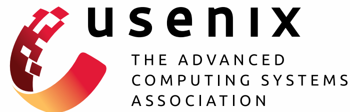

ShuffleWatcher: Shuffle-aware Scheduling in Multi-tenant MapReduce Clusters

Faraz Ahmad, *Teradata Aster and Purdue University;* Srimat T. Chakradhar, *NEC Laboratories America;* Anand Raghunathan and T. N. Vijaykumar, *Purdue University*

https://www.usenix.org/conference/atc14/technical-sessions/presentation/ahmad

This paper is included in the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference.

June 19–20, 2014 • Philadelphia, PA

978-1-931971-10-2

Open access to the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference is sponsored by USENIX.

ShuffleWatcher: Shuffle-aware Scheduling in Multi-tenant MapReduce Clusters*

Faraz Ahmad*,†, Srimat T. Chakradhar‡, Anand Raghunathan†, T. N. Vijaykumar†
*Teradata Aster, San Carlos, CA, USA, ‡NEC Laboratories America, Princeton, NJ, USA
†School of Electrical and Computer Engineering, Purdue University, IN, USA
faraz.ahmad@teradata.com, chak@nec-labs.com, {raghunathan,vijay}@ecn.purdue.edu

Abstract

MapReduce clusters are usually multi-tenant (i.e., shared among multiple users and jobs) for improving cost and utilization. The performance of jobs in a multi-tenant MapReduce cluster is greatly impacted by the all-Map-to-all-Reduce communication, or Shuffle, which saturates the cluster's hard-to-scale network bisection bandwidth. Previous schedulers optimize Map input locality but do not consider the Shuffle, which is often the dominant source of traffic in MapReduce clusters.

We propose *ShuffleWatcher*, a new multi-tenant MapReduce scheduler that shapes and reduces Shuffle traffic to improve cluster performance (throughput and job turn-around times), while operating within specified fairness constraints. ShuffleWatcher employs three key techniques. First, it curbs intra-job Map-Shuffle concurrency to shape Shuffle traffic by delaying or elongating a job's Shuffle based on the network load. Second, it exploits the reduced intra-job concurrency and the flexibility engendered by the replication of Map input data for fault tolerance to preferentially assign a job's Map tasks to localize the Map output to as few nodes as possible. Third, it exploits localized Map output and delayed Shuffle to reduce the Shuffle traffic by preferentially assigning a job's Reduce tasks to the nodes containing its Map output. ShuffleWatcher leverages opportunities that are unique to multi-tenancy, such overlapping Map with Shuffle across jobs rather than within a job, and trading-off intra-job concurrency for reduced Shuffle traffic. On a 100-node Amazon EC2 cluster running Hadoop, ShuffleWatcher improves cluster throughput by 39-46% and job turn-around times by 27-32% over three state-of-the-art schedulers.

1 Introduction

MapReduce frameworks are commonly used to process large volumes of data on clusters of commodity computers. MapReduce provides easy programmability, automatic data parallelization and transparent fault tolerance [13]. For cost-effectiveness and better utilization, MapReduce clusters are frequently *multi-tenant* (i.e., shared among multiple users and jobs).

The performance of MapReduce clusters is greatly affected by the Shuffle, an all-Map-to-all-Reduce communication, which stresses the network bisection bandwidth. Typical MapReduce workloads contain a significant fraction of Shuffle-heavy jobs (e.g., 60% and 20% of the jobs on the Yahoo and Facebook clusters, respectively, are reported to be Shuffle-heavy [9,39]). Shuffle-heavy MapReduce jobs typically process more data in the Shuffle and Reduce phases and hence run much longer than Shuffle-light jobs [2,3]. As such, Shuffle-heavy jobs significantly impact the cluster throughput. The execution of multiple, concurrent Shuffles due to multi-tenancy worsens the pressure on the network bisection bandwidth. While network switch and *link* bandwidth scale with hardware technology, *bisection* bandwidth is a global resource that is hard to scale up with the cluster's compute and storage resources (CPU, memory, disk). Even with recent advances in data center networks [4], large clusters are typically provisioned for *per-node* bisection bandwidth that is 5-20 times lower than the within-rack bandwidth [13,17,35,37,39].

Several previous multi-tenant schedulers address the problem of fairness among users or jobs (e.g., FIFO [21], Capacity [30], Fair [31] and Dominant Resource Fairness [16] schedulers). Other efforts improve cluster throughput by optimizing data locality in the Map phase [25,39] but do not address Shuffle, which is the dominant source of network traffic in MapReduce. Our goal is to improve performance (cluster throughput and job turn-around time) within specified fairness constraints addressing the Shuffle bottleneck. Recent efforts [10,12,32] propose techniques to manage data center network traffic without changing network load. In contrast, we actively shape and reduce the network load.

We propose ShuffleWatcher, a new multi-tenant MapReduce scheduler that improves performance by exploiting a key trade-off between intra-job concurrency and Shuffle locality. Previous multi-tenant schedulers adopt the approach of maximizing intra-job concurrency, while ensuring a fair division of resources among users. This approach is a carryover from single-tenant scheduling; when there is a single job, utilizing the entire cluster (highest intra-job concurrency) typically ensures fastest turn-around times. Hence, concurrency is not traded for locality (although locality optimizations without sacrificing concurrency are welcome). However, adopting this concurrency-centric scheduling approach in multi-tenancy is neither neces-

*work was done while Faraz Ahmad was at Purdue University.

sary (because concurrency may be exploited either within or across jobs), nor beneficial (because it often results in multiple concurrent, contending Shuffles, which saturate the network and degrade throughput).

ShuffleWatcher employs three key mechanisms that leverage the aforementioned trade-off. The first mechanism, called *Network-Aware Shuffle Scheduling (NASS)*, curbs intra-job concurrency at high network loads to shape the Shuffle traffic. Previous schedulers typically overlap a job's Shuffle with its own Map phase by creating and scheduling the Reduce tasks early in the Map phase. This rigidity in Reduce scheduling often results in multiple Shuffle-heavy jobs being concurrently scheduled, thereby saturating the network bisection bandwidth and hurting performance. We make the key observation that multi-tenancy presents a new degree of freedom to overlap the Shuffle and Map across jobs, rather than within a job. Accordingly, NASS curbs the intra-job Map-Shuffle concurrency at high network loads by delaying or elongating a job's Shuffle. This profitably shapes the network traffic to alleviate congestion, while still achieving Map-Shuffle overlap across jobs. To maintain fairness for the user whose Shuffle is delayed, ShuffleWatcher schedules tasks (from the same or another job) of the same user that do not stress the network. ShuffleWatcher defaults to favoring intra-job concurrency when the load is low.

The other two mechanisms of ShuffleWatcher exploit the fact that in multi-tenancy, each job inevitably experiences reduced concurrency due to resource sharing with other jobs. ShuffleWatcher employs *Shuffle-aware Map Placement (SAMP)* on the Map side to trade this reduced concurrency with higher Shuffle locality. SAMP is based on the following assertion. Given a favorable Shuffle and Reduce schedule, a job's Map assignment (e.g., to as few sub-clusters as possible) that optimizes Map plus Shuffle locality results in higher network traffic reductions compared to one that optimizes Map locality alone as done by previous schedulers. SAMP leverages input data replication to optimize the sum of Map and Shuffle locality. In contrast to previous schedulers, SAMP may forgo some Map locality to achieve higher Shuffle locality. The favorable Shuffle and Reduce schedule is ensured by the other two mechanisms of ShuffleWatcher.

