FLEX: A Slot Allocation Scheduling Optimizer for MapReduce Workloads

Joel Wolf, Deepak Rajan, Kirsten Hildrum, Rohit Khandekar, Vibhore Kumar, Sujay Parekh, Kun-Lung Wu, Andrey Balmin

FLEX: A Slot Allocation Scheduling Optimizer for MapReduce Workloads

Joel Wolf[1], Deepak Rajan[1], Kirsten Hildrum[1], Rohit Khandekar[1], Vibhore Kumar[1], Sujay Parekh[1], Kun-Lung Wu[1], and Andrey Balmin[2]

[1] IBM Watson Research Center, Hawthorne NY 10532, USA
{jlwolf,drajan,hildrum,rohitk,vibhorek,sujay,klwu}@us.ibm.com
[2] IBM Almaden Research Center, San Jose CA 95120, USA
abalmin@us.ibm.com

Abstract. Originally, MapReduce implementations such as *Hadoop* employed *First In First Out* (FIFO) scheduling, but such simple schemes cause job starvation. The *Hadoop Fair Scheduler* (HFS) is a slot-based MapReduce scheme designed to ensure a degree of fairness among the jobs, by guaranteeing each job at least some minimum number of allocated slots. Our prime contribution in this paper is a different, *flexible* scheduling allocation scheme, known as FLEX. Our goal is to optimize any of a variety of standard scheduling theory metrics (response time, stretch, makespan and *Service Level Agreements* (SLAs), among others) while ensuring the same minimum job slot guarantees as in HFS, and maximum job slot guarantees as well. The FLEX allocation scheduler can be regarded as an add-on module that works synergistically with HFS. We describe the mathematical basis for FLEX, and compare it with FIFO and HFS in a variety of experiments.

Keywords: MapReduce, Scheduling, Allocation, Optimization

1 Introduction

The MapReduce programming model [1] has been growing dramatically in popularity for a number of years now. There are many good reasons for this success. Most are related to MapReduce's inherent simplicity of use, even when applied to large applications and installations. For example, MapReduce work is designed to be parallelized automatically. It can be implemented on large clusters of commodity hosts, and it inherently scales well. Scheduling, fault tolerance and necessary communications are all handled automatically, without direct user assistance. Finally, and perhaps most importantly, the programming of MapReduce applications is relatively straight-forward, and thus appropriate for less sophisticated programmers. These benefits, in turn, result in lower costs.

Originally MapReduce was aimed at large (generally periodic) production batch jobs. As such, the natural goal would be to decrease the length of time required for a batch window. (A dual goal might be to maximize the number of jobs which could be accommodated within a fixed batch window, effectively

maximizing throughput.) For such scenarios, a simple job scheduling scheme such as *First In, First Out* (FIFO) works very well. FIFO was, in fact, the first scheduler implemented in Google's MapReduce environment and in Hadoop [2], an open-source MapReduce implementation. Over time, however, the use of MapReduce has evolved in the natural and standard manner, towards more user interaction. There are now many more *ad-hoc* query MapReduce jobs, and these share cluster resources with the batch production work. For users who submit these queries, expecting quick results, schemes like FIFO do not work well. That is because a large job can "starve" a small, user-submitted job which arrives even a little later. Worse, if the large job was a batch submission, the exact completion time of that job might not even be regarded as particularly important.

This basic unfairness associated with FIFO scheduling motivated Zaharia et. al. to build the *Hadoop Fair Scheduler* HFS [3],[5]. (See [4] for the Hadoop implementation.) It is the top layer of HFS that is most closely aligned with our current paper. To understand this, we briefly describe MapReduce in more detail, and then the goal and design of HFS itself.

MapReduce jobs, consist, as the name implies, of two processing phases. Each phase is broken into multiple independent *tasks*, the nature of which depends on the phase. In the *Map* phase the tasks consist of the steps of scanning and processing (extracting information) from equal-sized *blocks* of input data. Each block is typically replicated on disks in three separate racks of hosts (in Hadoop, for example, using the HDFS file system). The output of the Map phase is a set of key-value pairs. These intermediate results are also stored on disk. Each of the Reduce phase tasks corresponds to a partitioned subset of the keys of the intermediate results. There is a shuffle step in which all relevant data from all Map phase output is transmitted across the network, a sort step, and finally a processing step (which may consist of transformation, aggregation, filtering and/or summarization).

HFS can be said to consist of two hierarchical algorithmic layers, which in our terminology will be called the *allocation* layer and the *assignment* layer.

Allocation Layer. Each host is assumed to be capable of simultaneously handling some maximum number of Map phase tasks and some maximum number of Reduce phase tasks. These are the number of *Map slots* and *Reduce slots*, respectively. Aggregating these slots over all the hosts in the cluster, we compute the total number of Map slots, and similarly the total number of Reduce slots. The role of the allocation layer scheme is to partition the Map slots among the active Map jobs in some intelligent manner, and similarly the number of Reduce slots among the active Reduce jobs. In [3] and [5], the node that produces these allocations is known as the *master*. We will call the HFS allocation layer FAIR.

Assignment Layer. It is this layer that makes the actual job task assignment decisions, attempting to honor the allocation decisions made at the allocation level to the extent possible. Host *slaves* report any task completions at *heartbeat* epochs (on the order of a few seconds). Such completions free up slots, and also incrementally affect the number of slots currently assigned to the various jobs. The current slot assignment numbers for jobs are then subtracted from

the job allocation goals. This yields an effective ordering of the jobs, from most relatively underallocated to most relatively overallocated. For each currently unassigned slot, the HFS assignment model then finds an "appropriate" task from the most relatively underallocated job that has one, assigns it to the slot, and performs bookkeeping. It may not find an appropriate task for a job, for example, because of host and/or rack affinity issues. That is why HFS *relaxes* fidelity to the precise dictates of the master allocation goals for a time. This is known as *delay scheduling*, and described in [3] and [5].

Let S denote the total number of Map slots. Then the FAIR allocation scheme is *fair* in the following sense: It computes, for each of J Map phase jobs j, a minimum number m_j of Map slots. This minimum number is chosen so that the sum $\sum_{j=1}^{J} m_j \leq S$. (The minima are simply normalized if necessary.) This minimum number m_j acts as a fairness guarantee, because FAIR will always allocate a number of Map slots $s_j \geq m_j$, thereby preventing job starvation. Slack (the difference $S - \sum_{j=1}^{J} m_j$) is allocated in FAIR according to a very simple waterline-based scheme (which also emphasizes fairness). Analogous statements hold for the Reduce phase.

While HFS [3] mentions standard scheduling metrics such as throughput or average response time as a goal of the scheduler, and compares its scheduler to others in terms of these, FAIR makes no direct attempt to optimize such metrics. It is worth noting that schedules designed to optimize one metric will generally be quite different from those designed to optimize another.

