Orchestrating an Ensemble of MapReduce Jobs for Minimizing Their Makespan

Abhishek Verma, *Member, IEEE,* Ludmila Cherkasova, *Member, IEEE,* and Roy H. Campbell, *Fellow, IEEE*

Abstract—Cloud computing offers an attractive option for businesses to rent a suitable size MapReduce cluster, consume resources as a service, and pay only for resources that were consumed. A key challenge in such environments is to increase the utilization of MapReduce clusters to minimize their cost. One way of achieving this goal is to optimize the execution of Mapreduce jobs on the cluster. For a set of production jobs that are executed periodically on new data, we can perform an off-line analysis for evaluating performance benefits of different optimization techniques. In this work, we consider a subset of production workloads that consists of MapReduce jobs with no dependencies. We observe that the order in which these jobs are executed can have a significant impact on their overall completion time and the cluster resource utilization. Our goal is to automate the design of a job schedule that minimizes the completion time (makespan) of such a set of MapReduce jobs. We introduce a simple abstraction where each MapReduce job is represented as a pair of map and reduce stage durations. This representation enables us to apply the classic Johnson algorithm that was designed for building an optimal two-stage job schedule. We evaluate the performance benefits of the constructed schedule through an extensive set of simulations over a variety of realistic workloads. The results are workload and cluster-size dependent, but it is typical to achieve up to 10%-25% of makespan improvements by simply processing the jobs in the *right* order. However, in some cases, the simplified abstraction assumed by Johnson's algorithm may lead to a suboptimal job schedule. We design *a novel heuristic*, called *BalancedPools*, that significantly improves Johnson's schedule results (up to 15%-38%), exactly in the situations when it produces suboptimal makespan. Overall, we observe up to 50% in the makespan improvements with the new *BalancedPools* algorithm. The results of our simulation study are validated through experiments on a 66-node Hadoop cluster.

Index Terms—MapReduce, Hadoop, batch workloads, optimized schedule, minimized makespan.

1 Introduction

Private and public clouds offer a new delivery model with virtually unlimited computing and storage resources. An increasing number of companies are exploiting the MapReduce paradigm [1] and its open-source implementation Hadoop as a platform choice for efficient *Big Data* processing and advanced analytics over unstructured information. This new style of large data processing enables businesses to extract information and discover novel data insights in a non-traditional and game-changing way. For many companies, their core business depends on a timely analysis and processing of large quantities of new data. The data analysis applications might be of different complexities, resource needs, and data delivery deadlines. This diversity create competing requirements for program design, job scheduling, and workload management policies in MapReduce environments. However, in spite of different user objectives a common goal is to ease the use of the MapReduce framework and enhance Hadoop performance.

To ease the task of writing complex analytics programs several projects, e.g., Pig [2] and Hive [3], provide high-level SQL-like abstractions on top of MapReduce engines. Programs written in such frameworks are compiled into directed acyclic graphs (DAGs) of MapReduce jobs. Often these automatically generated programs are less efficient than the "hand-crafted" MapReduce jobs. There are continuous efforts [4], [5] to improve the implementation efficiency of "SQL-to-MapReduce" translators and optimize the completion time of automatically generated MapReduce jobs and workflows.

The job execution efficiency is particularly important for processing production workloads when a given set of MapReduce jobs and workflows (DAGs) need to be executed periodically for processing new data [6], e.g., hourly, daily, or weekly. Typically, the default FIFO scheduler is used for processing an ensemble of such production jobs since the main *performance objective* is to *minimize the overall execution time (makespan)* of a given set. Such production workloads are analyzed off-line for optimizing their execution. There is a slew of optimization methods introduced for improving data read/write efficiency in a set of production jobs. For different MapReduce jobs that operate over the same dataset, a more efficient job scheduling [7], [8] was proposed to merge their executions in such a way that the input data is only scanned once. In this work, we consider a subset of production workloads consisting of jobs with no dependencies. Such independent jobs arise, for example, while processing different datasets, or as a result of optimized Pig/Hive queries translated in a single MapReduce job[1]. For jobs with no dependencies

- *A. Verma and R. Campbell are with the Department of Computer Science, University of Illinois at Urbana-Champaign, IL, 61801.*
E-mail: {verma7, rhc}@illinois.edu
- *L. Cherkasova is with Hewlett-Packard Labs, Palo Alto, CA, 94304.*
E-mail: lucy.cherkasova@hp.com

1. Currently, 11 out of 17 queries in the PigMix benchmark (see http://wiki.apache.org/pig/PigMix) translate to a single MapReduce job.

we discuss a different cause of execution inefficiency inherent to the MapReduce computation that processes map and reduce tasks in two stages separated by a synchronization barrier. The order in which jobs are executed can have a significant impact on the overall processing time, and therefore, on the achieved cluster utilization. For data-dependent jobs, the successive job can only start after the current one is entirely finished. However, for data-independent jobs, once the previous job completes its map stage and begins the reduce stage, the next job can start executing its map stage with the released map resources in a pipelined fashion. Thus, there is an overlap in job executions when different jobs use complementary cluster resources: map and reduce slots. Note that a larger overlap in job executions leads to better job pipelining, increased cluster utilization, and an improved job execution time, while using the same number of machines. Our goal is to automate the construction of a job schedule that minimizes[2] the completion time of such a set of MapReduce jobs.

The paper makes the following key contributions:

- We apply the classic Johnson algorithm [9] to construct an optimized schedule for a set of independent MapReduce jobs. We represent each MapReduce job J_i by a pair of computed durations (m_i, r_i) of its map and reduce stages. This representation enables us to apply Johnson's algorithm that has been proposed for building an optimal two-stage job schedule [9]. For periodic production jobs we can perfrorm their automated profiling from *past executions*. When jobs in a batch need to process new datasets, we use the knowledge of extracted job profiles to pre-compute new estimates of jobs' map and reduce stage durations, and then construct an optimized schedule for *future executions*.
- We evaluate performance benefits of the constructed schedule through extensive simulations over a variety of realistic workloads. The performance results are workload and cluster-size dependent, but we typically achieve up to 10%-25% makespan improvements. Moreover, we design a special simulation framework based on the accurate MapReduce simulator SimMR [10] to predict the batch completion time and to analyze potential performance improvements of different schedules.
- Our analysis reveals that Johnson's schedule may lead to a suboptimal makespan in some cases. The proposed abstraction of MapReduce jobs as a pair of map and reduce stage durations obscures the amount of resources each job may be able to utilize. In many cases, there can be a set of jobs that can use only a subset of the cluster resources. These situations require a different approach for job ordering and present an opportunity for additional performance improvements. We design *BalancedPools*, a *novel heuristic* that efficiently utilizes characteristics and properties of MapReduce jobs in a given workload for constructing the optimized job schedule.
- The detailed evaluation of the proposed heuristic demonstrates makespan improvements of up to 15%-38% for situations where Johnson's schedule is suboptimal. The *BalancedPools* algorithm may provide up to 50% in the overall makespan improvements by partitioning a set of given jobs into two pools, each one with a similar makespan. The algorithm achieves this through a balancing act of a tailored resource allocation to each pool. The results of our simulation study are validated through experiments on a 66-node Hadoop cluster.

This paper is organized as follows. Section 2 provides a background on MapReduce, Hadoop, and the performance modeling framework. Section 3 introduces the job scheduling algorithms to minimize the batch makespan. Section 3.2 explains Johnson's algorithm and its application to MapReduce jobs scheduling. Section 3.3 introduces a novel *BalancedPools* heuristic. These job schedules are evaluated in Section 4. Section 5 describes the related work. Sections 6 and 7 discuss future directions and summarize our work.

2 Background

This section provides a basic background on the MapReduce framework [1] and outlines a MapReduce performance model introduced in ARIA [11] that is used in constructing the optimized job schedule. This model provides the estimates on the completion time of the map and reduce stages as a function of allocated resources.

2.1 MapReduce Framework

In the MapReduce framework, computation is expressed as two functions: *Map* and *Reduce*. *Map* takes an input pair of data with a type in one data domain and produces a list of pairs in a different domain: $map(k_1, v_1) \rightarrow list(k_2, v_2)$. The generated values associated with the same key k_2 (so-called intermediate data) are grouped together and then passed to the *Reduce* function. *Reduce* takes intermediate key k_2 with a list of values and processes them to form a new list of values in the same domain: $reduce(k_2, list(v_2)) \rightarrow list(v_3)$.

MapReduce jobs are distributed and executed across multiple machines: the map stage is partitioned into *map tasks* and the reduce stage is partitioned into *reduce tasks*. Each map task processes a logical split of input data that generally resides on a Hadoop distributed file system (HDFS). The map task reads the data, applies the user-defined map function on each record, and buffers the resulting output. This data is sorted and partitioned for different reduce tasks, and written to the local disk of

2. In this work, we use terms "minimize" or "optimize" to represent the objective function for constructing the job schedule. We show that the makespan minimization problem for MapReduce jobs is NP-hard. Therefore, we offer an efficient heuristic that constructs a job schedule with a significantly improved makespan (but which is not necessarily minimal or optimal).

