Towards Improving MapReduce Task Scheduling Using Online Simulation Based Predictions

Guanying Wang, Aleksandr Khasymski, Krish K. R., Ali R. Butt
Department of Computer Science, Virginia Tech
Email: {wanggy, khasymskia, kris, butta}@cs.vt.edu

***Abstract*—MapReduce is the model of choice for processing emerging big-data applications, and is facing an ever increasing demand for higher efficiency. In this context, we propose a novel task scheduling scheme that uses current task and system state information to drive online simulations concurrently within Hadoop, and predict with high accuracy future events, e.g., when a job would complete, or when task-specific data-local nodes would be available. These predictions can then be used to make more efficient resource scheduling decisions. Our framework consists of two components: (i) *Task Predictor* that predicts task-level execution times based on historical data of the same type of tasks; and (ii) *Job Simulator* that instantiates the real task scheduler in a simulated environment, and predicts expected scheduling decisions for all the tasks comprising a MapReduce job. Evaluation shows that our framework can achieve high prediction accuracy — 95% of the predicted task execution times are within 10% of the actual times — with negligible overhead (1.29%). Finally, we also present two realistic usecases, job data prefetching and a multi-strategy dynamic scheduler, which can benefit from integration of our prediction framework in Hadoop.**

I. Introduction

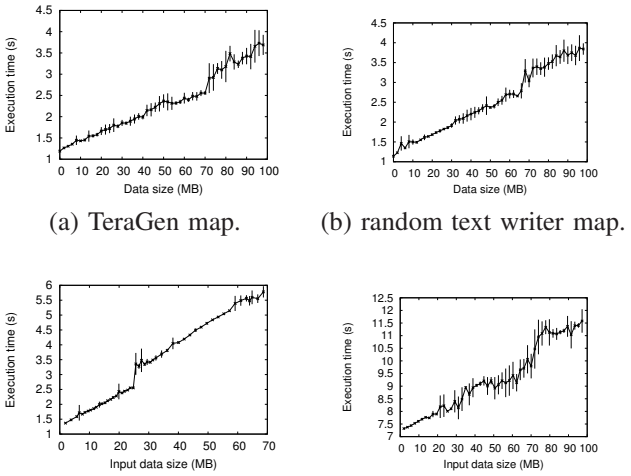
In recent years, MapReduce/Hadoop [1] has emerged as the de facto model for big data applications, and is employed by industry [2], [3], [4], [5] and academia [6], [7], [8] alike. Improving the efficiency of Hadoop is therefore crucial. Recent research is revisiting classic optimizations such as anticipatory scheduling and history-based prediction in the context of Hadoop. Delay scheduling [9] delays assignment of a task with non-optimal data locality, with the anticipation that a node with better locality may become available soon. PACMan [10] manages an in-memory cache on each Hadoop node, and tries to keep "hot" data blocks in the cache. Such systems typically rely on heuristics for resource management decisions, however, this can cause false positives/negatives when the heuristics fail to correctly capture the behavior of the current workload.

We make the observation that if future events can be accurately predicted in Hadoop, the information can be used to better drive resource management than the use of pure heuristics only, consequently improving overall system performance. For instance, if we can predict that no node with better data locality will become available in the near future, we can avoid the overhead of Delay Scheduling [9], and schedule the task immediately. Similarly, if we can predict when and where a task is likely to be run in PACMan [10], we can prefetch the needed data from disks into the memory cache right before the task starts. However, predicting behavior of an entire system is a challenging task. For example, in operating systems, external factors including user input and creation of new processes make future system state hard to predict. Similarly, high-performance computing applications usually involve complicated communication and dependencies between tasks, and hence not easily predictable. In contrast, MapReduce tasks are inherently independent with no inter-task communication or synchronization except for a well-defined shuffle phase, so task behavior depends only on local node resources. Moreover, MapReduce is a batch processing system where new tasks are added to a pending queue. Thus, the scheduler is aware of what workloads are in line to run in the near future. These properties are promising in enabling high-accuracy behavior prediction in a Hadoop.

In this paper, we present an online prediction framework that leverages the above properties of Hadoop that can drive more efficient task scheduling. Powered by history-based statistical prediction and online simulation, our framework can continuously predict future execution of tasks and jobs in live Hadoop. The online prediction framework comprises two components, *Task Predictor* and *Job Simulator*. The key insight in *Task Predictor* is that the execution time of a task is directly correlated with the size of the data the task processes. This allows us to derive a linear regression model for task execution time based on task input size, and apply the model to estimate execution time of pending tasks. *Job Simulator* predicts when a task will start to run and on which node, based on the current state of the system. *Job Simulator*, when invoked, replicates the current real system state and uses the execution time estimates from *Task Predictor* to simulate future states. The real MapReduce scheduler is modified to invoke *Job Simulator* and consider its outcomes before making scheduling decisions. Moreover, our simulator employs the same code as the real MapReduce scheduler to ensure that the simulated environment is as close to the real events as possible, and that the simulation can offer a good prediction for expected future system behavior.