1.1 Our Contributions

Our prime contribution in this paper is a different, *flexible* allocation scheme, known as FLEX. The goal of our FLEX algorithm is to optimize any of a variety of standard scheduling theory metrics while ensuring the same minimum job slot guarantees as in FAIR. Thus FLEX will also be fair in the sense of [3]: It avoids job starvation just as FAIR does. The metrics can be chosen by the system administrator on a cluster-wide basis, or by individual users on a job-by-job basis. And these metrics can be chosen from a menu that includes response time, makespan (dual to throughput), and any of several metrics which (reward or) penalize job completion times compared to possible deadlines. More generally, one could imagine negotiating *Service Level Agreements* (SLAs) with MapReduce users for which the penalty depends on the level of service achieved. (Formally, this can be described by a step function, each step corresponding to the penalty that would be incurred by missing the previous pseudo-deadline but making the next one.) FLEX can work towards optimizing any combination of these metrics while simultaneously ensuring fairness. It can minimize either the sum (and, equivalently, the average) of all the individual job penalty functions or the maximum of all of them. While we expect most system administrators to make a single choice of metric, there are natural examples of job-by-job choices. For example, one might employ different metrics for batch jobs and ad-hoc queries.

Note that the FLEX allocation scheduler can be regarded as either a standalone replacement to FAIR *or* as an add-on module that simply works synergis-

tically with FAIR: To understand this point imagine HFS handing FLEX the Map phase minimum slots $m_1, ..., m_J$. After optimizing the appropriate scheduling metrics, FLEX would hand back slot allocations $s_1, ..., s_J$ such that $s_j \geq m_j$ for all jobs j, and there is no slack: $\sum_{j=1}^{J} s_j = S$. These values s_j can then serve as *revised* minima to be handed to FAIR. Because there is no slack, the water-line component of FAIR will be neutralized. Thus FAIR will produce the properly optimized solution $s_1, ..., s_J$. A similar statement holds for the Reduce phase.

Additionally, our FLEX allocation scheme can handle maximum slots M_j for a given job j as well as minima, and they are implicitly dealt with in FAIR as well. For example, at the very end of the Map phase the number of remaining tasks for a particular job might be less than the total number of Map slots S. There is clearly no reason to allocate more slots than the current demand.

The remainder of this paper is organized as follows. An overview of the mathematical concepts behind our FLEX scheduler is given in Section 2. The algorithmic details of FLEX are given in Section 3. Simulation and cluster experiments designed to show the effectiveness of FLEX are described in Section 4. Related MapReduce scheduling work is described in Section 5. Finally, conclusions and future work are given in Section 6.

2 Scheduling Theory Concepts

We give an overview of the fundamental mathematical concepts behind multi-processor task scheduling, focusing on how they apply to MapReduce scheduling in general, and to FLEX in particular. We begin by describing five broad categories of penalty functions, including multiple variants of each. Then, we describe epoch-based scheduling. Finally, we define the notion of a *speedup* function, and how it pertains to the notions of *moldable* and *malleable* scheduling. The algorithmic details of FLEX (a penalty-function-specific, malleable, epoch-based scheduler) will be covered in the next section.

2.1 Penalty Functions

Think of each job as having a penalty function which measures the cost of completing that job at a particular time. Thus each of the five subfigures in Fig. 1 describes the form of a particular per job penalty function. The X-axis represents the completion time of that job, while the Y-axis represents the penalty. We point out that there can still be many potential problem variants for most of them. These combinatorial alternatives involve, for example, whether or not to incorporate non-unit weights into the penalty functions. (In some cases specific weight choices will have special meanings. In other cases they are used basically to define the relative importance of each job.) Also, it generally makes sense either to minimize the sum of all the per job penalty functions, or to minimize the maximum of all the per job penalty functions. The former case is referred to as a *minisum* problem, and the latter case as a *minimax* problem. The five penalty function categories are as follows.

Response Time. The metric illustrated in Fig. 1(a) is probably the most commonly employed in computer science. (The weight is the slope of the linear function.) Three natural examples come to mind. Solving the minisum problem would minimize either the average response time or the weighted average response time of all the jobs. In the unweighted case, the minimax problem would be used to minimize the *makespan* of the jobs. This is the completion time of the last job to finish and is appropriate for optimizing batch work. Suppose the *work* (or time required to perform job j in isolation) is W_j. Then the completion time of a job divided by W_j is known as the *stretch* of the job, a measure of how delayed the job will be by having to share the system resources with other jobs. Thus, solving a minisum problem while employing weights $1/W_j$ will minimize the average stretch of the jobs. Similarly, solving a minimax problem while employing weights $1/W_j$ will minimize the maximum stretch. Either of these are excellent *fairness* measures, and are in fact precisely the two metrics used in [6]. (They are not, however, formally considered in [3] and [5].)

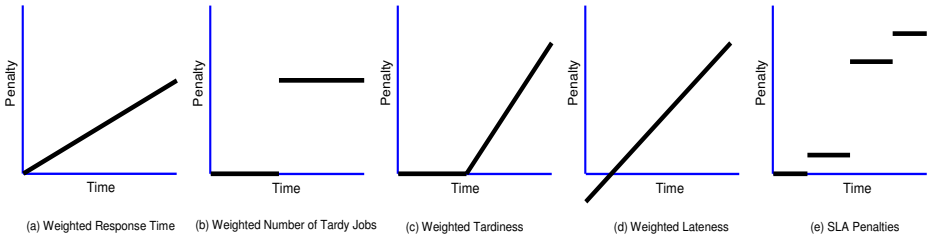


Fig. 1. Per Job Metrics

Number of Tardy Jobs. In this case each job j has a *deadline*, say d_j. In this case only the minisum problem is appropriate. The weight is the height of the "step" in Fig. 1(b). The unweighted case counts the number of jobs that miss their deadlines. But the weighted case makes sense as well.

Tardiness. Again, each job j has a deadline d_j. The tardiness metric generalizes the response time metric, which can be said to employ deadlines at time 0. Only tardy jobs are "charged", and the slope of the non-flat line segment in Fig. 1(c) is the weight. It makes sense to speak of either minisum or minimax tardiness problems, both either weighted or unweighted.

Lateness. Again, each job j has a deadline d_j. The lateness metric generalizes response time also, with the same explanation as that of tardiness above. As before, the weight is the slope of the line. Note that "early" jobs are actually rewarded rather than penalized, making this the only potentially negative metric. The minisum variant differs from the response time metric by an additive constant, and thus can be solved in exactly the same manner as that problem. But the minimax problem is legitimately interesting in its own right. See Fig. 1(d).

SLA Costs. In this metric each job j has potentially multiple pseudo-deadlines $d_{i,j}$ which increase with i. And the penalties $w_{i,j}$ increase with i also. This

yields the metric of Fig. 1(e), consisting of a step function for each job, clearly a generalization of the weighted number of tardy jobs metric. As in that case, only the minisum problem is appropriate. One can think of this metric as the total cost charged to the provider based on a pre-negotiated SLA contract.

