Tarazu: Optimizing MapReduce On Heterogeneous Clusters

Faraz Ahmad[†], Srimat Chakradhar[‡], Anand Raghunathan[†], T. N. Vijaykumar[†]
[†]School of Electrical and Computer Engineering, Purdue University, IN, USA
[‡]NEC Laboratories America, Princeton, NJ, USA
{fahmad,raghunathan,vijay}@ecn.purdue.edu chak@nec-labs.com

Abstract

Data center-scale clusters are evolving towards heterogeneous hardware for power, cost, differentiated price-performance, and other reasons. MapReduce is a well-known programming model to process large amount of data on data center-scale clusters. Most MapReduce implementations have been designed and optimized for homogeneous clusters. Unfortunately, these implementations perform poorly on heterogeneous clusters (e.g., on a 90-node cluster that contains 10 Xeon-based servers and 80 Atom-based servers, Hadoop performs worse than on 10-node Xeon-only or 80-node Atom-only homogeneous sub-clusters for many of our benchmarks). This poor performance remains despite previously proposed optimizations related to management of straggler tasks.

In this paper, we address MapReduce's poor performance on heterogeneous clusters. Our first contribution is that the poor performance is due to two key factors: (1) the non-intuitive effect that MapReduce's built-in load balancing results in excessive and bursty network communication during the Map phase, and (2) the intuitive effect that the heterogeneity amplifies load imbalance in the Reduce computation. Our second contribution is *Tarazu*, a suite of optimizations to improve MapReduce performance on heterogeneous clusters. Tarazu consists of (1) *Communication-Aware Load Balancing of Map computation (CALB)* across the nodes, (2) *Communication-Aware Scheduling of Map computation (CAS)* to avoid bursty network traffic and (3) *Predictive Load Balancing of Reduce computation (PLB)* across the nodes. Using the above 90-node cluster, we show that Tarazu significantly improves performance over a baseline of Hadoop with straightforward tuning for hardware heterogeneity.

Categories and Subject Descriptors: D.1.3 [**Software**]: Programming Techniques - *Concurrent Programming*
General Terms: Design; Measurement; Performance
Keywords: Heterogeneous clusters, MapReduce, Shuffle, Load imbalance, Cluster Scheduling.

1 Introduction

Data centers are the compute platforms of choice for the emerging era of cloud computing. Data center-scale clusters are evolving toward heterogeneous hardware — a mix of high-performance and low-power nodes of disparate hardware architectures. Such heterogeneity stems from the diversity in the performance needs faced by the clusters, ranging from the most stringent of service-level requirements (e.g., on-line search and real-time computations) to more lenient requirements (e.g., data analysis, email or batch jobs). Exploiting this diversity to provide differentiated price-performance (e.g., EC2 [2]) and to optimize power [9], real-estate foot print, and cost often leads to a mix of nodes designed for different power-performance points (e.g., Xeons and Atoms). Even when heterogeneity is not introduced by design, the common practice of phased addition of nodes to the cluster over time implies that the nodes are often some hardware generations apart resulting in performance variation (e.g, adding nodes over 5-7 years implies 2-3 generations apart). While heterogeneity could also be due to other factors such as background load variation [37], or to the use of GPUs and other accelerators, we focus on heterogeneous, general-purpose CPU architectures.

MapReduce [10], a programming model for data-intensive applications, greatly improves programmability by providing automatic data management and transparent fault detection and recovery. MapReduce's programmability has led to quick adoption (e.g., Google [10], Yahoo [14], Microsoft [18,35] and Facebook [11]). To date, however, most MapReduce implementations have been designed and optimized for homogeneous clusters (e.g., Hadoop [14]). Unfortunately, these implementations perform poorly on heterogeneous clusters. For example, on a heterogeneous cluster of 10 Xeon-based servers and 80 Atom-based servers, Hadoop performs 20-75% worse than 10-node Xeon-only or 80-node Atom-only homogeneous sub-clusters for six out of our eleven benchmarks. This poor performance occurs despite Hadoop's built-in load-balancing scheduler and straightforward tuning for hardware heterogeneity (e.g., customizing the number of processes per node for each type of node and optimizing the memory buffer sizes).

LATE [37] is a significant effort to optimize MapReduce on heterogeneous clusters, and focuses specifically on straggler management. For fault tolerance, MapReduce identifies stragglers (i.e., tasks that are slow due to a presumed problem with their nodes) and speculatively runs backup copies for faster completion. LATE points out that straggler management results in excessive speculation in the presence of heterogeneity, and proposes techniques to improve straggler management. Unfortunately, LATE (or, in general, straggler management) alone is not sufficient to address hardware heterogeneity which raises other important issues, as described in this paper. As such, LATE is unable to improve MapReduce performance on our heterogeneous cluster.

In this paper, we explain the reasons behind MapReduce's poor performance on heterogeneous clusters and propose *Tarazu*[1], a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASPLOS'12 March 3-7, 2012, London, England, UK.
Copyright © 2012 ACM 978-1-4503-0759-8/12/03...$10.00.

1. An Urdu word that means balance.

suite of optimizations to exploit the full performance potential of such clusters.

Our first contribution is the key finding that the poor performance is due to two key factors: (1) the non-intuitive effect that MapReduce's built-in load balancing of Map computation results in excessive network communication and (2) the intuitive effect that the heterogeneity amplifies the load imbalance in Reduce computation. These factors extend beyond the issues of stragglers and speculative execution.

The Map-side effect has three components: First, the built-in load balancing of Map computation leads to faster high-performance nodes stealing[2] work from slower low-power nodes. The pronounced difference in the nodes' compute capabilities implies that a considerable amount of Map work is stolen. Unfortunately, input data for the stolen work remains at low-power nodes, resulting in numerous *remote* Map tasks at high-performance nodes and greatly increased network load**.** Note that the stolen tasks are simply the tasks that have not begun execution — not stragglers, which are slow-running tasks. Skewing the input data distribution towards high-performance nodes to convert remote tasks into local tasks incurs many problems, as explained in Section 2. This remote task problem does not exist in homogeneous clusters which are naturally load balanced, and hence, run only a few remote tasks. Second, the remote Map tasks compete with the Shuffle — the all-to-all communication from Map tasks to Reduce tasks — for the network bandwidth, severely worsening the bandwidth pressure. Finally, to reduce inter-node communication while load balancing in current MapReduce implementations, each node exhausts its local Map tasks (i.e., tasks whose input data is on local disk) before stealing tasks from another node (remote tasks). This strategy leads to execution of remote tasks only when necessary. Unfortunately, in heterogeneous clusters, this strategy temporally concentrates all remote Map tasks at the end of the Map phase, creating bursty network load.

The Reduce-side load imbalance stems from the fact that MapReduce implementations distribute the keys equally among Reduce tasks assuming homogeneous nodes. However, this equal distribution creates load imbalance among the heterogeneous nodes which have disparate compute capability. We crystallize the above insights using a simple analytical model.

Our second contribution is Tarazu, which consists of a set of three schemes to address the performance problems. First, we propose *Communication-Aware Load Balancing of Map computation (CALB)* based on the key observation that MapReduce implementations overlap Map computation and Shuffle, and either Shuffle or Map computation is in the critical path of execution depending upon MapReduction's Shuffle load and cluster hardware characteristics. When Shuffle is critical, CALB prevents all task stealing and increases the chances of tasks being performed locally, thereby eliminating remote Map task traffic and aggravation of the Shuffle — the first and second Map-side components above. While this prevention increases computation at low-power nodes, CALB hides extra work under the critical Shuffle. When Map computation is critical, CALB allows controlled task stealing to load-balance Map computation and hides resulting remote task traffic under low-power nodes' computation. Thus, CALB shortens the critical path in both cases — by not aggravating the Shuffle in the former case and by load-balancing Map computation in the latter. Second, to avoid bursty remote task communication in Map-critical MapReductions — the third Map-side component above, we propose *Communication-Aware Scheduling of Map computation (CAS)* which spreads out remote tasks throughout the Map phase. We propose heuristics based on online measurements to estimate the information needed by CALB and CAS for making application- and cluster-aware decisions. Third, we propose *Predictive Load Balancing of Reduce computation (PLB)* by skewing the key distribution among Reduce tasks based on the compute capabilities of nodes. To address the challenge that PLB's skew factors need to be known *before* the Reduce phase begins, we use measurements from the Map phase to predict the Reduce phase's skew factor.

In summary, the key contributions of this paper are:

- identifying the key reasons for MapReduce's poor performance on heterogeneous clusters;
- *Communication-Aware Load Balancing of Map computation (CALB)* across nodes;
- *Communication-Aware Scheduling of Map computation (CAS)* for spreading out remote Map task traffic over time;
- *Predictive Load Balancing of Reduce computation (PLB)* across nodes; and
- on-line measurement-based heuristics to estimate information needed for making application- and cluster-aware decisions.

We evaluate Tarazu using Hadoop running a suite of eleven typical MapReduce applications. Using a heterogeneous cluster of 10 Xeon-based servers and 80 Atom-based servers, we show that on average Tarazu achieves 40% speedup over a baseline of Hadoop that includes straightforward tuning for hardware heterogeneity.

The rest of the paper is organized as follows. We discuss MapReduce's issues with heterogeneity in Section 2 and describe Tarazu in Section 3. We present our experimental methodology in Section 4, and our results in Section 5. We discuss some related work in Section 6 and conclude in Section 7.

2 Issues with heterogeneity

We start with a brief background on MapReduce and then discuss the issues with MapReduce on heterogeneous clusters.

2.1 Background: MapReduce

In MapReduce's *programming model* [10], programmers specify a *Map* function that processes input data to generate intermediate data in the form of <*key, value*> tuples, and a *Reduce* function that merges the values associated with a key.

