At T_1, since the reduce tasks have not started, *P* can only be estimated using the relative progress rates of map tasks (so far) on fast and slow nodes, *i.e.,* $P = \frac{S_{fa,m}}{S_{sl,m}}$, where $S_{fa,m}$ and $S_{sl,m}$ are the progress rates for map tasks on fast and slow nodes. This method is used in Tarazu [6].

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 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 ap-

[2] T_0 to T_4 are marked in Figure 1.

[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.



(a) Wordcount. (b) Sort.

Figure 3: Ratio of progress rates of map and reduce tasks on fast and slow nodes.

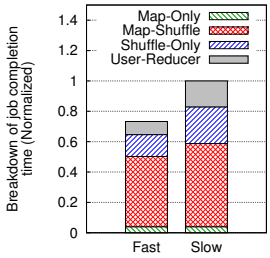


Figure 4: Job execution time and breakdown of Wordcount under D_3 (P=2).

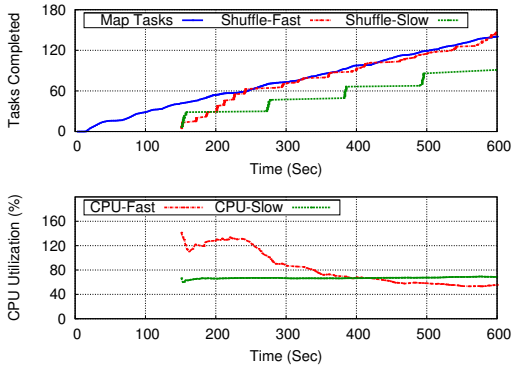


Figure 5: Shuffling progress and CPU utilization by each reducer on fast and slow nodes in D_3 (calculated P=2).

proximation to the 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.

4.3 Design Refinement

We implemented D_3 (details in §5, the calculated P=2 from Figure 3(a)) and reran Wordcount. The new execution time breakdowns, shown in Figure 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 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 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 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

$$P = \frac{S_{fa}}{S_{sl}} * \frac{1}{E_{fa}} \quad (1)$$

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 6 bottom) to the CPU utilization when shuffle (on fast nodes) has caught up with map tasks completed (T_s in Figure 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 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.

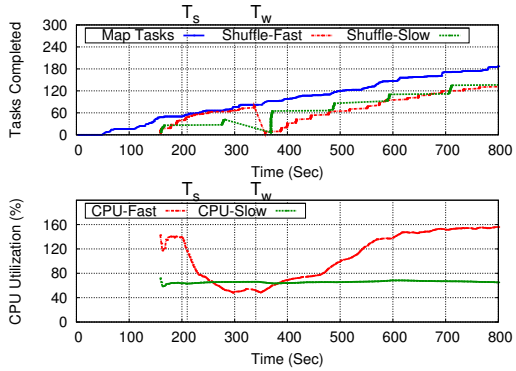


Figure 6: Shuffling progress and CPU utilization by each reducer on fast and slow nodes under D_3'.

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

5 Implementation

We implemented our new load rebalancing scheme D_3' in Hadoop v0.20.203.0 [4] 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.

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

$$V_s = \frac{10 \cdot (N_s + N_f)}{N_s + P \cdot N_f}, V_f = \frac{10 \cdot (N_s + N_f) \cdot P}{N_s + P \cdot N_f} \quad (2)$$

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.

6 Evaluation

We also implemented Tarazu [6] 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 Self-join* [6]. *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.

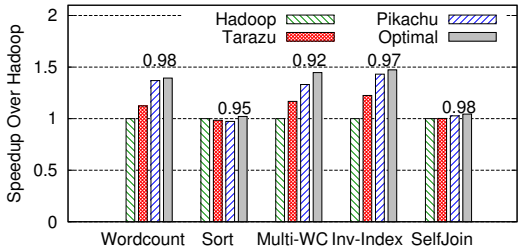


Figure 7: Speedup of of Tarazu and PIKACHU over Hadoop, under Config-1.

Performance on Local Cluster. Figure 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-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.