All of these metrics (except, perhaps, the last) has been studied extensively in the scheduling theory literature. See [7], [8] and [9]. We will borrow and adapt some of these techniques for use in our FLEX scheduling algorithm.

2.2 Epoch-Based Scheduling

FLEX is an example of an *epoch*-based allocation scheduler. This means that it partitions time into epochs of some fixed length T. So if time starts at $t = 0$ the epochs will start at times 0, T, $2T$, $3T$ and so on. Label these accordingly. The scheduler will produce allocations that will be in effect for one epoch, so that the eth epoch allocations will be honored from time eT to time $(e+1)T$. Obviously the work of FLEX for the eth epoch must be completed by the start time eT of that epoch. If T is relatively small it follows that FLEX must be comparably fast.

There will actually be two instances of FLEX running at any given time, one for the active Map phase jobs and one for the active Reduce phase jobs. We will describe the Map instance below, but the Reduce instance is comparable.

The Map instance of FLEX receives input describing the total number of Map slots in the system, the number of active Map jobs, the minimum and maximum number of slots per active Map job, and estimates of the remaining execution times required for each of the active Map jobs. Then the algorithm outputs high quality allocations of slots to these jobs. These allocations may be time-dependent in the sense that there may be several consecutive *intervals*, say I, of different allocation levels. In more detail, consider the eth epoch. The output will take the form $(s_{1,1}, \ldots, s_{1,J}, T_0, T_1), \ldots, (s_{I,1}, \ldots, s_{I,J}, T_{I-1}, T_I)$, where $T_0 = eT$, the ith interval is the time between T_{i-1} and T_i, and $s_{i,j}$ represents the number of slots allocated to job j in interval i. Allocations for the eth epoch will likely extend beyond the start time of the $(e+1)$st epoch. That is, we expect that $T_I > (e+1)T$. But any of these allocation decisions will be superseded by the decisions of newer epochs. In fact, it is expected that the completion time of even the *first* of the consecutive intervals in the eth epoch will typically exceed the length of an epoch, so that $T_1 > (e+1)T$. This means that generally only the first interval in the FLEX output will actually be enforced by the assignment model during each epoch.

An advantage of an epoch-based scheme is its resilience to inaccuracies in input data that might arise from a heterogeneous cluster environment. Epoch by epoch, FLEX automatically corrects its solution in light of better estimates and system state changes.

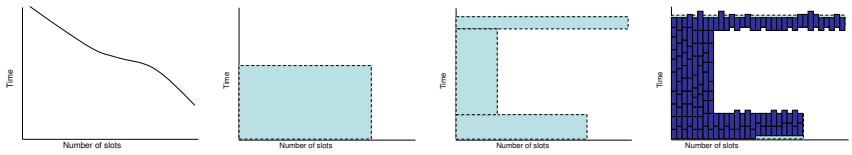
2.3 Speedup Functions

From a scheduling perspective a key feature of either the Map or Reduce phase of a MapReduce job is that it is *parallelizable*. Roughly speaking, it is composed of

many atomic tasks which are effectively independent of each other and therefore can be performed on a relatively arbitrary number of (multiple slots in) multiple hosts simultaneously. If a given job is allocated more of these slots it will complete in less time. In the case of the Map phase these atomic tasks correspond to the blocks. In the case of the Reduce phase the atomic tasks are created on the fly, based on keys. The FLEX scheme takes advantage of this additional structure inherent in the MapReduce paradigm.

We now describe the relevant scheduling theory concepts formally. See, for example, [9]. Consider a total of S homogeneous hosts. (In our scenario the hosts will actually be the Map or Reduce slots in the cluster.) A job is said to be *parallel* if it can be performed using some number of hosts $1 \leq s \leq S$ simultaneously, with an execution time E. (One can think geometrically of this job as a rectangle with width equal to the number of hosts s, and height equal to the execution time E.) A job is said to be *parallelizable* if it can be performed variously on an arbitrary number $1 \leq s \leq S$ of hosts simultaneously, with an execution time $F(s)$ that depends on the number of hosts allocated. The execution time function F is known as the *speedup* function. It can be assumed without loss of generality to be non-increasing, because if $F(s) < F(s+1)$ it would be better to simply leave one host idle. This would result in a new (replacement) speedup function $\bar{F}$ for which $\bar{F}(s+1) = \bar{F}(s) = F(s)$. One can think, also without loss of generality, of a job which can be performed on a subset $P \subseteq \{1, ..., S\}$ of possible allocations of hosts as being parallelizable: simply define $F(s) = \min_{\{p \in P | p \leq s\}} F(p)$, where, as per the usual convention, the empty set has minimum ∞. In this sense parallelizable is a generalization of parallel.

As we shall describe below, to a good approximation both the Map and Reduce phases of a MapReduce job have such speedup functions. Fig. 2(a) illustrates a possible speedup function in a MapReduce context. (Note that the X-axis is described in terms of allocated slots.)



(a) Speedup (b) Moldable Job (c) Malleable Job (d) Tasks

Fig. 2. Moldable and Malleable Scheduling

2.4 Moldable and Malleable Scheduling

The problem of scheduling multiple parallel jobs in order to minimize some given metric can be visualized as a problem of packing rectangles, with job start times as the decision variables. Parallel scheduling has been studied extensively in the

scheduling literature [9]. Problems for most of the natural metrics are NP-hard. See, for example, [10] for the makespan metric, and [11] for the weighted and unweighted response time metrics.

Parallelizable scheduling is a generalization of parallel scheduling. The simplest such variant is known as *moldable* scheduling. Here the problem is to schedule multiple parallelizable jobs to optimize a given metric. The number of hosts is treated as an additional decision variable, but once this allocation is chosen it cannot be changed during the entire execution of the job. The name comes because the rectangles themselves can be thought of as moldable: Pulling a rectangle wider (that is, giving a job more hosts) has the effect of making the rectangle less high (that is, executing the job faster). Fig. 2(b) illustrates a potential choice of host (slot) allocations for a moldable job in a MapReduce context. The top-right vertex in the rectangle corresponds to a point on the speedup function of Fig. 2(a). Moldable scheduling problems can sometimes be solved by using the parallel scheduling problem algorithm as a subroutine. See, for example, [12] for makespan, and [11] for weighted and unweighted response time. In each of these cases, the approximation factor of the parallel algorithm is retained.

Finally, one can generalize moldable scheduling as well, to so-called *malleable* scheduling. Here the problem is to schedule multiple parallelizable jobs to optimize a given metric, as before. But instead of making a permanent decision as to the host allocations each job can proceed in multiple intervals. Different intervals can involve different allocations. Each interval contributes a portion of the total work required to perform the job. Fig. 2(c) illustrates a potential three interval choice of host (slot) allocations for a malleable job in a MapReduce context. See, for example, [13] for makespan.