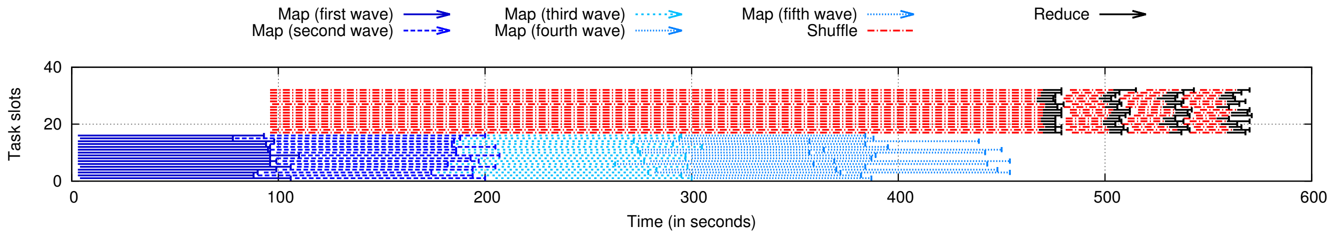


Fig. 1. WikiTrends application executed in a Hadoop cluster with 16 map and 16 reduce slots.

the machine executing the map task. The reduce stage consists of three phases: shuffle, sort and reduce. In the shuffle phase, the reduce tasks fetch the intermediate data files from the already completed map tasks, thus following the "pull" model. In the sort phase, the intermediate files from all the map tasks are sorted. After all the intermediate data is shuffled, a final pass is made to merge all these sorted files. Thus, the shuffle and sort phases are interleaved, and in this paper, we combine these activities under the shuffle phase. Finally, in the reduce phase, the sorted intermediate data is passed to the user-defined reduce function. The output from the reduce function is generally written back to HDFS.

Job scheduling in Hadoop is performed by a master node, which manages a number of worker nodes in the cluster. Each worker has a fixed number of map slots and reduce slots, which can execute tasks. The number of map and reduce slots is statically configured (typically, one or two per core or disk). The workers periodically send heartbeats to the master to report the number of free slots and the progress of tasks that they are currently running. Based on the availability of free slots and the scheduling policy, the master assigns map and reduce tasks to slots in the cluster.

2.2 Job Profile and MapReduce Performance Model

Each MapReduce job consists of a specified number of map and reduce tasks. The job execution time depends on the amount of resources (represented by map and reduce slots) allocated to the job. The amount of allocated resources may drastically impact the job progress over time and its duration. While there could be different possible job executions (due to a non-determinism in tasks' execution or due to a different amount of allocated map and reduce slots for a job execution), these job executions all consist of executing the same map and reduce tasks and they process the same amount of data.

Figures 1 and 2 visualize two different executions of a MapReduce job that is allocated different amount of resources. In this example, we consider *WikiTrends* application. It analyzes Wikipedia article traffic logs that were collected (and compressed) every hour. *WikiTrends* counts the number of times each article has been visited in the given input dataset, i.e., the article access frequency (popularity count) over time. The example's input dataset has 71 files that correspond to 71 map tasks, and the application is defined with 64 reduce tasks.

First, we execute WikiTrends with 16 map and 16 reduce slots. Figure 1 visualizes the WikiTrends execution with allocated 16 map and 16 reduce slots. Therefore, the job execution has 5 map waves ($\lceil 71/16 \rceil$ and they comprise the map stage) and 4 reduce waves ($\lceil 64/16 \rceil$ that constitute the reduce stage). The first shuffle (i.e., the shuffle phase of the first reduce wave) overlaps with a significant portion of the map stage as we can see from this figure. At the same time, there is a strict barrier between map and reduce task processing: a reduce task (a reduce function) execution may only start when all map tasks are completed and the intermediate data has been shuffled to the reducer. In other words, there is a strict barrier between map and reduce stage processing of the same job.

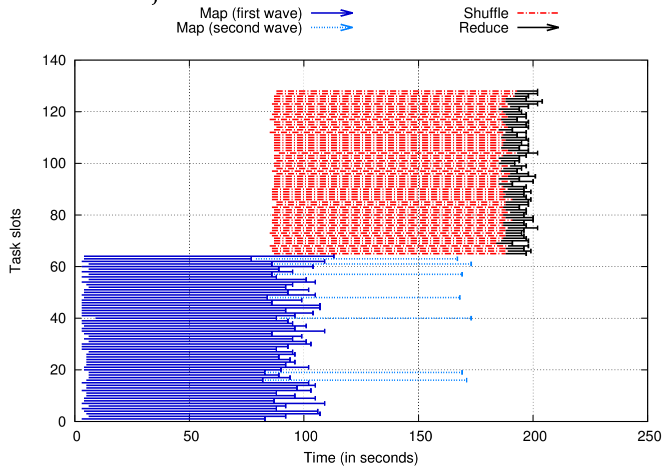


Fig. 2. WikiTrends with 64 map and 64 reduce slots.

Now, we execute WikiTrends with 64 map and 64 reduce slots allocated to the application. This job execution consists of two map waves ($\lceil 71/64 \rceil$) and a single reduce wave as shown in Figure 2. The second map wave processes only 7 map tasks. The shuffle phase of the reduce stage can be completed only when all map tasks are done (overlapping with both preceding map waves).

For the purpose of this work, we define the job execution time as **the sum of the complementary, non-overlapping map and reduce stage execution times**, i.e., we represent the duration of the first shuffle by using a fraction of its non-overlapping time with the duration of the map stage.

In this work, we apply the model that was introduced in ARIA [11]. ARIA offers a MapReduce performance model that is based on *theoretical performance bounds* described as follows. Let a job J be represented as a set of n tasks processed by k servers (or by k slots in MapReduce environments). Let $T_1, T_2, \ldots, T_n$ be the duration of n tasks of a given job. The assignment of tasks to slots is done using a simple, online, *greedy* algorithm, i.e., assign each task to the slot with the earliest finishing time.

Let $avg = (\sum_{i=1}^{n} T_i)/n$ and $max = \max_i \{T_i\}$ be the *average* and *maximum duration* of the n tasks respectively. Then the completion time of a greedy task assignment is proven to be at least:

$$T^{low} = \frac{n \cdot avg}{k}$$

and at most

$$T^{up} = \frac{(n-1) \cdot avg}{k} + max.$$

The lower bound is trivial, as the best case is when all n tasks are equally distributed among the k slots (or the overall amount of work $n \cdot avg$ is processed as fast as possible by k slots). Thus, the overall makespan is at least $n \cdot avg/k$.

For the upper bound, let us consider the worst case scenario, i.e., the longest task $\hat{T} \in \{T_1, T_2, \ldots, T_n\}$ with duration max is the last processed task. In this case, the time elapsed before the final task $\hat{T}$ is scheduled is at most the following: $(\sum_{i=1}^{n-1} T_i)/k \leq (n-1) \cdot avg/k$. Thus, the makespan of the overall assignment is at most $(n-1) \cdot avg/k + max$.[3]

The difference between lower and upper bounds represents the range of possible job completion times due to non-determinism and scheduling. As motivated by described theoretical bounds, in order to approximate the overall completion time of a MapReduce job, we need to estimate the *average* and *maximum* task durations for different execution phases of the job, i.e., map, shuffle/sort, and reduce phases. We use the fact that production jobs are run periodically, and from the past run we automatically build a compact job profile that reflects performance characteristics of the underlying application during all execution phases.

Let us consider job J that is partitioned into N_M^J map tasks and N_R^J reduce tasks. Let J be already executed in a given Hadoop cluster. Below, we explain both our profiling approach and the proposed MapReduce performance model for estimating the job completion time as a function of allocated resources. Let S_M^J and S_R^J be the number of map and reduce slots allocated to the **future** execution of job J.

The **map stage** consists of a number of map tasks. If the number of tasks is greater than the number of slots, the task assignment proceeds in multiple rounds, which we call *waves*. From the distribution of the map task durations of the past run, we compute the average duration M_{avg} and the maximum duration M_{max}. Then the lower and upper bounds on the duration of the entire map stage in the future execution with S_M^J map slots (denoted as T_M^{low} and T_M^{up} respectively) are estimated as follows:

$$T_M^{low} = \frac{N_M^J \cdot M_{avg}}{S_M^J}$$

$$T_M^{up} = \frac{(N_M^J - 1) \cdot M_{avg}}{S_M^J} + M_{max}$$

3. Similar ideas were explored in the classic papers on scheduling, e.g., to characterize makespan bounds [12].

The **reduce stage** consists of the *shuffle* and *reduce* phases, and their execution time bounds can be computed similarly.

The *shuffle phase* begins only after the first map task has completed. The shuffle phase completes when the entire map stage is complete and all the intermediate data generated by the map tasks has been shuffled to the reduce tasks and has been sorted. The shuffle phase of the *first* reduce wave may be significantly different from the shuffle phase that belongs to the next reduce waves. This happens because the shuffle phase of the first reduce wave overlaps with the entire map stage, and hence its depends on the number of map waves and their durations. Therefore, from the past execution, we extract two sets of measurements: (Sh_{avg}^1, Sh_{max}^1) for shuffle phase of the first reduce wave (called, *first shuffle*) and $(Sh_{avg}^{typ}, Sh_{max}^{typ})$ for shuffle phase of the other waves (called, *typical shuffle*). Moreover, we characterize a first shuffle in a special way and include only the non-overlapping portion (with map stage) in our metrics: Sh_{avg}^1 and Sh_{max}^1. This way, we carefully estimate the latency portion that contributes explicitly to the job completion time. The *typical shuffle* phase is computed as follows:

$$T_{Sh}^{low} = \left(N_R^J / S_R^J - 1\right) \cdot Sh_{avg}^{typ}$$

$$T_{Sh}^{up} = \left((N_R^J - 1)/S_R^J - 1\right) \cdot Sh_{avg}^{typ} + Sh_{max}^{typ}$$

The *reduce phase* begins only after the shuffle phase is complete. From the distribution of the reduce task durations of the past run, we compute the *average* and *maximum* metrics: R_{avg} and R_{max} that are used to compute the lower and upper bounds of completion times of the reduce phase.