Specifically, this paper makes the following contributions:

- We design an online framework for making predic-



(a) TeraGen map. (b) random text writer map.

(c) TeraSort map. (d) TeraSort reduce.

Figure 1. Task execution time versus data size for representative Hadoop applications.

tions based on real-time state of a Hadoop system. The framework provides a basis for designing more advanced prediction algorithms and simulations.

- We develop a simulation engine that can predict task scheduler decisions that will be made in the near future, and leverage this information to derive more efficient scheduling decisions.
- We implement the online prediction framework as a patch for Hadoop release 0.20.203.0. We intend to release the patch to the community to enable further research in efficient MapReduce Systems.
- We evaluate our framework using representative applications and show that it achieves high prediction accuracy — 95% of the predicted task execution times are within 10% of the actual times — with negligible overhead (1.29%).

II. Online Simulation Framework

In this section, we present the main components of our online simulation framework. We first describe the basis for our task execution time predictions, followed by how we utilize this information to simulate the system and predict when and where a particular task will be scheduled.

A. Estimating Task Execution Time Using Linear Regression

Our main objective is to predict the execution time for an entire job. However, since jobs in Hadoop comprise of one or more tasks, we first focus on predicting the execution time for individual tasks making up a job.

Empirical observation and intuition suggest that the total computation and I/O time for a MapReduce task is correlated with the input data size of the task. To better understand this relationship, we run a set of representative MapReduce applications [11] with varying input data size and observe the resulting execution time. For this purpose, we use a single worker node configured with one map slot and one reduce slot to eliminate the effect of any parallelization. We also configured MapReduce to start running reduce tasks only after all map tasks have finished, which ensures that only one task is running on a given node at any time. For each application, we create jobs to process 50 different data sizes. The results of this study are shown in data size versus execution time plots in Figure 1. Each data point on the graphs shows the average execution time and standard deviation observed for tasks with a given data size.

We observe that most jobs show linear correlation between data size and task execution time. In some jobs, all associated tasks have similar input data sizes and corresponding execution times. One notable difference is observed for (c) TeraSort map, where the result shows two different linear correlations, one occurring for input data size below 27 MB (approximately) and the other above that. The reason is that in a job such as TeraSort that involves both map and reduce phases, when input data size is less than a threshold, 27 MB for our test, a map task can write to a single output file and no merge is necessary. Whereas when input data size is larger than the threshold, the map task must write to multiple files, and later merge the results. This produces the different patterns as observed in the graph. The threshold may vary for different jobs and tasks, but the pattern is expected to be similar for map-reduce phase jobs.

Implementation of Task Predictor: Based on the above observations, we predict execution time of a task using information about previously finished tasks of the same type. We develop *Task Predictor* to first derive a performance model from tasks that have already finished, and then apply the model to predict execution time of new tasks. *Task Predictor* observes data sizes and corresponding execution times for all previous tasks of a particular type, and uses linear regression to determine the correlation between them. With the determined correlation model, we can then predict execution time of a task of the same type using its input data size.

Limitations: A limitation of *Task Predictor* is that it must have observed the complete execution of a type of task before it can reasonably predict the execution time for a new task of the type. If *Task Predictor* encounters a task it has not observed before, it simply estimates the execution time based on the input size of the task and a pre-specified default value. Such prediction may not be very accurate, but serves as a starting point, which is then refined when the same type of task is encountered on its next occurrences.

Also note that although we use a linear regression model based on input data size in our current implementation of *Task Predictor*, our framework can easily incorporate more complex models, such as those presented in [12], [7], [8].

Algorithm 1 Pseudo code for *Job Simulator* driving engine.

```
while queue is not empty AND not all jobs have finished do
    event ← next event in the queue
    advance virtual clock to when event occurs
    if event is a heartbeat event then
        SimTT prepare a status update
        SimTT calls SimJT.heartbeat()
        SimJT processes the heartbeat
        SimJT actions ← actions for SimTT
        SimJT response ← a heartbeat response with actions
        SimJT send response back to SimTT
        SimTT.performActions()
    else if event is a task finish event then
        SimTT mark task as COMPLETED
        if a map task finishes then
            SimTT map_slots ← map_slots + 1
        else if a reduce task finishes then
            SimTT reduce_slots ← reduce_slots + 1
        end if
    end if
end while
```

B. Predicting Job Schedules

Once we have obtained estimated task execution times, we use them to predict tasks scheduling decisions for the execution of an entire job as follows. We design a realistic simulator, *Job Simulator*, that captures all the information used by Hadoop task scheduler, instantiates the same task scheduler code in the simulated environment as that used for the real scheduling, and drives a simulation to mimic the scheduling decisions that are likely to occur in the real system given the current state. By speeding up virtual time, *Job Simulator* provides a prescient look of how the system will behave in the near future if the scheduler were to continue its course with the currently batched jobs. This also allows *Job Simulator* to predict when and where particular tasks would be scheduled. The real scheduler can then use the predicted information from *Job Simulator* to make more efficient decisions if needed. Finally, *Job Simulator* is updated periodically and the above process is repeated.