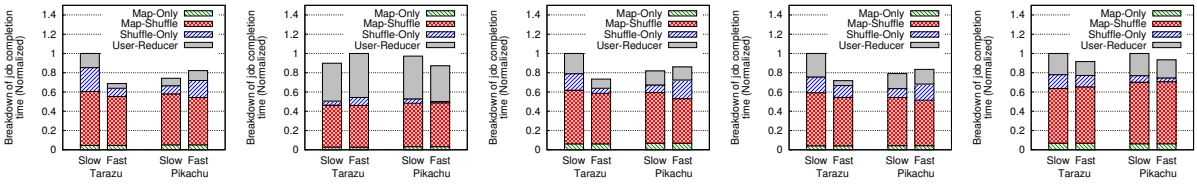
Table 1 summarizes the partition ratios calculated by Tarazu (T), PIKACHU (P) and Optimal (O). The partition ratio calculated using PIKACHU is closer to Optimal compared to Tarazu. Table 1 also shows the overhead incurred by PIKACHU, measured as the extra data shuffled by all the nodes in PIKACHU compared to Hadoop. We see PIKACHU incurs a low overhead 0.96-4.75% in re-shuffling and re-sorting.

Figure 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) 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

Table 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-Overhead	3.86%			4.13%			4.58%			4.75%			0.96%		



(a) Wordcount (b) Sort (c) Multi-Wordcount (d) Inverted Index (e) Selfjoin

Figure 8: Job completion time breakdown on fast and slow nodes (Normalized to Tarazu).

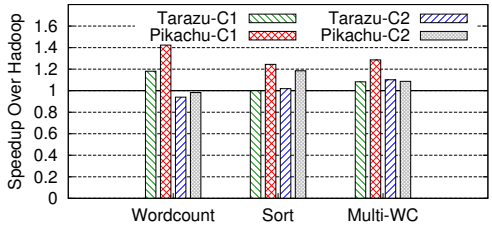


Figure 9: Speedup of Tarazu and PIKACHU over Hadoop on the 60-node EC2 cluster on 2 configurations.

180 GB data (2880 map tasks). Figure 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.

7 Related Work

Many implementations, extensions, and domain specific libraries of MapReduce have been developed to support large-scale data processing [3, 4, 14, 2, 16, 5]. None of them explicitly study optimizing MapReduce execution on heterogeneous hardware. LATE [18] 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 [13, 12] and proposed a fair scheduler [12] 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 [6] has already been discussed previously.

8 Conclusion

We showed that the prior-art MapReduce scheduler for heterogeneous clusters, Tarazu, poorly balances the load among reducers on fast and slow nodes. We proposed PIKACHU, which strikes a balance between accuracy and overhead in estimating the load adjustment and doubles Tarazu's improvement over Hadoop. We have released PIKACHU at `http://github.com/mapreduce-pikachu`.

Acknowledgment. This work was supported in part by NSF grant CNS-1065456.

References

[1] Amazon ec2. `aws.amazon.com/ec2/`.
[2] Apache mahout: Scalable machine learning and data mining. `http://mahout.apache.org`.
[3] Facebook hive. `hadoop.apache.org/hive`.
[4] Hadoop. `http://lucene.apache.org/hadoop`.
[5] X-rime: Hadoop based large scale social network analysis. `http://xrime.sourceforge.net/`.
[6] F. Ahmad, *et al.* Tarazu: optimizing mapreduce on heterogeneous clusters. In *ASPLOS '12*.
[7] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Why let resources idle? aggressive cloning of jobs with dolly. In *HotCloud'12*.
[8] G. Ananthanarayanan, *et al.* Reining in the outliers in map-reduce clusters using mantri. In *OSDI'10*.
[9] E. Bortnikov, *et al.* Predicting execution bottlenecks in map-reduce clusters. In *HotCloud'12*.
[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04*.
[11] B. Farley, *et al.* More for your money: exploiting performance heterogeneity in public clouds. In *SoCC '12*.
[12] G. Lee, *et al.* Heterogeneity-aware resource allocation and scheduling in the cloud. In *HotCloud'11*.
[13] G. Lee, *et al.* Topology-aware resource allocation for data-intensive workloads. In *APSys '10*.
[14] C. Olston, *et al.* Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08*.
[15] C. Reiss, *et al.* Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC '12*.
[16] Y. Yu, *et al.* Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI'08*.
[17] M. Zaharia, *et al.* Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys '10*.
[18] M. Zaharia, *et al.* Improving mapreduce performance in heterogeneous environments. In *OSDI'08*.