Finally, we can put together the lower and upper bounds of the entire reduce stage (T_R^{low}, T_R^{up}) by summing up durations of shuffle and reduce phases. We also define the *average of lower and upper bounds* as follows:

$$T_R^{low} = Sh_{avg}^1 + T_{Sh}^{low} + (N_R^J \cdot R_{avg}/S_R^J)$$

$$T_R^{up} = Sh_{max}^1 + T_{Sh}^{up} + ((N_R^J - 1) \cdot R_{avg}/S_M^J + R_{max})$$

$$T_M^{avg} = (T_M^{low} + T_M^{up})/2 \quad \text{and} \quad T_R^{avg} = (T_R^{low} + T_R^{up})/2.$$

It was shown in [11], [13] that the extracted *average* of task' durations is quite stable over time and that the *average of lower and upper bounds* is a good approximation of the stage completion time: it is within 10% of the measured durations. For the rest of the paper, we use T_M^{avg} and T_R^{avg} as the estimates of the map and reduce stage execution time.

3 Optimized Batch Scheduling

In this section, we discuss the problem of increasing the cluster utilization via minimizing the overall completion time for a given set of MapReduce jobs. We present a simple but effective *abstraction* of the MapReduce job execution that enables us to apply the classic Johnson algorithm for building an optimized job schedule. Then we discuss possible inefficiencies of this abstraction and a novel heuristic as an alternative solution.

3.1 Problem Definition

Each MapReduce job consists of a specified number of map and reduce tasks. The job execution time and specifics of the execution depend on the amount of resources (map and reduce slots) allocated to the job. Section 2.2 presents an example of Wikitrends application processing. Figure 1 shows a detailed visualization of how 71 map and 64 reduce tasks of this application are processed in the Hadoop cluster with 16 map and 16 reduce slots. Instead of the detailed job execution at the task level, we introduce a simple **abstraction**, where each MapReduce job J_i is defined by durations of its map and reduce stages m_i and r_i, i.e., $J_i = (m_i, r_i)$. Section 2.2 presents our profiling approach and performance model for computing the estimates of average map and reduce stage durations when the job is executed on a new dataset. This model is applied to derive the proposed new abstraction $J_i = (m_i, r_i)$.

Let us consider the execution of two (independent) MapReduce jobs J_1 and J_2 in a Hadoop cluster with a **FIFO scheduler**. There are no data dependencies between these jobs. Therefore, once the first job completes its map stage and begins reduce stage processing, the next job can start its map stage execution with the released map resources in a pipelined fashion (see Figure 3). There is an “overlap” in executions of map stage of the next job and the reduce stage of the previous one.

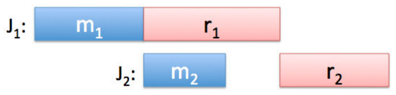


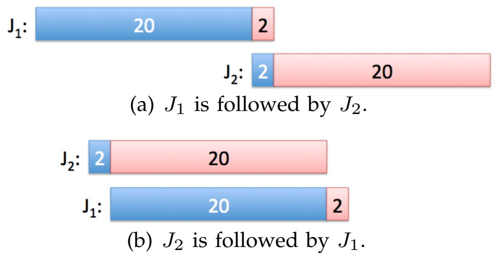
Fig. 3. Pipelined execution of two jobs J_1 and J_2.

We note that some execution orders of such jobs may lead to a significantly less efficient resource usage and an increased processing time. As a motivating example, let us consider two independent MapReduce jobs that utilize all the given Hadoop cluster's resources and that result in the following map and reduce stage durations: $J_1 = (20s, 2s)$ and $J_2 = (2s, 20s)$. In the Hadoop cluster with the FIFO scheduler, they can be processed in two possible ways:

- Job J_1 is followed by job J_2 (as shown in Figure 4 (a)). The reduce stage of J_1 overlaps with the map stage of J_2 leading to overlap of only $2s$. Thus, the total completion time of processing two jobs is $20s + 2s + 20s = 42s$.
- Job J_2 is followed by job J_1 (as shown in Figure 4 (b)). The reduce stage of J_2 overlaps with the map stage of J_1 leading to a much better pipelined execution and a much larger overlap of 20s. Thus, the total makespan is $2s + 20s + 2s = 24s$.

Thus, there can be a significant difference in the overall job completion time (75% in the example above) depending on the execution order of the jobs.

Let $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ be a set of n MapReduce jobs with no data dependencies between them. We consider **the following problem**: determine an order of execution (a schedule) of jobs $J_i \in \mathcal{J}$ such that the makespan (completion time) of the entire set is minimized.



(a) J_1 is followed by J_2.

(b) J_2 is followed by J_1.

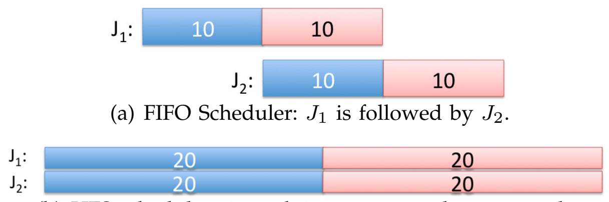
Fig. 4. Impact of different job schedules on the completion time.

Note, that once the job schedule is produced, we execute it with the Hadoop FIFO scheduler.

Remark: A reasonable question to ask is why do we choose to determine the jobs' order for execution with the Hadoop FIFO scheduler? What would happen if one would try to execute a given set of jobs (concurrently) using the Hadoop Fair Scheduler (HFS) [14]? In case of multiple ready jobs, HFS starts executing map stages of these jobs **concurrently** by allocating a fair fraction of cluster resources to each job. Therefore, HFS does not optimize jobs pipelining to improve the overlap of map and reduce stage executions among different jobs as described above. Below, we provide a simple example with two jobs to demonstrate that executing these jobs with HFS results in a suboptimal job execution.

Let us consider two jobs J_1 and J_2 with similar stage execution profiles, i.e., $J_1 = J_2 = (10, 10)$ on the Hadoop cluster with 30x30 map and reduce slots respectively. We assume that both jobs can utilize all 30x30 map and reduce slots available in the cluster.

Under FIFO scheduler two possible job sequences are: *i)* J_1 is followed by J_2 and *ii)* J_2 is followed by J_1. Both of these executions result in the same overall makespan of 30 time units as shown in Figure 5 (a).



(a) FIFO Scheduler: J_1 is followed by J_2.

(b) HFS scheduler: J_1 and J_2 are executed concurrently.

Fig. 5. Jobs' completion time: FIFO vs HFS schedulers.

Figure 5 (b) shows executing a given set of jobs with Hadoop Fair scheduler. We use the same color scheme for map and reduce stages, the height of the stages reflects the amount of resources used by the jobs, and the width (length) represents the stage duration. Under HFS each job gets a half of the cluster slots: 15x15 map and reduce slots respectively. Therefore, both map and reduce stages of each job take two times longer to complete: $J_1 = J_2 = (20, 20)$. It means that each job's completion time is 40 time units, and the overall makespan is 40 time units compared to 30 time units under FIFO scheduler. Therefore, the Fair scheduler results in the job execution with a suboptimal makespan compared to a specially constructed job schedule under the FIFO scheduler.

3.2 Johnson's Algorithm

In 1953, Johnson [9] proposed an optimal algorithm for two stage production schedule. In the original problem formulation, a set of production items and two machines (S_1 and S_2) are given. Each item must pass through stage one that is served by machine S_1, and then stage two that is served by machine S_2. Each machine can handle only one item at a time. The production item i in the set is represented by two positive numbers (s_i^1, s_i^2) that define service times for the item to pass through stages one and two respectively[4].

There is a striking similarity between the problem formulation described above and the problem that we would like to solve: building a schedule that minimizes the makespan of a given set of MapReduce jobs. We can represent each MapReduce job J_i in our batch set $\mathcal{J}$ by a pair of computed durations (m_i, r_i) of its map and reduce stages, and these stage durations fairly define the "busy" processing times by the map and reduce slots respectively. This abstraction enable us to apply Johnson's algorithm (offered for building the optimal two-stage jobs' schedule) to our scheduling problem for a set of MapReduce jobs.

Now, we explain the essence of Johnson's algorithm in terms of MapReduce jobs.

Let us consider a collection $\mathcal{J}$ of n jobs, where each job J_i is represented by the pair (m_i, r_i) of map and reduce stage durations respectively. Let us augment each job $J_i = (m_i, r_i)$ with an attribute D_i defined as follows:

$$D_i = \begin{cases} (m_i, \texttt{m}) & \text{if } \min(m_i, r_i) = m_i, \\ (r_i, \texttt{r}) & \text{otherwise.} \end{cases}$$

The first argument in D_i is called the *stage duration* and denoted as D_i^1. The second argument is called the *stage type* (map or reduce) and denoted as D_i^2.

Algorithm 1 shows how an optimal schedule can be constructed using Johnson's algorithm. First, we sort all the n jobs from the original set $\mathcal{J}$ in the ordered list L in such a way that job J_i precedes job J_{i+1} if and only if $min(m_i, r_i) \leq min(m_{i+1}, r_{i+1})$. In other words, we sort in descending order the jobs using the stage duration attribute D_i^1 in D_i (it represents the smallest duration of the two stages). Then the algorithm works by taking jobs from list L and placing them into the schedule σ from the both ends (head and tail) and proceeding towards the middle. If the stage type in D_i is m, i.e., represents the map stage, then the job J_i is placed from the head of the schedule, otherwise from the tail. The complexity of Johnson's Algorithm is dominated by the sorting operation and thus is $\mathcal{O}(n \log n)$.

