

Supplemental Material: DynamicMR: A Dynamic Slot Allocation Optimization Framework for MapReduce Clusters

Shanjiang Tang, Bu-Sung Lee, Bingsheng He

Abstract—MapReduce is a popular computing paradigm for large-scale data processing in cloud computing. However, the slot-based MapReduce system (e.g., Hadoop MRv1) can suffer from poor performance due to its unoptimized resource allocation. To address it, this paper identifies and optimizes the resource allocation from three key aspects. First, due to the pre-configuration of distinct map slots and reduce slots which are not fungible, slots can be severely under-utilized. Because map slots might be *fully* utilized while reduce slots are empty, and vice-versa. We proposes an alternative technique called *Dynamic Hadoop Slot Allocation* by keeping the slot-based model. It relaxes the slot allocation constraint to allow slots to be reallocated to either map or reduce tasks depending on their needs. Second, the speculative execution can tackle the straggler problem, which has shown to improve the performance for a single job but at the expense of the cluster efficiency. In view of this, we propose *Speculative Execution Performance Balancing* to balance the performance tradeoff between a single job and a batch of jobs. Third, delay scheduling has shown to improve the data locality but at the cost of fairness. Alternatively, we propose a technique called *Slot PreScheduling* that can improve the data locality but with no impact on fairness. Finally, by combining these techniques together, we form a step-by-step slot allocation system called *DynamicMR* that can improve the performance of MapReduce workloads substantially. The experimental results show that our DynamicMR can improve the performance of Hadoop MRv1 significantly while maintaining the fairness, by up to $46\% \sim 115\%$ for single jobs and $49\% \sim 112\%$ for multiple jobs. Moreover, we make a comparison with YARN experimentally, showing that DynamicMR outperforms YARN by about $2\% \sim 9\%$ for multiple jobs due to its ratio control mechanism of running map/reduce tasks.

Keywords—MapReduce, Hadoop Fair Scheduler, Slot PreScheduling, Delay Scheduler, DynamicMR, Slot Allocation.

APPENDIX A
DESIGN AND IMPLEMENTATION OF SLOT PRESCHEDULING

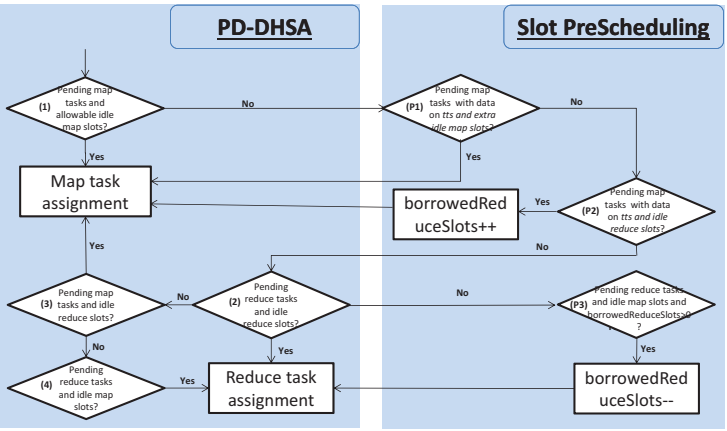


Fig. 1: The overview design and implementation of Slot PreScheduling and its combination with PD-DHSA. The labels (1)-(4), (P1)-(P3) in the graph corresponds to Case (1)-(4) in Section 2.1.2 of the main file and Case (P1)-(P3) in Section A, respectively.

Figure 1 gives an overview design and implementation of Slot PreScheduling and its incorporation with PD-DHSA. It is an extended graph on top of Figure 4 of the main file. Given a tasktracker *tts* connecting to the jobTracker in a heartbeat, the overall execution flow is as follows:

Case (P1): The slot allocation work starts from Case (1) of PD-DHSA. When the attempt of Case (1) fails, instead of going forwards to Case (2) as in Figure 4 of the main file, it comes to Case (P1) of Slot PreScheduling. It checks whether the condition of the first case in Section 2.3.2 of the main file holds. If yes, the Slot PreScheduling will perform the map task assignment for the current job. Otherwise, it checks whether the condition of the second case in Section 2.3.2 of the main file holds or not, by moving towards to Case (P2).

Case (P2): It checks whether there are idle reduce slots and pending map tasks for the current job with data on *tts*. If the condition holds, it attempts to perform map task assignment work on *tts* by borrowing idle reduce slots. Moreover, there is a variable *borrowedReduceSlots*, which counts the current number of reduce slots that has been borrowed by Slot PreScheduling. To limit the maximum number of idle reduce slots that can be borrowed by Slot PreScheduling, we provides user with a configurable threshold variable *maximumBorrowableReduceSlots*, initialized to be infinity by default. The Slot PreScheduling can proactively perform map task assignment if and only if *borrowedReduceSlots* does not exceed *maximumBorrowableReduceSlots*. Otherwise, the Slot PreScheduling will deliver the task assignment work to PD-DHSA (i.e., Case (2)).

Case (P3): When the condition for Case (2) of PD-DHSA

- *S.J. Tang, B.S. Lee, B.S. He are with the School of Computer Engineering, Nanyang Technological University, Singapore. E-mail: {stang5, ebslee, bshe}@ntu.edu.sg.*

does not hold, Slot PreScheduling will begin to work by checking whether it is possible to assign pending reduce tasks with idle map slots in case that the variable *borrowedReduceSlots* is larger than zero. If yes, Slot PreScheduling will perform reduce task assignment with idle map slots.