The optimal malleable schedule for a particular scheduling problem instance will have a cost less than or equal to that of the optimal moldable schedule for the same problem instance, since moldable schedules are also malleable. In this paper we are attempting to find an optimal malleable scheduling solution for each of the relevant objective functions. But we will first solve the moldable scheduling problem. The solution there will then help us to solve the more general malleable problem.

2.5 Malleable Scheduling and the Model

Now we explore how well the MapReduce model fits the malleable scheduling framework. There are several aspects to the answer.

To begin with, the parallel job structure in MapReduce is a direct consequence of the decomposition into small, independent atomic tasks. Consider the assignment layer. Its role is to approximate at any given time the decisions of the allocation layer by assigning job tasks. Consider Fig. 2(d). Over time, the assignment layer will nearly replicate the malleable allocation decisions given in Fig. 2(c) by a set of job task to slot decisions. The approximation is not perfect, for several reasons: First, slots may not free up at precisely the time predicted by the allocation layer. So new task slots won't be assigned perfectly: at the bottom of each interval the rectangle may be slightly jagged. Likewise, slots may

not end at precisely the time predicted by the allocation layer. Thus at the top of each interval the rectangle may also be slightly jagged. Finally, the assignment layer may relax adherence to the exact allocation goals for the various jobs, for example, in order to ensure host or rack locality [3],[5].

But the first two are modest approximations *because* individual task times are small relative to the total time required for the job. And discrepancies between the allocation layer goals and the assignment layer are modest by definition. In summary, as illustrated in Figures 2(c-d), the malleable scheduling model fits the reality of the MapReduce paradigm closely.

Moreover, because the tasks are independent, the total amount of work involved in a job is essentially the sum of the work of the individual tasks. Therefore, in conjunction with the statements above, we can actually assume a speedup function of a very special kind: The speedup should be close to *linear*, meaning a speedup function of the form $F(s) = C/s$ between its minimum and maximum numbers of slots. (Here C is a constant for a job proportional to the amount of work required to perform that job.) Note that this is not a statement particularly affected by the factors such as locality. Such factors would affect the constant, not the shape of the speedup function.

Speedup functions for individual jobs must be estimated in order for FLEX to do its job. Fortunately, by assuming relative uniformity among the job task times, we can repeatedly extrapolate the times for the remaining tasks from the times of those tasks already completed. The refined estimates should naturally become more accurate as the job progresses, epoch by epoch. Periodic jobs can also be seeded with speedup functions obtained from past job instances.

3 FLEX Algorithmic Details

With these preliminaries we are ready to describe FLEX. As already noted, FLEX works identically for either the Map or the Reduce phase, so we describe it in a phase-agnostic manner. The FLEX algorithm depends on two key ideas:

1. Given any ordering of the jobs we will devise a *Malleable Packing Scheme* (MPS) to compute a high quality schedule. In fact, it turns out that for any of our possible metrics, there will exist a job ordering for which the MPS schedule will actually be optimal. (The job ordering is strictly input to the packing scheme. There is no guarantee, for example, that the jobs will complete in that order in the MPS schedule.)
2. Unfortunately, finding the job ordering that yields an optimal MPS solution is difficult. We instead find a high quality ordering for any of our possible metrics in one of two ways: One approach is to employ an essentially *generic* algorithm to solve a so-called *Resource Allocation Problem* (RAP). The solution to this RAP will actually be optimal in the context of moldable scheduling, assuming positive minima for each job. In terms of malleable schedules the solution will not be optimal. This leads to the second approach. We create better schemes which are *specific* to selected metrics. (We must omit these details due to lack of space.)

Combining all of this into a single algorithm, FLEX creates a generic ordering and at most a modest number of specific orderings, feeding each of them to MPS. The final output is the best solution MPS found. Now we give the details.

3.1 Malleable Packing Scheme

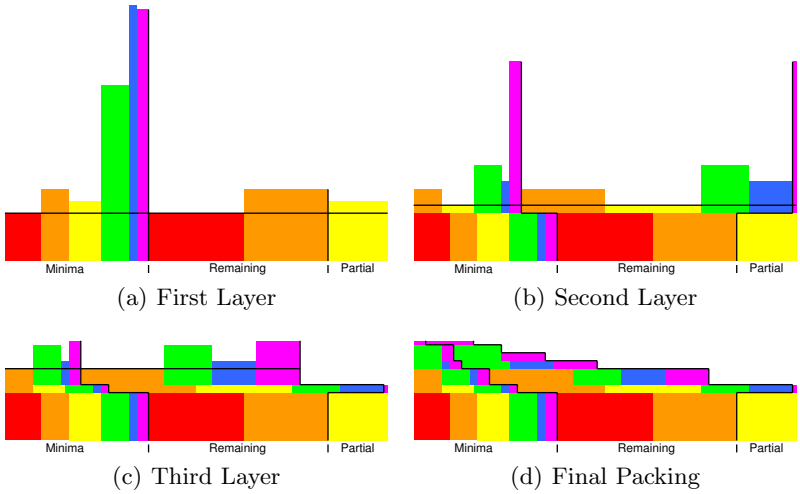
```
1: Set time T_0 = 0
2: Create list L = {1, ..., J} of jobs, ordered by priority
3: for i = 1 to J do
4:    Allocate m_j slots to each job j ∈ L
5:    Set L = L, with implied ordering
6:    Compute slack s = S − Σ_{j∈L} m_j
7:    while s > 0 do
8:       Allocate min(s, M_j − m_j) additional slots to highest priority job j ∈ L
9:       Set L = L \ {j}
10:      Set s = s − min(s, M_j − m_j)
11:   end while
12:   Find first job j ∈ L to complete under given allocations
13:   Set T_i to be the completion time of job j
14:   Set L = L \ {j}
15:   Compute remaining work for jobs in L after time T_i
16: end for
```

Fig. 3. Malleable Packing Scheme Pseudocode

Fig. 3 contains the pseudocode for the malleable packing scheme. Given a priority ordering, the scheme proceeds iteratively. At any iteration a *current* list $\mathcal{L}$ of jobs is maintained, ordered by priority. Time is initialized to $T_0 = 0$. The current list $\mathcal{L}$ is initialized to be all of the jobs, and one job is removed from $\mathcal{L}$ at the completion time T_i of each iteration i. Call the time interval during iteration i (from time T_{i-1} to T_i) an *interval*. The number of slots allocated to a given job may vary from interval to interval, thus producing a malleable schedule.

The ith iteration of the algorithm involves the following steps: First, the scheme allocates the minimum number m_j of slots to each job $j \in \mathcal{L}$. This is feasible, since the minima have been normalized, if necessary, during a precomputation step. (This idea is employed by the FAIR scheduler [4] in addition to our scheme.) After allocating these minima, some slack may remain. This slack can be computed as $s = S - \sum_{j \in L} m_j$. The idea is to allocate the remaining allowable slots $M_j - m_j$ to the jobs j in priority order. The first several may get their full allocations, and those jobs will be allocated their maximum number of slots, namely $M_j = m_j + (M_j - m_j)$. But ultimately all S slots may get allocated in this manner, leaving at most one job with a "partial" remaining allocation of slots, and all jobs having lower priority with only their original, minimum number of slots. (The formal details of these steps are given in the pseudocode.) Given this set of job allocations, one of the jobs j will complete first, at time T_i.