Algorithm 1 Johnson's Algorithm

Input: A set $\mathcal{J}$ of n MapReduce jobs. D_i is the attribute of job J_i as defined above.
Output: Schedule σ (order of jobs execution.)

```
1:  Sort in descending order the original set $\mathcal{J}$ of jobs into the
    ordered list $L$ using their stage duration attribute $D_i^1$
2:  $head \leftarrow 1, tail \leftarrow n$
3:  for each job $J_i$ in $L$ do
4:      if $D_i^2$ = m then
5:          // Put job $J_i$ from the front
6:          $\sigma_{head} \leftarrow J_i$, head ← head + 1
7:      else
8:          // Put job $J_i$ from the end
9:          $\sigma_{tail} \leftarrow J_i$, tail ← tail - 1
10:     end if
11: end for
```

4. In fact, Johnson's schedule is also optimal for the case when $s_i^2 = 0$. This extension of the classic result is important because some of MapReduce jobs only have map stage (that corresponds to the execution on machine one) and do not have a reduce stage (that corresponds to the execution on machine two).

Let us illustrate the job schedule construction with Johnson's algorithm for a simple example with five MapReduce jobs defined in Figure 6. These jobs are augmented with additional computed attribute D_i shown in the last column. At first, this collection of jobs is

J_i	m_i	r_i	D_i
J_1	4	5	(4, m)
J_2	1	4	(1, m)
J_3	30	4	(4, r)
J_4	6	30	(6, m)
J_5	2	3	(2, m)

Fig. 6. Example of five MapReduce jobs.

J_i	m_i	r_i	D_i
J_2	1	4	(1, m)
J_5	2	3	(2, m)
J_1	4	5	(4, m)
J_3	30	4	(4, r)
J_4	6	30	(6, m)

Fig. 7. The ordered list L of five MapReduce jobs.

sorted into a list L according to the attribute D_i^1 (i.e., first argument of D_i). The sorted list of jobs is shown in Figure 7. Then we follow Johnson's algorithm and start placing the jobs in the schedule σ from both ends toward the middle, and construct the following schedule:

- J_2 is represented by D_2=(1, m). Since D_2^2 = m then J_2 goes to the head of σ, and $\sigma = (J_2, ...)$.
- J_5 is represented by D_5=(2, m). Again, J_5 goes to the head of σ, and $\sigma = (J_2, J_5, ...)$.
- J_1 is represented by D_1=(4, m), and it goes to the head of σ, and $\sigma = (J_2, J_5, J_1, ...)$.
- J_3 is represented by D_3=(4, r). Since D_3^2 = r then J_3 goes to the tail of σ, and $\sigma = (J_2, J_5, J_1, ..., J_3)$.
- J_4 is represented by D_4=(6, m) and it goes to the head of σ, and $\sigma = (J_2, J_5, J_1, J_4, J_3)$.

Job ordering $\sigma = (J_2, J_5, J_1, J_4, J_3)$ defines Johnson's schedule for the execution with the minimum overall makespan. For our example, the makespan of the optimal schedule is 47. We can re-write σ by showing job processing time details: σ=((1, 4), (2, 3), (4, 5), (6, 30), (30, 4)). This job schedule is shown in Figure 8.

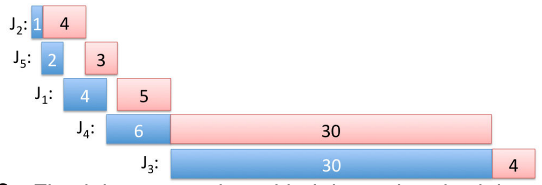


Fig. 8. Five jobs processing with Johnson's schedule.

Intuitively, for achieving a larger overlap between the jobs, the optimal schedule executes jobs in the increased map stage durations order (the subsequence J_2, J_5, J_1, J_4) and then from some point, in the decreased reduce stage durations order (the subsequence J_4, J_3).

The worst schedule is defined by the reverse order of the optimal one, i.e., $(J_3, J_4, J_1, J_5, J_2)$. The worst job schedule has a makespan of 78 (this is 66% increase in the makespan compared to the optimal time). Indeed, the optimal schedule may provide significant savings.

3.3 BalancedPools Heuristic Algorithm

While the simple abstraction for MapReduce jobs proposed in Section 3.2 enables us to apply the elegant Johnson algorithm for constructing the optimized job schedule, it raises the following questions about the abstraction:

- How well does this abstraction correspond to the reality of complex execution of MapReduce jobs?
- How accurate is the *computed* makespan of Johnson's schedule for estimating the *measured* makespan of a given set of MapReduce jobs?
- What are the situations where the generated Johnson schedule might lead to suboptimal results?

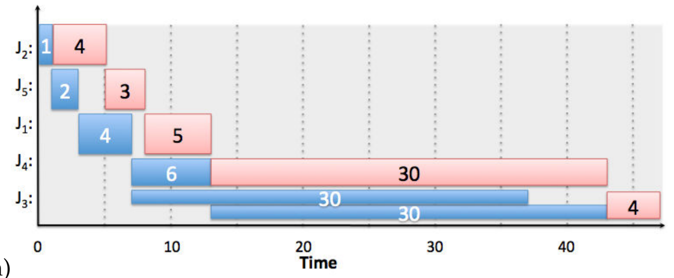
When a MapReduce job is represented as a pair of map and reduce stage durations, it obscures the number of tasks that comprise the job's map and reduce stages and the number of slots that process these tasks. For example, Figure 1 shows how 71 map and 64 reduce tasks of Wikitrends application are processed in the Hadoop cluster with 16 map and 16 reduce slots. Note, that the last, fifth wave of the map stage has for processing only 7 tasks (71-16x4). Thus, out of 16 available map slots only 7 slots are used by the current application and the remaining 9 map slots can be immediately used for processing of the next job. Therefore, processing of the next job's map stage may start before the previous job completes its map stage. As a result, while the job schedule defined by the Johnson algorithm might be still the optimal one, the makespan computed by the Johnson algorithm might be pessimistic compared to the real execution of the job schedule on the Hadoop cluster. To provide better estimates for the makespan of a given set of MapReduce jobs under different job schedules, we use the MapReduce simulator SimMR [10] that can faithfully replay MapReduce job traces at the tasks and slots level: the completion times of simulated jobs are within 5% of the original ones as was shown in [10].

Let us revisit MapReduce job processing and discuss situations where Johnson's schedule might provide a suboptimal solution. Consider the set of five jobs shown in Figure 6-8 (see Section 3.2). Below we describe *two different scenarios* that, in spite of their differences, lead to the same job profiles and stage durations as shown in Figure 6. Therefore, if we apply Johnson's algorithm, it will produce the same job schedule $\sigma = (J_2, J_5, J_1, J_4, J_3)$ for minimizing the makespan of this set. In both scenarios, we consider a Hadoop cluster with 30 worker nodes, each configured with a single map and single reduce slot, i.e., with 30 map and 30 reduce slots overall.

Scenario1: Let each job in the set be comprised of 30 map and 30 reduce tasks. Thus, each job utilizes either all map or all reduce slots during its processing. In this scenario, there is a perfect match between the assumptions of the classic Johnson algorithm for two-stage production system and MapReduce job processing.

Scenario2: Let jobs J_1, J_2, and J_5 be comprised of 30 map and 30 reduce tasks, and jobs J_3 and J_4 consist of 20 map and 20 reduce tasks. Figure 9(a) visualizes the execution of these five MapReduce jobs according to the generated Johnson schedule $\sigma = (J_2, J_5, J_1, J_4, J_3)$.

We use a different color scheme for map (blue/dark) and reduce (red/light) stages, the height of the stages reflects the amount of resources used by the jobs, the width (length) represents the stage duration, the jobs appear at the time line as they are processed by the schedule.



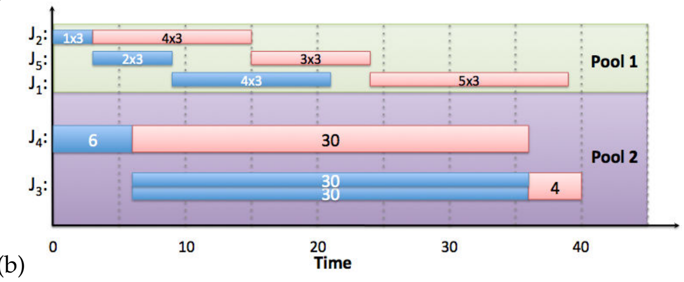


Fig. 9. Example with five MapReduce jobs: (a) job processing with Johnson's schedule; (b) an alternative with BalancedPools.

While the first three jobs J_2, J_5, and J_1 utilize all map and all reduce slots during their processing, the last two jobs J_4 and J_3 only use 20 map and 20 reduce slots, and hence map stage processing of J_3 starts earlier than the map stage of J_4 is completed because there are 10 map slots available in the system. The first 10 tasks of J_3 are processed concurrently with 20 map tasks of J_4. When J_4 completes its map stage and releases 20 map slots, then the next 10 map tasks of J_3 get processed. However, this slightly modified execution leads to the same makespan of 47 time units as under *Scenario1* because processing of J_3's reduce stage can only start when the entire map stage of J_3 is finished.

We claim that Johnson's schedule for *Scenario2* described above is suboptimal, by outlining a better solution. Let us partition these five jobs into two pools with the following amount of resources allocated to each pool:

1) *Pool1* with **10** map and **10** reduce slots and jobs J_1, J_2, and J_5;
2) *Pool2* with **20** map and **20** reduce slots and J_3, J_4.

First of all, a different amount of resources allocated to jobs in *Pool1* changes these jobs' map and reduce stage durations. Each of these jobs has 30 map and 30 reduce tasks. When processing 30 tasks with 10 slots, their execution takes three times longer[5]: both map and reduce stages are processed in three waves, compared with a single wave for the stage execution with 30 slots. Then for jobs in each pool, we apply Johnson's algorithm to generate the optimized schedules:

1) *Pool1* is processed according to $\sigma_1 = (J_2, J_5, J_1)$. This schedule has a makespan of 39 time units;
2) *Pool2* is executed according to $\sigma_2 = (J_4, J_3)$. This schedule results in the makespan of 40 time units.