A dedicated thread in the Hadoop *JobTracker* process is added to run *Job Simulator*. First, we take a snapshot of the current status of *JobTracker* and instantiate a simulated *JobTracker* and task scheduler component, *SimJT*, in *Job Simulator*. *SimJT* contains replicated information for each running job, tasks in each job, running tasks, etc., from the real *JobTracker*. We also instantiate a simulated *TaskTracker* object (*SimTT*) in *Job Simulator* for each *TaskTracker* that is active in the real system at the time when the current round of simulation started. Figure 2 shows the architecture of *Job Simulator*. *JobTracker* and task scheduler in *Job Simulator* both run the same code as their counterparts in the real system. *SimTT*s are simulated objects that are controlled by a discrete event simulator engine. *SimTT*s communicate with *SimJT*, similar to how *TaskTracker*s communicate with the *JobTracker*. One challenge we faced was that the original *JobTracker* code only works with physical time, but *SimJT* had to run with virtual time during the simulations. We refactored the *JobTracker* code to make it compatible with both physical and virtual time, thus enabling the use of the same code directly within *SimJT*. The simulation engine maintains a priority event queue sorted by the virtual time when each event is scheduled to occur. The engine advances the virtual time to the point when the next event in the queue will occur and processes the event. During the processing, more events can be inserted into the queue as events that will occur in the virtual future. The engine repeats the process until the queue is empty, or in our case, when all jobs in the system are completed.

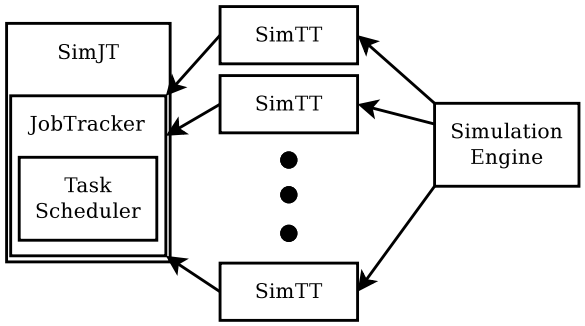


Figure 2. Overview of *Job Simulator* architecture.

Algorithm 1 shows the pseudo code for the simulation engine. We currently implement two types of events: a `heartbeat` event and a `task finish` event. On receiving a `heartbeat` from *SimJT*, the associated *SimTT* creates an up-to-date status and sends a heartbeat message to *SimJT*. *SimJT* processes heartbeat messages, calls the task scheduler to make scheduling decisions if necessary, and returns a heartbeat response with actions to the *SimTT*. *SimTT* then performs the actions such as, a launch-map-task action, or a all-maps-completed action (to launch a reduce task). The processing of a heartbeat response message is done after *SimTT* has processed all actions requested in the message. On a `task finish`, *SimTT* simply marks the task as COMPLETED and frees the slots occupied by the task. The engine then moves on to process the next event in the queue. The procedure is repeated until all jobs in the simulation complete. To avoid using too much resources on the physical machine, each simulation stops after a long enough virtual period (default 1 hour) has elapsed, typically within about 10 seconds in physical time. This amount of simulation is sufficient, as status of the real system may

change due to newly submitted jobs, machine failure or recovery, random noise, etc. Running the simulation for any longer will likely result in it diverging significantly from the actual system behavior.

Predictions Based on Online Simulation: Job Simulator can predict each scheduling decision to be made by the task scheduler. If the virtualized environment supplied to the task scheduler mimics the real environment, the task scheduler will make the same scheduling decision in simulation as it will in the real system with high accuracy since the simulation implements the same code as the real system. Every decision is valuable information, and can be used to improve overall system performance. When all tasks of a job finish in the simulation, we can predict the total job finish time as well as the start and finish time of all of its tasks.

The simulator engine drives *SimTT*, which in turn calls *SimJT* and the task scheduler on a `heartbeat`. The engine and *SimTT* can be viewed as a virtualized environment, which surrounds *SimJT* and the task scheduler, as if *JobTracker* and task scheduler are running in a real system. Furthermore, since we do not modify the existing task scheduler code, *Job Simulator* is compatible with any deterministic task scheduler. To make a task scheduler compatible with our *Job Simulator*, the scheduler must implement a `copy()` method to create a new task scheduler object that is a snapshot of itself and the scheduler must support virtual time. We have ported the default JobQueueTaskScheduler and Fair Scheduler (naive fair scheduler as discussed in [9]) to work with *Job Simulator*.