APPENDIX B
DYNAMICMR IMPLEMENTATION

We implement DynamicMR based on HFS source code. Table 1 lists some key functions for DynamicMR. We have a taskScheduler class *DynamicMRFairScheduler*, acting as the core role for task scheduling. For the DHSA component, it provides two functions *assignTasks_poolInDependentMode()* and *assignTasks_poolDependentMode()* implemented for PI-DHSA and PD-DHSA, respectively. Moreover, there are two functions *prescheduler_mapTask()* and *prescheduler_reduceTask()* for map tasks and reduce tasks preScheduling, respectively. For SEPB, we implement it in PoolSchedulable class. *DyamicSpeculativeTaskScheduler()* dynamically determines when to schedule speculative tasks for utilization efficiency optimization.

As mentioned at the beginning of Section 2 of the main file before, the three components, DHSA, SEPB, Slot PreScheduling are loosely coupled. They can work alone or together. To achieve that, we provide users with some configurable arguments as shown in Table 2. For DHSA, users can choose the types (e.g., PI-DHSA, PD-DHSA) they want to use. Meanwhile, users can specify the maximum number of idle map (reduce) slots (e.g., *mapred.idle.mapslots.borrowedforReduceTasks*, *mapred.idle.reduceslots.borrowedforMapTasks*) that can be borrowed for reduce (map) tasks at a moment. The DHSA can be disabled by simply setting them to zero. For SEPB, we provide users with two alternative modes (i.e., cluster-level and pool-level), specified by *FairSchedulerSpeculativeJobsPendingTasksCheckedMode*. Instead of checking all pending tasks for all jobs, we allow users flexibly to specify how many percentage of runnable jobs they want to check for pending tasks before allowing to schedule speculative tasks (e.g., *poolPercentageOfSubmittedJobsCheckedForPendingTasks* and *poolPercentageOfSubmittedJobsCheckedForPendingTasks*). It can be disabled by configuring them with zero. Finally, for Slot PreScheduling, users can enable it by setting mapred.map.tasks.preScheduler.enabled to be *true*. We also allow users to specify the maximum number of reduce slots (e.g., *mapred.preScheduler.maximumNumBorrowedReduceSlots*) that can be preempted by map tasks for data locality improvement.

APPENDIX C
DISCUSSION ON DHSA

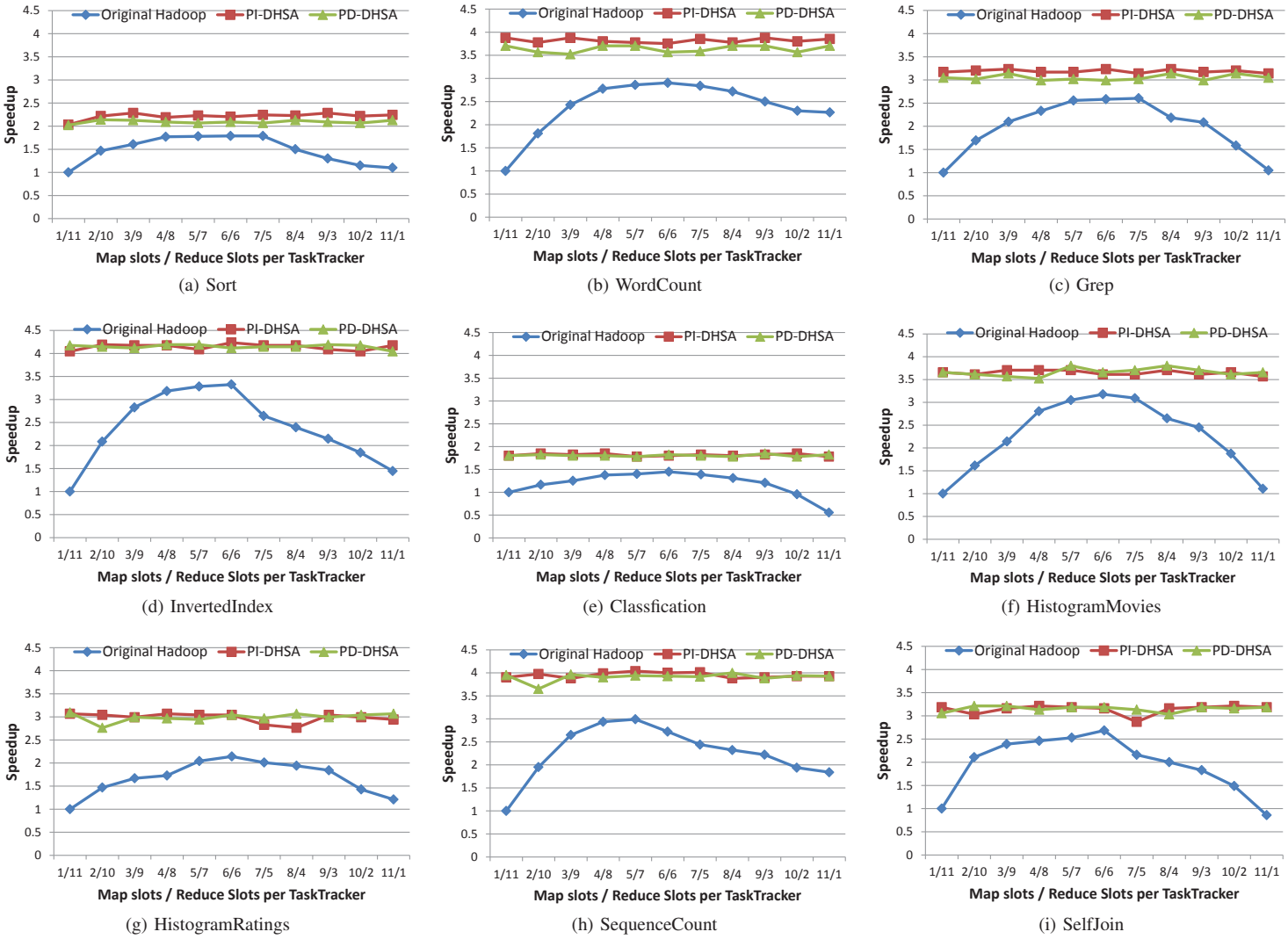
DHSA VS Static Slot Optimization. Traditional Hadoop MRv1 adopts the static slot configuration method for map and reduce tasks and has a strict constrain that map tasks can only use map slots and reduce tasks can only run on reduce slots, making the performance sensitive to the slot configurations, i.e., different slot configurations will results in varied performance, as illustrated in Figure 2. Moreover, for FIFO scheduling, different job submission orders also have varied performance [24]. In contrast, for Fair scheduling, different weights (i.e., shares) of pools can cause varied performance for the whole jobs. It implies that, we can improve the performance for Hadoop MRv1 statically via optimizing slot configuration, job submission orders, and pool weights. Compared to DHSA, such static methods have the following serious shortcomings:

First, the static optimization methods need to know some information about MapReduce workloads (e.g., the number of map/reduce tasks per job/pool, the execution time for map/reduce tasks per job) in advance, which is however often un-obtainable without running tasks in practice. Instead, it can only be used for periodically running jobs (e.g., in data warehouse). Thus, there is a generality problem for static methods. In contrast, DHSA improves the performance without needs of any information about MapReduce jobs in advance and thus keeps the generality feature of Hadoop.

Second, the static optimization methods can improve the performance of MapReduce jobs but substantially not enough. Consider the static slot configuration for example, no matter how to optimize the slot configuration per slave node, it still cannot avoid the case that there are pending tasks and idle slots (e.g., for a single job case, the map slots will be idle in the

Algorithm 1 The dynamic task assignment policy for tasktracker under PI-DHSA.

When a heartbeat is received from a compute node n:
1: compute its clusterUsedMapSlots, clusterUsedReduceSlots, mapSlotsDemand, reduceSlotsDemand, mapSlotsLoadFactor and reduceSlotsLoadFactor.
2: */*Case 1: both map slots and reduce slots are sufficient.*/*
3: **if** (*mapSlotsLoadFactor* $\leqslant$ 1 and *reduceSlotsLoadFactor* $\leqslant$ 1) **then**
4: //No borrow operation is needed.
5: **end if**
6: */*Case 2: both map slots and reduce slots are not enough.*/*
7: **if** (*mapSlotsLoadFactor* $\geqslant$ 1 and *reduceSlotsLoadFactor* $\geqslant$ 1) **then**
8: //No borrow operation is needed.
9: **end if**
10: */*Case 3: map slots are enough, while reduce slots are insufficient. It calculates borrowed map slots for reduce tasks.*/*
11: **if** (*mapSlotsLoadFactor* $<$ 1 and *reduceSlotsLoadFactor* $>$ 1) **then**
12: *currentBorrowedMapSlots* = *clusterUsedMapSlots* - *clusterRunningMapTasks*;
13: *extraReduceSlotsDemand* = min{ max{ floor{ *clusterMapCapacity* * *percentageOfBorrowedMapSlots*} - currentBorrowedMapSlots, 0}, *reduceSlotsDemand* - *clusterReduceCapacity*}
14: *updatedMapSlotsLoadFactor* = (*mapSlotsDemand* + *extraReduceSlotsDemand*) / *clusterMapCapacity*;
15: **end if**
16: */*Case 4: map slots are insufficient, while reduce slots are enough. It calculates borrowed reduce slots for map tasks.*/*
17: **if** (*mapSlotsLoadFactor* $>$ 1 and *reduceSlotsLoadFactor* $<$ 1) **then**
18: *currentBorrowedReduceSlots* = *clusterUsedReduceSlots* - *clusterRunningReduceTasks*;
19: *extraMapSlotsDemand* = min{ max{ floor{ *clusterReduceCapacity* * *percentageOfBorrowedReduceSlots*} - currentBorrowedReduceSlots, 0}, *mapSlotsDemand* - *clusterMapCapacity*}
20: *updatedReduceSlotsLoadFactor* = (*reduceSlotsDemand* + *extraMapSlotsDemand*) / *clusterReduceCapacity*;
21: **end if**
22: compute availableMapSlots and availableReduceSlots based on the updated map/reduce load factor and used slots.