(Ties among jobs may be adjudicated in priority order.) Now job j is removed from $\mathcal{L}$, and the necessary bookkeeping is performed to compute the remaining work past time T_i for those jobs remaining in $\mathcal{L}$. After J iterations (and J intervals) the list $\mathcal{L}$ will be depleted and the malleable schedule created.



(a) First Layer (b) Second Layer

(c) Third Layer (d) Final Packing

Fig. 4. Malleable Packing Scheme

Fig. 4(a-c) show the first three "intervals" in the malleable packing iterative process for a 6-job example. The priority ordering is given as a rainbow coloring (red, orange, yellow, green, blue and violet). Note the jobs getting their minimum number of slots, the jobs which get their remaining number of slots (up to their maximum), and the single job getting a partial number of slots between its minimum and its maximum. At each of these intervals the completion time T_i is shown as a horizontal line. (The interval extends from the previous time T_{i-1} to the current time T_i, as noted, and the portion of the schedule beyond this time is discarded.) Note that each new iteration involves one fewer job – this job may or may *not* be the next job in the priority ordering. The final malleable packing is shown in Fig. 4(d). In this example there are 6 intervals.

The following lemma states that under linear speedup assumption, for any of the metrics considered here, there exists an ordering of jobs such that the MPS schedule is optimum. The proof of this lemma is omitted due to lack of space.

Lemma 1. *Assume that each job satisfies linear speedup, i.e., the speedup function of job j has the form $F_j(s) = C_j/s$ where C_j is a constant depending on j. Fix any minisum or minimax aggregate objective function of non-decreasing per job penalty functions of the completion time of the jobs. There exists an ordering $\mathcal{L}^*$ of the jobs such that the* MPS *algorithm in Fig. 3 with $\mathcal{L} = \mathcal{L}^*$ outputs an optimum schedule for the given objective function.*

3.2 Finding a High-Quality Priority Ordering

Considering Lemma 1, we would like to find an ordering $\mathcal{L}$ on jobs for which the MPS algorithm performs well. Unfortunately, finding optimum such an ordering is NP-hard for many of our metrics.

We will therefore determine the priority ordering of jobs to use in the MPS by solving a simplified variant of the overall scheduling problem. Specifically, the completion time ordering of the jobs for this algorithm will become the input priority ordering to MPS. In the case of the generic scheme the simplified problem will be a moldable version of the original problem. (In the case of the metric-specific schemes a closer, malleable version of the problem is considered.)

Generic Scheme. The good news is that regardless of the specific metric chosen, finding the optimal moldable schedule can be formulated as a separable resource allocation problem (RAP). In the first category (*minisum*), we wish to minimize $\sum_{j=1}^{J} F_j(s_j)$. In the second category (*minimax*), we wish to minimize $\max_{1 \le j \le J} F_j(s_j)$. In both categories the minimization is subject to the two constraints $\sum_{j=1}^{J} s_j \le S$ and $m_j \le s_j \le M_j$ for all $1 \le j \le J$. (See [14] for details on RAPs, including all of the schemes listed below.) Separability here means that each summand $F_j(s_j)$ is a function of a single decision variable s_j, and such resource allocation problems are relatively straightforward to solve exactly.

Minisum. Minisum separable RAPs can be solved by one of several means, the key being whether or not the functions involved are *convex*. In fact, if the individual functions happen to be convex the problem can be solved by one of three standard algorithms: These are the schemes by Fox [15], Galil and Megiddo (*GM*) [16], and Frederickson and Johnson (*FJ*) [17], which can be regarded as fast, faster and (theoretically) fastest, respectively. The *FJ* scheme, which we employ, has complexity $O(\max(J, J\log(S/J))$. If the individual functions, on the other hand, are not convex, the problem may still be solved by a simple dynamic programming (*DP*) algorithm. This scheme has complexity $O(JS^2)$.

Checking for convexity of the underlying summands is easy for the various metrics we consider. Specifically, only the (weighted) number of tardy jobs and the (weighted) total tardiness require the more expensive *DP*. All others minisum metrics can use *FJ*.

Minimax. For minimax separable RAPs the situation is cleaner. As long as the functions F_j are non-decreasing, as all of our metrics are, the minimax problem can be reduced naturally to a minisum convex separable RAP: The transformation is via the functions $G_j(s_j) = \sum_{i}^{s_j} F_j(i)$. Note that this function is convex *because* the original function $F_j(s_j)$ is non-decreasing. Also note that the first differences of G_j correspond to F_j. An easy telescoping argument then shows that the solution to the separable convex RAP of minimizing $\sum_j G_j(s_j)$ subject to the constraints $\sum_j s_j \le S$ and $m_j \le s_j \le M_j$ corresponds precisely to the solution to the minimax RAP for $F_j(s_j)$.

This means that one can apply any of the fast schemes, such as *FJ*, to each of the minimax problems.

4 Experiments

In this section we will describe the performance of our FLEX scheduler by means of a number of experiments. These include both simulation and actual experiments on a cluster of hosts.

We should note that our experiments are in some sense *unfair* to FAIR. That is because FAIR was not designed to behave well with respect to any of the possible metrics on which we are evaluating it. It is merely trying to be fair to the smaller jobs, by incorporating minimum constraints. On the other hand, we are being *more* than fair to FIFO. First, let us be precise by what we will mean by FIFO here. One can imagine three FIFO variants of various levels of freedom and intelligence. We choose the variant which does not obey the minimum constraints but does take advantage of the maximum constraints. This seems to be in the spirit of FIFO, and also at least modestly intelligent. So we are being very fair to FIFO because we are comparing it with two schemes (FAIR and FLEX) which must obey the fairness constraints imposed by the minima. By the way, notice that the FIFO scheme we use can be regarded as an MPS schedule with priority orderings determined by job arrival times, where the minima are set to 0.

4.1 Simulation Experiments

First we will describe the experimental methodology employed. Each experiment represents multiple jobs, in either the Map or Reduce phase.

The amount of work for each job is chosen according to a bimodal distribution that is the sum of two (left) truncated normal distributions. The first simulates "small" jobs and the second simulates "large" jobs. The second mean is chosen 10 times larger than the first, and the standard deviations are each one third of the means. The ratio of small to total jobs is controlled by a parameter which is allowed to vary between 0 and 100%. Once work values are chosen for each job the values are normalized so that the total amount of work across all jobs in any experiment is a fixed constant. Assuming an average amount of work associated with each task in a job the total work for that job can then be translated into a number of slots. If the slot demand for a job is less than the total number of slots available the maximum constraint is adjusted accordingly. Otherwise it is set to the total number of slots. The minimum number of slots, on the other hand, is chosen according to a parameter which controls the total fraction of slack available. This slack parameter can essentially vary between 0 and 100%, though for fairness we insist on a non-zero minimum per job. The actual minimum for each job is chosen from another truncated normal distribution, with a mean chosen based on the slack parameter and a standard deviation one third of that. If the job minimum computed in this manner exceeds the maximum it is discarded and resampled.