Limitation: One limitation of *Job Simulator* is that it can predict the execution time only of the jobs that are submitted when the simulation starts. In contrast to real Hadoop, *Job Simulator* does not have a job client, and no new jobs can be added during a simulation run. However, this is not a problem since the simulation is rerun periodically and new jobs can be captured in the next round. *Job Simulator* also cannot predict hardware and network failures. In every simulation, all jobs are simulated with the assumption of no failure. When a failure does occur in the real system, we rely on the subsequent simulation runs to include the failure and simulate its impact. This is also not a major hurdle, as the simulation can be re-invoked soon after the system recovers, and *Job Simulator* will quickly adapt to the new system state.

Performance Impact of Job Simulator: In order to be practically useful, the simulation time must not be longer than the interval between two simulations executions. In a large cluster, *JobTracker* might be too busy running simulations and not be able to keep up with heartbeat messages from *TaskTracker*s. In order to minimize performance impact on the *JobTracker*, *Job Simulator* can be separated from *JobTracker* as a stand-alone process or even run on another node. The new *Job Simulator* process can communicate with *JobTracker* to get status update and send simulation results via periodical heartbeat messages. Thus, *Job Simulator* process will minimize the overhead on *JobTracker* process, and utilize processing power of multi-core processors or even processing power of another node.

III. Evaluation

We have implemented the online prediction framework including both *Task Predictor* and *Job Simulator* as a patch for Apache Hadoop release 0.20.203.0 in about 6000 lines of code. In this section, we evaluate the prediction accuracy of *Task Predictor* and *Job Simulator* under two schedulers: JobQueueTaskScheduler (the default Hadoop FCFS scheduler) and Fair Scheduler [9]. We also investigate the performance impact of our approach on *JobTracker*.

We conducted our experiments on a small cluster with 1 *JobTracker* and 3 *TaskTracker*s. Nodes are connected via a 1000 Mbps link. Each *TaskTracker* is configured with 2 map and 2 reduce slots. We configured MapReduce to launch reduce tasks only after all map tasks have finished. Speculative execution is turned off.

A. Prediction Accuracy of Task Predictor

In the first set of tests, we evaluate the accuracy of *Task Predictor*'s task execution time predictions. We run a workload with 10 grep jobs and 10 word-count jobs, and record the predicted and actual execution time for each associated task. The jobs are submitted together in the beginning of the test. We run the same 20-job workload twice, first for training, and then for the testing. *Job Simulator* is turned off in this experiment. We run the same experiment under both FCFS scheduler and Fair Scheduler.

Map tasks: The results for map tasks under the studied schedulers are shown in Figure 3 and 4. The graphs show normalized error expressed as percentage of predicted execution time against actual execution time of a task. A positive error means that a predicted value is larger than the actual value, while a negative error means that the predicted value is smaller. The predictions are ordered by the order in which each task finished as observed in the Hadoop task log. Overall, we observe that 95% of the predictions are within 10% of the actual measurements, and 75% of all errors are within 5% of real execution time. These results are promising in the showing the efficacy of our approach.

Reduce tasks: Prediction accuracy of our approach for reduce tasks under the two schedulers also sees high accuracy, 95% of all prediction errors are within 10% of actual measurements. We observe few significant outliers, e.g., a task running for 3 seconds predicted to run for 7.8 seconds. We believe that such outliers can be reduced with more training that is possible in a long running scheduler with much more historic information. Moreover, given high accuracy for most of the tasks, we expect the impact of such missed predictions to be small.

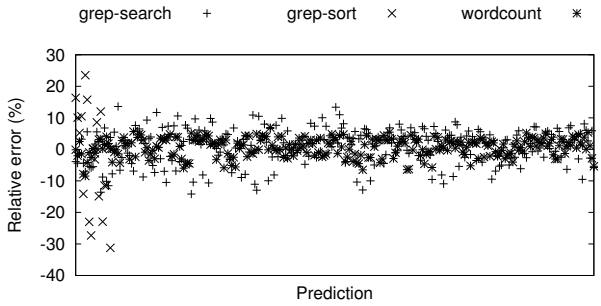


Figure 3. Prediction errors for map tasks under FCFS scheduler.

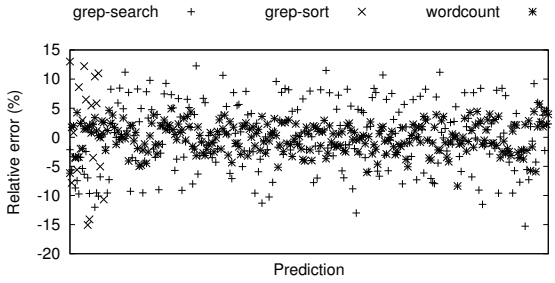


Figure 4. Prediction errors for map tasks under Fair Scheduler.