ShuffleWatcher employs *Shuffle-aware Reduce placement (SARP)* on the Reduce side to achieve higher Shuffle locality. Previous schedulers assign Reduce tasks to whichever node becomes free, assuming a uniform Map placement, and consequently, Map output distribution throughout the cluster. Such an assumption that may generally hold true for single-tenancy is no longer valid for multi-tenancy due to reduced concurrency per job. As a result, previous multi-tenant schedulers unnecessarily spread out a job's Shuffle in the cluster. SARP is based on the following assertion. Given a favorable Map and Shuffle schedule, a job's Reduce assignment that optimizes Shuffle locality results in higher network traffic reductions compared to one that randomly distributes Reduce tasks. The favorable Map and Shuffle schedule is ensured by the other two mechanisms of ShuffleWatcher. In ShuffleWatcher, most of the Map tasks finish before the Reduce tasks are scheduled whenever NASS delays the Shuffle and, therefore, the distribution of the intermediate data is known. SARP preferentially assigns each job's Reduce tasks to sub-clusters based on how much of the job's intermediate data they contain. Thus, SARP localizes most of the Shuffle and reduces cross-bisection Shuffle traffic.

We implement ShuffleWatcher in Hadoop [21] combined with Fair Scheduler [31]. On a 100-node Amazon EC2 cluster, ShuffleWatcher achieves 46% higher throughput and 48% reduced network traffic compared to Delay Scheduling [39] while improving job turn-around times by 32%. One may think that by trading-off intra-job concurrency for Shuffle locality, ShuffleWatcher may sacrifice turn-around times to gain throughput; on the contrary, by improving Shuffle locality and temporally balancing Shuffle traffic, ShuffleWatcher improves turn-around times, not only on average but of all 300 jobs in our experiments.

The rest of the paper is organized as follows. We provide a brief overview of multi-tenant scheduling in Section 2, and describe ShuffleWatcher in Section 3. We present our experimental methodology in Section 4, and results in Section 5. We discuss related work in Section 6 and conclude in Section 7.

2 Background, Challenges and Opportunities

We provide a brief background on scheduling in MapReduce clusters and discuss challenges and opportunities offered by multi-tenant clusters.

2.1 MapReduce Job Execution

We begin with the aspects of a MapReduce job's execution that are relevant to multi-tenant scheduling.

When executing a MapReduce job, Map and Reduce tasks are scheduled to maximize concurrency (i.e., occupy the entire cluster or as much as possible). Consequently, the all-Map-to-all-Reduce Shuffle results in an all-nodes-to-all-nodes communication, which stresses the network bisection bandwidth [13,17,35,37,39].

To improve performance, a job's Shuffle is overlapped with its own Map phase (i.e., the Shuffle of data produced by earlier Map tasks occurs while later Map tasks execute). To achieve this overlap, the scheduler must assign Reduce tasks (which perform the Shuffle) to nodes very early in the Map phase, before most of the Map tasks have even begun execution. Two key implications of this approach are: (1) The Shuffle's schedule is

fixed rigidly relative to the Map phase and cannot change dynamically in response to network load, and (2) the distribution of intermediate data is not known when the Reduce tasks are assigned. In single-tenancy, because the Map tasks are spread out across the entire cluster, a random assignment of Reduce tasks to nodes is close to optimal because it not only exploits full concurrency. but also achieves the overlap with the Map phase.

To reduce network traffic by exploiting locality, the scheduler attempts to assign a Map task to either the node or the rack that holds the task's input data. However, Shuffle locality is not considered because Reduce tasks are scheduled well before the distribution of Map output is known to gain the above-mentioned benefits.

2.2 Multi-tenant Scheduling

In a multi-tenant environment, users submit jobs to the scheduler, which enqueues and assigns the jobs' Map and Reduce tasks to nodes based on a specified fairness policy. Many fairness policies have been proposed. For example, Fair Scheduler [31] uses a share-based order among users, while Dynamic Resource Fairness (DRF) [16] ensures that the critical resource shares are equalized across users. From a scheduling perspective, the fairness policy effectively determines to which user's jobs should an available node be allocated. Among the chosen user's tasks, the earliest enqueued task that fits resources of the available node is scheduled. Under multi-tenancy, a single job does not have access to the full resources of the cluster, and therefore sees an inevitable reduction in concurrency.

Previous multi-tenant schedulers preserve Map-input locality by scheduling Map tasks on nodes or racks that have the corresponding input data. They also retain the approach of overlapping Map and Shuffle phases within each job, spread out the Map and Reduce tasks of a job for maximal concurrency, and assign the tasks to nodes irrespective of how much Shuffle data they consume.

In this context, we wish to improve throughput and job latency while obeying the specified fairness criteria.

2.3 Challenges and Opportunities in Multi-tenant Scheduling

The key challenge in multi-tenancy is that the execution of multiple concurrent shuffle-heavy jobs severely stresses the network bisection bandwidth. Previous schedulers optimize for Map-input traffic but not for Shuffle traffic. Moreover, they do not differentiate between Shuffle-heavy and Shuffle-light jobs and may concurrently schedule multiple Shuffle-heavy jobs, worsening the impact of network saturation. Such saturation affects all running jobs (not just the Shuffle-heavy jobs), and severely degrades cluster throughput as well as individual job turn-around times.

While network switch and link bandwidth scale with

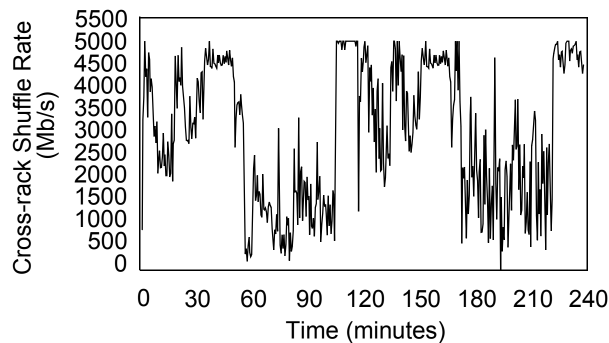

FIGURE 1: Shuffle profile in 100-node EC2 multi-tenant cluster

hardware technology, bisection bandwidth is a global resource that is expensive to scale up with the cluster's computational resources. Previous work [4] has proposed new topologies that achieve high bisection bandwidth without requiring custom, high-end switches. Nevertheless, provisioning for peak network bisection bandwidth requirements is still quite expensive, and wasteful because the full bandwidth is not utilized at all times [42]. Hence, clusters typically provide lower bandwidths at the aggregation and core layers of the network topology than at the cluster edges (i.e., the links to nodes). This bandwidth over-subscription results in significant cost savings. Large clusters usually have bandwidth over-subscription ratios ranging from 5:1 to 20:1 or even higher. Therefore, when all nodes are concurrently communicating (as in Shuffle), the bisection bandwidth available *per node* is still much less than bandwidth available within a rack (e.g., 50-200 Mbps compared to 1 Gbps within rack [13,17,35,37,39]).

While multi-tenancy poses the above challenge, it also offers new opportunities.

- Multi-tenant workloads often include a significant fraction of shuffle-light jobs [9,39], which may be overlapped with shuffle-heavy jobs without exacerbating the load on the cluster network. Current schedulers are Shuffle-unaware, resulting in periods of relatively high and low Shuffle activity in the cluster. Figure 1 shows the measured Shuffle traffic vs. time in a 100-node Amazon EC2 cluster running a workload mix representative of Yahoo and Facebook clusters [9,39]. From the figure, we see that network load is quite bursty and saturates the network during some periods, while leaving it under-utilized during other periods. This creates an opportunity to create a more temporally balanced network load.
- Unlike single-tenancy, where intra-job Map-Shuffle overlap is critical and delaying the Shuffle invariably hurts performance, multi-tenancy affords the possibility of achieving such overlap across jobs, creating an opportunity to flexibly schedule a job's Shuffle.
- In multi-tenancy, each job gets only a fraction of the cluster resources for its execution. Such reduced concurrency results into a skewed intermediate data distribu-

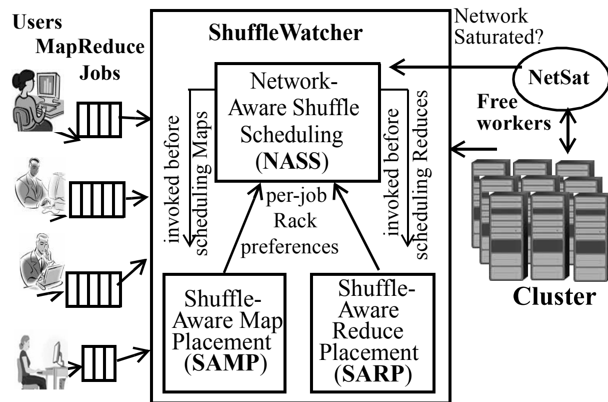

FIGURE 2: Overview of ShuffleWatcher

tion, creating an opportunity to exploit Shuffle locality.