(a) Sort (b) WordCount (c) Grep

(d) InvertedIndex (e) Classfication (f) HistogramMovies

(g) HistogramRatings (h) SequenceCount (i) SelfJoin

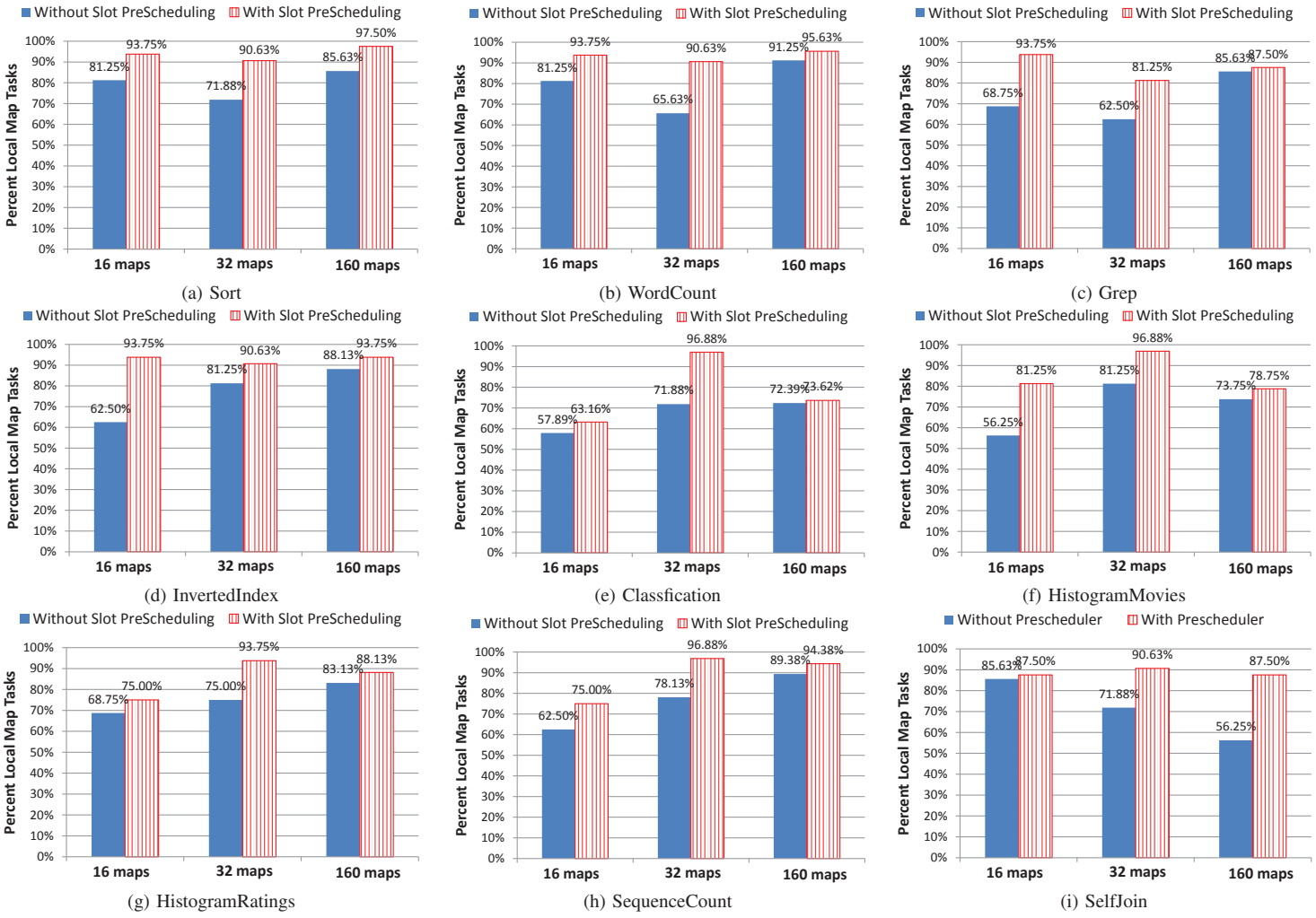
Fig. 2: The performance improvement by DHSA under various slot configuration for MapReduce workloads.

TABLE 1: The key functions in the DynamicMR.

Class	Function Description
DynamicMRFairScheduler	`List<Task> assignTasks_poolInDependentMode()` Assign map/reduce tasks across pools dynamically with PI-DHSA. See Algorithm 1 in the main file for details.
	`List<Task> assignTasks_poolDependentMode()` Assign map/reduce tasks across pools dynamically with PD-DHSA. See Algorithm 2 in the main file for details.
	`ArrayList<Task> prescheduler_mapTask()` Preschedule *local* map tasks using idle map and reduce slots, for Case *P1* and *P2* at *Design and Implementation* in Section A.
	`ArrayList<Task> prescheduler_reduceTask()` Preschedule reduce tasks using map slots for Case *P3* illustrated in Figure 1 in Section A.
PoolSchedulable	`Task task DyamicSpeculativeTaskScheduler()` Schedule speculative tasks dynamically for utilization efficiency. See detailed implementation in Section A.

TABLE 2: The user configurable arguments for DynamicMR.

Types	Argument	Location
DHSA	`mapred.idle.mapslots.borrowedforReduceTasks` Fraction of the number of unused map slots (0.0 ∼ 1.0) that can be used for reduce tasks, when map slots is not enough.	mapred-site.xml
	`mapred.idle.reduceslots.borrowedforMapTasks` Fraction of the number of unused reduce slots (0.0 ∼ 1.0) that can be used for map tasks, when reduce slots is not enough.	
	`PoolsFairShareSlotsAdaptiveMode` The choice of DHSA. For PI-DHSA, set it to ***PoolInDependent***; and otherwise configure it with ***PoolDependent*** for PD-DHSA.	fair-scheduler.xml
SEPB	`mapred.submitted.jobs.checkedforPendingTasks` Fraction of the total number of jobs (0.0 ∼ 1.0) to be checked for the number of pending map (reduce) tasks at cluster-level, needed by DynamicMR to decide whether it is time to run the speculative task or not for a batch of jobs.	mapred-site.xml
	`poolPercentageOfSubmittedJobsCheckedForPendingTasks` Fraction of the total number of jobs within a pool (0.0 ∼ 1.0) to be checked for the number of pending map (or reduce) tasks, needed by DynamicMR to decide whether it is time or not to run the speculative task for a batch of jobs from a pool.	fair-scheduler.xml
	`FairSchedulerSpeculativeJobsPendingTasksCheckedMode` Set it to be ***GlobalLevel*** when we check pending tasks for all jobs in a cluster. Otherwise, set it to be ***PoolLevel*** if we want to check pending tasks for all jobs within each pool.	
Slot PreScheduling	`mapred.map.tasks.preScheduler.enabled` If ***true***, it preSchedules local map tasks on the node when there are no allowable map slots.	mapred-site.xml
	`mapred.preScheduler.maximumNumBorrowedReduceSlots` The maximum number of reduce slots that can be borrowed for map tasks with preScheduler, 100000 by default.	