As we have noted, there are many possible metrics which can be employed by FLEX, even assuming that all jobs use the same type. The natural minisum problems include average response time, number of tardy jobs, average tardiness, average lateness, and total SLA costs. Each but the last of these can be weighted

or unweighted, and the special weight associated with stretch adds yet another metric. The natural minimax problems include maximum response time, tardiness and lateness. And again, each can be weighted or unweighted, and maximum stretch involves a special weight. All job weights except for the stretch-related weights are chosen from a uniform distribution between 0 and 1. Deadlines, pseudo-deadlines and step heights are chosen from similar uniform distributions, with the obvious constraints on multiple steps for SLAs. (The pseudo-deadlines and step heights are simply reordered into monotone sequences.)

A few of our metrics can produce solutions which are zero or even negative (for lateness). In fact, this is one reason why approximation algorithms for these metrics cannot be found. Taking the ratio of a FLEX, FAIR or FIFO solution to the optimal solution (or any other one) may produce a meaningless result. There is actually no "nice" way to deal with this, and for that reason we choose to simply ignore such experiments when they occur, reporting on the others.

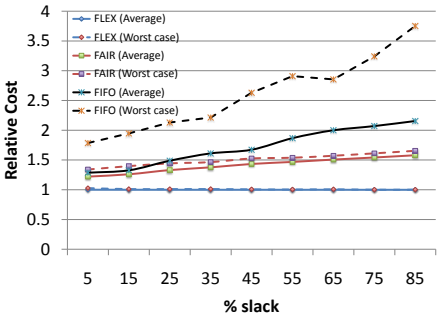
We start with simulation experiments involving 10 jobs and 100 Map or Reduce slots. We choose a modest number of jobs so that we can compare each of the three solutions to the optimal solution, which we compute by a brute force exhaustive search. However, by comparing FAIR and FIFO to FLEX directly in much larger experiments we have shown that our conclusions remain intact.

The base case for our experiments is the average response time metric, a small job parameter of 80% and a slack parameter of 75%. Average response time is the most commonly used metric, and the small job and slack parameters should be common cases. Then we vary along each of the three axes, first slack, then small jobs and finally on the choice of metric itself. Each of the experiments shown in the three figures result in 6 data points, namely the average and worst case ratios of FAIR, FIFO and FLEX to the optimum solutions in 100 simulations.

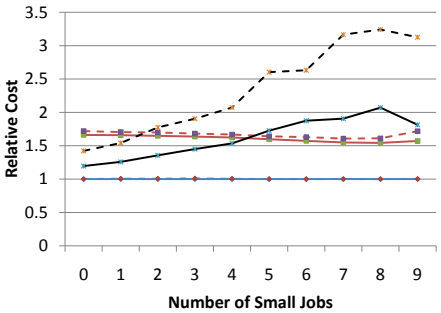
For our base case FIFO exhibits average case performance which is 207% of optimal, and worst case performance which is 324% of optimal. For FAIR the numbers are 154% and 161%, respectively. FLEX is virtually indistinguishable from optimal, with a worst case performance less than .1% higher.

Fig. 5(a) compares the average response time performance as we vary slack percentage on the X-axis. We hold the small job parameter to 80%. The Y-axis on this and the other figures is the ratio of each scheme to the value of the optimal solution, so that 1.0 is a lower bound. Larger percentages give FAIR, FLEX and the optimal solution greater flexibility, but FIFO is agnostic with respect to slack. The FIFO curves increase with slack *because* the optimal solution in the denominator decreases. Note that FIFO is not robust: The worst case performance differs significantly from the average case performance because it is highly dependent on the arrival order of the large jobs. Early large job arrivals destroy the response times of the small jobs. Both FAIR and FLEX, on the other hand, are able to deal with job arrivals robustly, and their average and worst case performance are therefore close to each other. The FAIR curves increase monotonically with greater slack, primarily because the optimal solution improves faster than the FAIR solution does. But FLEX finds a solution very close to the optimal in all cases, so both the average and worst case curves overlay the line at 1.0.

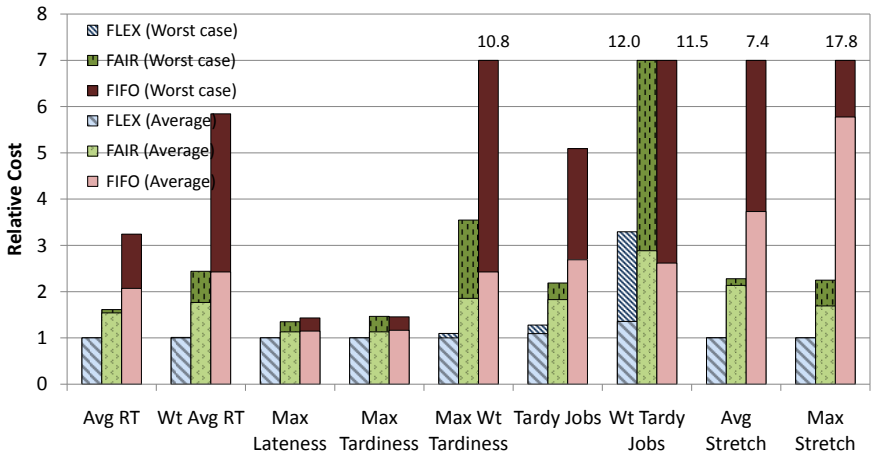
Fig. 5(b) compares the average response time performance as we vary the number of small jobs. We hold the slack parameter to 75%. We do not plot the case with 10 small jobs because the jobs would be picked in this case from a homogeneous distribution. So once normalized by the total amount of work these data points would be comparable to the case with 0 small jobs. Once again FIFO is not robust, exhibiting worst case performance significantly worse than its average case performance. It performs poorest at about 80% small jobs, and is at its best at the two homogeneous end points. On the other hand, the FAIR and FLEX curves are basically flat, and have individually similar average and worst case performance. The FAIR solutions are, however, uniformly about 50% higher than the virtually optimal FLEX solutions.