ShuffleWatcher exploits these opportunities to shape and to reduce the Shuffle traffic as described in the next section. Note that delaying Shuffle, or localizing Map and Reduce tasks of a job can be achieved without losing fairness by exploiting the choice among a user's many jobs and tasks. For example, a user whose Reduce task is delayed to alleviate network load need not lose her turn. Instead, ShuffleWatcher schedules a Map or Reduce task of the same user whose input or intermediate data is present on the node. In effect, ShuffleWatcher operates within the confines of a specified fairness policy.

Many of the opportunities described above require curbing a single job's concurrency. We show that such curbing can be done without hurting (and on the contrary, often improving) both cluster throughput and job turn-around times. Multi-tenancy implies that the cluster is shared among multiple jobs, so the concurrency available to each job is anyhow restricted. The aforementioned choice among a user's jobs and tasks is typically sufficient to fully utilize cluster's resources, and any loss in concurrency for a job is more than offset by the significant performance improvement due to Shuffle locality.

3 ShuffleWatcher

Figure 2 shows a high-level overview of ShuffleWatcher. Like other multi-tenant schedulers, ShuffleWatcher receives job submissions from one or more users. The scheduler monitors the status of nodes in the cluster, and schedules Map and Reduce tasks to them as they become available. ShuffleWatcher consists of three components: *Network-Aware Shuffle Scheduling (NASS)* (Section 3.1), *Shuffle-Aware Map Placement (SAMP* (Section 3.2), and *Shuffle-Aware Reduce Placement (SARP)* (Section 3.3).

3.1 Network-aware Shuffle Scheduling (NASS)

Figure 3 shows a high-level overview of steps performed by NASS, which is invoked whenever a worker node in the cluster requests a new task. First, NASS picks a user to whom the node should be allocated as per

Invoked when a worker on rack *r* requests a task

1. Select user based on fairness policy.
2. if (NetworkSaturated) {
3. find a task of selected user in the following order:
4. Map task for which *r* is in *PreferredMapRacks* (from SAMP)
5. Local Map task of any job
6. Any available Map task
7. Reduce task of any Shuffle-light job
8. Reduce task of Shuffle-heavy job for which *PreferredReducesPerRack[r]* is not met (from SARP)
9. Any available Reduce task
10. }
11. else {
12. find a task of selected user in the following order:
13. Reduce task of Shuffle-heavy job for which *PreferredReducesPerRack[r]* is not met (from SARP)
14. Reduce task of any Shuffle-heavy job
15. Any available Reduce task
16. Map task for which *r* is in *PreferredMapRacks* (from SAMP)
17. Local Map task of any job
18. Any available Map task
19. }

FIGURE 3: NASS Algorithm

fairness criteria (line 1) which can be based on any of the policies proposed previously [16,21,30,31]. Tasks only from the selected user's jobs are considered in the remaining steps (lines 2-19) of NASS to ensure the user's fair share.

The remaining steps in NASS, which differ significantly from previous MapReduce schedulers, are responsible for shaping the Shuffle traffic by exploiting the concurrency-locality trade-off (Section 2). This trade-off is driven by the network load as monitored by a daemon, called *NetSat*, which periodically determines each node's cross-rack traffic of all jobs due to the Shuffle, remote Map input reads, and Reduce output writes. NetSat compares the ratio of the traffic and the cross-rack bandwidth available to the node against a threshold, called *NWSaturationThreshold*, to set a flag, called *NetworkSaturated,* when the ratio exceeds the threshold. We found that *NWSaturationThreshold* can be in the broad 75-100% range and result in less than 1% difference in cluster throughput. While our current NetSat implementation uses only the limited notions of within-rack and cross-rack traffic, more precise information about the network topology or network congestion monitoring mechanisms, when available, can be used. Similarly, while *NetSat* currently monitors network traffic only due to MapReduce jobs in the cluster, the daemon can be modified to account for traffic from other applications running concurrently in the cluster (e.g., interactive workloads and MPI jobs).

If *NetworkSaturated* is true, NASS orders the Map and Reduce tasks so as to reduce the load on the network (lines 3-9). In this ordering, NASS curbs intra-job Map-Shuffle concurrency by preferring Map tasks and delay-

Invoked when a new job *j* is submitted

1. Sort racks in decreasing order of input data for *j*
2. *TmpMapRacks* = {}
3. *CrossRackTraffic* = infinity
4. do {
5. remove first rack *r* in sorted list and add to *TmpMapRacks*
6. estimate *CrossRackTraffic* = remote Map traffic + cross-rack Shuffle //assumes SARP is used.
7. } while (*CrossRackTraffic* decreases)
8. *PreferredMapRacks* = *TmpMapRacks*
9. compute *TentativeReducesPerRack* for SARP assuming Map tasks are scheduled on *PreferredMapRacks*

FIGURE 4: SAMP Algorithm

ing Reduce tasks and the associated Shuffle. NASS looks for a Map task of the selected user in the following categories listed in the order of increasing network load (lines 4-6): Map tasks for which the rack of the node requesting work is in SAMP's *PreferredMapRacks* (line 4) as described in Section 3.2; remaining local Map tasks (line 5), and remaining Map tasks (line 6). For local Map tasks, NASS obeys the locality-driven Map scheduling typically used in previous schedulers where node-local and rack-local tasks are explored in that order. In some cases, SAMP may explicitly decide that incurring some remote Map tasks reduces the total (remote Map + Shuffle) traffic. Such Map tasks are covered in SAMP's *PreferredMapRacks*. Within each category, the tasks are ordered by job arrival times.

If there is no available Map task, NASS looks for a Reduce task of the selected user in the order of increasing Shuffle volume (lines 7-9). The jobs are categorized into *Shuffle-heavy* and *Shuffle-light* based on the Shuffle-to-Map-input volume ratio, called *ShuffleInputRatio* (ratio > 1.0 indicates a Shuffle-heavy job). Our current implementation initializes this ratio to be 1.0 and dynamically updates the value as the Map phase of a job progresses. This ratio could also be provided by the user, if known in advance, or tracked from previous runs of the job. NASS's preferred order for Reduce tasks is: Shuffle-light (line 7), Shuffle-heavy from a job for which fewer than the desired number of Reduces as identified by SARP's *PreferredReducesPerRack* have been executed (line 8) as described in Section 3.3, followed by any Reduce task (line 9).

If *NetworkSaturated* is false, NASS defaults to favoring high intra-job concurrency by prioritizing Reduce tasks (and hence the Shuffle) over Map tasks (lines 12-18). Accordingly, NASS prioritizes Shuffle-heavy Reduce tasks preferred by SARP, followed by Reduce tasks of any Shuffle-heavy job to fully utilize the available bandwidth followed by Reduce tasks of any Shuffle-light job (lines 13-15). In the absence of a Reduce task, NASS schedules a Map task following the same preference order as in the saturated-network case to improve locality (lines 16-18). Because NASS is guaranteed to choose either a Map task (line 6 and line 18) or a Reduce task (line 9 and line 15) of the selected user irrespective of network saturation, NASS maintains fairness. One may think that because NASS maintains per-user fairness but not per-job fairness, NASS may either hurt the turn-around times of some jobs or not perform well under per-job fairness. We address both these concerns in our results and show that neither of these concerns is true (Section 5.2 and Section 5.4, respectively).