(a) Sort (b) WordCount (c) Grep

(d) InvertedIndex (e) Classfication (f) HistogramMovies

(g) HistogramRatings (h) SequenceCount (i) SelfJoin

Fig. 3: The data locality improvement for Slot PreScheduling.

Job ID	Name	Job ID	Name	Job ID	Name
J_1	WordCount	J_{11}	WordCount	J_{21}	Classification
J_2	Sort	J_{12}	InvertedIndex	J_{22}	SelfJoin
J_3	Grep	J_{13}	SequenceCount	J_{23}	Grep
J_4	InvertedIndex	J_{14}	SelfJoin	J_{24}	InvertedIndex
J_5	Classification	J_{15}	HistogramMovies	J_{25}	SequenceCount
J_6	HistogramMovies	J_{16}	Sort	J_{26}	Sort
J_7	HistogramRatings	J_{17}	HistogramRatings	J_{27}	WordCount
J_8	SequenceCount	J_{18}	Grep	J_{28}	HistogramRatings
J_9	SelfJoin	J_{19}	InvertedIndex	J_{29}	Classification
J_{10}	Classification	J_{20}	HistogramRatings	J_{30}	SequenceCount

TABLE 3: The batch jobs information. The detailed information for each job is given by Table 3 of the main file. Our experiments consider four workloads: 5 jobs ($J_1 \sim J_5$), 10 jobs ($J_1 \sim J_{10}$), 20 jobs ($J_1 \sim J_{20}$), 30 jobs ($J_1 \sim J_{30}$).

Algorithm 2 The dynamic task assignment policy for tasktracker under PD-DHSA.

When a heartbeat is received from tasktracker tts:

Compute its totalSlotsDemand, totalSlotsCapacity, trackerSlotsCapacity, trackerRunningTasksNum and trackerCurrentSlotsCapacity.
/* Return when there are no idle slots. */
if trackerRunningTasksNum $\geqslant$ trackerCurrentSlotsCapacity **then**
    return NULL;
end if
for (i = 0; i $<$ trackerCurrentSlotsCapacity - trackerRunningTasksNum; i++) **do**
    Sort pools by distance below min and fair share
    for (Pool p : pools) **do**
        / Case (1): allocate map slots for map tasks from Pool p*/*
        if (there are pending map tasks and idle map slots) **then**
            attempt to allocate map slots for map tasks (considering data locality) and jump out of loop if allocation succeeded.
        end if
        / Case (2): allocate reduce slots for reduce tasks from Pool p*/*
        if (Case (1) failed and there are pending reduce tasks and idle reduce slots) **then**
            attempt to allocate reduce slots for reduce tasks and jump out of loop if allocation succeeded.
        end if
        / Case (3): allocate reduce slots for map tasks from Pool p*/*
        if (Case (2) failed and there are pending map tasks) **then**
            attempt to allocate reduce slots for map tasks (considering data locality) and jump out of loop if allocation succeeded.
        end if
        / Case (4): allocate map slots for reduce tasks from Pool p*/*
        if (Case (3) failed and there are pending reduce tasks) **then**
            attempt to allocate map slots for reduce tasks and jump out of loop if allocation succeeded.
        end if
    end for
    / Case (5): schedule the non-local map tasks when its node-local tasks cannot be satisfied. */*
    if (Case (1)-(4) failed) **then**
        for (Pool p : pools) **do**
            if (there are pending map tasks) **then**
                attempt to allocate map/reduce slots to map tasks (not considering data locality) and jump out of loop if allocation succeeded.
            end if
        end for
    end if
end for

Algorithm 3 The pseudo-code for SEPB.

When there is a idle slot:

The speculative scheduler (e.g., LATE) checks first whether there are new straggled tasks.
if there are straggled tasks and *runningSpeculativeTasks* $<$ *SpeculativeCap* **then**
    Check a set of multiple jobs for pending tasks.
    if there are pending tasks for multiple jobs **then**
        Allocate the idle slot to a pending task.
    else
        Create a speculative task and allocate the idle slot to it.
    end if
else
    Allocate the idle slot to a pending task.
end if

Algorithm 4 The pseudo-code for PreScheduler.

When a heartbeat is received from a compute node n:

Compute *allowableIdleMapSlots*, *extraIdleMapSlots*, *idleReduceSlots*.
for (Job j in job set J of sorted fair-share priority order) **do**
    if Job j has local data in node n but *no allowableIdleMapSlots* **then** ▷ The condition to consider PreScheduler.
        if *extraIdleMapSlots* $>$ 0 **then** ▷ Case 1: using *extraIdleMapSlots*
            Allocate map slot to Job j.
        end if
        if *EnableDHSA=true* && *idleReduceSlots* $>$ 0 **then** ▷ Case 2: using *idleReduceSlots* in DHSA.
            Allocate map slot to Job j.
        end if
    end if
end for

reduce-phase computation). Such a problem does also exist for job ordering optimization and pool weight optimization methods. In contrast, with DHSA, we can guarantee that all slots must be busy whenever there are pending tasks, making slot utilization at maximum.