MapReduce's *execution model* has four phases [10]. In the first execution phase, *Map* computation produces <*key,value*> tuples. This phase is completely parallel and can be distributed easily over a cluster. The second phase performs an all-Map-to-all-Reduce personalized communication, called *Shuffle*, in which all tuples for a particular key are sent to a single Reduce task. Because there are usually many more keys than Reduce tasks, each Reduce task may process more than one key. The third phase *sorts* the tuples on the key field, essentially grouping all the tuples for the same key. This grouping does not occur naturally because the tuples for the differ-

2. While task stealing generally implies one thread running the tasks already assigned to another thread, we use the term to indicate a node processing data that is resident on the disk of another node to which the processing is likely to be assigned due to locality reasons.

ent keys meant for a Reduce task may be jumbled. Finally, the fourth phase of *Reduce* computation processes all the tuples for a key and produces the final output. Like Map tasks, Reduce tasks run in parallel.

A run-time system automatically partitions the input data, schedules the Map tasks across the nodes, orchestrates the Shuffle, and schedules the Reduce tasks. The run-time system also provides fault tolerance by re-executing computations when machines fail or become abnormally slow.

The growing interest in heterogeneous clusters comprising nodes designed for different performance-power points raises the question of how MapReduce implementations would perform on such clusters. Before we address this issue, we briefly describe the aspects of the MapReduce run-time framework that are most relevant to our context.

First, MapReduce frameworks employ a dynamic load-balancing scheduler, which tracks the execution status of nodes and assigns new tasks to free nodes. In doing so, the scheduler also considers locality. For example, the scheduler prefers tasks that process data resident on a node's local disk over tasks that require data from other nodes in the cluster (remote tasks). Second, in order to utilize maximum parallelism in both the Map and Reduce phases, Map and Reduce tasks occupy the entire cluster instead of space-sharing the cluster. Consequently, the all-Map-to-all-Reduce Shuffle amounts to all-nodes-to-all-nodes communication which stresses the scarce network bisection bandwidth [10, 32, 31, 36, 13]. While network switch bandwidth does scale with hardware technology, bisection bandwidth is a global resource which is hard to scale up with the number of nodes (e.g., around 50 Mbps per node for current clusters with thousands of nodes [10, 32, 31]). To alleviate this problem, the Shuffle is overlapped with Map computation where Reduce tasks "pull" intermediate data from earlier Map tasks while later Map tasks execute. To achieve this overlap, the scheduler creates and assigns Reduce tasks to nodes early during the Map phase (Map and Reduce tasks time-share the nodes, although Reduce computation does not start until Map and Shuffle phases have finished). Finally, in addition to these basic optimizations, MapReduce frameworks also have mechanisms for dealing with straggler tasks (tasks that take an unusually long time to execute and delay the completion of a phase [10]), by speculatively creating backup copies of tasks and terminating all outstanding copies when one copy finishes.

2.2 Reasons for poor performance on heterogeneous clusters

At first glance, it might appear that the dynamic load-balancing approach makes MapReduce frameworks inherently well-suited to heterogeneous clusters: slower (low-power) nodes would be assigned fewer tasks, while faster (high-performance) nodes would be assigned more tasks, leading to good load balance subject to the limits imposed by task granularity [10]. To test this hypothesis, we ran a suite of eleven MapReduce benchmarks on a 90-node, heterogeneous cluster comprising 10 Xeon-based nodes and 80 Atom-based nodes, using Hadoop [14] extended with suitable straggler optimizations [3,37]. We also tuned Hadoop for hardware heterogeneity by customizing the number of worker threads per node for each node type to account for nodes' differing core counts and optimizing memory buffer sizes to match nodes' memory capacities. We observed that the 90-node heterogeneous cluster was 20-75% slower than 10-node Xeon-only or 80-node Atom-only homogeneous sub-clusters for six out of eleven of our benchmarks. In other words, adding heterogeneous hardware resources actually degrades performance. This poor performance underscores the fact that MapReduce frameworks designed and optimized for homogeneous clusters do not directly scale to heterogeneous clusters.

Table 1: Map task distribution in homogeneous vs. heterogeneous clusters

	Homogeneous		Heterogeneous	
	Xeon-only	Atom-only	Xeons	Atoms
local	~100%	98%	45%	95%
remote	~0%	2%	55%	5%

The poor performance of MapReduce is due to two key factors, one non-intuitive and the other intuitive.

2.2.1 Factor 1 (non-intuitive): Interaction between load balancing and network traffic in Map

The MapReduce scheduler schedules remote Map tasks when a node no longer has any local data to process. In the context of a cluster with high-performance and low-power nodes, high-performance nodes finish their local Map tasks before low-power nodes, leading to the scheduling of many remote Map tasks that read input data from low-power nodes and incur greatly increased network load. Table 1 shows the break down of Map tasks into local and remote tasks for our above 90-node cluster. We see that because the homogeneous sub-clusters (Xeon-only and Atom-only) are well-balanced naturally, there are only a few remote Map tasks. In contrast, in the heterogeneous cluster, almost half of the Xeon-based nodes' Map tasks are remote (stolen from the Atom-based nodes). Because the Atom-based nodes are slower, they execute only a few remote tasks. We clarify that (1) the input data traffic for these Map tasks is distinct from the Shuffle which is the Map-to-Reduce communication, and (2) the remote tasks correspond to low-power nodes' tasks that have not begun execution — not stragglers which are slow-running tasks.

This network traffic problem is exacerbated in MapReductions with heavy Shuffle, where remote Map traffic must compete with the Shuffle for network bandwidth (recall that the Shuffle is overlapped with the Map phase). Figure 1 depicts the activities of high-performance and low-power nodes (Y axis) over time (X axis) during Map and Reduce phases. In the Map phase, the Shuffle is overlapped with Map tasks in the nodes. The high-performance nodes' remote Map traffic competes with the Shuffle. The local tasks at low-power nodes are slowed down due to the I/O processing required to service the high-performance nodes' remote tasks.

Furthermore, remote Map traffic is concentrated at the end of the Map phase because remote tasks are scheduled only after a node has finished processing its local data, as shown in Figure 1 for high-performance nodes. Figure 2 plots the remote Map traffic (Y axis) over Map phase execution time (X axis) showing the traffic burst at the Map phase end. Thus, despite being locality-aware (which in general reduces communication), the scheduler greatly increases network traffic in heterogeneous clusters.

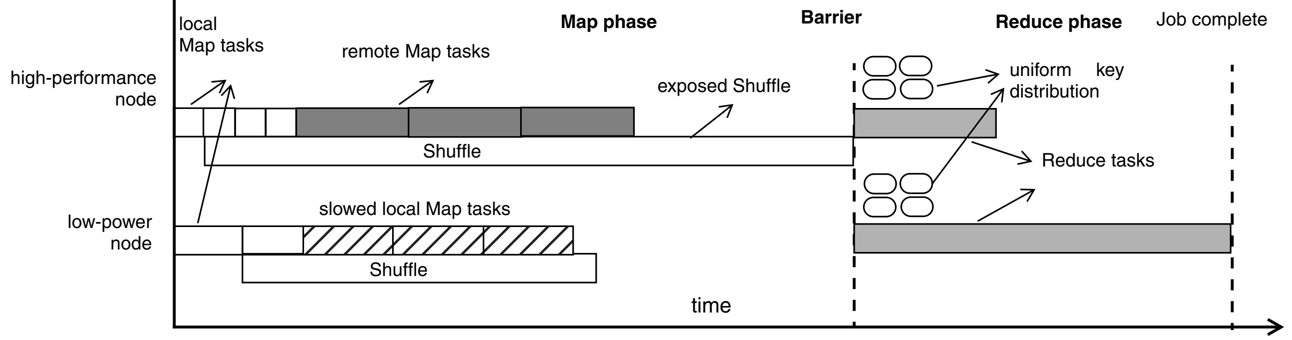

FIGURE 1: MapReduce performance issues on heterogeneous clusters

2.2.2 Factor 2 (intuitive): Reduce phase imbalance amplified by heterogeneity

The run-time system attempts to load balance the Reduce phase by assigning keys uniformly to each of the Reduce tasks, and scheduling tasks to the nodes in the cluster. The ability to dynamically load balance Reduce tasks is drastically limited by the fact that the number of Reduce tasks is typically only a small multiple of the number of nodes in the cluster in order to limit output fragmentation [10]. In the context of heterogeneous clusters, Reduce phase load imbalance is amplified due to the different processing speeds of different node types. Although the keys are distributed uniformly, low-power nodes take longer. The Reduce phase depicted in Figure 1 illustrates this issue. It is difficult for the scheduler to rectify this imbalance given the small number of Reduce tasks at each node. Treating Reduce tasks on low-power nodes as stragglers and running backup copies on high-performance nodes would effectively amount to not utilizing low power nodes in the Reduce phase. Furthermore, different Reduce tasks could process different amounts of data in both homogeneous and heterogeneous clusters, depending on the distribution of the values per key. However, addressing the value-per-key variability is harder than the variability due to heterogeneity, as discussed in Section 3.3.

2.3 A simple analytical model

To crystallize the above insights, we propose a simple performance model for MapReduce execution on a heterogeneous cluster. For ease of illustration, we describe our model for two types of nodes: high performance nodes (*hp*) and low-power nodes (*lp*). Our model assumes information such as the average execution times of Map and Reduce tasks on different types of nodes and total volume of Shuffle data (e.g., collected through profiling). The model expresses a MapReduction's execution time as a function of various known parameters and key unknowns — the ratio in which Map tasks and Reduce tasks are partitioned among different node types. The inputs to the model are as follows:

- N_{hp} (N_{lp}) — number of high-performance (low-power) cores (not nodes) in the cluster;
- *ChunkSize* — size of one input data chunk in bytes;
- D_{hp} (D_{lp}) — number of input data chunks stored on high-performance (low-power) nodes, without counting replicas;
- $Tmap_{l,hp}$ ($Tmap_{r,hp}$) — average execution time of local (remote) Map task on a high-performance node;
- $Tmap_{l,lp}$ ($Tmap_{r,lp}$) — average execution time of local (remote) Map task on a low-power node;
- *Nkeys* — total number of keys processed by Reduce;
- $Tred_{hp}$ — average time taken to perform Reduce (including Sort) on a single key on a high performance node;
- $Tred_{lp}$ — average time taken to perform Reduce (including Sort) on a single key on a low power node;
- $BW_{bisection}$ — bisection bandwidth of the cluster in bytes/sec; and
- *ShuffleData* — total Shuffle traffic in bytes.