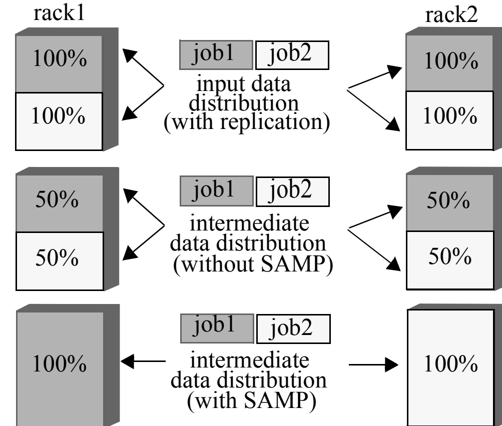

FIGURE 5: Cross-rack Shuffle Reduction with SAMP

3.2 Shuffle-aware Map Placement (SAMP)

Recall from Section 2.3 that in multi-tenancy, each job experiences reduced concurrency resulting into a skewed intermediate data distribution. The Map-input locality driven Map scheduling employed in previous MapReduce schedulers [21, 39] does not consider Shuffle locality. SAMP goes beyond previous schedulers in two ways: (i) it leverages input data replication to localize intermediate data for a job, by prioritizing the execution of some replicas over others, and (ii) it allows remote execution of Map tasks when the resulting remote Map input traffic is outweighed by the Shuffle traffic reduction due to localized intermediate data. Such restriction of a job's Map tasks to a subset of nodes or racks achieves high Shuffle locality at the expense of full intra-job concurrency, which is anyway not available in multi-tenancy. SAMP relies on NASS and SARP to exploit Shuffle locality in later phases of a job execution.

The procedure used by SAMP is shown in Figure 4. SAMP is triggered once per job, at the time of job submission. Based on the locations of a job's input data, SAMP prepares a sorted list of racks in decreasing order of the amount of the job's input data that they contain (line 1). SAMP initializes a list of racks, *TmpMapRacks*, (line 2) and a variable *CrossRackTraffic* to measure cross-rack traffic (line 3). Next, SAMP keeps adding racks to *TmpMapRacks* in the sorted order, and computes the resulting *CrossRackTraffic* as the sum of remote Map traffic incurred and cross-rack Shuffle vol-

Invoked when job *j* schedules its Reduce tasks

```
1.if (fraction of Maps completed > MapCompletionThreshold)
2.    for each rack r
3.      PreferredReducesPerRack[r] = NumReduces * Intermediate
            data size on rack r / Current Intermediate data size of j
4.} else
5.    for each rack r
6.      PreferredReducesPerRack[r] = TentativeReducesPerRack[r]
```

FIGURE 6: SARP Algorithm

ume until this sum is minimized (lines 4-7). Remote Map traffic is estimated from the fraction of the job's input data that does not reside on the racks in *TmpMapRacks*, whereas cross-rack Shuffle volume is estimated based on the job's *ShuffleInputRatio* (Section 3.1) and assuming SARP's Reduce placement. The final list of racks *TmpMapRacks* is assigned to *PreferredMapRacks* (line 8) which is then communicated to NASS for task scheduling (Section 3.1). SAMP also computes an estimated number of Reduce tasks for each rack (*TentativeReducesPerRack)* assuming that the Map tasks are scheduled on *PreferredMapRacks*. *TentativeReducesPerRack* is used by SARP when SARP is invoked after only a few Map tasks complete, (i.e., intermediate data locations are unavailable) (Figure 6, lines 4-6).

To highlight the advantages of SAMP, Figure 5 shows a simple example with two jobs, *job1* and *job2*, whose input data is replicated and available on two racks, *rack1* and *rack2*. Because the nodes of each rack become available to execute tasks at roughly the same rate, previous schedulers would assign equal numbers of Map tasks for each job to the nodes within each rack, without incurring remote Map traffic. However, such scheduling results in uniform intermediate data distribution for both jobs, creating little opportunity for SARP to reduce cross-rack Shuffle traffic. In contrast, SAMP's *PreferredMapRacks* selection (*rack1* for *job1*, *rack2* for *job2*) results in a schedule that places all the intermediate data for *job1* on *rack1* and for *job2* on *rack2*, which SARP can exploit to reduce cross-rack Shuffle traffic for both jobs. In this example, note that SAMP exploits input data replication to avoid remote Map traffic.

3.3 Shuffle-aware Reduce Placement (SARP)

Recall from Section 2.3 that the intermediate data distribution is likely to be skewed in a multi-tenant cluster. The scheduling of Reduce tasks solely based on node availability increases the volume of cross-rack Shuffle traffic. Shuffle-aware Reduce placement (SARP) exploits SAMP's Map assignment and NASS's delayed Reduce scheduling to localize most of the Shuffle within racks. SARP achieves this localization by computing a preferred number of Reduce tasks on each rack based on the amount of intermediate data the rack holds.

SARP's algorithm is shown in Figure 6. SARP is

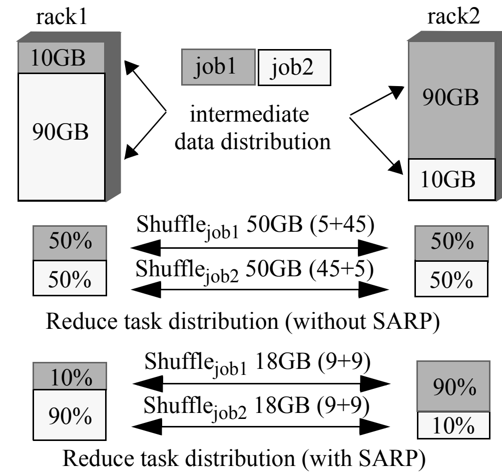

FIGURE 7: Cross-rack Shuffle reduction with SARP

invoked when the Reduce tasks of a job are first enabled for scheduling. This enabling is done when either the Map phase is complete or there are unoccupied slots for the user to schedule a task. SARP first checks whether sufficient number of Map tasks have been completed for the job and significant intermediate data has been accumulated, in order to decrease the chances of poor rack preferences. This check is done by comparing the fraction of completed Maps to a threshold, *MapCompletionThreshold* (line 1). Because NASS schedules Reduce tasks as late as possible, this criterion is satisfied in the common case. SARP then computes (lines 2-3) the preferred number of Reduce tasks for the job on a rack, *PreferredReducesPerRack*, by multiplying the job's total number of Reduce tasks (typically specified by the user in current systems) with the fraction of intermediate data residing on the rack. We found that *MapCompletionThreshold* can be in the broad 5-25% range for less than 1% difference in cluster throughput.

In the (uncommon) case when SARP is invoked for a job before sufficient Maps have been executed, SARP relies on SAMP's predictive analysis to decide how many Reduce tasks should be executed on each rack (*TentativeReducePerRack* (Section 3.2)) (lines 5-6).

SARP computes the preferred number of Reduce tasks per rack but does not specify which Reduce task is scheduled on which rack. Because the intermediate data on each rack is likely to contain most or all of the keys (of MapReduce key-value pairs), any Reduce task can be scheduled on a given rack. Therefore, NASS chooses as many available Reduce tasks as specified by SARP.

One may think that SARP's localization may unbalance load across nodes or racks. Because a free node is always assigned some work (preferred or otherwise -line 9 or 18 in Figure 3), such imbalance does not occur.

Figure 7 illustrates a simple example showing SARP's Shuffle traffic reduction in a cluster for two jobs whose data resides only on two racks. When the Reduce

Table 1: Benchmarks Characterization

Shuffle-heavy	terasort(5%), ranked-inverted-index(10%), self-join(10%), word-sequence-count(10%), adjacency-list(5%)
Shuffle-medium	inverted-index(10%), term-vector(10%)
Shuffle-light	grep(15%), wordcount(10%), classification(5%), histogram-movies(5%), histogram-ratings(5%)

tasks are evenly distributed among the racks, both jobs need to transfer half (50GB) of their intermediate data from one rack to the other. With SARP, each job needs to transfer only 18GB of intermediate data across the racks reducing the cross-rack Shuffle traffic by 64%. The reductions in Shuffle traffic is even greater for cases where previous schedulers' random placement assigns all Reduce tasks of job1 to rack1 and of job2 to rack2.