(a) Effect of % of slack slots



(b) Effect of number of small jobs



(c) Optimization of a variety of metrics

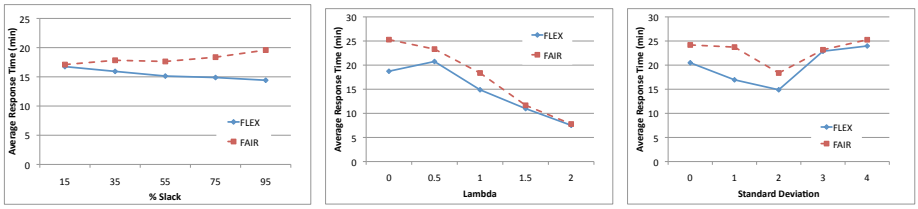
Fig. 5. Comparison of FLEX, FAIR, and FIFO

Finally, Fig. 5(c) compares the performance of a representative sample of the other metrics, with a small job parameter of 80% and a slack parameter of 75%. In this figure we have overlaid the average and worst case ratios for each metric

and scheme. Note that the worst case ratio is by definition at least as large as the average case ratio. If the former is not visible it means that the two coincide. The worst case ratios for some FIFO (and one FAIR) runs are so high that we have truncated the values for readability. From the figure we see that the average response time is a relatively *easy* metric for FLEX to optimize. Both FIFO and FAIR are unaware of these metrics, so their solutions do not vary. On the other hand, the optimal solution in the denominator does change. FLEX does very well compared to the others on weighted average response time, maximum weighted tardiness, weighted and unweighted number of tardy jobs, and both important stretch metrics.

4.2 Cluster Experiments

In these experiments, we compare FLEX to the implementation of FAIR in HFS, and present results for the average response time metric. In addition to being the most commonly used metric, it needs no additional synthetically created input data (weights, deadlines, etc.). We do not present results for FIFO since it ignores the minimum number of slots for each job, and since the simulation experiments show its performance is not competitive anyway.



(a) Effect of % of slack slots (b) Effect of job inter-arrival times (c) Effect of variance in job sizes

Fig. 6. Comparison of FLEX and FAIR in real runs

From a code standpoint, we have currently built FLEX as an extension of FAIR allocation layer of HFS (Hadoop version 20.0), overriding its calculation of minimum slot allocations to each job. We use FAIR's infrastructure for calculating minimum slots and deficits to be used in the assignment layer to schedule tasks. Since FLEX requires estimates of total work for a job, whenever a new phase for a job is ready to be executed, we ensure that its number of minimum slots is at least 1. By doing so, at least one task is executed as soon a slot is available (to estimate the total work for that phase). At the completion of each task, we log its processing time, and re-estimate the remaining work for each phase (extrapolating based on the average seen so far, and the number of tasks left). We do not modify FAIR in any other way.

Our experiments are run on a cluster of 26 hosts with 3GHz Intel Xeon processors. This configuration is heterogeneous - 13 4-core blades on a rack and

13 8-core blades directly connected to the network switch. We pre-generate 1.8 TB of synthetic data, to be used as input by our synthetic workload which consists of a job written in Jaql [18]. This job is completely parametrizable. (Both the size of the Maps and Reduces can be modified independently). To simulate jobs of a variety of sizes, we use different versions of this job. For each experiment, we picked a subset of 10 jobs that can be completed on our cluster in about 45 minutes. In all the figures presented, the Y-axis is the average response time in minutes, and the X-axis is the parameter being modified.

Minimum slots are assigned to the jobs by allocating all jobs to the same pool, and configuring the pool's minimum number of slots. We set the same minimum for Map slots and Reduce slots. Observe that the minimum slots given to each job at any time depends on the number of running jobs, since jobs in a pool share the minimum slots. We present experiments with slack of $15\%, 35\%, 55\%, 75\%, 95\%$ in Fig. 6(a). When slack is small, FLEX does not do much better than FAIR because the optimization problem is more constrained. However, FLEX does significantly better better as slack is increased, outperforming FAIR by almost 50% when slack is large.

We simulate an on-line problem (where jobs are submitted at various times) by modifying the inter-arrival times based on an exponential distribution. The time between submission of job i and job $i+1$ is chosen from an independently drawn exponential distribution of mean λ minutes. We present results for $\lambda = 0, 0.5, 1, 1.5, 2$ in Fig. 6(b). When $\lambda = 0$, all jobs are submitted at time 0. Observe that both FLEX and FAIR perform better as the inter-arrival time is increased, since jobs are competing less for resources. Values of $\lambda > 3$ result in very little difference between the performance of FAIR and FLEX, since there is not much overlap between the jobs. For smaller λ, FLEX outperforms FAIR significantly, almost 50% when $\lambda = 0$.

We measure the heterogeneity of job sizes in each experiment using the standard deviation (σ) of the estimated work of the jobs in the sample. We present experiments with $\sigma = 0, 1, 2, 3, 4$ in Fig. 6(c). When $\sigma = 0$, all jobs are identical, and when $\sigma = 4$, the smallest job is roughly one-tenth the size of the largest job. As before, FLEX results in lower average response time than FAIR, as low as 50% better in some cases ($\sigma = 1$).

For larger values of σ ($\sigma > 3$), the performance of both FAIR and FLEX degrades, and FLEX is not much better than FAIR in terms of response time. Initially this puzzled us, but looking at the experiments carefully, we realized this was an artifact of the fact that FAIR and FLEX calculate allocations for Maps and Reduces separately, and that Reduces are usually started before Maps finish in an attempt to save shuffle time. This is counter-productive in the following scenario: Suppose a large job is running its Map tasks. When Reduce slots become available Hadoop starts scheduling its Reduce tasks. However, if this job is sufficiently large (more Reduce tasks than slots), then it eventually takes over all the Reduce slots. Now, when other (potentially smaller) jobs arrive, both FAIR and FLEX will start scheduling their Map tasks (FLEX more aggressively than FAIR), but their Reduce slots cannot be scheduled until the large job can

finish its Reduces, which will only happen after all its Maps are completed. As the ratio between the size of large and small jobs increases, this is more and more likely to happen. One easy way of solving this problem is by separating the shuffle part of the Reduce phase from the computation part (*copy-compute splitting* [3],[5]), but this is not currently implemented in FAIR, and is, in any case, outside the scope of this paper. Nevertheless, we do no worse that FAIR even in these cases, and significantly better in all other scenarios.

We also compared FLEX with FAIR in experiments involving 20 and 40 jobs, respectively. We observed that FLEX continues to outperform FAIR by almost 20%. We also repeated our experiments using GridMix2 (a standard Hadoop benchmark with 172 jobs in each workload), with an average 30% improvement in response time over FAIR. Even in the largest experiments, we observed that FLEX takes less than 10 milliseconds per run, compared to an epoch interval of about 500 milliseconds. This illustrates that our algorithms and implementations scale as the problem size increases.

5 Related Work

While MapReduce scheduling is relatively new, there have already been a number of interesting papers written. We have room to outline just a few key papers. We have already discussed [3] and [5], which are most related to our current work. The latter paper focuses more on the delay scheduling idea for the assignment level scheme. In our terminology the delay scheduling technique slightly *relaxes* the allocation level decisions in order to achieve better host and/or rack locality. It does so by delaying, within time limits, slot assignments in jobs where there are no currently local tasks available.