Figure 9(b) visualizes these job executions. Jobs in *Pool1* and *Pool2* are processed concurrently (each set follows its own schedule). The cluster resources are partitioned between the two pools in a tailored manner. Using this approach, the overall makespan for processing these five jobs is 40 time units, that is almost 20% improvements compared to 47 time units using Johnson's schedule.

This example and the proposed solution exploits additional properties specific to MapReduce environments and the execution of MapReduce jobs. In particular, the job stage durations closely depend on the amount of allocated resources (map and reduce slots). In this way, we can change the jobs' *appearance* (see Figures 1, 2 that visualize two different executions of Wikitrends with different amount of resources). The main objective function of such an algorithm is to partition the jobs into two pools with specially tailored resource allocations such that the makespan of jobs in these pools are balanced, and the overall completion time of jobs in both pools is minimized. In general, the problem of balancing the map and reduce tasks in slots to achieve the minimum makespan for a set of MapReduce jobs is NP-hard. This can be easily proved by a simple polynomial reduction from the 3PARTITION problem [15]. We design a heuristic called the *BalancedPools* algorithm. The pseudo-code of the algorithm is shown below. As shown in Algorithm 2, we iteratively partition the jobs into two pools and then try to identify the adequate resource allocations for each pool such that the makespans of these pools are balanced. Within each pool we apply Johnson's algorithm for job scheduling, where map and reduce stage durations are computed with the performance model described in Section 2.2. The pool makespan is estimated (accurately within 5%) with MapReduce simulator SimMR [10] as a part of the algorithm. The use of the simulator in the solution is absolutely necessary and justified. As we demonstrated, the makespan computation that follows Johnson's schedule and its simple abstraction may result in a significant inaccuracy, and more accurate estimates might be obtained only via MapReduce simulations at the task/slot level. The complexity of the algorithm is $\mathcal{O}(n^2 \log n \log M)$. However, SimMR can simulate a 1000 job workload on a 100 node Hadoop cluster in less than 2 seconds. The designed algorithm can be extended to a larger number of pools (at a cost of a significantly higher complexity). We plan to investigate performance benefits versus algorithm computation costs in our future work. Since a job distribution is typically bimodal (i.e., a collection of "small" and "large" jobs as shown in studies [16], [17]) we use two pools. Note, that we assume a homogeneous Hadoop cluster. The job execution with two pools is implemented using Capacity scheduler [18] that allows resource partitioning into different pools with a separate job queue for each pool.

Algorithm 2 BalancedPools Algorithm

Input: 1) List J of n MapReduce jobs.
2) M: Number of machines in the cluster.
Output: Optimized Makespan

```
1: Sort $J$ based on increasing number of map tasks
2: BestMakespan ← SIMULATE($J$, JOHNSONORDER($J$), M)
3: for split ← 1 to $n-1$ do
4:     // Partition $J$ into list of small Jobs$_\alpha$ and big Jobs$_\beta$
5:     Jobs$_\alpha$ ← $(J_1, \cdots, J_{split})$
6:     Jobs$_\beta$ ← $(J_{split+1}, \cdots, J_n)$
7:     SizeBegin ← 1, SizeEnd ← $M$
8:     // Binary search for the pool size that balances completion times of both pools
9:     repeat
10:        SizeMid ← (SizeBegin + SizeEnd)/2
11:        Makespan$_\alpha$ ← SIMULATE(Jobs$_\alpha$,JOHNSONORDER(Jobs$_\alpha$),SizeMid)
12:        Makespan$_\beta$ ← SIMULATE(Jobs$_\beta$,JOHNSONORDER(Jobs$_\beta$),M-SizeMid)
13:        if Makespan$_\alpha$ < Makespan$_\beta$ then
14:            SizeEnd ← SizeMid
15:        else
16:            SizeBegin ← SizeMid
17:        end if
18:    until SizeBegin = SizeEnd
19:    Makespan ← MAX(Makespan$_\alpha$, Makespan$_\beta$)
20:    if Makespan < BestMakespan then
21:        BestMakespan ← Makespan
22:    end if
23: end for
```

4 EVALUATION

This section evaluates the benefits of Johnson's schedule and the novel *Balanced Pools* algorithm for minimizing the makespan of a set of MapReduce jobs using a variety of synthetic and realistic workloads derived from the Yahoo! M45 cluster and Facebook workloads. First, we compare different schedules using simulations. Then, we validate the simulation results by performing similar experiments in a 66-node Hadoop cluster.

4.1 Workloads

We use the following workloads in our experiments:

1) Yahoo! M45: This workload represents a mix of 100 MapReduce jobs that is based on the analysis performed on the Yahoo! M45 cluster [19], and is generated as follows:

- Each job consists of the number of map and reduce tasks drawn from the distribution $\mathcal{N}(154, 558)$ and

5. For simplicity of presentation in the corresponding figures, we scale the duration of the job by 3 times when the amounts of allocated resources are 3 times smaller. In fact, we can accurately compute the increase in the job completion time using the extracted job profiles and the performance model outlined in Section 2.)

$\mathcal{N}(19, 145)$ respectively, where $\mathcal{N}(\mu, \sigma)$ is the normal distribution with mean μ and standard deviation σ.

- Map and reduce task durations are defined by $\mathcal{N}(50, 200)$ and $\mathcal{N}(100, 300)$ respectively[6].
- To avoid that map (reduce) stage durations look similar for different jobs (since they are drawn from the same distribution), an additional *scale factor* is applied to map (reduce) task durations of each job.

To perform a sensitivity analysis, we have created two job sets (100 jobs each) based on Yahoo! M45 workload:

1) *Unimodal* set that uses a single scale factor for the overall workload, i.e., the scale factor for each job is drawn uniformly from $[1, 10]$.
2) *Bimodal* set where a subset of jobs (80%) are scaled using a factor uniformly distributed between $[1, 2]$ and the remaining jobs (20%) are scaled using $[8, 10]$. This mimics workloads that have a large fraction of *short* jobs and a small subset of *long* jobs.

2) Facebook: This workload is based on the detailed description [16] of MapReduce job sizes and their distribution in production at Facebook in October 2009. In order to simulate and execute this workload on the Hadoop cluster, we follow the approach suggested by Zaharia et. al. [16] for creating a representative and executable workload. We use four job types defined by the Hive benchmark [3] that represent typical operations performed by Hadoop clusters:

1) *Text search:* It finds a certain pattern in an input dataset.
```
SELECT * FROM grep WHERE field LIKE
'%XYZ%'
```
2) *Select:* It selects pages with a certain pageRank.
```
SELECT pageRank, pageURL FROM rankings
WHERE pageRank > 5
```
3) *Aggregation:* It computes Ad revenue for each IP address in a dataset.
```
SELECT sourceIP, SUM(adRevenue) FROM
uservisits GROUP BY sourceIP
```
4) *Join:* It joins the page rank and user visits table based on the page URL.
```
SELECT INTO Temp sourceIP,
   AVG(pageRank) AS avgPageRank,
   SUM(adRevenue) AS totalRevenue
FROM rankings AS R, userVisits AS UV
   WHERE R.pageURL = UV.destURL
```

These jobs process three datasets `grep`, `rankings` and `uservisits`. We generate 100 jobs according to the Facebook job distribution shown in Table 1 and by creating datasets with correct sizes.

The input sizes are defined by the number of map tasks per job (i.e., the number of map tasks multiplied by 64 MB block). We observe that there is a large number (68%) of small (≤10 map tasks) jobs and a few jobs (4%) that are very large (≥2400 map tasks).

6. The study [19] did not report statistics of individual task durations. We use a greater range for reduce tasks since they combine shuffle, sort, reduce phase processing, and time for writing three data copies back to HDFS.

Bin	Job Type	Map Tasks	Reduce Tasks	# Jobs Run
1	Select	1	NA	38
2	Text search	2	NA	16
3	Aggregation	10	3	14
4	Select	50	NA	8
5	Text search	100	NA	6
6	Aggregation	200	50	6
7	Select	400	NA	4
8	Aggregation	800	180	4
9	Join	2400	360	2
10	Text search	4800	NA	2

TABLE 1
Job sizes in the Facebook workload (100 jobs, Table 3 in [16]).

3) Synthetic: Additionally, we create two synthetic workloads to perform a sensitivity analysis.

Synthetic1 consists of jobs having a number of map and reduce tasks drawn uniformly from $[1, 100]$ and $[1, 50]$ respectively. The map and reduce task durations are normally distributed using $\mathcal{N}(100, 1000)$ and $\mathcal{N}(200, 2000)$ respectively. We create two versions of *Synthetic1* workload: *i) Unimodal*, where each job is scaled using a factor uniformly distributed between $[1, 10]$ and *ii) Bimodal*, where 80% of the jobs are scaled using a factor uniformly distributed between $[1, 2]$ and the remaining 20% of jobs are scaled using $[8, 10]$.

Synthetic2 consists of jobs having a number of map and reduce tasks drawn uniformly from $[1, 100]$ and $[1, 50]$ respectively. The map and reduce task durations are normally distributed using $\mathcal{N}(20, 100)$ and $\mathcal{N}(50, 200)$ respectively. For *Synthetic2* workload we consider a *Bimodal* mode only: where 90% of the jobs are scaled using a factor uniformly distributed between $[1, 3]$ and the remaining 10% of jobs are scaled using $[8, 10]$.

We generate *Synthetic1* and *Synthetic2* workloads with 10, 20, and 100 jobs. We aim to verify whether the number of jobs might impact the schedule performance.