Dynamic slot allocation. Underlying PI-DHSA and PD-DHSA are two different concepts or definitions of fairness. PI-DHSA follows strictly the definition of fairness given by traditional HFS, i.e., the slots are fairly shared across pools within each phase (e.g., map-phase, reduce-phase), but independent across phases. In contrast, PD-DHSA gives a new definition of fairness from the perspective of pools, i.e., each pool shares the total number of map and reduce slots from the map phase and reduce phase fairly with other pools. Due to varied definitions of fairness, there are different priorities and possibilities for slot movements between map-phase and reduce-phase (i.e., moving map slots to reduce-phase for reduce tasks, and vice versa) for PI-DHSA and PD-DHSA. For PI-DHSA, the map slots always satisfy the map tasks first before giving to reduce tasks, and vice versa. Thus, the inter-phase slot movement can only occur when one typed slots (e.g., map slots) are enough while the other typed slots (e.g., reduce slots) are insufficient, i.e., two cases: Case 2 and Case 3 in Section 2.1.1 of the main file. In contrast, for PD-DHSA, all the slots of a pool always attempt to satisfy the tasks within the pool first before yielding to other pools. The inter-phase slot movement appears when

it exists a pool whose one typed slots (e.g., map slots) share is larger than its corresponding typed slots demand while its share of the other typed slots (e.g., reduce slots) is not enough. However, all four scenarios (e.g., Case 1, Case 2, Case 3, Case 4) mentioned in Section 2.1.1 of the main file could have such a kind of pools, implying that the inter-phase slot movement can occur in all the four scenarios. That is, it is more likely to have inter-phase slot movement for PD-DHSA than PI-DHSA during the dynamic slot allocation process.

APPENDIX D
MOTIVATION FOR SLOT PRESCHEDULING

The delay scheduler was original proposed by Zaharia et al. [37] and has implemented in Hadoop to improve the data locality. However, in the real implementation of Hadoop Fair Scheduler (HFS) and FIFO scheduler (e.g., Hadoop Version 1.2.), the load balance is considered on task scheduling (including delay scheduling). In HFS and FIFO, their task assignment processes both follow that, when a tasktracker connects to the jobTracker in a heartbeat, instead of allowing all its idle slots to be allocated directly, it will first consider the load balance issue across slave nodes by determining dynamically how many running tasks should be allowed per slave node during runtime (i.e., *allowable idle slots*), implying that it may allow only a part of idle slots of a tasktracker to be allocated rather than all in order for load balancing, or even don't allow any idle slots to be allocated. For example, in Figure 6 of the main file, the current load for Tasktracker 2 and 4 are over and at the load balancing line, respectively. In that case, Hadoop will not allow any slot allocation and just skip Tasktracker 2 and 4 directly before arriving at scheduling part (e.g., delay scheduling, fair scheduling, speculative scheduling, etc), although they have idle slots. Thus, when J_1 is in the headed allocation list based on their share priority, it is not allowed to schedule tasks to TaskTracker 2 and 3 as there is *no allowable idle slots* due to load balancing constrain. For Tasktracker 1 and 3, delay scheduler will skip J_1 for a while since there is no local data available for J_1. That is, no matter which tasktracker connects to the jobTracker in a heartbeat, J_1 will be delayed to schedule within a time limit. Based on these observations by reviewing the Hadoop source code, we thus proposed PreScheduler to maximize the data locality while not hurting fairness by relaxing the load balance constrain and allowing J_1 to use the extra idle slots (i.e, idle slots above the workload balancing line in Figure 6 of the main file) when the Tastracker 2 or 4 connected.

APPENDIX E
LOAD BALANCE AND FAIRNESS EVALUATION FOR SLOT PRESCHEDULING

Figure 11 and 12 of the main file have illustrated that Slot Prescheduling can improve the data locality and performance of MapReduce jobs. To validate our claim in Section 2.3 of the main file that Slot PreScheduling can improve the data locality at the expense of load balance across slave nodes while having no negative impact on the fairness of MapReduce jobs, we perform an experiment with one to ten jobs (J_1 to J_{10}) from Table 3, with each job in a separate pool. A cluster is considered as *load balanced* if the numbers of running tasks across slave nodes (i.e., taskTrackers) are the same. We define a *unbalanced degree* as the average of square deviation of running tasks across slave node over the whole computation time t, i.e.,

$$UnbalancedDegree(t) = \frac{\int_0^t f(t)}{t}.$$

where $f(t)$ denotes the square deviation at instant time t, i.e.,

$$f(t) = \sum_{i=1}^{n} (l(i,t) - \bar{l}(t))^2.$$

where n, $l(i,t)$, and $\bar{l}(t)$ denote the number of tasktrackers, the load for the i^{th} tasktracker, and the mean load for tasktrackers at time t, respectively.

We call it *fair* for the cluster when all demanding pools share (possess) the same amount of resources. Similarly, we measure the *unfairness degree* using the average of square deviation of allocated resources across pools over the whole computation time t, i.e.,

$$UnfairnessDegree(t) = \frac{\int_0^t g(t)}{t}.$$

where $g(t)$ denotes the square deviation of allocated resources at instant time t, i.e.,

$$g(t) = \sum_{i=1}^{p} (s(i,t) - \bar{s}(t))^2.$$

where p, $s(i,t)$, and $\bar{s}(t)$ denote the number of pools, the allocated resources for the i^{th} pool, and the mean allocated resources for pools at time t, respectively.