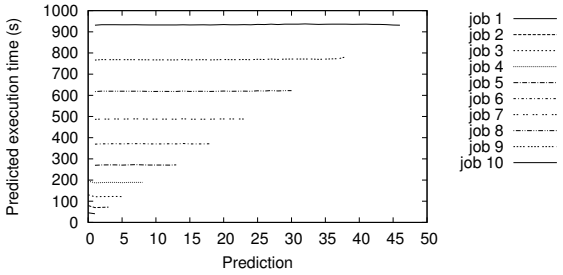


Figure 5. Prediction of job execution time under FCFS scheduler.

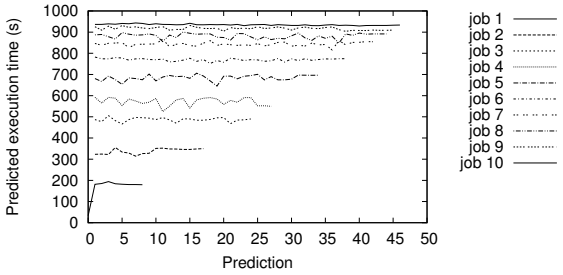


Figure 6. Prediction of job execution time under Fair Scheduler.

B. Prediction Accuracy of Job Simulator

In the next test, we study the accuracy of *Job Simulator*. For this purpose, we run a workload of 10 word-count jobs twice, first for training *Task Predictor* as before, and then for testing *Job Simulator*. In this case, we did not use grep jobs as they involve dependency between jobs. For every test, we record predicted execution time of each task and each job. We show the results of predicted execution time of each of the 10 jobs under the two studied schedulers in Figures 5 and 6. Individual lines show how predicted execution time of each job changes as the workload executes. A line stops when a job completes as no further predictions are made for that job. Flat lines, as seen for FCFS in Figure 5, show that our predictions do not change over time and are accurate from the beginning of the workload run. Error in predicted execution time of each job is observed to be within 10 seconds for the 900 seconds workload. Results for Fair Scheduler also show stable prediction for each job, with error observed to be within 40 seconds. Given the 900 seconds workload runtime, we note that we can predict finish times of jobs 15 minutes earlier (as long as no other job is submitted during our simulation).

To further understand the accuracy of *Job Simulator*, we divide prediction of task execution time provided by *Task Predictor* and prediction of task start time provided by *Job Simulator*. We compare the start time of each task predicted by *Job Simulator* against the actual start time of the task. Since *Job Simulator* runs periodically, the information is most useful for a short time window in the near future, when the scheduler can act on the information. To test this, we determine all tasks that start to run in a 30-second or 60-second window after each simulation run, and compare the error of actual start time and predicted start time of these tasks. Figures 7 and 8 show average prediction error of start time of all tasks within each window. FCFS results show almost perfect predictions with average errors of less than 2 seconds for the 900 second workload for both the cases of 30-second windows and 60-second windows. Fair Scheduler results, however, show much higher average errors, up to 70 seconds. This is because Fair Scheduler is more sensitive to small differences in task execution time. A small difference may result in a task from a different job scheduled, or a task scheduled to another node or after a long interval. Some tasks are predicted out-of-order as compared to actual execution trace, so the error could be very large.

To avoid bias due to high-error tasks, we calculated average percentage of tasks within each window that are predicted to start within an error bound. Moreover, map tasks must be predicted to run on the same nodes that they are actually scheduled on. The result is shown in Figure 9 and shows that under Fair Scheduler, nearly 80% of tasks in a 30-second window are predicted correctly with an error of less than 2 seconds. Hence, we observe that *Job Simulator* is accurate for most of the tasks, even though 20% tasks are predicted to run out-of-order with much higher errors.

C. Performance Overhead of Online Simulation

To study the overhead of our online prediction framework caused by the periodic running of the online simulation,

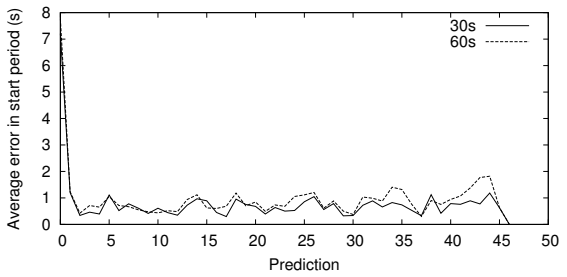


Figure 7. Average prediction error for task start times within a short time window under FCFS scheduler.

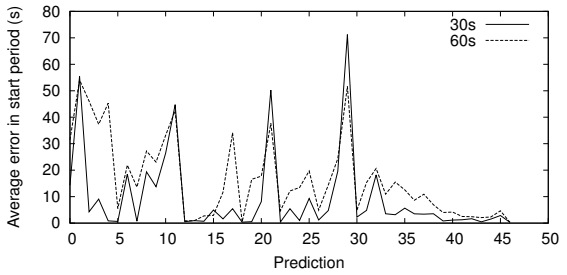


Figure 8. Average prediction error for task start times within a short time window under Fair Scheduler.