Zaharia et al. [19] propose a revised speculative execution scheme for MapReduce systems. The concept of dealing with straggler tasks by occasionally launching speculative copies (restarts) of that task is fundamental to the MapReduce paradigm, but the LATE scheme introduced here is shown to have better performance than that of the default Hadoop scheme.

As noted earlier, Agrawal et al. [6] cleverly amortize the costs of multiple jobs sharing the same input data. Since this is a potentially very large fraction of the total time needed to perform MapReduce jobs, the technique could yield significant benefits. The shared scan scheme introduced attempts to optimize either the average or maximum stretch, two metrics described in this paper.

Israd et al. [20] propose the *Quincy* scheduler to achieve data locality and fairness for concurrent distributed jobs. They map the scheduling problem to a graph data-structure, where edge weights and capacities encode the competing demands of data locality and fairness; and use a min-cost flow algorithm to compute a schedule. They evaluate the performance based on several metrics like makespan, system normalized performance, slowdown norm, etc.

Saldholm and Lai [21] propose a resource allocation scheme for MapReduce workflows in order to improve the system efficiency metric – the average ratio of actual application performance in a shared system to the application perfor-

mance in a dedicated system. Their scheme prioritizes MapReduce jobs in order to remove bottlenecks in different stages of the workflows.

6 Conclusions and Future Work

In this paper we have described FLEX, a flexible and intelligent allocation scheme for MapReduce workloads. It is flexible in the sense that it can optimize towards any of a variety of standard scheduling metrics, such as average response time, makespan, stretch, deadline-based penalty functions, and even SLAs. It is intelligent in the sense that it can achieve performance close to the theoretical optimal, and this performance is robust with respect to the underlying workloads, even in a heterogeneous environment. The FLEX allocation scheduler can be regarded as either a standalone plug-in replacement to the FAIR allocation scheme in HFS or as an add-on module that works synergistically with it. We have compared the performance of FLEX with that of FAIR and FIFO in a wide variety of experiments.

There are a number of future work items which we are currently pursuing.

- While we have designed a very generic FLEX scheme capable of optimizing any minisum or minimax scheduling objective, we are continuing to improve the quality of some of the specific schemes. Such schemes are very metric-dependent and they can be notoriously difficult. Results in this area should be of both practical and theoretical importance.
- As with any algorithm, the output quality depends on the quality of the input. For best FLEX performance we need to improve the extrapolation techniques for estimating the remaining work in a job. The estimation routine needs to be accurate, dynamic, and more aware of past history.
- We have only begun to tackle the assignment level scheduling for MapReduce workloads. The assignment level should modestly relax the decisions made at the allocation level in the hopes of achieving better locality, memory usage, threading and multicore utilization. This is a very challenging problem from the system design perspective.
- The shared scan idea described in [6] is an exciting new direction for MapReduce. We have some possibly novel ideas for shared scans.
- We are interested in providing schedulers for flowgraphs consisting of multiple MapReduce jobs. Similar metric choices would apply to these problems, so it could be a natural and ultimate extension of our current work.

References

1. Dean, J., Ghemawat, S.: Mapreduce: Simplified Data Processing on Large Clusters. ACM Transactions on Computer Systems 51(1), 107–113 (2008)
2. Hadoop, `http://hadoop.apache.org`
3. Zaharia, M., Borthakur, D., Sarma, J., Elmeleegy, K., Schenker, S., Stoica, I.: Job Scheduling for Multi-user Mapreduce Clusters. Technical Report EECS-2009-55, UC Berkeley Technical Report (2009)

4. Hadoop Fair Scheduler Design Document, http://svn.apache.org/repos/asf/hadoop/mapreduce/trunk/src/contrib/fairscheduler/designdoc/fair_scheduler_design_doc.pdf
5. Zaharia, M., Borthakur, D., Sarma, J., Elmeleegy, K., Shenker, S., Stoica, I.: Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In: EuroSys '10: Proceedings of the 5th European Conference on Computer Systems, pp. 265–278. ACM, New York (2010)
6. Agrawal, P., Kifer, D., Olston, C.: Scheduling Shared Scans of Large Data Files. Proceedings of the VLDB Endowment 1(1), 958–969 (2008)
7. Pinedo, M.: Scheduling: Theory, Algorithms and Systems. Prentice Hall, Englewood Cliffs (1995)
8. Blazewicz, J., Ecker, K., Schmidt, G., Weglarz, J.: Scheduling in Computer and Manufacturing Systems. Springer-Verlag, Secaucus (1993)
9. J. Leung, E.: Handbook of Scheduling: Algorithms, Models, and Performance Analysis. CRC, Boca Raton (2004)
10. Coffman, E., Garey, M., Johnson, D., Tarjan, R.: Performance Bounds for Level-oriented Two-dimensional Packing Problems. SIAM Journal on Computing 9(4), 808–826 (1980)
11. Schwiegelshohn, U., Ludwig, W., Wolf, J., Turek, J., Yu, P.: Smart SMART Bounds for Weighted Response Time Scheduling. SIAM Journal on Computing 28, 237–253 (1999)
12. Turek, J., Wolf, J., Yu, P.: Approximate Algorithms for Scheduling Parallelizable Tasks. In: SPAA '92: Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 323–332. ACM, New York (1992)
13. Blazewicz, J., Kovalyov, M., Machowiak, M., Trystram, D., Weglarz, J.: Malleable Task Scheduling to Minimize the Makespan. Annals of Operations Research 129, 65–80 (2004)
14. Ibaraki, T., Katoh, N.: Resource Allocation Problems: Algorithmic Approaches. MIT Press, Cambridge (1988)
15. Fox, B.: Discrete Optimization via Marginal Analysis. Management Science 13, 210–216 (1966)
16. Galil, Z., Megiddo, N.: A Fast Selection Algorithm and the Problem of Optimum Distribution of Effort. Journal of the ACM 26(1), 58–64 (1979)
17. Frederickson, G., Johnson, D.: Generalized Selection and Ranking. In: STOC '80: Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing, pp. 420–428. ACM, New York (1980)
18. Jaql Query Language for JavaScript Object Notation, http://code.google.com/p/jaql
19. Zaharia, M., Konwinski, A., Joseph, A., Katz, R., Stoica, I.: Improving Mapreduce Performance in Heterogeneous Environments. In: 8th USENIX Symposium on Operating Systems Design and Implementation, pp. 29–42. USENIX Association (2008)
20. Isard, M., Prabhakaran, V., Curry, J., Wieder, U., Talwar, K., Goldberg, A.: Quincy: Fair Scheduling for Distributed Computing Clusters. In: SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, pp. 261–276. ACM, New York (2009)
21. Sandholm, T., Lai, K.: Mapreduce Optimization using Regulated Dynamic Prioritization. In: SIGMETRICS '09: Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems, pp. 299–310. ACM, New York (2009).