4.2 Simulation Results

In this section, we analyze the proposed job schedule algorithms and their performance using the simulation environment SimMR [10] that was designed for evaluation and analysis of different workload management strategies in MapReduce environments. SimMR can replay execution traces of real workloads collected in Hadoop clusters as well as generate and execute synthetic traces based on statistical properties of workloads. SimMR is comprised of three components: *i)* Trace Generator that creates a replayable MapReduce workload; *ii)* Simulator Engine that accurately emulates the job master functionality in Hadoop and job execution at task/slot level; and *iii)* a pluggable scheduling policy that dictates the scheduler decisions on job ordering and the amount of resources allocated to different jobs over time.

For all our experiments, we first generate a set of jobs (batch) according to given distributions of map and reduce tasks. This set of jobs defines the production workload of interest. The main goal is to process this given workload in a shortest time and to determine the jobs' execution order that minimizes the overall processing time. We simulate the given workload using clusters of different size.[7] Often system administrators need to adjust cluster resources for the batch execution to get the desirable makespan results. We plan to compare performance and the accuracy of the following strategies:

- *Min* strategy – it provides a *theoretical* makespan under Johnson's (optimal) schedule. In other words, if given MapReduce jobs hypothetically satisfy two-stage system assumptions (i.e., completely utilize cluster resources during both stages) then the overall makespan can be directly derived from the abstraction $J_i{=}(m_i, r_i)$. Stage durations (m_i, r_i) are computed with bounds-based ARIA's model as a function of job profile and cluster resources (see Section 2.2).
- *Max* strategy – it shows a *theoretical* makespan under reverse Johnson's (worst) schedule. Assumptions and computations are similar to *Min* strategy;
- *MinSim* – it reflects a *simulated* makespan under Johnson's (optimal) schedule. The generated job schedule is exactly the same as under *Min* strategy. The main difference is that the job execution under this schedule is carefully simulated using SimMR. The simulator devotedly imitates the Hadoop allocation decisions of tasks to slots in a given size cluster. It accurately replays each task execution using its actual time duration. It aims to reflect what would be real jobs' executions according to Johnson's schedule in a given cluster.
- *MaxSim* – it shows a *simulated* makespan under reverse Johnson's (worst) schedule. The execution conditions of this strategy are similar to *MinSim*.
- *BalancedPools*– it provides a *simulated* makespan of the new job schedule constructed with the *BalancedPools* heuristic. The new algorithm suggests partitioning cluster resources and jobs into two pools. Then Johnson's schedule is constructed for each subset of jobs and its resource pool. Finally, SimMR accurately simulates the job execution on the corresponding sub-cluster according to the generated job schedule, and reports the overall makespan for two pools.

Figure 10 shows the results for the *Synthetic1* workload with 100 jobs and *Unimodal* and *Bimodal* distributions. The Y-axis reflects the estimated makespan for different size Hadoop clusters on X-axis (without loss of generality, we assume 1 map and 1 reduce slot per node). The achievable jobs' makespan is a function of the cluster size. When available resources in the cluster are plentiful and different jobs might be executed concurrently

7. We consistently use the **same generated workload** for comparing the *relative* performance of different job schedules. Later, we show that schedule performance depends on both workload and cluster size. Even when we generate workloads using the same distributions, their makespans may be quite different for the same schedule strategies. Our goal is to compare the relative performance of different algorithms for optimizing the **same** workload. Experiments with 10, 20, and 100 jobs (drown from the same distribution) enhance the study with relative performance comparison of different schedule algorithms when processing the same workload type.

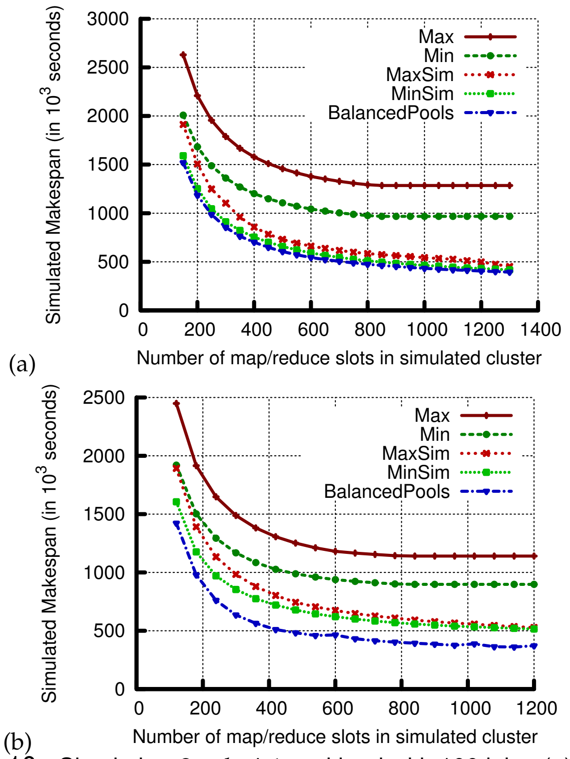


Fig. 10. Simulating *Synthetic1* workload with 100 jobs: (a) *Unimodal* and (b) *Bimodal*.

instead of being sequential the benefits of the smart schedule are decreasing. However, for different workloads the points of diminishing return are different. This sensitivity analysis is especially useful for evaluating the required cluster size to support the specified (targeted) makespan for a set of given jobs.

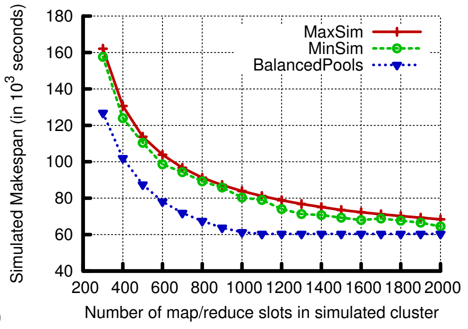
The graphs reflect five lines. Two top lines: *Min* and *Max* show theoretical makespans under Johnson's (optimal) schedule and reverse Johnson's (worst) schedule respectively. In these strategies, we represent a job as $J_i{=}(m_i, r_i)$ and have no information how many map (reduce) slots the job may utilize over time. Under this abstraction, even if the map (reduce) stage of the previous job does not utilize all the map (reduce) slots in the cluster – the next job's stage only "starts" when the previous job's stage completes. This may lead to significantly over-estimating the overall makespan.

MinSim and *MaxSim* show accurately simulated makespans with SimMR for a set of given MapReduce jobs under Johnson's schedule and reverse Johnson's schedule respectively. We should stress that once we consider MapReduce jobs at the tasks/slots level Johnson's schedule and reverse Johnson's schedule do not guarantee the optimal and worst makespan for this set of jobs. The difference between *MinSim* and *MaxSim* reflects a lower bound of potential benefits (since the "worst" makespan might be much worse than under *MaxSim*).

Finally, *BalancedPools* curve reflects the accurately simulated makespan of the job schedule constructed with the new *BalancedPools* heuristic.

Figure 10 confirms that the simplified abstraction $J_i{=}(m_i, r_i)$ and makespan computations that use it (i.e., *Min* and *Max*) are inaccurate for estimating the real makespan of MapReduce jobs (due to lack of tasks/slots information), and in the rest of the graphs we omit these lines. This comparison strongly justifies the introduction of the simulator SimMR in the new heuristic for accurate makespan estimates.

Figure 10(a) shows up to 25% of makespan decrease with Jonhson's schedule (*MinSim*) compared to *MaxSim* for *Unimodal* case. The *BalancedPools* schedule behaves similar to Johnson's in this case. However, results are very different for the *Bimodal* workload shown in Figure 10(b). The *BalancedPools* heuristic provides up to 38% of makespan improvements compared to Johnson's schedule (it is suboptimal for this workload). *BalancedPools* achieves significant additional makespan improvements compared to Johnson's algorithm for a variety of different cluster sizes.

(a)


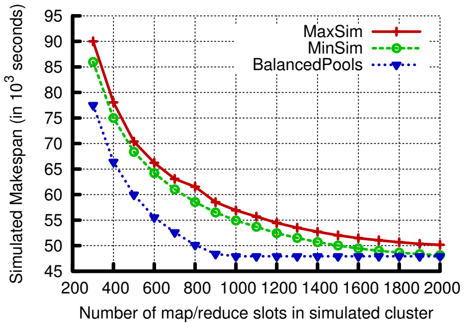
(b)


Fig. 11. Simulating *Synthetic1* workload (*Bimodal*) with (a) 20 jobs and (b) 10 jobs.

Figures 11 (a-b) show the results for the *Synthetic1* workload with *Bimodal* distribution when two smaller workloads with 20 and 10 jobs were generated to verify the impact of the workload size. While these workload sets are much smaller we can see a significant makespan reduction achieved under the new *BalancedPools* algorithm (25%-30% for many cluster size values).

Figures 12 (a-c) show the results for the *Synthetic2* workload with *Bimodal* distribution for three different size workloads with 100, 20 and 10 jobs respectively. *Synthetic2* workload has a smaller fraction of larger jobs (10% compared to 90% of shorter jobs) and smaller variation in map and reduce tasks durations compared to *Synthetic1* workload. While *Synthetic1* and *Synthetic2* workloads are quite different the new *BalancedPools* heuristic is able to take advantage of workload properties in each case and automatically generate the job schedule and corresponding resource allocations that offer optimized job executions. There is a significant difference in the overall job completion times under the non-optimized job schedule and a new *BalancedPools* algorithm. There could be up to 60% completion time increase compared to the optimized makespan. Figures 12 (b-c) show the results for two smaller workloads with 20 and 10 jobs respectively. Even for much smaller workload sets we can still see up to 25%-40% makespan improvements under the new job schedule.