Figure 4 presents the unbalanced and unfairness results for batch jobs. We see in Figure 4(b) that the Slot PreScheduling has better fairness results, but it has a bit worse load balance degree given by Figure 4(a), validating our statement.

APPENDIX F
DISCUSSION ON THE PERFORMANCE OF DIFFERENT PERCENTAGES OF BORROWED MAP AND REDUCE SLOTS

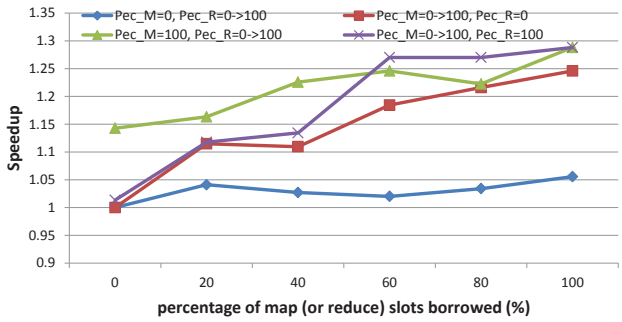
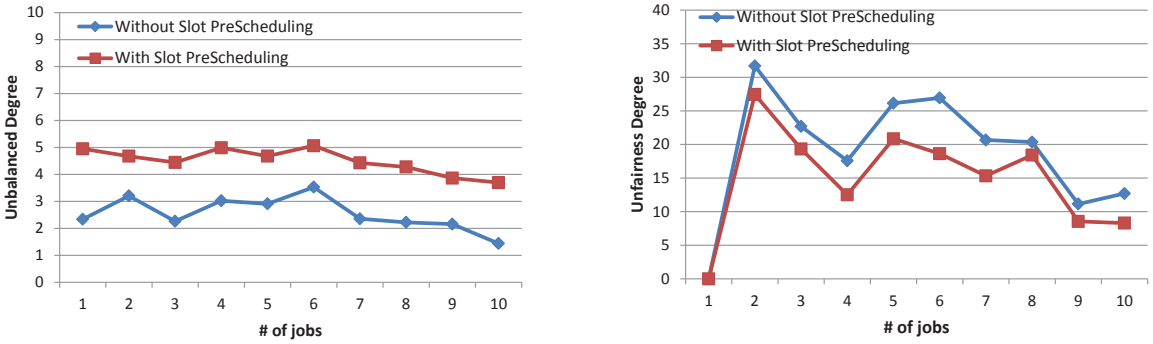


Fig. 5: The performance results with different percentages of map (or reduce) slots borrowed.



(a) the unbalanced degree for the cluster

(b) the unfairness degree for pools

Fig. 4: The results of unbalanced degree and unfairness degree for batch jobs.

Instead of borrowing all unused map (or reduce) slots for overloaded reduce (or map) tasks in our DHSA, we provide users with two configuration arguments *percentageOfBorrowedMapSlots* and *percentageOfBorrowedReduceSlots* to limit the amount of borrowed map/reduce slots, and ensure that tasks at the map/reduce phase are not starved. It is meaningful and important when users want to reserve some unused slots for incoming tasks, instead of lending all of them to other phases or pools. To show its impact on the performance, we perform an experiment with *sort* benchmark (320 map tasks and 200 reduce tasks) by varying values of arguments.

Let Pec_M and Pec_R denote *percentageOfBorrowedMapSlots* and *percentageOfBorrowedReduceSlots* respectively. Figure 5 presents the performance results under varied argument configurations. All speedup results are calculated with respect to the case when $Pec\_M = 0$ and $Pec\_R = 0$. We consider four cases: (1). Vary the value of Pec_R from 0 to 100 while fix $Pec\_M = 0$; (2). Vary the value of Pec_R while set $Pec\_M = 100$; (3). Vary the value of Pec_M while fix $Pec\_R = 0$; (4). Vary the value of Pec_M while fix $Pec\_R = 100$. We can see that, the performance improves by increasing either Pec_M or Pec_R. Particularly, there is a significant performance improvement(i.e., approximate 29%) when we fix the value of Pec_R and increase the value of Pec_M. It is because there are plenty of reduce tasks (e.g., 200 reduce tasks) but only 18 reduce slots. Thus increasing the percentage value of map slots (Pec_M) that can be borrowed would let more reduce tasks be scheduled using borrowed map slots, reducing the number of computation waves of reduce tasks and improving the utilization as well as performance of the Hadoop cluster.

REFERENCES

[1] F. Ahmad, S. Y. Lee, M. Thottethodi, T. N. Vijaykumar. *PUMA: Purdue MapReduce Benchmarks Suite*. ECE Technical Reports, 2012.

[2] 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 OSDI'10, pp. 1-16, 2010.

[3] Apache Hadoop NextGen MapReduce (YARN). http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[4] J. Chao, R. Buyya. *MapReduce Programming Model for .NET-Based Cloud Computing*. In Euro-Par'09, pp. 417-428, 2009.

[5] Q. Chen, C. Liu, Z. Xiao, *Improving MapReduce Performance Using Smart Speculative Execution Strategy*. IEEE Transactions on Computer, 2013.

[6] J. Dean and S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*, In OSDI'04, pp. 107-113, 2004.

[7] Z.H. Guo, G. Fox, M. Zhou, Y. Ruan.*Improving Resource Utilization in MapReduce.* In IEEE Cluster'12. pp. 402-410, 2012.

[8] Z. H. Guo, G. Fox, and M. Zhou.*Investigation of data locality and fairness in MapReduce*. In MapReduce'12, pp, 25-32, 2012.