The unknowns (to be determined) are:

- $Nmap_{l,hp}$ ($Nmap_{r,hp}$) — number of local (remote) Map tasks executed on high-performance nodes;
- $Nmap_{l,lp}$ ($Nmap_{r,lp}$) — number of local (remote) Map tasks executed on low-power nodes; and
- $Nkeys_{hp}$ ($Nkeys_{lp}$) — number of keys processed by Reduce on high-performance (low-power) nodes.

Our model is described by the following linear equations for each of the Map, Shuffle and Reduce phases:

Map phase: Due to symmetry, we assume perfect load balance of Map tasks within high-performance and low-power nodes (but not across the two types of nodes). We intentionally ignore stragglers because it is not our intent to address them (previously proposed straggler management techniques may be used). Consequently, data chunks that reside on a high-performance node must be processed either by local Map tasks or by remote Map tasks on low-power nodes, and vice-versa for data chunks on a low-power node. Further, we assume that a chunk is located on a high-performance node if a single replica is available on a high-performance node. Hence,

$$Nmap_{l,hp} + Nmap_{r,lp} = D_{hp}$$
$$Nmap_{l,lp} + Nmap_{r,hp} = D_{lp}$$

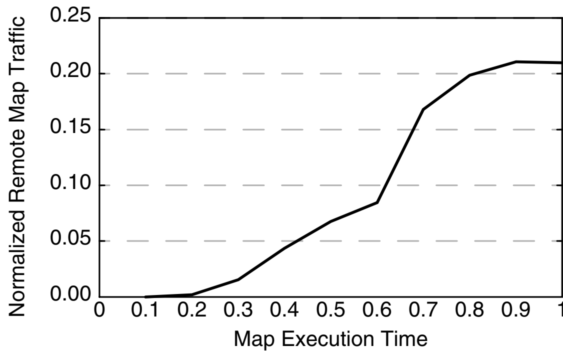

FIGURE 2: Remote task traffic

Map Finish Time (*MFT*) can be computed as:

$$MFT_{hp} = (Nmap_{l,hp}*Tmap_{l,hp}+Nmap_{r,hp}*Tmap_{r,hp})/ N_{hp}$$
$$MFT_{lp} = (Nmap_{l,lp} * Tmap_{l,lp} + Nmap_{r,lp} * Tmap_{r,lp}) / N_{lp}$$

$$MFT = \max(MFT_{hp}, MFT_{lp}) \qquad \text{(EQ 1)}$$

Shuffle phase: For simplicity, we assume that all of the remote Map reads and Shuffle data cross the cluster bisection, as given by:

$$D_{bisection} = (Nmap_{r,lp} + Nmap_{r,hp})* ChunkSize + ShuffleData \qquad \text{(EQ 2)}$$

To compute Shuffle Finish Time (*SFT*), we make simplifying assumptions that (1) the remote Map reads are spread out evenly throughout the Map phase while, in practice, the reads are concentrated at the Map phase end as discussed in Section 2.2.1 and (2) the Shuffle starts at the beginning of the Map phase (though in real implementations, the Shuffle starts a little later in the phase). Because the Map computation overlaps with the Shuffle and remote Map reads, *SFT* can be computed as:

$$SFT = \max(D_{bisection}/BW_{bisection}, MFT) \qquad \text{(EQ 3)}$$

Because the remote Map reads are bursty, their overlap with the Map is imperfect causing *SFT* to be longer in practice. If the Shuffle is hidden completely under Map then *SFT = MFT*; otherwise, there is some exposed Shuffle and *SFT > MFT.*

Reduce phase: Reduce Finish Time (RFT) is given by:

$$RFT = SFT + \max (Nkeys_{lp}*Tred_{lp}/N_{lp}, Nkeys_{hp}* Tred_{hp}/N_{hp}) \qquad \text{(EQ 4)}$$

We note that the model consists of linear equalities and inequalities in the unknown variables (max can be expressed as multiple linear inequalities). Therefore, the model can be solved exactly to minimize the total execution time (Reduce finish time, *RFT*), subject to the above constraints. Intuitively, this model captures the following trade-offs engendered by MapReduce execution on heterogeneous clusters:

- The ratio of input data stored on high-performance nodes to that on low-power nodes may be different from the ratio of the nodes' processing rates ($Tmap_{l,lp}/Tmap_{l,hp}$).
- The Map phase could be load balanced by executing remote Map tasks ($Nmap_{r,lp} > 0$ or $Nmap_{r,hp} > 0$), as is typical in current MapReduce frameworks, but doing do may increase the data crossing the cluster bisection due to remote Map reads ($D_{bisection}$), possibly delaying the Shuffle finish time (*SFT*).
- For Shuffle-light MapReductions, *ShuffleData* is small, and the Shuffle is hidden under the Map phase. For Shuffle-heavy MapReductions with large *ShuffleData*, the Shuffle is exposed.
- The Reduce phase can be load-balanced by assigning different number of keys to each type of nodes ($Nkeys_{hp}$, $Nkeys_{lp}$).

While the above model is useful to understand the trade-offs involved in MapReduce execution on heterogeneous clusters, the extensive information required (application and cluster characteristics, and task execution times) make it infeasible to use in practice. Requiring a priori application profiling on the target cluster greatly diminishes the agility and ease of use that is inherent to MapReduce. Therefore, we propose to use some heuristics based on online measurements to achieve the same goal in Tarazu.

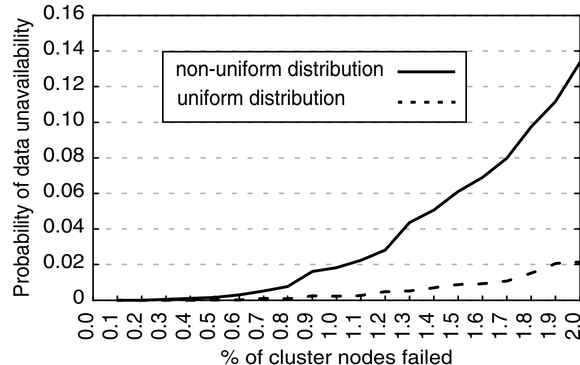

FIGURE 3: Probability of data unavailability with skewed distribution

2.4 Alternative approaches

One may argue that the aforementioned issues could be addressed using alternative approaches, including skewed data distribution, backup tasks, re-organization of the cluster hardware, and employing fine-grained Reduce tasks. We consider each of these in turn and argue their merits and de-merits in order to justify our approach.

Because node disk capacities are not directly correlated to compute capabilities (low-power nodes are available with similar disk capacities as high-performance nodes), it is natural to distribute data uniformly across the disks. However, one could skew Map input data distribution such that the amount of input data stored on each node is proportional to the node's processing capability. Doing so would imply that a high-performance node has more local Map data than a low-power node so that high-performance nodes do not exhaust local data much earlier than low-power nodes, resulting in far fewer remote Map tasks and far less network traffic. However, determining a good skew factor is hard in practice because the relative processing speeds of different types of nodes, and hence the skew factor, vary significantly from application to application. For example, in our suite of 11 MapReduce benchmarks, the ratio of Map task processing times on Atom-based to Xeon-based nodes vary from 1.5:1 to 4.5:1. Thus, despite skewing the data distribution, a significant number of remote tasks will remain, leading to the performance issue outlined above (see Section 5.5). Furthermore, skewing the data distribution incurs other significant problems: (i) Skewing would cause under-utilization of the cluster storage capacity because commodity disk capacities are unlikely to match our skew factors. In fact, Hadoop uses a "disk re-balancer" to prevent any unintentional skew to achieve good storage utilization [38]. (ii) Skewing would compromise fault tolerance achieved by data replication (nodes that store more data will have a much higher impact when they fail). Based on a Monte-Carlo simulation of node failures in a 1000-node cluster (100 high-performance nodes and 900 low-power nodes) with 3-way data replication of 640 GB, we determined that the probability of data becoming unavailable when 1% of the nodes fail is 0.0022 for uniform data distribution, and 0.0182 (an increase of 8.3X) when the distribution is skewed in the ratio of 1:4 between low-power and high-performance nodes. In Figure 3, we show the probability of data becoming unavailable (Y axis) when 0-2% of the nodes fail

(X axis) for the two data distribution schemes (higher node failure rates are unlikely). To avoid these reliability problems, we explore a solution that addresses the more realistic scenario where a significant number of remote tasks need to be executed, with or without skewed data distribution.

Backup tasks could be used to improve the load balancing capability of the scheduler in the presence of heterogeneity: upon completing all their local tasks, high-performance nodes can speculatively execute backup copies of tasks running on low-power nodes, reducing the chance of imbalance. Doing so, however, would imply that low-power nodes are under-utilized**.** In addition, the backup tasks may even hurt performance by creating additional load on the network.

Increasing the replication factor for data would increase the probability that data is local to a node and would reduce the number of remote tasks. This option, however, imposes a significant penalty on the cluster storage capacity.