While apparently the results are workload and cluster size dependent, we observe consistent significant improvements under the new *BalancedPools* algorithm.

Figure 13 shows results of simulating the Yahoo! M45 workload (*Unimodal* and *Bimodal* types). Interestingly, the classic Johnson schedule provides diminishing returns in both cases for Yahoo!'s workload. We can see only up to 12% of makespan improvements for most experiments. Again, the proposed *BalancedPools* heuristic significantly outperforms Johnson's algorithm: by 10%-30% in most cases. It offers the overall makespan improvements up to 38% for the *Bimodal* Yahoo! M45 workload as shown in Figure 13(b).

Figure 13(c) shows results for the Facebook workload. The Facebook workload has a large number (68%) of short jobs and a few jobs (4%) that are very large. This makes the Facebook workload resemble the *Bimodal* workloads that we analyzed earlier. The simulation results stress this similarity.

Also, the Facebook workload has 74% jobs with no reduce tasks, which are clubbed together at the end of the schedule by the Johnson's algorithm. Hence, there is relatively lesser (2%) difference between MinSim and MaxSim as the makespan is dominated by these jobs. However, since Facebook workload resembles the *Bimodal* workload the *BalancedPool* algorithm again provides significant performance improvements: it decreases the makespan by 14% for smaller cluster sizes and shows up to 33% of makespan improvements for larger cluster sizes shown in Figure 13(c).

Performance benefits under Johnson's algorithm and the *BalancedPools* heuristic are clearly workload and cluster size dependent. The proposed framework automatically constructs the optimized job schedule and provides the estimates of its makespan as a function of allocated resources.

4.3 Testbed Results

We have implemented the *BalancedPools* heuristic using Capacity scheduler [18] that enables cluster resource partitioning into different pools with a separate job queue for each pool. We validate the simulation results through experiments on a 66-node Hadoop cluster with the following configuration. Each node is an HP DL145 G3 machine with four AMD 2.39GHz cores, 8 GB RAM and two 160GB 7.2K rpm SATA hard disks. The machines are set up in two racks and interconnected with gigabit Ethernet. We use Hadoop 0.20.2 with two machines for JobTracker and NameNode, and remaining

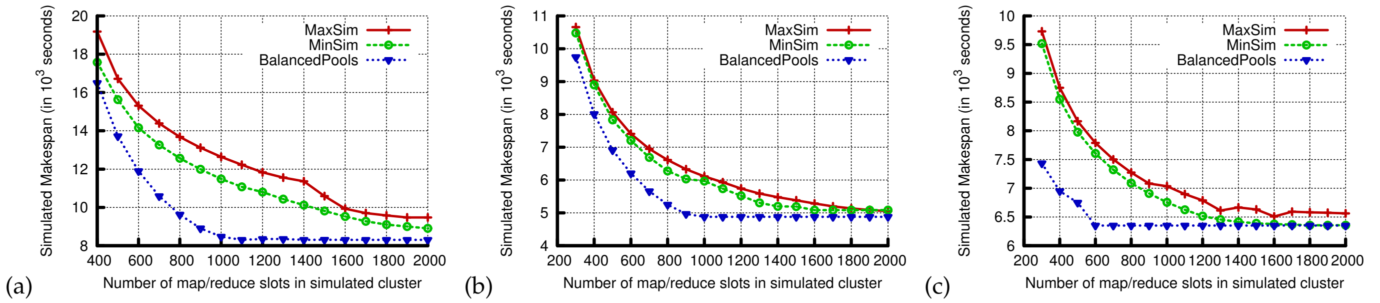


Fig. 12. Simulating *Synthetic2* workload (*Bimodal*) with (a) 100 jobs, (b) 20 jobs, and (c) 10 jobs.

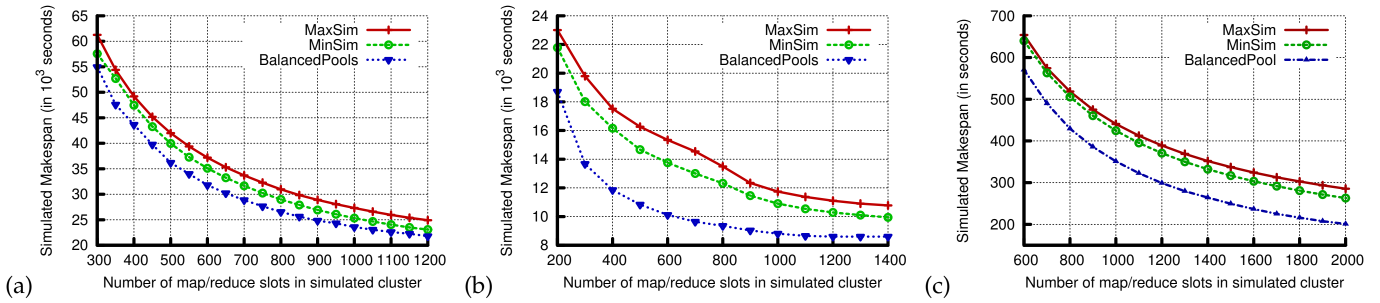


Fig. 13. (a) Simulating Yahoo!'s *Unimodal* workload, (b) Simulating Yahoo!'s *Bimodal* workload, (c) Simulating Facebook's workload,

64 machines as worker nodes. Each slave is configured with four map and four reduce slots, i.e. our cluster has 256 map and 256 reduce slots overall. The default HDFS blocksize is 64MB and the replication level is set to 3. We disable speculation as it did not lead to any significant improvements.

Figure 14 shows three groups of results. We executed the Facebook workload in our testbed with three different job schedules: 1) Johnson' schedule denoted as *Johnson*, 2) reverse Johnson' schedule denoted as *R-Johnson*, and 3) *BalancedPools* schedule. The dark error bars show the variation in measured times across 5 trials. Then we simulated these job schedules using the same cluster configuration in SimMR (i.e., with 256 map and 256 reduce slots in the Hadoop cluster).

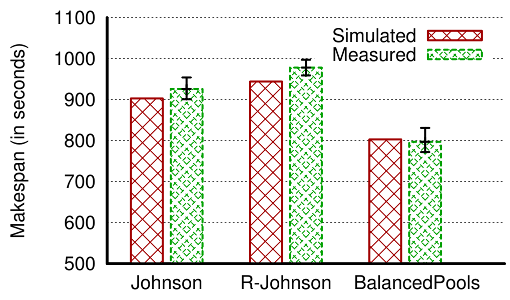


Fig. 14. Facebook workload: simulated vs. executed in the testbed.

The *BalancedPools* algorithm suggests splitting the jobs from the Facebook workload described in Table 1 in the following two pools:

- *Pool_1*: the smallest 68 jobs (*bins 1, 2* and *3*) into a small pool of *3 map* and *3 reduce slots*, and
- *Pool_2*: the largest 32 jobs (*bins 4 through 10*) into a large pool of *253 map* and *253 reduce slots*.

We enforce these pools by creating two queues using *Capacity* scheduler and submitted these jobs independently and concurrently to these two resource pools. The results are shown in Figure 14. The errors between the simulated and the measured makespans are less than 4%. The *BalancedPools* schedule reduces the makespan by 13% compared to Johnson's schedule, and by 18% compared to the reverse Johnson's schedule.

These results closely follow the simulation results shown in Figure 13(c). This validates of our general projections for performance benefits of Johnson's and *BalancedPools* schedules.

5 Related Work

Scheduling of incoming jobs and the assignment of processors to the scheduled jobs has been an important factor for optimizing the performance of parallel and distributed systems. It has been studied extensively in scheduling theory (see a variety of papers and textbooks on the topic [20], [21], [22], [23], [24], [25], [26], [27]). Designing an efficient distributed server system often assumes choosing the "best" task assignment policy for a given model and user requirements. However, the question of "best" job scheduling or task assignment policy is still open for many models. Typically, the choice of the algorithm is driven by performance objectives. If the performance goal is to minimize mean response time, then the optimal algorithm is to schedule the shortest job first. However, if there is a requirement of fairness in jobs' processing, then *processor-sharing* or *round-robin* scheduling might be preferable. For minimizing the *makespan* a promising approach is to schedule the longest job first ([12]). There is a whole body of scheduling research which focuses on minimizing makespan for jobs with precedence constraints ([28], [21]).

For large-scale heterogeneous distributed systems such as the Grid, job scheduling is one of the main component of resource management. Most work in the Grid-related job scheduling space aims to empirically evaluate scheduling heuristics: backfilling [29], adaptive scheduling [24], and task grouping [27].

Job scheduling and workload management in MapReduce environments is a new topic, but it has already received much attention. Originally, Hadoop was designed for periodically running large batch workloads with a *FIFO* scheduler. As the number of users sharing the same MapReduce cluster increased, a new *Capacity* scheduler [18] was introduced to support more efficient cluster sharing. Capacity scheduler partitions the resources into pools and provides separate queues and priorities for each pool. In order to maintain fairness between different users, the *Hadoop Fair Scheduler* (HFS) [16] was proposed.

However, while both HFS and Capacity scheduler allow sharing of the cluster among multiple users and their applications, these schedulers do not provide any special support for achieving the application performance goals and the service level objectives for a set of jobs like minimizing their makespan.

There are a few research prototypes of Hadoop schedulers that aim to optimize explicitly some given scheduling metric, e.g. FLEX [30] and ARIA [11]. FLEX extends HFS by proposing a special slot allocation schema that aims to optimize explicitly some given scheduling metric. FLEX relies on the speedup function of the job (for map and reduce stages) that produces the job execution time as a function of allocated slots. This function aims to represent the application model, but it is not clear how to derive this function for different applications and for different sizes of input datasets. FLEX does provide a support for optimizing the completion time of multiple jobs. In our earlier work, we proposed a framework, called ARIA [11], that implements a novel *SLO-scheduler* in Hadoop. This scheduler employs the *Earliest Deadline First* job ordering and determines the amount of resources that needed for meeting the job's deadlines. However, ARIA does not have an optimization engine to achieve additional performance objectives such as minimizing the completion time for a set of jobs.