[9] Z. H. Guo, G. Fox, and M. Zhou. *Investigation of Data Locality in MapReduce*. In IEEE/ACM CCGrid'12, pp, 419-426, 2012.

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

[11] M. Hammoud and M. F. Sakr. *Locality-Aware Reduce Task Scheduling for MapReduce*. In IEEE CLOUDCOM'11. pp. 570-576, 2011.

[12] M. Hammoud, M. S. Rehman, M. F. Sakr. *Center-of-Gravity Reduce Task Scheduling to Lower MapReduce Network Traffic*. In IEEE CLOUD'12, pp. 49-58, 2012.

[13] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. *Starfish: A Self-tuning System for Big Data Analytics*. In CIDR'11, pp. 261C272, 2011.

[14] H. Herodotou and S. Babu, *Profiling, What-if Analysis, and Costbased Optimization of MapReduce Programs*. In Proc. of the VLDB Endowment, Vol. 4, No. 11, 2011.

[15] S Ibrahim, H Jin, L Lu, B He, S Wu. *Adaptive Disk I/O Scheduling for MapReduce in Virtualized Environment*, In IEEE ICPP'11, pp.335-344, 2011.

[16] Y. C. Kwon, M. Balazinska, B. Howe, and J. Rolia. *SkewTune: mitigating skew in mapreduce applications*. In SIGMOD'12. pp. 25-36, 2012.

[17] Max-Min Fairness (Wikipedia). http://en.wikipedia.org/wiki/Max-min_fairness.

[18] B. Moseley, A. Dasgupta, R. Kumar, T. Sarl, *On scheduling in map-reduce and flow-shops*. In SPAA'11, pp. 289-298, 2011.

[19] C. Oğuz, M.F. Ercan, *Scheduling multiprocessor tasks in a two-stage flow-shop environment*. Proceedings of the 21st international conference on Computers and industrial engineering, pp. 269-272, 1997.

[20] B. Palanisamy, A. Singh, L. Liu and B. Jain, *Purlieus: Localityaware Resource Allocation for MapReduce in a Cloud*, In SC'11, pp. 1-11, 2011.

[21] J. Polo, C. Castillo, D. Carrera, et al. *Resource-aware Adaptive Scheduling for MapReduce Clusters*. In Middleware'11, pp. 187-207, 2011.

[22] J. Tan, X. Q. Meng, L. Zhang. *Coupling task progress for MapReduce resource-aware scheduling*. In IEEE Infocom'13, pp. 1618-1626, 2013.

[23] J. Tan, S. C. Meng, X. Q. Meng, L. Zhang. *Improving ReduceTask data locality for sequential MapReduce jobs*. In IEEE Infocom'13, pp. 1627-1635, 2013.

[24] S.J. Tang, B.S. Lee, and B.S. He. *MROrder: Flexible Job Ordering Optimization for Online MapReduce Workloads*. In Euro-Par'13, pp. 291-304, 2013.

[25] S.J. Tang, B.S. Lee, R. Fan and B.S. He. *Dynamic Job Ordering and Slot Configurations for MapReduce Workloads*, CORR (Technical Report), 2013.

[26] S.J. Tang, B.S. Lee, and B.S. He. *Dynamic Slot Allocation Technique for MapReduce Clusters*. In IEEE Cluster'13, pp. 1-8, 2013.

[27] S.J. Tang, B.S. Lee, B.S. He, H.K. Liu. *Long-Term Resource Fairness: Towards Economic Fairness on Pay-as-you-use Computing Systems*. In ACM ICS'14, 2014.

[28] J. Polo, Y. Becerra, et al. *Deadline-Based MapReduce Workload Management*, IEEE Transactions on Network and Service Management, 2013.

[29] PUMA Datasets. https://sites.google.com/site/farazahmad/pumadatasets.

[30] M. A. Rodriguez, R. Buyya. *Deadline based Resource Provisioning and Scheduling Algorithm for Scientific Workflows on Clouds*, IEEE Transaction on Cloud Computing, 2014.

[31] A. Verma, L. Cherkasova, R.H. Campbell, *Orchestrating an Ensemble of MapReduce Jobs for Minimizing Their Makespan*, IEEE Transaction on dependency and secure computing, 2013.

[32] A. Verma, L. Cherkasova, R. Campbell. *Two Sides of a Coin: Optimizing the Schedule of MapReduce Jobs to Minimize Their Makespan and Improve Cluster Performance*. In IEEE MASCOTS, pp. 11-18, 2012.

[33] Y. Wang, W. Shi, *Budget-Driven Scheduling Algorithms for Batches of MapReduce Jobs in Heterogeneous Clouds*, IEEE Transaction on Cloud Computing, 2014

[34] T. White. *Hadoop: The Definitive Guide, 3rd Version*. O'Reilly Media, 2012.

[35] M. Zaharia, A. Konwinski , A.D. Joseph , R. Katz , I. Stoica, *Improving MapReduce performance in heterogeneous environments*. In OSDI'08, pp.29-42, 2008.

[36] M. Zaharia, D. Borthakur, J. Sarma, K. Elmeleegy,S. Schenker,I. Stoica, *Job Scheduling for Multi-user Mapreduce Clusters*. Technical Report EECS-2009-55, UC Berkeley Technical Report (2009).

[37] M. Zaharia, D. Borthakur, J. Sarma, K. Elmeleegy,S. Schenker,I. Stoica, *Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling*. In EuroSys'10, pp. 265-278, 2010.

[38] C. Zhou, B.S. He, *Transformation-based Monetary Cost Optimizations for Workflows in the Cloud*, IEEE Transaction on Cloud Computing, 2014.