Increasing the number of Reduce tasks (finer-grained Reduce tasks executed in multiple batches) could improve the Reduce phase's load balance because high-performance nodes would automatically execute more Reduce tasks. However, typical MapReduce jobs use a single batch of coarse-grained Reduce tasks due to two reasons. (1) With multiple batches, the Shuffle cannot hide under Map computation in later batches (Map completes before the first Reduce batch) and Reduce computation is usually too short to fully hide the Shuffle. The exposed Shuffle often makes multiple Reduce batches slower than one batch. (2) Unlike a MapReduction's input and final output, the intermediate Map output is not replicated for fault tolerance. To reduce the probability of loss of the intermediate data and subsequent re-run of Map tasks, the Reduce phase follows soon after Map in the single-batch case, whereas multiple batches often delay the Reduce phase.

In summary, MapReduce application developers expect their applications' performance to scale in proportion to the cluster's compute resources, irrespective of whether the hardware is homogeneous or heterogeneous. Based on the above insights, we propose a suite of optimizations called Tarazu.

3 Tarazu

Recall from above that (1) the Map-side built-in load balancing results in numerous remote Map tasks which increase the network load (i.e., $D_{bisection}$ in EQ 2 and EQ 3), compete with the Shuffle, and create bursty network load by being concentrated at the end of the Map phase, and (2) hardware heterogeneity amplifies the Reduce-side load imbalance across the nodes (i.e., increases the max term for *RFT* in EQ 4). To address these issues, we propose Tarazu, a suite of optimizations to improve MapReduce performance on heterogeneous clusters. As illustrated in Figure 4, Tarazu consists of three components:

- C*ommunication-Aware Load Balancing of Map computation (CALB),* which regulates the use of remote Map tasks based on whether Map or Shuffle is likely to be in the critical path (Section)
- *Communication-Aware Scheduling of Map computation (CAS)* to spread out remote Map tasks throughout the Map phase (Section 3.2), and
- *Predictive Load Balancing of Reduce computation (PLB)* across the heterogeneous nodes (Section 3.3).

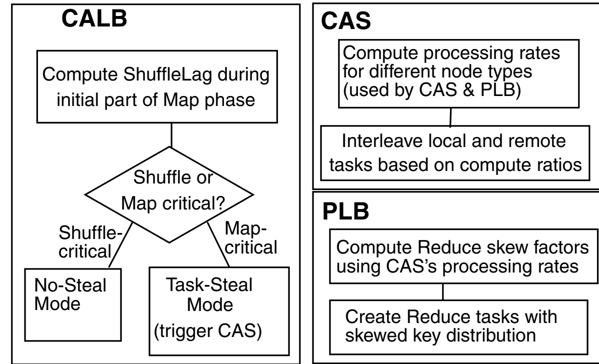

FIGURE 4: Overview of Tarazu

3.1 Communication-Aware Load Balancing of Map (CALB)

CALB is based on the key observation that due to the overlap between Map computation and Shuffle, either the Shuffle or the Map computation is in the critical path, depending upon the MapReduction's Shuffle load and the cluster hardware characteristics. As shown in Figure 4, CALB uses on-line measurements during the initial part of the Map phase to choose one of two operating modes, depending on whether Shuffle or Map is critical. In the initial part, all MapReductions start in the *task-steal* mode which allows controlled task stealing to load-balance the Map computation (i.e., lower *MFT* in EQ 1).

For MapReductions where the Shuffle is critical, CALB switches to *no-steal mode* where CALB prevents task stealing for most of the Map phase, preventing further aggravation of the Shuffle traffic (i.e., lower $D_{bisection}$ in EQ 2 and EQ 3) and increasing the chances of tasks being executed locally. When the Shuffle traffic falls below a threshold, called *shuffleEndThreshold*, (i.e., towards the end of the Map phase), CALB allows task stealing to load-balance any remaining tasks (naturally, faster nodes steal work from slower nodes). Figure 5 shows how the no-steal mode operates for the example of Figure 1. Like Figure 1, Figure 5 depicts the activities of high-performance and low-power nodes (Y axis) over time (X axis) during Map and Reduce phases. The Shuffle is critical in this example. For most of the Map phase, high-performance nodes do not steal tasks, allowing the low-power nodes to run the tasks locally. At the end of the Map phase when the Shuffle ends, high-performance nodes pick up a few remote tasks.

The *no-steal* mode has three benefits: (i) There are no remote Map tasks to compete with the Shuffle (Figure 5), (ii) Avoiding remote tasks also reduces the I/O processing overhead at senders and receivers. (iii) Because of no task stealing, slower nodes perform more work compared to the case with task stealing. Despite being slower, nodes' extra work is hidden under the critical Shuffle which exists irrespective of whether task-stealing occurs or not (Figure 5). By avoiding the contention due to remote tasks and their traffic, CALB enables the critical Shuffle to proceed faster, resulting in faster overall execution.

One may think that the residual task stealing at the end of the Map phase in the no-steal mode could create a burst of remote task traffic. However, this problem does not arise because (1) the number of remote tasks at this point is usually small (e.g., 1-2% of Map

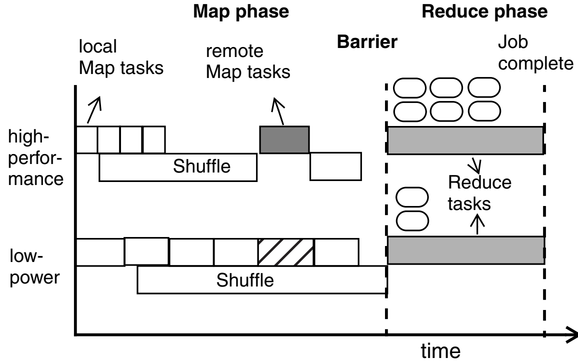

FIGURE 5: Communication-aware load balancing of Map (CALB) *no-steal* mode

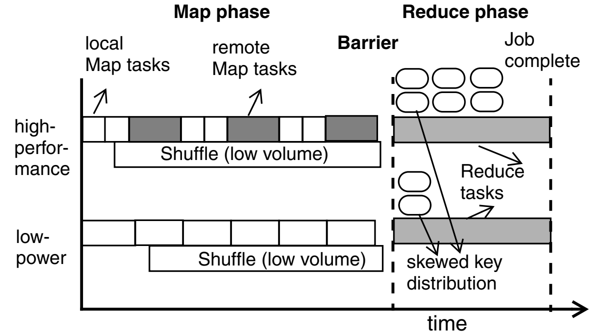

FIGURE 6: Communication-aware scheduling of Map (CAS) and Predictive Load Balancing of Reduce (PLB)

tasks) and (2) most of the Shuffle has finished by that time, so the remote traffic does not interfere with the Shuffle.

For MapReductions where the Map computation is critical, CALB continues in the task-steal mode. Note that the number of remote tasks needed and when to schedule them are the concern of CAS, which is discussed in the next sub-section. Figure 6 shows the task-steal mode for an example where Map computation is critical. As before, the Y axis shows the nodes' activities and the X axis shows time. In the Map phase, high-performance nodes steal tasks from low-power nodes. CALB hides the resultant high-performance nodes' remote task traffic under the low-power nodes' computation (Figure 6). By load-balancing Map computation, CALB shortens the critical path in this case.

The remaining issue is the detail of determining CALB's mode for a given application-cluster combination based on whether the Shuffle or the Map computation is critical. To this end, we observe that the Shuffle is critical when Map tasks complete their computation at a faster rate than their communication to Reduce tasks. Fortunately, MapReduce implementations already monitor, for fault tolerance purposes, the number of Map tasks that have completed their computation ($numComputed_i$) and the number that have completed their communication ($numCommunicated_i$) for each node i (these are sent at each heartbeat by worker nodes to the master node). We aggregate these numbers for the entire cluster and denote the aggregate values as $numComputed_{all}$ and $numCommunicated_{all}$. The difference between the two numbers, which we call *shuffleLag*, indicates the extent to which the Shuffle lags the Map computation. If *shuffleLag* increases over time, it implies that the Shuffle is likely to be critical.

We monitor the trend in *shuffleLag* over a time window between two points during the Map phase, denoted by *measureBegin* and *measureEnd*. Recall that the Shuffle communication starts only after Reduce tasks have been created, which occurs after some Map tasks are underway (Section 2). Consequently, we set *measureBegin* as the point where at least some Map tasks have completed their communication so that we obtain reliable readings for $numCommunicated_i$. *measureEnd* is constrained by the fact that the earlier the window ends, the fewer the samples whereas the later the window ends, the less time there is for taking the necessary action. We set *measureEnd* to achieve a good compromise between these opposing requirements. If the slope of the time series of *shuffleLag* measurements over the monitoring interval is positive, CALB enters no-steal mode; otherwise, it enters task-steal mode. CALB uses this simple approach to decide whether the Shuffle or the Map computation is critical for an application-cluster combination.

In no-steal mode, CALB detects the end of the Shuffle by continuing to monitor *shuffleLag* beyond *measureEnd*. The end is detected when *shuffleLag* as a fraction of $numComputed_{all}$ falls below *shuffleEndThreshold*.

It may seem that CALB may sub-optimally choose the task-steal mode for straddler MapReductions where the Map computation is only slightly more critical than the Shuffle. However, because all MapReductions start with task-steal mode in which they stay till *measureEnd*, our measurements ($numCommunicated_i$) include the effect of remote tasks due to task stealing on the network traffic in the measurement time window. CALB chooses task-steal mode only if the Map computation lags the Shuffle in the presence of the remote tasks. Therefore, CALB's choice of task-steal mode is not likely to be sub-optimal. Moreover, because Shuffle or Map criticality does not change within a MapReduction, deciding the source of criticality once is enough without the need for repeated, dynamic checks.

Although we have used two types of nodes — high-performance and low-power nodes — to illustrate our ideas, our description of CALB does not assume any specific number of types of nodes. As such, CALB is applicable to a heterogeneous cluster with any number of node types.