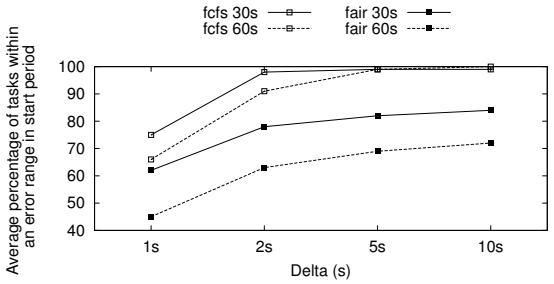


Figure 9. Percentage of relatively accurate predictions within a short window.

Job Simulator interval	average exec. time (s)	maximum exec. time (s)	heartbeat proc. rate
off	735	1293	1.00
60s	728	1297	0.98
30s	743	1302	0.96
20s	751	1310	0.94
10s	769	1331	0.89
5s	825	1384	0.80

Table I
OVERHEAD OF RUNNING *Job Simulator* MEASURED IN TERMS OF AVERAGE JOB EXECUTION TIME, MAXIMUM JOB EXECUTION TIME AND HEARTBEAT PROCESSING RATE.

we run the same workload of 10 grep and 10 word-count jobs with and without *Job Simulator*. We vary the simulation period and measure the impact of running the extra simulations on our workload performance. We use the Fair Scheduler for this purpose (though not shown, FCFS shows similar results). We summarize average job execution time, maximum job execution time (workload execution time), and heartbeat processing rate (calculated by number of heartbeats processed divided by length of experiment) in Table I. Running *Job Simulator* every 20 seconds incurs a 1.29% overhead in workload execution time and a 5.29% reduction in heartbeat processing rate. In larger clusters, we expect higher overhead on *JobTracker* and as a result propose separating *Job Simulator* process in order to lower the overhead on *JobTracker*.

In summary, our results show that *Job Simulator* can help in improving scheduling performance while imparting small overhead.

IV. Case Studies: Dynamic Scheduling and Data Prefetching

In this section, we introduce two use cases for our online simulations framework, namely data caching and prefetching, and dynamic scheduler selection (Figure 10), and outline how it can be employed to improve the overall performance and efficiency in each case.

A. Data Caching and Prefetching

Data caching for MapReduce systems has been the focus of recent research. For instance, PACMan [10] is a caching service for data-intensive parallel computing frameworks such as MapReduce. While PACMan is effective in reducing average completion time of jobs by over 50%, the authors also note that data processed by over 30% of tasks is accessed only once, which cannot benefit from caching. Thus, even with large amounts of RAM, e.g. 20 GB per node in PACMan, caching efficiency can still be improved.

Our system can facilitate much higher efficiency than generalized caching. Rather than caching previously accessed data in memory and hoping that some tasks will access the cached data, we can use our system to predict which data blocks will be accessed on which node. Then, we can prefetch or retain only the needed data blocks into memory just before tasks start to run. Thus, the I/O latency of the data access is hidden from the task when it starts, and depending on the accuracy of our predictions (that is high as shown in our evaluation), we can achieve significantly higher hit ratio. Moreover, data that is not likely to be accessed again can be discarded immediately, e.g., the 30% of jobs observed in PACMan will have their data prefetched and see a performance gain, but their data will be discarded after the first use to free the cache and benefit other jobs. Thus, using our approach also eliminates the need for reserving large amounts of RAM for caching, which would otherwise be needed to support such jobs.

Prefetching is good at reducing data loading time for all tasks with modest RAM usage. However, it can impose increased load on disk if we discard data from RAM as soon as the processing is finished. Prefetching works best if a long queue of jobs are waiting to run. In contrast, caching can reduce load on disks by absorbing recurring access to the

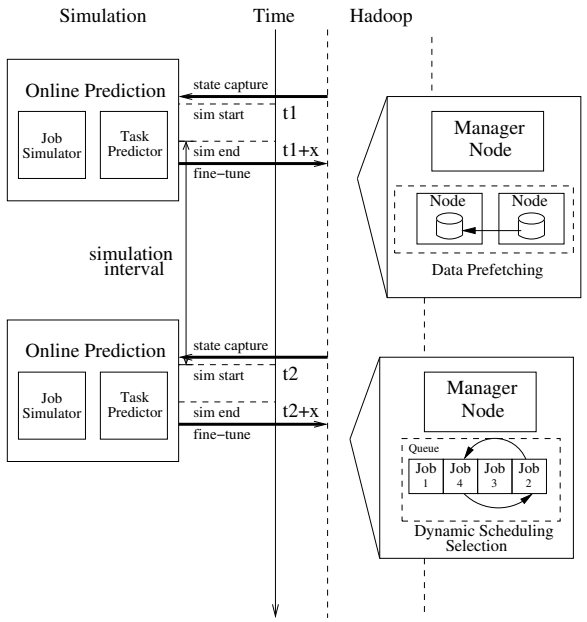


Figure 10. Use Cases of Online Prediction Framework.

same data block from disks. To get the best performance and efficiency, scheduler should work together with both caching and prefetching.