3.4 Discussion

We end our description of ShuffleWatcher by providing a few additional insights and clarifications.

One possible alternative to ShuffleWatcher is simply to reduce the Shuffle traffic by assigning Reduce tasks to machines or racks in proportion to the distribution of the intermediate (Map output) data or the input data. While ShuffleWatcher achieves the same effect by preferentially assigning Reduce tasks to the nodes containing the intermediate data, the alternative approach is simpler. However, such an approach does not consider other equally-important aspects of multi-tenancy, such as job latency, cluster utilization, and fairness. For example, to schedule Reduce tasks based solely on the intermediate data distribution, the scheduler must delay the Shuffle until the Map phase is complete, entirely losing the opportunity for intra-job Shuffle-Map concurrency and potentially increasing job latency. Alternately, scheduling Reduce tasks based solely on the input data distribution eliminates exploiting any additional skew in the intermediate data. In both cases, fixing the assignment of Reduce tasks to machines leaves the scheduler with limited flexibility. If none of a job's tasks can be executed on a free machine due to the assignment, then resources are under-utilized. As such, the scheduler must reduce Shuffle volume while considering these other important aspects, which preclude a simple or fixed assignment and necessitate the more complete approach of ShuffleWatcher. Finally, the alternative approach does not temporally shape the Shuffle traffic and therefore does not capture a significant part of ShuffleWatcher's improvements (more than 40% in Figure 12).

The mechanisms for tracking job execution, fault tolerance, straggler identification and backup task execution, are not modified by ShuffleWatcher.

In rare cases, Shuffle-aware scheduling employed by ShuffleWatcher may result in a particular job getting starved due to unavailability of preferred racks (caused by load or failure), while other jobs from the same user are executed to satisfy the fairness constraint. ShuffleWatcher addresses this problem by tracking job submission times at the granularity of a window such that a user's jobs submitted in an earlier window are prioritized over those submitted in a later one, overriding the heuristics in NASS, SAMP and SARP. The window width acts like a time-out interval and can be set as some multiple of the average job completion time. A window of 10 minutes was enough to prevent starvation of any jobs in our cluster.

Table 2: Distribution of job sizes

Input job sizes	% jobs	Input job sizes	% jobs
< 100MB	20%	100GB - 200GB	10%
100MB- 1GB	19%	200GB - 500GB	7%
1GB - 20GB	21%	500GB - 1TB	8%
20GB - 100GB	10%	> 1TB	5%

By default, ShuffleWatcher improves performance while strictly obeying the constraints provided by any fairness policy (we evaluate ShuffleWatcher using two such policies in Section 5). However, ShuffleWatcher can be operated under relaxed fairness constraints (e.g., as employed in Delay Scheduling [39]). We evaluate the impact of relaxed fairness constraints in Section 5.

Finally, although ShuffleWatcher performs additional steps (Figure 2) compared to current schedulers, these steps do not impact scalability as they are executed either periodically at a low frequency (NetSat) or only once per job (SAMP and SARP). The computation in NASS is quite simple, and adds negligible overheads.

4 Experimental Methodology

We implement ShuffleWatcher in Hadoop (version 1.0.0) [21], and evaluate on a 100-node testbed in Amazon's Elastic Compute Cloud (EC2) [5].

4.1 Testbed

In the 100-node cluster, we use "extra-large" instances, each with 4 virtual cores and 15 GB of memory. EC2 does not provide any information about the underlying network topology or physical locations of the instances. In large clusters, the cross-rack bandwidth is usually much lower than the within-rack bandwidth [13, 21, 25, 39]. To emulate a cluster with realistic bandwidths and to distinguish the nodes from each other based on their location (e.g., rack-local versus rack-remote), we divide our cluster into 10 sub-clusters of 10 nodes each. We identify the sub-clusters by their elastic IP addresses assigned based on their location in EC2. We use the network utility tools *tc* and *iptables* to limit the aggregate bandwidth from one sub-cluster to another to 500 Mbps (50 Mbps is the typical per-node bisection bandwidth [13, 37, 35, 12]), without limiting the band-

widths within each sub-cluster. Because the aggregate bandwidth, and not individual link bandwidths, is limited, a subset of nodes can fully utilize the entire aggregate bandwidth when the other nodes are not using their links. We measure the bandwidth within each sub-cluster to be around 250 Mbps, resulting in the ratio of cross-rack and within-rack per-node bandwidths to be 5:1 at peak network utilization. This ratio being at the lower end of typical over-subscription ranging from 5:1 to 20:1 or even higher makes our results conservative; a higher ratio would mean lower bisection bandwidth making ShuffleWatcher even more important. Similarly, using a shared cluster such as EC2 instead of a dedicated cluster makes our results conservative and realistic because the shared cluster comes with network traffic interference from jobs outside ShuffleWatcher's control which would be the case in real deployments. This interference impacts the accuracy of *NetSat's* estimate of network saturation, despite which ShuffleWatcher achieves significant improvements.

4.2 Multi-tenant Scheduler Implementations

We implement ShuffleWatcher on top of two fairness schemes, Fair Scheduler [31] and Dominant Resource Fairness (DRF) [16]. We compare ShuffleWatcher to these baselines as well as Delay Scheduler [39]. Delay Scheduler implementation is open-source and is available with Hadoop release. Delay Scheduler is implemented on top of Fair Scheduler and exploits relaxed fairness among users. For a fair comparison with Delay Scheduler, we configure ShuffleWatcher with Delay Scheduler's relaxed fairness constraints. For the Fair Scheduler-based implementations, each node concurrently runs four Map tasks and two Reduce tasks. DRF is another scheduler based on generalized min-max fairness algorithm [16]. Because DRF's implementation is not publicly available, we develop one. To determine a job's CPU and memory requirements for DRF, we run each of our benchmark jobs individually and monitor the maximum resources needed, as done in [16].

For ShuffleWatcher, we set *NWSaturationThreshold*, the per-rack link utilization threshold to measure network saturation (Section 3.1), to 400 Mbps which is 80% of the admissible bandwidth capacity. We set *MapCompletionThreshold,* the fraction of Map tasks to be completed after which SARP computes the preferred locations for Reduce tasks based on actual intermediate data rather than on SAMP's prediction (Section 3.3), to be 15%. Because NASS considers SAMP's preferences during scheduling, the actual intermediate data and SAMP's predictions are so close that our performance improvements are not sensitive to variations in this parameter. We choose the default distributed file system (HDFS) block size of 64 MB and replication factor of 3.

Table 3: Traffic Volume (GB) under Fair Scheduler

Job Type	Total Shuffle	Cross-Rack Shuffle	Remote Map Traffic	Total Cross-Rack Traffic
Shuffle-heavy	1261	1108	187	1295
Shuffle-medium	70	63	38	101
Shuffle-light	13	11	48	59

4.3 Workloads

We use typical workloads consisting of benchmarks drawn from the Hadoop release and [2]. Based on *ShuffleInputRatio* (Section 3.1), we characterize the benchmarks as Shuffle-heavy, Shuffle-medium or Shuffle-light in Table 1. The table also shows the percentage of jobs of each type in the workload. The variation in job mixes and the variation in job input sizes (Table 2) are based on real workloads from Yahoo and Facebook [9].

We set the number of users to 30 for our 100-node cluster, consistent with the Facebook cluster usage reported in [39] (200 users for a 600-node cluster). Job submission follows an exponential distribution [39, 9, 16]. Each user picks a job from our suite in Table 1, although with different input datasets. After testing with different mean job inter-arrival times, we set the mean to be 40 seconds to utilize the cluster maximally for the base case (Delay Scheduler). We use the same job arrival rates for all schedulers.

Because we are interested only in the steady-state period of the cluster under full load, we ignore the load ramp-up and ramp-down periods. We run each experiment for a steady-state duration of 4 hours.