3.2 Communication-Aware Scheduling of Map (CAS)

While CALB decides whether or not allowing remote tasks would be beneficial for a given MapReduction, CAS determines how many remote tasks are needed and when to execute them in the task-steal mode (Figure 4). Recall from Section 2 that in current MapReduce implementations, task stealing occurs at the end of the Map phase, creating a surge of traffic. Also, as discussed in Section 2.3, this bursty traffic increases *SFT* (EQ 3). To avoid this problem, CAS spreads out the remote tasks (during initial part of the Map phase, and in CALB's task-steal mode) by interleaving them with local tasks. Figure 6 shows that high-performance nodes uniformly interleave their local and remote tasks. In addition to avoiding the bursty traffic, CAS has other benefits: (1) By interleaving remote tasks with local tasks in the Map phase, CAS achieves better overlap between remote task communication and local task computation on both sender and receiver sides (low-power and high-performance nodes, respectively, in Figure 6). (2) The remote tasks read input data faster by avoiding bursts. These

benefits shorten Map computation, which is the critical path relevant for CAS.

In current MapReduce implementations, remote tasks are executed by a node only after it runs out of local tasks, and hence it is known for sure that remote tasks are required. In contrast, CAS requires task stealing to occur throughout the Map phase when the number of remote tasks is not known. To address this issue, we measure the average Map task execution times for each node type in the cluster and compute the ratios of the execution times for each pair of types ($mapComputeRatio_{i,j}$ defined as execution time of type *j* / execution time of type *i).* Because we need only one of $mapComputeRatio_{i,j}$ and $mapComputeRatio_{j,i}$, we choose numerator node type *j* to be the slower one. (In the simple case where there are only two node types, *mapComputeRatio* is a single number.) CAS uses these ratios to determine the number of remote tasks to be moved from one node to another. Because current MapReduce implementations already track the identity of the pair of source and destination nodes involved in task stealing, CAS can apply the specific pair's *mapComputeRatio*.

We continually monitor the execution times of local Map tasks that have completed their execution and calculate $mapComputeRatio_{i,j}$ for all *i* and *j*. As CAS triggers task stealing throughout the Map phase, each task stealing instance uses the ratio available at the time of the instance.

Unlike CALB which is naturally independent of the number of node types in the cluster, CAS needs per-pair ratios. However, because this number is likely to be small in real-world clusters, computing the ratios poses insignificant overhead.

Finally, CAS applies some more optimizations. First, to avoid overwhelming a node and causing thrashing, CAS limits the maximum number of remote tasks that can concurrently read input data from a node to *remoteServeThreshold*. Second, CAS prevents slower nodes from running local tasks whose input data is replicated at a faster node by using the replica locations of data (which is tracked by current MapReduce implementations for locality purposes). Third, CAS prevents task stealing from a node if the node has fewer remaining tasks than *localRemainingThreshold.* The rationale is that the node will likely execute the tasks faster than communicating the data for remote execution.

3.3 Predictive load balancing of Reduce (PLB)

While CALB and CAS optimize the Map phase, PLB achieves better load balance in the Reduce phase by skewing the intermediate key distribution among the Reduce tasks based on the type of the node on which a Reduce task runs (Figure 4). Such load balancing reduces the max term for *RFT* in EQ 4. While current implementations create as many bins per Map task as there are Reduce tasks, PLB creates more hash bins (by a factor of *binMultiplier)* as the number of Reduce tasks to achieve the skew (e.g., *binMultiplier* = 4). PLB uniformly distributes keys to the bins and then assigns as many bins to each Reduce task on node *i* as is dictated by the skew factor, $reduceSkewFactor_i$. For example, if a fast node is three times as fast as a slow node, then $reduceSkewFactor_i$ for the fast node is 3 and that for the slow node is 1. Therefore, a Reduce task on a fast node gets three times as many bins as a Reduce task on a slow node, as shown in the Reduce phase in Figure 6. To implement assigning multiple bins per Reduce task, we modify the MapReduce implementation to allow multiple sends from a Map task to a Reduce task (the baseline implementation assigns and sends only one bin from a Map task to a Reduce task). Note that although we create more hash bins per Map task than the baseline, we have the same number of Reduce tasks as the baseline and therefore do not incur any extra output fragmentation (Section 2.2.2).

Ideally, $reduceSkewFactor_i$ should be the ratio of the processing speeds of Reduce computation on the types of nodes in the cluster. However, because Reduce tasks are launched early in the Map phase to avoid delaying the Shuffle, the processing speeds are not available at the beginning of the Reduce phase. Moreover, starting with a default skew factor and changing it during the Reduce phase would incur the prohibitive cost of re-shuffling of data. Consequently, we use CAS's $mapComputeRatio_{i,j}$ for each pair of node types to compute $reduceSkewFactor_i$ and normalize this factor with respect to that of the slowest node. Thus, we use measurements from the Map phase to predict the Reduce phase's skew factor (we could use generic performance ratios of the node types but doing so would likely be no better than using the Map phase ratios).

While PLB skews the key distribution, the imbalance in the Reduce tasks may also come from the variability in the actual number of values per key. However, the requirement that the number of values per key has to be learnt early in the Map phase could make any estimates inaccurate. Therefore, we do not pursue this option.

Finally, PLB's skewed key distribution results in the Reduce output being skewed across nodes. However, the Reduce output is not uniform even in homogeneous clusters due to the variability in the number of values per key. In fact, MapReduce implementations provide a disk re-balancer to address this issue (Section 2).

3.4 Other issues

We discuss a few remaining issues related to Tarazu.

Because our schemes merely affect the creation and scheduling of the tasks, MapReduce's fault tolerance schemes and speculative execution of stragglers remain intact.

Although our implementation is based on Hadoop, our schemes are general and apply to other implementations [11,18,27]. Further, our schemes would default to Hadoop for homogeneous clusters. CALB's no-steal mode would amount to a few remote tasks at the end of the Map phase, similar to Hadoop. In CALB's task-steal mode, CAS would not kick in because a homogeneous cluster has only one type of nodes. Consequently, CALB's task-steal mode would also default to current Hadoop. PLB would not kick in due to the same reasons as CAS.

Our schemes are applicable to heterogeneous clusters with any number of types of nodes, and may be useful under different forms of hardware heterogeneity. For example, clusters where some nodes are GPU accelerated while others are not will also be subject to the issues that we address.

Despite the disadvantages of doing so, Map input data distribution may be skewed such that more data is placed on faster nodes to reduce remote Map tasks (Section 2). If the skewing is only moderate where faster nodes still finish before slower nodes, CALB would work as described. However, CAS's execution time ratios would have to be adjusted by the data skew factor (e.g., if CAS's *mapComputeRatio* ratio is 3:1 and faster nodes have twice as much data as slow nodes, then the new ratio would be 1.5:1). On

Table 2: Hadoop parameters

mapred.tasktracker.map.tasks.maximum	8 for Xeons, 2 for Atoms
mapred.tasktracker.reduce.tasks.maximum	2
io.sort.factor	5
mapred.inmem.merge.threshold	100
mapred.job.shuffle.merge.percent	0.50
DFS block size	64 MB
DFS replication factor	3
Maps to be completed before reduce creation	5%
Speculative execution enabled	Yes
Heartbeat interval	6 sec

the other hand, if faster nodes have so much data that they finish after slower nodes, then CALB's no-steal mode would proceed as before with a few remote tasks migrating from fast nodes to slow nodes at the end of the Map phase. In CALB's task-steal mode, CAS's adjusted ratios would flip directions and force the slower nodes to steal tasks from the faster nodes. Because skewing is only for the Map input data, PLB is not affected.

4 Experimental Methodology

We evaluate Tarazu by modifying Hadoop's public-domain MapReduce implementation (version 0.20.2) [14]. We use a heterogeneous cluster of 90 servers comprising 10 Xeon-based and 80 Atom-based server nodes.

4.1 MapReduce Implementations

Hadoop implements MapReduce as a run-time system linked with user-supplied Java classes for Map, Combiner (optional), and Reduce functionality. Hadoop uses a single master (JobTracker) for the whole cluster to create and schedule Map and Reduce tasks, monitor node health, and trigger backup tasks if needed.

We compare Tarazu against Hadoop as well as LATE [37]. For a fair comparison, we tune Hadoop for hardware heterogeneity by customizing the number of processes per node for each node type to account for the differences in core counts across node types and optimizing memory buffer sizes to match the memory sizes (e.g, our Xeon-based and Atom-based nodes have 8 cores with 48 GB DRAM per node and 2 cores with 4 GB DRAM per node, respectively, and Atom-based nodes thrash or run out of memory with default Hadoop settings). We list these and other parameters in Table 2. Further, for better performance, we replace Hadoop's synchronous write of the final Reduce output, which is replicated for fault tolerance, with asynchronous write in all cases (Hadoop, LATE, and Tarazu). We ensure that our measurements include the time for asynchronous writes in the background to finish.

We implement LATE and use the above tuned parameters. Further, we use the same values for *SpeculativeCap* and *SlowTaskThreshold*, as recommended in the paper. We tune *SlowNodeThreshold* so that LATE can correctly detect Xeon-based and Atom-based nodes as fast and slow nodes, respectively. We use the same heuristics to estimate the tasks' progress rates and finish times as in [37]. We ensure that LATE always schedules backup tasks on a Xeon-based node.

We run Tarazu with the parameters listed in Table 3. The master node samples $numComputed_i$ and $numCommunicated_i$ at every heartbeat for each node type in the time interval between *measureBegin* and *measureEnd* (Section), computes $mapComputeRatio_{i,j}$ for each pair of node types to determine the number of remote Map tasks (Section 3.2) and $reduceSkewFactor_i$ for each node type to skew the key distribution to Reduce tasks across the node types (Section 3.3). In PLB, we use the same heuristic as Hadoop to launch the Reduce tasks, with the additional requirement that at least one Map task on each node type is complete. This requirement ensures that we have valid $reduceSkewFactor_i$ for assigning hash bins to Reduce tasks.