We will use an example to illustrate how caching and prefetching can improve performance and efficiency of Hadoop. Suppose 3 jobs as shown in Table II are submitted to a MapReduce system. The finish column shows estimated finish time of each job in the baseline MapReduce system. With these numbers, in a baseline system, the 3 jobs account for 340 cluster-seconds in a baseline MapReduce system. With caching enabled, only job 3 can benefit from caching because it accesses data $d1$ which was also accessed by job 1. Job 1 and job 2 access data for the first time, so these data cannot be cached and cannot be sped up. Assume job 3 get a speed up of 50%, and it runs for 50 seconds. Therefore, The 3 jobs account for 290 cluster-seconds in total with caching enabled. If we can predict task locations and prefetch data blocks for all tasks, we can preload data into RAM for each job and we can speed up each job by 50%, regardless of whether data has ever been accessed before. The 3 jobs account for 170 cluster-seconds in total with prefetching enbaled. In terms of performance, Prefetching outperforms baseline by 100% and caching by 70%. Prefetching can improve performance of the system, but it imposes more load on underlining disk resource. Because we prefetch data $d1$ again for job 3, $d1$ is read twice – once for job 1 and once for job 3. Total data read for prefetching is 3 GB, the same as baseline. In comparison, with caching enabled, we do not need to read data $d1$ again for job 3, because $d1$ is already cached in RAM. Total data read for caching is 2 GB, a 50% improvement over baseline and prefetching.

Job	Start	Finish	Accessing data
1	0	100	d1 (1GB)
2	10	150	d2 (1GB)
3	200	300	d1 (1GB)

Table II
JOBS IN A MAPREDUCE SYSTEM.

Job	Baseline	w/ caching	w/ prefetching	w/ both
1	100	100	50	50
	1GB	1GB	1GB	1GB
2	140	140	70	70
	1GB	1GB	1GB	1GB
3	100	50	50	50
	1GB	0GB	1GB	0GB
Overall	340	290	170	170
	3GB	2GB	3GB	2GB

Table III
PERFORMANCE AND RESOURCE CONSUMPTION OF THE JOBS.

In fact, caching and prefetching can work together and make the system optimal. Consider the same example with caching and prefetching both enabled. Consider with prefetching enabled, and we keep data prefetched and processed in RAM, as we would do with caching enabled. All jobs can be sped up by 50%, and total CPU used is 170 cluster-seconds. For data read from disks, we don't need to read $d1$ again for job 3, so total data read from disks is 2 GB. Overall, Table III summarizes the pros and cons of caching and prefetching.

B. Dynamic Scheduling

The default scheduler in MapReduce, JobQueueTaskScheduler, is a first-come-first-serve (FCFS) scheduler. Under JobQueueTaskScheduler, all jobs are sorted by submission order into a queue, and the scheduler always picks new tasks from the first job in the queue, until the first job finishes and the second job is promoted to be the new first job. A major drawback of FCFS is that subsequent jobs must wait until preceding jobs finish. If the first job in the queue is a large job, subsequent small jobs must wait for a long period before they are executed. A new class of schedulers, including Quincy [13] and Delay Scheduling [9], tries to solve the problem of long delays for small jobs in FCFS. Multiple jobs are allowed to run concurrently and share the resource of a cluster in term of task slots fairly, so small jobs are not blocked by long-running large jobs. Wang et. al. [11] have shown different workloads perform differently under different scheduler. Hence, the scheduling strategy should be determined at runtime, based on the properties of the jobs currently running in the cluster and the ones waiting in the queue.

Our online prediction framework can solve exactly that problem, by predicting the execution time of the current workload under different scheduler policies in faster virtual time. In Section III we have shown that running the simulation every 20 seconds is sufficiently frequent to be accurate, as running time of most tasks is in the minutes. Thus, the system can provide feedback in time to make the next real scheduling decision. Of course, this technique should not be

taken to an extreme where the system is forced to switch back and forth between two schedulers that produce very similar results for the current workload. To prevent this, and any overhead associated with that scenario, the system should change to a different scheduler only if that results in significant performance gain.