Starfish [31] applies *dynamic instrumentation* to collect a detailed run-time monitoring information about job execution that enables predicting the job execution time under different configuration parameters. It offers a workflow-aware scheduler that correlate data placement with task scheduling to optimize the workflow completion time. In our work, we propose complementary optimizations based on optimal scheduling of independent jobs to minimize the overall completion time.

The closest work to ours is [32]. It formalizes MapReduce scheduling as a generalized version of the classical two-stage flexible flow-shop problem with identical machines. It provides a 12-approximate algorithm for the offline problem of minimizing the total *flowtime*, which is the sum of the time between the arrival and completion of each job. In our work, we pursue a different performance objective and propose heuristics for minimizing the maximum completion time for a set of jobs.

6 Discussion and Future Work

Job scheduling in MapReduce environments may pursue a variety of performance goals, and therefore, might be driven by different *Service Level Objectives* (SLOs).

Hadoop Fair Scheduler [14] pursues *fairness* of resource allocations among different jobs (or user classes). It allocates equally (or weighted) proportional shares to running MapReduce jobs.

New job schedulers such as ARIA [11], FLEX [30], and RAS [33] offer interfaces for specifying (soft) completion time objectives for MapReduce jobs. While these schedulers employ different models, utility functions, and resource allocation approaches their main common goal is to support job completion time guarantees.

However, when system administrators have a well-defined set of production jobs that periodically process newly collected datasets (logs) then the administrators' performance objective is quite different. Instead of specifying a completion time of individual jobs in the set, they are rather interested in minimizing the overall makespan of the entire set of jobs via a smart job scheduling (or job ordering). The existing Hadoop schedulers do not address this problem. We consider and formalize the makespan minimization problem, and offer a new solution for dealing with it.

For future work, there are interesting extensions of the makespan problem beyond the formulation described in this paper. For example, assume that there is a set of ready production jobs $\mathcal{J}$. We construct an optimized job schedule for $\mathcal{J}$, and start executing it at time T. Then during the set $\mathcal{J}$ execution (i.e., at a time $T+\Delta$) there is an additional subset of jobs $\mathcal{G}$ ready for processing (e.g., their input datasets are ready at this time point). How to construct a job schedule that minimizes the overall makespan for processing the combined set of jobs $G \cup J$?

We envision that there could be other interesting optimization metrics to pursue such as improving the mean job delay (defined as a job execution time that includes a job waiting time to be processed). Classic past results for a single server (with non-pre-emptive jobs) are that if the performance goal is to minimize the mean response time, then the optimal algorithm is to schedule the shortest job first. We believe that an interesting future research direction is to revisit and redefine this problem for MapReduce jobs.

7 Conclusion

Data center operators employ a variety of computing technologies and management techniques to improve resource efficiency and performance of their data centers. Large-scale Hadoop clusters with data-intensive MapReduce applications represent new entities in the enterprise data center infrastructure. Design of new schedulers and resource-aware job management for Hadoop has been an active research topic in industry and academia during the last years. In this work, we consider the problem of finding a schedule that minimizes the overall completion time of a given set of independent MapReduce jobs. We designed a novel framework and a new *heuristic*, called *BalancedPools*, that efficiently utilize characteristics and properties of MapReduce jobs in a given workload for constructing the optimized job schedule. Currently, we are evaluating this heuristic with a variety of different MapReduce workloads to measure achievable performance gains. Data analysis tasks are often specified with higher-level SQL-type abstractions like Pig or Hive, that may result in MapReduce jobs with dependencies. The next step is to address a more general problem of minimizing makespan of batch workloads that additionally include workflows (DAGs) of MapReduce jobs.

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] C. Olston *et al.*, "Pig latin: a not-so-foreign language for data processing," in *Procs of SIGMOD*, 2008.

[3] A. Thusoo *et al.*, "Hive - a Warehousing Solution over a Map-Reduce Framework," in *Proc. of VLDB*, 2009.

[4] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi, "Query optimization for massively parallel data processing," in *Proc. of SOCC*, 2011.

[5] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang, "YSmart: Yet Another SQL-to-MapReduce Translator," in *Proc. of ICDCS'2011*.

[6] A. Thusoo *et al.*, "Data warehousing and analytics infrastructure at Facebook," in *Proc. of SIGMOD*, 2010.

[7] T. Nykiel *et al.*, "MRShare: sharing across multiple queries in MapReduce." in *Proc. of VLDB*, 2010.

[8] X. Wang, C. Olston, A. Sarma, and R. Burns, "CoScan: Cooperative Scan Sharing in the Cloud," in *Proc. of SOCC*, 2011.

[9] S. Johnson, "Optimal Two- and Three-Stage Production Schedules with Setup Times Included," *Naval Res. Log. Quart.*, 1954.

[10] A. Verma, L. Cherkasova, and R. H. Campbell, "Play It Again, SimMR!" in *Proc. of Intl. IEEE Cluster'2011*.

[11] ——, "ARIA: Automatic Resource Inference and Allocation for MapReduce Environments," in *Proc. of ICAC,*, 2011.

[12] R. Graham, "Bounds for certain multiprocessing anomalies," *Bell System Tech. Journal*, vol. 45, pp. 1563–1581, 1966.

[13] A. Verma, L. Cherkasova, and R. H. Campbell, "Resource Provisioning Framework for MapReduce Jobs with Performance Goals," in *Proc. of the 12th ACM/IFIP/USENIX Middleware Conference*, 2011.

[14] "Fair Scheduler Guide." [Online]. Available: http://hadoop.apache.org/docs/r0.20.2/fair_scheduler.html

[15] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*. WH Freeman & Co., 1979.

[16] M. Zaharia *et al.*, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. of EuroSys*, 2010.

[17] S. Rao *et al.*, "Sailfish: A framework for large scale data processing," in *Proc. of the ACM Symposium on Cloud Computing, 2012*.

[18] "Capacity Scheduler Guide." [Online]. Available: http://hadoop.apache.org/common/docs/r0.20.1/capacity_scheduler.html

[19] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An Analysis of Traces from a Production MapReduce Cluster," in *Proc of CCGrid'2010*.

[20] J. Blazewicz, *Scheduling in computer and manufacturing systems*. Springer-Verlag, New York, USA, 1996.

[21] G. Blelloch, P. Gibbons, and Y. Matias, "Provably efficient scheduling for languages with fine-grained parallelism," *Journal of the ACM (JACM)*, vol. 46, no. 2, pp. 281–321, 1999.

[22] R. Blumofe and C. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.

[23] C. Chekuri and M. Bender, "An efficient approximation algorithm for minimizing makespan on uniformly related machines," *Integer Programming and Combinatorial Optimization*, pp. 383–393, 1998.

[24] C. Chekuri and S. Khanna, "Approximation algorithms for minimizing average weighted completion time," *Handbook of Scheduling: Algorithms, Models, and Performance Analysis. CRC Press, Inc., Boca Raton, FL, USA*, 2004.

[25] B. Lampson, "A scheduling philosophy for multiprocessing systems," *Communications of the ACM*, vol. 11, no. 5, 1968.

[26] J. Leung, *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press, 2004.

[27] L. Tan and Z. Tari, "Dynamic task assignment in server farms: Better performance by task grouping," in *Proc. of the Intl. Symp. on Computers and Communications (ISCC)*, 2002.

[28] T. Adam, K. Chandy, and J. Dickson., " A comparison of list schedules for parallel processing systems." *Communications of the ACM (CACM)*, vol. 17, no. 12, 1974.

[29] D. Lifka, "The anl/ibm sp scheduling system," in *Job Scheduling Strategies for Parallel Processing*. Springer, 1995.

[30] J. Wolf *et al.*, "FLEX: A Slot Allocation Scheduling Optimizer for MapReduce Workloads," *ACM/IFIP/USENIX Intl. Middleware Conference*, 2010.

[31] 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.

[32] B. Moseley, A. Dasgupta, R. Kumar, and T. Sarlós, "On scheduling in map-reduce and flow-shops," in *Proc. of SPAA*, 2011.

[33] J. Polo *et al.*, "Resource-aware Adaptive Scheduling for MapReduce Clusters," in *Proc. of the 12th ACM/IFIP/USENIX Middleware Conference*, 2011.

Abhishek Verma is currently a PhD student in the Computer Science department at the University of Illinois at Urbana-Champaign. He received his MS in Computer Science from the same university in 2010. Prior to that, he earned his BTech in Computer Science and Engineering from NIT Trichy, India in 2008. His research interests include large scale data intensive processing and performance modeling.

Lucy Cherkasova Dr. Ludmila Cherkasova is a principal scientist at HP Labs, Palo Alto, USA. Her current research interests are in developing quantitative methods for the analysis, design, and management of distributed systems (such as emerging systems for Big Data processing, internet and enterprise applications, virtualized environments, and next generation data centers). She is the ACM Distinguished Scientist and is recognized by the Certificate of Appreciation from the IEEE Computer Society.

Roy H. Campbell is the Sohaib and Sara Abbasi Professor of Computer Science at the University of Illinois at Urbana-Champaign, where he leads the Systems Research Group. He received his PhD in Computer Science from University of Newcastle upon Tyne in 1977. His current research projects include assured cloud computing, security assessment of SCADA networks, operating system dependability and security, and active spaces for ubiquitous computing.