4.2 Platform

We use a 90-node cluster comprising 10 Xeon-based and 80 Atom-based nodes. Each Xeon-based node is a dual-socket machine, where each socket has a quad-core Xeon E5620. Overall, each node has 8 cores running at 2.4 GHz, 12 MB L2, 48 GB of DDR3 RAM, and a 1-TB SATA hard disk. Each Atom-based node has an Intel Dual Core Atom D510 processor with each core at 1.66GHz, 2MB L2, 4 GB of DDR3 RAM, and a 500-GB SATA hard disk. All the systems run Ubuntu 10.04. Because of real system artifacts such as differences in disk seek times and OS scheduling variations, the execution time for the same job can vary across runs. However, we did not see any significant variations in our runs.

Our cluster uses two levels of non-blocking Gigabit Ethernet switches to connect the 90 nodes, which results in a per-node bisection bandwidth (around 1 Gbps) that is much higher than that available in typical large-scale clusters (around 50 Mbps [10, 32, 31], as discussed in Section 2.1). While the Shuffle stresses the network bisection in typical large clusters [36, 13] especially in the case of Shuffle-heavy MapReductions, our cluster's high bandwidth unrealistically eliminates the impact of the Shuffle. For example, our Shuffle-heavy MapReductions running on a homogeneous Amazon EC2 cluster of 128 Xeon-based nodes spend about 20% of their total execution times in exposed Shuffle as compared to less than 5% on our homogeneous Xeon-node sub-cluster (the clusters use similar nodes and therefore the Map computation should hide the Shuffle to the same extent in the two clusters). To address this issue, we simulate realistic bisection bandwidth by dividing our cluster into nine sub-clusters of 10 nodes each (in some experiments, we use Xeon-only (10 nodes) and Atom-only (80 nodes) clusters where we divide the Xeon-only cluster into two five-node sub-clusters and the Atom-only cluster into eight 10-node sub-clusters). We use the network-utility tools *tc* and *iptables* to limit per-node bandwidths from one sub-cluster to another to 50 Mbps without limiting the bandwidths within each sub-cluster.

Table 3: Tarazu parameters

CALB *measureBegin*	0.4
CALB *measureEnd*	0.5
CALB *shuffleEndThreshold*	0.02
CAS *remoteServeThreshold*	1
CAS *localRemainingThreshold*	2
PLB *binMultiplier*	4

Table 4: Benchmark Characteristics

Benchmark	Input size (GB)	Input data	#Maps & #Reduces	Run time on Hadoop	Shuffle volume (GB)	mapCompute Ratio	Critical path
self-join	250	synthetic	3720 & 180	1929	246	1.76	Shuffle
tera-sort	300	synthetic, random	4500 & 180	2353	300	2.02	Shuffle
ranked-inverted-index	205	multi-wordcount output	3204 & 180	2322	219	2.50	Shuffle
kmeans	100	Netflix data, $k = 6$	1602 & 6	4608	106	4.43	compute
inverted-index	250	wikipedia	3764 & 180	1714	57	3.40	compute
term-vector	250	wikipedia	3764 & 180	1874	59	3.36	compute
word-count	250	wikipedia	3764 & 180	1035	49	3.17	compute
multi-word-count	250	wikipedia	3764 & 180	2703	248	4.40	compute
histogram-movies	215	Netflix data	3223 & 180	416	2×10^{-3}	1.44	compute
histogram-ratings	215	Netflix data	3223 & 180	685	1.2×10^{-3}	2.50	compute
grep	250	wikipedia	3764 & 180	459	1.3×10^{-3}	1.71	compute

4.3 Benchmarks

Because there are only three interesting MapReductions — *tera-sort*, *word-count*, and *grep* — in the Hadoop release, we add eight more MapReductions covering Shuffle-heavy and Shuffle-light categories.

4.3.1 Shuffle-heavy MapReductions

Our Shuffle-heavy MapReductions include *self-join, tera-sort*, *ranked-inverted-index*, *k-means*, *term-vector*, *inverted-index, word-count*, and *multi-word-count*. *Self-join* is similar to the candidate generation part of the *a priori* data mining algorithm to generate association among *k+1* fields given the set of *k*-field associations [1]. *Tera-sort* is based on NOWsort [6] for sorting 100-byte tuples with first 10 bytes of each record as the key and the remaining 90 bytes as the value. *Ranked-inverted-index* takes lists of words and their frequencies in a file, and generates lists of files containing the given words in decreasing order of frequency. *k-means* is a popular data-mining algorithm used to cluster input data into *k* clusters. *k-means* iterates to successively improve clustering [21]. *Inverted-index* takes a list of documents as input and generates word-to-document indexing. *Term-vector* determines the most frequent words in a host (above some cut-off) to aid analyses of the host's relevance to a search. *Word-count,* a well-known application, counts all unique words in a set of documents. *Multi-word-count* generates a count of all unique sets of three consecutive words in the set of documents.

We summarize input data sizes, dataset descriptions, run times of baseline Hadoop, and the Shuffle volume in Table 4. The Shuffle volume is high despite using Combiners. The table also shows *mapComputeRatio*, the ratio of local Map execution time on the Atom-based nodes to that on the Xeon-based nodes (Section 3.2). These ratios show the extent of our cluster's heterogeneity as seen by the MapReductions.

Although all MapReductions have large Shuffle volume, the critical path is not the same for all the MapReductions running on the baseline Hadoop. In *self-join*, *tera-sort*, and *ranked-inverted-index,* the Shuffle is exposed well past the Map computation in the Xeon-based nodes. Though the Shuffle is exposed less in the slower Atom-based nodes, the Shuffle remains the critical path in these MapReductions. On the other hand, the Map computation in both Xeon-based and Atom-based nodes completely hides the Shuffle for *k-means*, *inverted-index, term-vector, word-count,* and *multi-word-count.* Consequently, the Map computation is the critical path in these five MapReductions. The last column of Table 4 lists the critical path for each MapReduction (i.e., whether Map computation or Shuffle is critical).

4.3.2 Shuffle-light MapReductions

Our Shuffle-light MapReductions include *histogram-movies*, *histogram-ratings*, and *grep*. We classified movies based on their ratings using Netflix's data [26]. While *Histogram-movies* generates a histogram of movies based on their average ratings (Netflix data), *Histogram-ratings* generates a histogram of the ratings. *grep*, a well-known application, searches for an input string in a set of documents. These MapReductions are also summarized in Table 4, In all these MapReductions, the Map computation in both Xeon-based and Atom-based nodes completely hides the Shuffle. Consequently, the Map computation is the critical path in all of the Shuffle-light MapReductions.

5 Experimental Results

We start by comparing Tarazu against Hadoop and LATE, followed by isolating the impact of CALB, CAS and PLB. We then show the impact of Tarazu's sensitivity to the extent of hardware heterogeneity by varying the mix of Xeons and Atoms in our cluster. The above experiments use uniform Map input data distribution on the cluster which is default for Hadoop DFS. Finally, we show the impact of skewed input data distribution on Tarazu's speedups.

5.1 Performance

We compare Tarazu with Hadoop and LATE both of which include straight-forward tuning for hardware heterogeneity (Section 4.1). We also present results for Hadoop on each of the homogeneous sub-clusters of 10 Xeon-based nodes (*Xeon-only*) and of 80 Atom-based nodes (*Atom-only*).

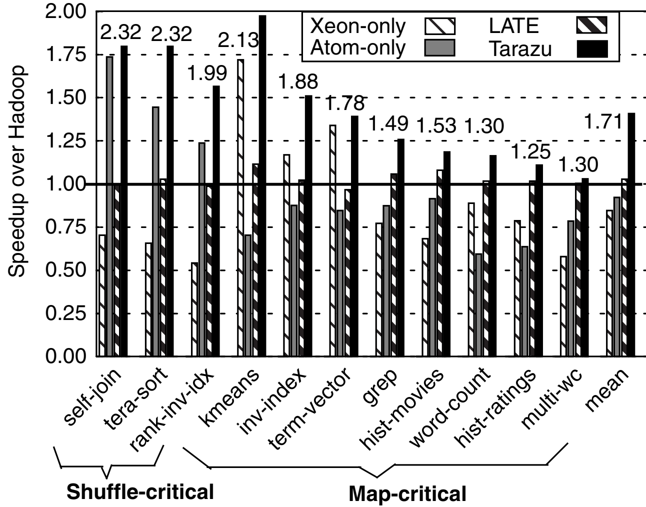

FIGURE 7: Performance comparison

In Figure 7, the Y-axis shows the speedups of Xeon-only, Atom-only, LATE, and Tarazu over Hadoop. The X-axis shows our benchmarks, of which the first three from the left are Shuffle-critical and the rest are Map-critical. The numbers above the bars show the speedups for an ideal case which (1) achieves perfect Map-Shuffle overlap for Shuffle-critical MapReductions and perfect load balance of the Map computation across Xeon-based and Atom-based nodes for Map-critical MapReductions and (2) achieves perfect load balance across Xeon-based and Atom-based nodes for the Reduce computation. We create the ideal case by post-processing Hadoop's execution logs. The ideal speedups range from 1.25 to 2.32, illustrating that there is significant scope for improvement over Hadoop. The high ideal speedups for Shuffle-critical MapReductions highlight the severity of the network bottleneck.