5 Experimental Results

We first show Shuffle's importance (Section 5.1) and then compare ShuffleWatcher against three baselines, namely Fair Scheduler, Delay Scheduler, and DRF Scheduler (Section 5.2). We then isolate the impact of NASS, SARP and SAMP (Section 5.3) Finally, we show the impact of varying the number of jobs per user (Section 5.4) and the job mix (Section 5.5) on ShuffleWatcher's improvements.

5.1 Importance of the Shuffle

Table 3 shows the actual volumes of the total Shuffle (within-rack and cross-rack), cross-rack Shuffle and remote Map traffic for Shuffle-heavy, Shuffle-medium and Shuffle-light jobs under Fair Scheduler. We show cross-rack traffic as a proxy for the bisection bandwidth demand. From the first two columns, we see that most of the Shuffle (~ 90%) is across racks. From the last three columns, we see that the cross-rack Shuffle volume of the Shuffle-heavy jobs has a significant contribution (>75%) of the total cross-rack traffic, and the contribu-

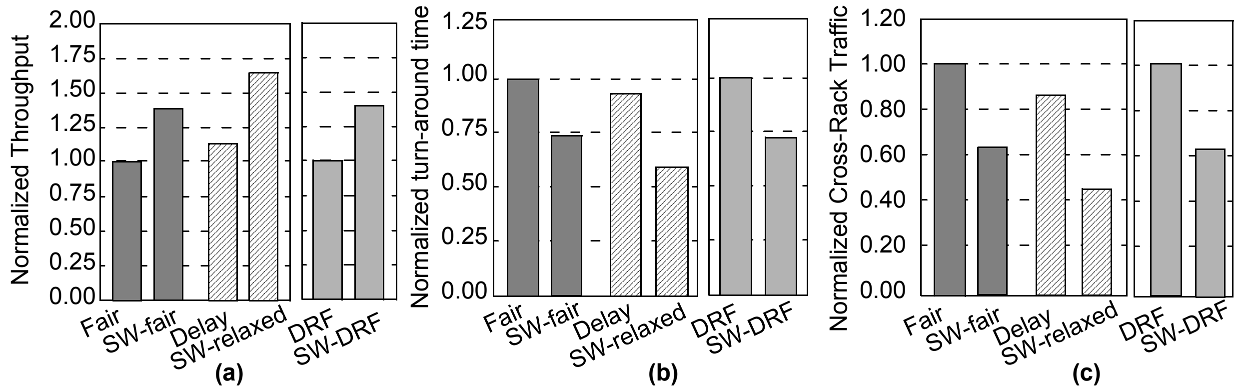

FIGURE 8: Performance comparison

tion of the remote Map traffic is less than 20% of the total cross-rack traffic. These numbers confirm the Shuffle's dominance and justify our focus on the Shuffle.

5.2 Performance

Figure 8 shows ShuffleWatcher operating under two fairness policies — Fair Scheduler (SW-fair) and DRF Scheduler (SW-DRF). To ensure a fair comparison with Delay Scheduler (Delay) which relaxes fairness constraints, we also show ShuffleWatcher configured with similarly relaxed constraints (SW-relaxed). We use relaxed fairness interval of five seconds, consistent with [39]. In the three graphs in Figure 8(a-c) with two sub-graphs per graph, the Y-axes represent throughput, turn-around time and cross-rack traffic, respectively. The first sub-graph shows Fair, SW-fair, Delay, and SW-relaxed which use Fair Scheduler's fairness policy and therefore are normalized to Fair. The second sub-graph shows DRF and SW-DRF which use the DRF policy and therefore are normalized to DRF.

ShuffleWatcher (SW-fair) achieves significant improvements over Fair Scheduler, with 39% higher throughput (Figure 8(a)), 27% lower turn-around time (Figure 8(b)) and 36% lower cross-rack traffic (Figure 8(c)). Compared to Delay Scheduler (Delay), ShuffleWatcher (SW-relaxed) achieves more improvements. Specifically, SW-relaxed is 46%, 32%, and 48% better than Delay in throughput, turn-around time, and cross-rack traffic, respectively (computed from the graphs). SW-DRF also achieves similar performance improvements as SW-fair, showing that our improvements are largely independent of the underlying fairness policy. In the rest of the paper, we report results only for SW-fair, because results for SW-DRF are similar.

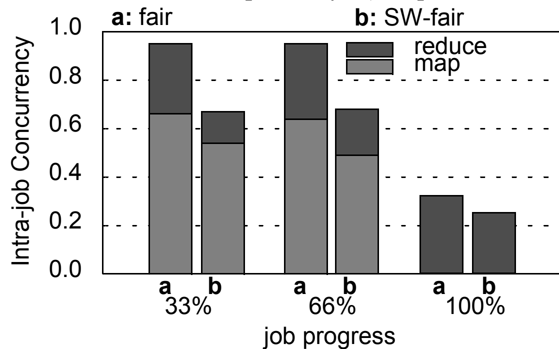

FIGURE 9: Intra-job concurrency

Figure 9 shows the average intra-job concurrency in Fair Scheduler and ShuffleWatcher measured as the fraction of the allocated per-user slots (resources) occupied by a job's Map and Reduce tasks during first, middle and last thirds of the job's work completion. Because typically Map tasks are numerous and Reduce tasks are fewer, Fair Scheduler's concurrency fraction goes from nearly one for Map phase in the first two-thirds of a job's execution (i.e., one job's Map tasks occupy almost all of the user's slots) to less than half for the Reduce phase in the last third of a job's execution (i.e., one job's Reduce tasks leave vacant slots which are occupied by the user's other jobs). The graph shows that ShuffleWatcher trades off intra-job concurrency for Shuffle locality. While the lower Map and Reduce concurrencies due to SAMP and SARP are obvious, these lower concurrencies also mean lower Map-Shuffle concurrency due to NASS.

Figure 10 shows the measured cross-rack Shuffle traffic over time in our testbed for one of our ShuffleWatcher runs. Comparing this profile with that for Fair Scheduler in Figure 1, we see that the network traffic with ShuffleWatcher is relatively balanced.

To show that ShuffleWatcher does not hurt any job's turn-around time by trading-off intra-job concurrency

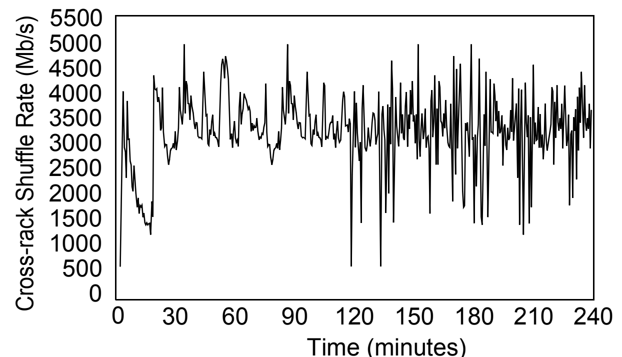

FIGURE 10: ShuffleWatcher: Shuffle profile

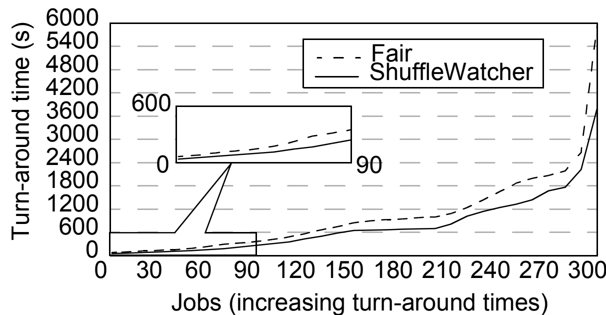

FIGURE 11: Turn-around times of individual jobs

for throughput, Figure 11 plots the turn-around times of individual jobs in Fair Scheduler and ShuffleWatcher. The Y-axis shows the turn-around times and the X-axis shows the jobs ordered in increasing turn-around times under Fair Scheduler (for clarity, jobs 0-90 are shown separately in a blow-up). We see that ShuffleWatcher improves the turn-around time of every one of our 300 jobs, irrespective of the job's size or its Shuffle intensity. This result confirms that by trading-off intra-job concurrency for Shuffle locality in the presence of high contention in multi-tenancy, ShuffleWatcher improves both throughput and turn-around times.