V. Related Works

Job Simulator shares its goal of integrated simulation with another MapReduce simulator, Mumak [14], but differs from it in that Mumak is designed to run offline driven by a trace, whereas our *Job Simulator* runs online along with the real *JobTracker* and is driven by the live workload on the cluster. In *Job Simulator*, we must predict execution time of a new task based on historical. Another difference is that *Job Simulator* runs periodically, and each time it runs, we must take snapshots of *JobTracker*, task scheduler, and worker nodes and replicate them into *Job Simulator*.

Several recent works [12], [7], [8] are based on predefined performance models within each MapReduce task. We adopt a simple linear model in this paper. Since MapReduce jobs are a collection of a large number of smaller tasks, simple linear model is accurate enough for a computation framework like Hadoop. There has been other extensive previous research in simulation of MapReduce workloads and setups [14], [15], [12], [16], [17], [18].

Job Simulator is unique in its goal of predicting a live MapReduce task. Compared to all other MapReduce simulators, our prediction framework is arguably more realistic, easier to verify and evaluate, and can directly benefit system performance. Our framework predicts what is about to happen in the current system in the near future, and therefore predictions can be verified and evaluated. The results from the prediction can be readily used to improve performance of the live system. In contrast other simulators either try to match what has already happened in the past, or simulate a particular cluster environment offline.

VI. Conclusion

In this paper, we have described a simulation-based online prediction framework for Hadoop. We design and employ our simulator to predict near-future system behavior based on the current state of the Hadoop scheduler. The information can then be incorporated into the scheduler to better allocate jobs to nodes, and achieve overall higher performance. We evaluate the proposed simulation framework using TeraGen, TeraSort, grep, and wordcount. We find that for studied applications, 95% of the predicted task execution times are within 10% of the actual values, and 80% of predicted task start times (in a 30-second window) are within 2 seconds of the actual start times. In our future work, we plan to leverage the prediction framework to implement prefetching for MapReduce to improve latency of initial I/O, and a dynamic multi-strategy scheduler that can switch between multiple scheduling strategies based on current workload.

Acknowledgment

This work was sponsored in part by the NSF under Grant No: CNS-1016408, CNS-1016793, CCF-0746832, and CNS- 1016198.

References

[1] Apache Software Foundation., "Apache Hadoop," Feb. 2011. [Online]. Available: http://hadoop.apache.org/

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI*, 2004, pp. 137–150.

[3] Baldeschwieler, Eric, "Hortonworks Manifesto." [Online]. Available: http://hortonworks.com/blog/our-manifesto/

[4] D. Borthakur, S. Rash, R. Schmidt, A. Aiyer, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, and A. Menon, "Apache hadoop goes realtime at Facebook," in *SIGMOD*, Jun. 2011, p. 1071.

[5] Amazon, "Amazon Elastic MapReduce (EMR)." [Online]. Available: http://aws.amazon.com/elasticmapreduce/

[6] H. Monti, A. R. Butt, and S. S. Vazhkudai, "CATCH: A Cloud-based Adaptive Data Transfer Service for HPC," in *Proc. IPDPS*, 2011.

[7] 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*, 2011, pp. 261–272.

[8] H. Herodotou, "Hadoop Performance Models," Duke University, Tech. Rep. CS-2011-05, Feb. 2011.

[9] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proc. EuroSys*, 2010, pp. 265–278.

[10] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "Pacman: coordinated memory caching for parallel jobs," in *NSDI*, 2012, pp. 20–20.

[11] G. Wang, A. R. Butt, H. Monti, and K. Gupta, "Towards Synthesizing Realistic Workload Traces for Studying the Hadoop Ecosystem," in *MASCOTS*, 2011.

[12] G. Wang, A. R. Butt, P. Pandey, and K. Gupta, "A simulation approach to evaluating design decisions in mapreduce setups," in *MASCOTS*, 2009.

[13] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proc. sosp*, 2009, pp. 261–276.

[14] A. C. Murthy, "Mumak: Map-Reduce Simulator," MAPREDUCE-728, Apache JIRA, 2009. [Online]. Available: http://issues.apache.org/jira/browse/MAPREDUCE-728

[15] A. Verma, L. Cherkasova, and R. H. Campbell, "Play It Again, SimMR!" in *2011 IEEE International Conference on Cluster Computing*. IEEE, Sep. 2011, pp. 253–261.

[16] F. Teng, L. Yu, and F. Magoulès, "SimMapReduce: A Simulator for Modeling MapReduce Framework," in *2011 Fifth FTRA International Conference on Multimedia and Ubiquitous Engineering*. IEEE, Jun. 2011, pp. 277–282.

[17] Y. Liu, M. Li, N. K. Alham, and S. Hammoud, "HSim: A MapReduce simulator in enabling Cloud Computing," *Future Generation Computer Systems*, May 2011.

[18] S. Hammoud, "MRSim: A discrete event based MapReduce simulator," in *2010 Seventh International Conference on Fuzzy Systems and Knowledge Discovery*. IEEE, Aug. 2010, pp. 2993–2997.