From the figure, we see that Hadoop on either of the homogeneous Xeon-only and Atom-only sub-clusters performs better than the baseline Hadoop on the heterogeneous cluster for the leftmost six out of eleven MapReductions. This result, one of the key motivations for this paper, shows that current MapReduce implementations do not perform well on heterogeneous clusters. LATE performs only slightly better than Hadoop by scheduling backup tasks on faster Xeon-based nodes for the stragglers running on slower Atom-based nodes. However, because remote Map task traffic and Reduce-side load imbalance are significant problems in heterogeneous clusters, regulating stragglers alone is insufficient to address hardware heterogeneity. On other hand, by addressing these problems, Tarazu achieves significant improvement with a mean speedup of 1.4, which includes high speedups for Shuffle-critical MapReductions (mean of 1.72, not shown) and good speedups for Map-critical MapReductions (mean of 1.3, not shown).

We also implemented and ran Mantri [3] which improves upon LATE's straggler identification. Though not intended for heterogeneous clusters, Mantri's accurate straggler identification may improve performance. However, for our cluster, we found that Mantri does not fare better than LATE for the same reasons stated above.

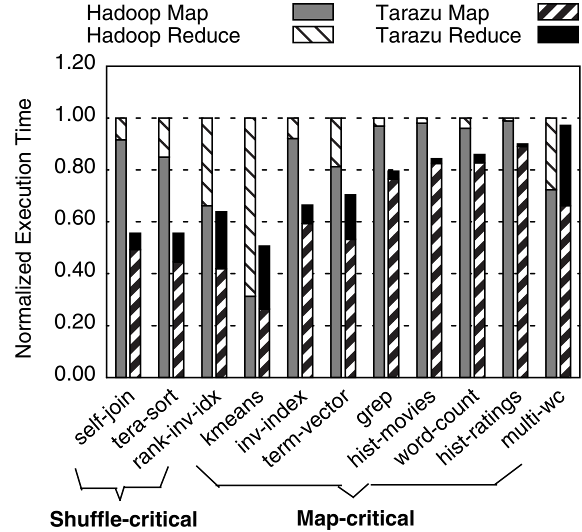

FIGURE 8: Impact of CALB, CAS, and PLB

Finally, although Tarazu performs well, it still lags behind the ideal case due to two main reasons: (1) For Shuffle-critical MapReductions, CALB's on-line measurements for deciding the mode are performed only after some of the Map tasks have executed, by which time some unnecessary remote Map tasks may have been executed. (2) For Map-critical MapReductions, CAS's scheduling is not perfect and incurs some network contention among the remote tasks on different nodes.

In the following sections, we isolate the effects of CALB, CAS, and PLB, and show other data to explain Tarazu's speedups.

5.2 Effect of CALB and CAS

Figure 8 isolates the impact of CALB, CAS, and PLB. Because CAS is triggered only in CALB's task-steal mode (Section), we show the impact of CALB and CAS together. In Figure 8, the Y-axis shows Tarazu's execution times normalized to that of Hadoop. The execution times are broken up into two components, one for the Map phase including the Shuffle and the other for the Reduce phase. CALB and CAS target the Map phase and PLB targets the Reduce phase. Recall that on the Map side, the issues are remote Map traffic which competes with the Shuffle and is bursty, and on the Reduce side, the issue is load imbalance.

CALB predictions of whether each benchmark is Map- or Shuffle-critical agree perfectly with the characterization listed in Table 4. CALB's no-steal mode targets Shuffle-critical MapReductions (CAS is not triggered in this case). From the figure, we see that CALB significantly shortens the Map phase for these MapReductions, contributing to a large fraction of Tarazu's speedups. To

Table 5: Impact of CALB on remote tasks

Benchmark	Fraction of remote tasks on Xeons		Reduction in bisection traffic for Xeons
	Hadoop	Tarazu	
self-join	0.20	0.06	57%
rank-inv-idx	0.29	0.03	52%
tera-sort	0.24	0.08	50%

Table 6: Impact of CAS on remote tasks

Benchmark	Remote task execution times (s)		
	Hadoop	Tarazu	% speedup
kmeans	87.58	82.55	6%
inverted-index	58.59	43.12	36%
term-vector	57.71	43.04	34%
grep	56.00	37.47	49%
histogram-movies	49.31	43.17	14%
word-count	56.56	42.14	34%
histogram-ratings	54.93	45.46	21%
multi-word-count	99.95	94.50	6%

explain these improvements, Table 5 shows the fraction of remote tasks over all Map tasks on Xeon-based nodes in Hadoop and Tarazu, and Tarazu's reduction over Hadoop in the Xeon-based nodes' bisection traffic (Atom-based nodes run only a few remote tasks even in Hadoop).

We see that CALB significantly reduces the number of remote Map tasks and their traffic. By preventing remote tasks on the Xeon-based nodes, CALB allows Atom-based nodes to execute tasks locally hidden under the Shuffle. By avoiding the aggravation of the critical Shuffle due to remote traffic, CALB improves these MapReductions.

CALB's task-steal mode triggers CAS for Map-critical MapReductions. From Figure 8, we see that CAS significantly shortens the Map phase and is responsible for a large fraction of Tarazu's speedups for these MapReductions. To explain these speedups, we compare Hadoop and Tarazu in Figure 9 in terms of the distribution of remote Map traffic over Map execution time and in Table 6 in terms of the remote task execution times. We see that while Hadoop's traffic is bursty at the end of the Map phase, CAS evenly distributes the traffic. As a result, CAS achieves much faster remote task execution (Table 6) which translates to faster completion of the Map phase.

5.3 Effect of PLB

Figure 8 shows PLB's impact on the Reduce phase. In baseline Hadoop, the Reduce phase is much shorter than the Map phase for most of our MapReductions except *ranked-inverted-index, k-means, term-vector,* and *multi-word count*. Consequently, PLB's contributions to Tarazu's speedups are less than those of CALB

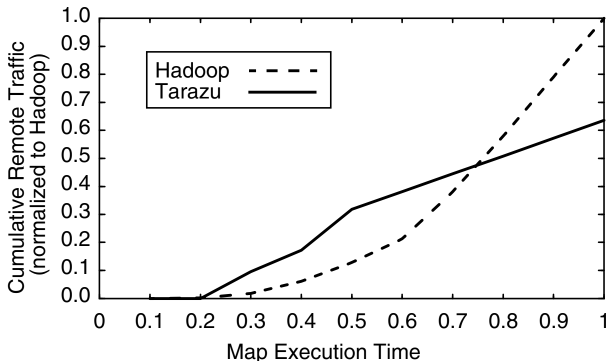

FIGURE 9: Impact of CAS on remote task traffic

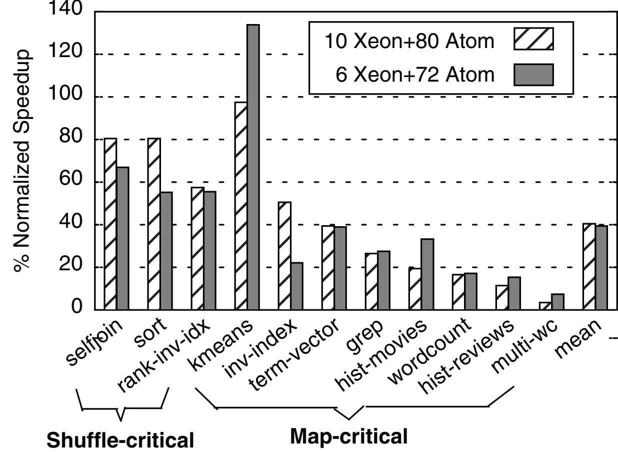

FIGURE 10: Sensitivity to heterogeneity mix

and CAS. PLB improves *k-means* significantly. *k-means* has only six Reduce tasks (number of centroids = 6, as shown in Table 4). Hadoop assigns Reduce tasks to nodes on a first-come-first-served basis where tasks have a high probability of being assigned to the slower Atom-based nodes which are more in number than the Xeon-based nodes (80 versus 10). In contrast, PLB assigns all Reduce tasks to the Xeon-based nodes. In *multi-word-count*, PLB slightly degrades performance because the ratio used for key distribution, $reduceSkewFactor_i$, which is predicted using measurements from the Map phase (Section 3.3), is inaccurate (the Xeon-based nodes are 4.4 times faster than the Atom-based nodes for *multi-word-count*'s Reduce tasks whereas the predicted skew is 1.5). For the other MapReductions, the prediction is reasonably accurate.

5.4 Sensitivity to extent of heterogeneity

We show Tarazu's sensitivity to the extent of heterogeneity by considering two different ratios of the number of Xeon-based nodes (8 cores/node) to that of Atom-based nodes (2 cores/node) — 10 to 80 (default) and 6 to 72. In first cluster, 33% of the cores are Xeons whereas in the second cluster this fraction is 25%. We note that extreme ratios where Xeons or Atoms are dominant make the clusters nearly homogeneous and hence are not interesting. Further, because Atoms are 3-4 times slower than Xeons, ratios with fewer Atom-based cores than Xeon-based cores also lead to Xeon-dominant clusters. Figure 10 shows Tarazu's speedups over Hadoop on these two clusters. We see that Tarazu works well on both the clusters.