5.3 Impact of NASS, SARP and SAMP

Figure 12 and Figure 13 isolate the impact of NASS, SARP and SAMP by showing a breakdown of ShuffleWatcher's throughput improvements and traffic reduction. Because SARP cannot work without NASS and SAMP cannot work without NASS and SARP, our breakdown is additive in the order of NASS, SARP, and SAMP. We omit the turn-around times breakdown which is similar to the throughput breakdown. In Figure 12 and Figure 13, the Y-axes show throughput and cross-rack traffic, respectively, for ShuffleWatcher normalized to those for Fair Scheduler. We show the breakdown for Shuffle-heavy, Shuffle-medium and Shuffle-light jobs separately, and all the jobs together, to give better insight into ShuffleWatcher's improvements.

From Figure 12, we see that the contribution of each technique is significant across all the three types of jobs. Going from Shuffle-heavy to Shuffle-light, the overall improvement and NASS's contribution increase. Without ShuffleWatcher, the Shuffle-light jobs' short run times are greatly degraded by interference from Shuffle-heavy jobs. ShuffleWatcher, and NASS in particular, reduce this interference, resulting in the observed trend.

The cross-rack traffic breakdown graph in Figure 13 splits the cross-rack traffic into Shuffle traffic and remote Map traffic. From the graph, we see that NASS does not reduce the cross-rack traffic (recall that NASS only shapes, but does not reduce, the traffic). However, SARP, which leverages NASS to improve Reduce-side locality, reduces the cross-rack traffic of Shuffle-heavy,

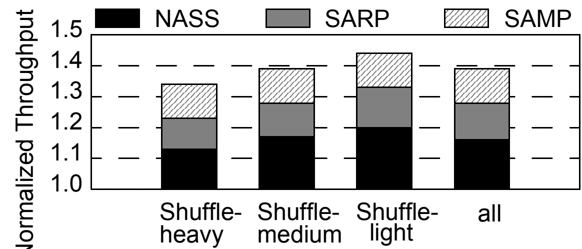

FIGURE 12: Throughput Impact of NASS,SARP,SAMP

Shuffle-medium and Shuffle-light jobs by 16%, 13% and 4%, respectively, with total traffic reduction of 15%. The reduction due to SARP in Shuffle-light jobs' cross-rack traffic is insignificant because most of the traffic of these jobs is due not to the Shuffle but to remote Map tasks (Table 3) which are not impacted by SARP. Similarly, SAMP, which leverages NASS and SARP to improve Map-side locality, reduces the cross-rack Shuffle traffic by 38%, 23% and 4% for the three job types, while reducing the total traffic by 36%. From the graph, we see that SAMP incurs a small increase in the remote Map traffic for Shuffle-heavy jobs (~3%) to localize the Shuffle to fewer racks, but reduces the total cross-rack traffic volume. Such small increase shows that SAMP successfully exploits data replication to localize the Shuffle without incurring significant remote Map traffic overhead. The graph also shows that the total cross-rack traffic reduction for all the jobs together closely follows that for the Shuffle-heavy jobs which contribute a significant portion of the traffic (Table 3).

5.4 Impact of varying jobs per user

Because ShuffleWatcher exploits the choice among a given user's jobs (i.e., per-user fairness) to adapt to the network load, ShuffleWatcher may not perform well with only one job per user, which is equivalent to per-job fairness. To address this concern, Figure 14(a) shows ShuffleWatcher's sensitivity to the number of jobs per user. We use the same mean job arrival rate as before, but vary the number of jobs per user as 1, 9, 12 (default), and 18. We evaluated the case of one job per user in a local 16-node cluster and the rest of the cases in EC2 as

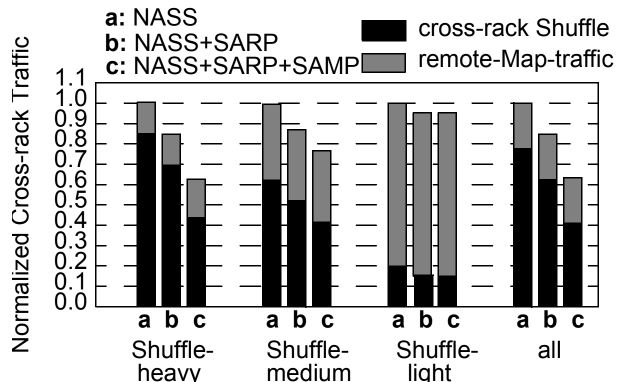

FIGURE 13: Traffic Reduction of NASS,SARP,SAMP

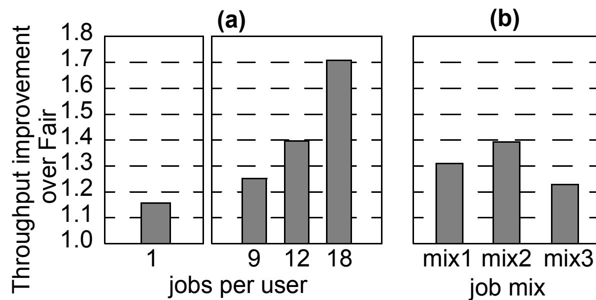

FIGURE 14: Sensitivity to (a) number of jobs per user and (b) job mix

before (we added the first case later, hence the different set up). Therefore, we isolate the one job per user in the left sub-graph while the rest are shown in the right.

The figure shows that even with one job per user ShuffleWatcher achieves 16% higher throughput over Fair Scheduler by choosing between the job's Map and Reduce tasks based on network loading. This result shows that ShuffleWatcher performs well even under per-job fairness. From the right sub-graph, we see that ShuffleWatcher achieves higher improvements with more jobs per user (i.e., under per-user fairness). With more jobs per user, ShuffleWatcher has more choices in its scheduling decisions and, therefore, achieves better network traffic shaping and reduction.

5.5 Sensitivity to the variation in job mix

We show ShuffleWatcher's sensitivity to the job mix in the workload. Figure 14(b) shows throughput improvements over Fair Scheduler for three different mixes of Shuffle-light, Shuffle-medium, and Shuffle-heavy jobs: *mix1* with 20%, 20% and 60%, respectively; *mix2* with 40%, 20%, and 40%, respectively (default); and *mix3* with 60%, 20% and 20%, respectively.

From the figure, we see that ShuffleWatcher improves throughput by 31% and 22% for *mix1* and *mix3*, respectively, compared to 39% for *mix2*. The improvements for *mix1* and *mix3* are significant but lower than that for *mix2* because of reduced opportunity. Compared to *mix2*, *mix1*'s larger fraction of Shuffle-heavy jobs means higher network pressure with fewer low-utilization periods; and *mix3*'s larger fraction of Shuffle-light jobs means lower network saturation. Nevertheless, significant improvements over a wide range of job mixes demonstrate ShuffleWatcher's effectiveness.

5.6 Execution on a dedicated cluster

In addition to 100-node EC2 runs, we also performed runs on a dedicated 16 Xeon-nodes cluster to isolate the interference from jobs outside ShuffleWatcher's control. We scaled down job arrival rate, job sizes, and number of users to match the cluster configuration. We divided the cluster into 4 sub-clusters of 4 nodes each and limited the per-node bisection bandwidth to be the same as in the EC2 cluster (50Mbps). Our results exceeded the performance achieved in the EC2 cluster. ShuffleWatcher achieved 46% higher throughput, 32% lower turn-around time and 48% lower cross-rack traffic over Fair Scheduler. Compared to Delay Scheduler, ShuffleWatcher was 54%, 38%, and 56% better in throughput, turn-around time, and cross-rack traffic, respectively.