5.5 Effect of skewed input data distribution

Recall from Section 2.4 that skewing the input data can reduce the number of remote Map tasks by placing more data on high-performance nodes than on low-power nodes. We show the effect of such skewed input data distribution on Tarazu in Figure 11. The input data is skewed on Xeon-based and Atom-based nodes in the ratio of 3:1 which is the mean $mapComputeRatio_{Xeon,\ Atom}$ for our benchmarks (Table 4). In the figure, the Y-axis shows Tarazu's speedups over Hadoop. From the figure, we see that Tarazu achieves a mean speedup of 1.22, highlighting the fact that a single skew ratio is not optimal for all the benchmarks and therefore skewing does not eliminate Tarazu's opportunity. As expected, this

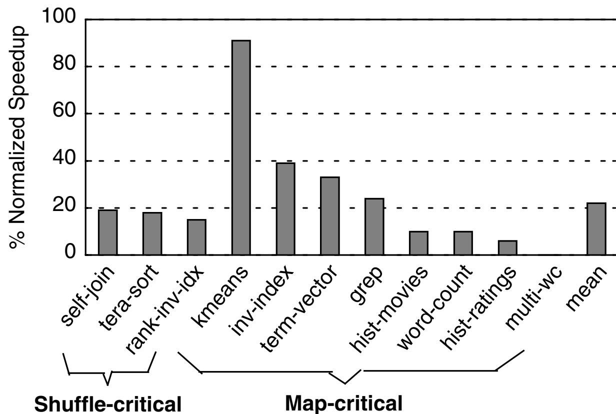

FIGURE 11: Impact of skewed input data distribution

mean speedup is lower than that in the non-skewed case (1.4 in Figure 7) due to fewer remote Map tasks in Hadoop. However, as mentioned in Section 2.4, skewing input data compromises reliability and hence may not be a viable approach. Tarazu achieves better performance than skewing without degrading reliability.

6 Related Work

The interest in heterogeneous clusters has been spurred by recent studies of servers designed with low-power processors [4,20,24]. A number of start-up companies are developing servers using low-power processors, such as [7,30,17]. Hamilton [15] and Lang et al. [23] argue that not all workloads perform well on clusters built with low-power nodes, and [23] specifically demonstrates that enterprise databases do not. While these studies consider only homogeneous clusters constructed with low-power components, or "wimpy" nodes, Chun et al. [9] argue that hybrid data centers that mix nodes designed to a multitude of power/performance points, can handle diverse workloads in an energy-efficient manner. Towards this end, NapSAC, an energy-proportional web server cluster designed using three classes of servers based on Xeon, Atom, and ARM processors was proposed in [22], together with provisioning and load balancing algorithms for such a heterogeneous cluster. Further efforts along this direction were reported in [28]. Our work fundamentally differs from [22,28] in the nature of the workload targeted (interactive and highly time-varying versus batch), and therefore the challenges involved in efficiently utilizing a heterogeneous cluster are quite different.

Several implementations of MapReduce frameworks have been developed that support extensions such as relational data processing [14,11,34,35,27]. In addition, domain-specific libraries that implement various algorithms on top of MapReduce frameworks have been developed [5,33]. Due to the abundance of workloads and applications based on MapReduce, optimizing their performance on heterogeneous clusters is of great importance.

LATE [37] was the first work to point out and address the shortcomings of MapReduce in heterogeneous environments. They specifically focus on the observation that under heterogeneity, the mechanisms built in to MapReduce frameworks for identifying and managing straggler tasks break down, and propose better techniques for identifying, prioritizing, and scheduling backup copies for slow tasks. Techniques to improve the accuracy of progress estimation for tasks in MapReduce were reported in [8]. Mantri [3] explores various causes of outlier tasks in further depth, and develops cause- and resource-aware techniques to act on outliers more intelligently, and earlier in their lifetime.

As described in Section 2, our work addresses fundamentally different issues than [3,8,37]. We demonstrate that despite straggler optimizations, the performance of MapReduce frameworks on clusters with architectural heterogeneity remains poor, and we identify and address the causes of this poor performance. Therefore, we also report results comparing our techniques to baselines that already incorporate straggler optimizations. MapReduce frameworks have also been developed for other types of heterogeneous computing platforms such as multi-core processors and accelerators [16,29,25]. In addition to the fact that our work is complementary to these efforts, we note that the use of accelerators in a subset of the nodes of a cluster would also result in cluster-level heterogeneity, necessitating the use of our techniques (e.g., nodes with GPUs would process data at a faster rate than nodes without GPUs, leading to the performance issues that we outline).

Finally, many recent papers optimize multi-tenant MapReduce jobs on homogeneous clusters [19,12,36]. These papers propose schedulers which, in one way or another, prioritize local tasks over remote tasks across multiple jobs. In a heterogeneous cluster, however, high-performance nodes would run out of local tasks in all the jobs, incurring the remote task problem addressed by Tarazu.

7 Conclusion

Current MapReduce frameworks perform poorly on heterogeneous clusters. We analyzed the performance of MapReduce workloads on a heterogeneous cluster composed of high-performance and low-power nodes, and identified the key causes of poor performance as follows: (1) the load balancing used in MapReduce causes excessive and bursty network communication during the Map phase, which competes with the Shuffle for the scarce bisection bandwidth and (2) the heterogeneity amplifies the Reduce's load imbalance. Addressing these issues require judicious run-time decisions based on application and cluster characteristics (e.g., the decisions of whether, how much, and when to steal tasks depend upon whether Map or Shuffle is critical, and the relative rates of Map computation and communication).

We proposed Tarazu to perform these decisions via Communication-Aware Load Balancing of Map computations (CALB), Communication-Aware Scheduling of Map computations (CAS-CAS), and Predictive Load-Balancing of Reduce computations (PLB). Tarazu incorporates application and cluster characteristics through various heuristics driven by on-line measurements We implemented the proposed techniques in the Hadoop framework, and showed that, on a 90-node heterogeneous cluster and across a suite of 11 benchmarks, Tarazu achieves an average speedup of 40% over Hadoop.

Due to its effectiveness, Tarazu will be valuable in realizing the potential of heterogeneous clusters for the important class of large-scale data-intensive applications. The growing importance and prevalence of heterogeneous clusters suggest that Tarazu will be an important tool for these applications in the future.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. *Proceedings of 20th Intl. Conference on Very Large Data Bases, VLDB*, 1994.

[2] Amazon EC2. http://aws.amazon.com/ec2.

[3] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in Map-Reduce clusters using Mantri. In *Proceedings of the Usenix Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[4] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[5] Apache Mahout: Scalable machine learning and data mining. http://mahout.apache.org.

[6] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. High-performance sorting on networks of workstations. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 243–254, Tucson,Arizona, May 1997.

[7] Calxeda, Inc. http://www.calxeda.com.

[8] Q. Chen, D. Zhang, M. Guo, Q. Deng, and S. Guo. SAMR: A Self-adaptive MapReduce Scheduling Algorithm in Heterogeneous Environment. In *Proceedings of the International Conference on Computer and Information Technology (CIT)*, 2010.

[9] B.-G. Chun, G. Iannaccone, G. Iannaccone, R. Katz, G. Lee, and L. Niccolini. An Energy Case for Hybrid Datacenters. In *SOSP Workshop on Power Aware Computing and Systems (HotPower)*, 2009.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, pages 107–113, Jan. 2008.

[11] Facebook Hive. http://hadoop.apache.org/hive.

[12] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation(NSDI)*, 2011.

[13] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *Proceedings of the SIGCOMM conference on Data Communication*, pages 51–62, 2009.

[14] Hadoop. http://lucene.apache.org/hadoop/.

[15] J. Hamilton. When Very Low-Power, Low-Cost Servers Don't Make Sense. In *http://perspectives.mvdirona.com/2010/05/18/WhenVeryLowPowerLowCostServersDontMakeSense.aspx*, 2010.

[16] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A MapReduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel Architectures and Compilation Techniques*, pages 260–269, 2008.

[17] HP Labs. Project Moonshot. In *http://www.hp.com/hpinfo/newsroom/press_kits/2011/MoonshotInfrastructure/index.html*, 2011.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

[19] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the 22nd symposium on Operating systems principles*, SOSP, pages 261–276, USA, 2009.

[20] V. Janapa Reddi, B. C. Lee, T. Chilimbi, and K. Vaid. Web search using mobile cores: Quantifying and mitigating the price of efficiency. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2010.

[21] J.Hartigan. *Clustering Algorithms. Wiley*, 1975.

[22] A. Krioukov, P. Mohan, S. Alspaugh, L. Keys, D. E. Culler, and R. H. Katz. NapSAC: Design and implementation of a power-proportional web cluster. *ACM Computer Communication Review*, 41(1), 2011.

[23] W. Lang, J. M. Patel, and S. Shankar. Wimpy node clusters: What about non-wimpy workloads? In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, 2010.

[24] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt. Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments. In *Proceedings of the 35th Annual International Symposium on Computer Architecture* (ISCA), pages 315–326, 2008.

[25] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: A programming model for heterogeneous multi-core systems. In *Proceedings of the 13th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[26] Netflix movies data. *http://www.netflixprize.com/download*.

[27] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 international conference on Management Of Data*, SIGMOD, pages 1099–1110, 2008.

[28] M. M. Rafique, N. Ravi, S. Cadambi, S. T. Chakradhar, and A. R. Butt. Power Management for Heterogeneous Clusters: An Experimental Study. In *Proceedings of the IEEE International Conference on Green Computing*, 2011.

[29] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *13th International Symposium on High Performance Computer Architecture, HPCA*, pages 13–24, 2007.

[30] SeaMicro, Inc. http://www.seamicro.com.

[31] A. Vahdat, M. Al-Fares, N. Farrington, R. N. Mysore, G. Porter, and S. Radhakrishnan. Scale-Out Networking in the Data Center. *IEEE Micro*, 30:29–41, July 2010.

[32] D. Weld. Lecture notes on MapReduce (based on Jeff Dean's slides). *http://rakaposhi.eas.asu.edu/cse494/notes/s07-map-reduce.ppt*, 2007.

[33] X-RIME: Hadoop based large scale social network analysis. http://xrime.sourceforge.net/.

[34] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: Simplified relational data processing on large clusters. In *Proceedings of the SIGMOD international conference on Management Of Data*, 2007.

[35] 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 *Proceedings of International Symposium on Operating System Design and Implementation (OSDI)*, 2008.

[36] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 265–278, 2010.

[37] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the Usenix Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[38] Hadoop rebalancer. http://hadoop.apache.org/common/docs/r0.17.2/hdfs_user_guide.html#Rebalancer.