6 Related work

Several previous efforts have targeted improving MapReduce performance, including better straggler management [7], improved computation-communication overlap [3,11,36], improved aggregation of intermediate data [43], optimizations for heterogeneous clusters [2,41], and runtime optimizations for iterative MapReductions [8,14,40]. However, these proposals target single-tenancy whereas ShuffleWatcher exploits opportunities that are specific to multi-tenancy.

In the domain of multi-tenancy, Hadoop [21] offers a FIFO scheduler to run jobs in a sequential manner. Capacity Scheduler [30], Fair Scheduler [31] and Dominant Resource Fairness [16] propose different fairness models and schedulers for resource allocation among users. In contrast to their target of achieving fairness, our goal is to improve performance within the given fairness constraints. Delay Scheduling [39] and Quincy [25] target reducing network traffic by optimizing Map-input locality, but not the Shuffle which is by far the most dominant source of traffic in MapReduce. ShuffleWatcher targets the Shuffle by trading-off intra-job concurrency for Shuffle locality to perform better than these previous techniques. Mesos [23] and Yarn [38] facilitate resource provisioning among multiple frameworks that share a cluster (e.g., MPI and MapReduce). These systems decouple resource allocation from job scheduling and can benefit from ShuffleWatcher's scheduling. Purlieus [29] and CAM [26] achieve locality via synergistic placement of virtual machines and input data. However, such static techniques cannot address dynamic variations in the Shuffle traffic.

In the domain of data center networks, researchers have proposed network architectures to improve cluster bisection bandwidth (e.g.,[1,4,15,17,18,19,20,28,33]). Many of these architectures require specialized hardware and/or communication protocols, and thereby incur additional cost especially because bisection bandwidth is inherently hard to scale up. Finally, a few recent papers propose better management of network traffic [10,12,32] without changing the network load, which is often high enough to limit their effectiveness, whereas ShuffleWatcher actively shapes and reduces the network load.

7 Conclusion

We proposed ShuffleWatcher, a Shuffle-aware, multi-tenant scheduler, which counter-intuitively trades-off intra-job concurrency for Shuffle locality. Shuffle-

Watcher employs three mechanisms: *Network-Aware Shuffle Scheduling (NASS), Shuffle-Aware Reduce Placement (SARP),* and *Shuffle-Aware Map Placement (SAMP)* which exploit this trade-off and improve performance by shaping and reducing the Shuffle traffic while working within the specified fairness constraints.

On a 100-node Amazon EC2 cluster running Hadoop, ShuffleWatcher improves cluster throughput by 39-46% and job turn-around time by 27-32% over three state-of-the-art schedulers. Despite trading-off intra-job concurrency for Shuffle locality, ShuffleWatcher does not sacrifice turn-around times to gain throughput; on the contrary, by improving Shuffle locality in the presence of high contention in multi-tenancy, ShuffleWatcher improves turn-around times, not only on average but also of every one of 300 jobs in our experiments. ShuffleWatcher improves both cluster throughput and job latency and, therefore, will be valuable in emerging multi-tenant environments.

References

[1] H. Abu-Libdeh et al. Symbiotic routing in future data centers. In *Proceedings of SIGCOMM,* 2010.

[2] F. Ahmad et al. Tarazu: Optimizing MapReduce on heterogeneous clusters. In *Proceedings of ASPLOS,* 2012.

[3] F. Ahmad et al. MapReduce with Communication Overlap (MaRCO). In *JPDC,* 2012.

[4] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of SIGCOMM,* 2008.

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

[6] Amazon Elastic MapReduce (Amazon EMR). http://aws.amazon.com/elasticmapreduce/.

[7] G. Ananthanarayanan et al. Reining in the outliers in Map-Reduce clusters using Mantri. In *Proceedings of OSDI,* 2010.

[8] Y. Bu et al. The Haloop approach to large-scale iterative data analysis. *The VLDB Journal*, 21(2):169-190, Apr. 2012.

[9] Y. Chen et al. The case for evaluating MapReduce performance using workload suites. In *Proceedings of MASCOTS,* 2011.

[10] M. Chowdhury et al. Managing data transfers in computer clusters with orchestra. In *Proceedings of SIGCOMM*, pages 98-109, 2011.

[11] T. Condie et al. MapReduce online. In *Proceedings of NSDI,* 2010.

[12] P. Costa et al. Camdoop: exploiting in-network aggregation for big data applications. In *Proceedings of NSDI,* 2012.

[13] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, Jan. 2008.

[14] J. Ekanayake et al. Twister: a runtime for iterative MapReduce. In *Proceedings of HPDC,* 2010.

[15] N. Farrington et al. Helios: a hybrid electrical/optical switch architecture for modular data centers. In *Proceedings of SIGCOMM,* 2010.

[16] A. Ghodsi et al. Dominant Resource Fairness: Fair allocation of multiple resource types. In *Proceedings of NSDI,* 2011.

[17] A. Greenberg et al. VL2: A scalable and flexible data center network. In *Proceedings of SIGCOMM,* 2009.

[18] C. Guo et al. Bcube: a high performance, server-centric network architecture for modular data centers. In *Proceedings of SIGCOMM,* 2009.

[19] C. Guo et al. Dcell: a scalable and fault-tolerant network structure for data centers. In *Proceedings of SIGCOMM,* 2008.

[20] L. Gyarmati and T. A. Trinh. Scafida: a scale-free network inspired data center architecture. *SIGCOMM Comput. Commun. Rev.*, 40(5):4-12.

[21] Hadoop. http://hadoop.apache.org.

[22] Hadoop On Demand. http://hadoop.apache.org/docs/r0.18.3/hod_admin_guide.html.

[23] B. Hindman et al. Mesos: a platform for fine-grained resource sharing in the data center. In *Proceedings of NSDI,* 2011.

[24] M. Isard et al. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of EuroSys,* 2007.

[25] M. Isard et al. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of SOSP,* 2009.

[26] M. Li et al. Cam: a topology aware minimum cost flow based resource manager for MapReduce applications in the cloud. In *Proceedings of HPDC,* 2012.

[27] Torque Resource Manager. http://www.adaptivecomputing.com/products/open-source/torque/.

[28] R. Niranjan Mysore et al. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of SIGCOMM,* 2009.

[29] B. Palanisamy et al. Purlieus: locality-aware resource allocation for MapReduce in a cloud. In *Proceedings of SC,* 2011.

[30] Hadoop Capacity Scheduler. http://hadoop.apache.org/docs/r0.20.2/capacity_scheduler.html.

[31] Hadoop Fair Scheduler. http://hadoop.apache.org/docs/r0.20.2/fair_scheduler.html.

[32] A. Shieh et al. Sharing the data center network. In *Proceedings of NSDI,* 2011.

[33] A. Singla et al. Jellyfish: Networking data centers randomly. In *Proceedings of NSDI,* 2012.

[34] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience: Research Articles, UK, Feb. 2005. John Wiley and Sons Ltd.

[35] A. Vahdat et al. Scale-Out Networking in the Data Center. *IEEE Micro*, 30:29-41, July 2010.

[36] A. Verma et al. Breaking the MapReduce Stage Barrier. In *Proceedings of IEEE CLUSTER,* 2010.

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

[38] Yarn. http://hadoop.apache.org/docs/r0.23.0/hadoop-yarn/hadoop-yarn-site/YARN.html.

[39] M. Zaharia et al. Delay Scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of EuroSys,* 2010.

[40] M. Zaharia et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of NSDI,* 2012.

[41] M. Zaharia et al. Improving MapReduce performance in heterogeneous environments. In *Proceedings of OSDI*, 2008.

[42] J. Mudigonda, P. Yalagandula, and J. C. Mogul. Taming the flying cable monster: a topology design and optimization framework for data-center networks. In *Proceedings of USENIX ATC,* 2011.

[43] S. Rao, et.al. Sailfish: a framework for large scale data processing. In *Proceedings of SoCC,* 2012.