CIRCUMFLEX: A Scheduling Optimizer for MapReduce Workloads With Shared Scans

Joel Wolf
IBM T.J. Watson Research
Hawthorne, NY 10532
jlwolf@us.ibm.com

Andrey Balmin
IBM Almaden Research
San Jose, CA 95120
abalmin@us.ibm.com

Deepak Rajan
Lawrence Livermore Labs
Livermore, CA 94550
rdeepak@gmail.com

Kirsten Hildrum
IBM T.J. Watson Research
Hawthorne, NY 10532
hildrum@us.ibm.com

Rohit Khandekar
IBM T.J. Watson Research
Hawthorne, NY 10532
rohitk@us.ibm.com

Sujay Parekh
Bank of America
New York, NY 10080
sujay.parekh@gmail.com

Kun-Lung Wu
IBM T.J. Watson Research
Hawthorne, NY 10532
klwu@us.ibm.com

Rares Vernica
Hewlett-Packard Laboratories
Palo Alto, CA 94304
rares.vernica@hp.com

ABSTRACT

We consider MapReduce clusters designed to support multiple concurrent jobs, concentrating on environments in which the number of distinct datasets is modest relative to the number of jobs. Many datasets in such scenarios wind up being scanned by multiple concurrent Map phase jobs. As has been noticed previously, this scenario provides an opportunity for Map phase jobs to *cooperate*, sharing the scans of these datasets, and thus reducing the costs of such scans. Our paper has two main contributions. First, we present a novel and highly general method for sharing scans and thus amortizing their costs. This concept, which we call *cyclic piggybacking*, has a number of advantages over the more traditional *batching* scheme described in the literature. Second, we describe a significant but natural generalization of the recently introduced FLEX scheduler, for optimizing schedules within the context of this cyclic piggybacking paradigm. The overall approach, including both cyclic piggybacking and the FLEX generalization, is called CIRCUMFLEX. We demonstrate the excellent performance of CIRCUMFLEX via a variety of simulation experiments.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*scheduling*

General Terms

Algorithms, Theory

Keywords

Mapreduce, optimization, scheduling, shared scans

1. INTRODUCTION

Google's MapReduce [2] and its open source implementation Hadoop [3] have become highly popular in recent years. There are many reasons for this: simplicity of use, automatically parallelizable, naturally scalable, implementable on commodity hardware. Important built-in features include fault tolerance, communications and scheduling.

We focus on the problem of scheduling MapReduce work in this paper. Early MapReduce implementations, including Hadoop, employed *First In First Out* (FIFO) scheduling, which is known to have problems with job starvation in most environments. The Hadoop *Fair Scheduler* (FAIR) is a slot-based MapReduce scheme designed to avoid starvation by ensuring that each job is allocated at least some minimum number of slots [11, 10, 12]. (Slots are the basic unit of resource in a MapReduce environment.) By contrast, the *Flexible Scheduler* (FLEX) [8] can optimize a wide variety of standard scheduling theory metrics while ensuring the same minimum job slot guarantees as FAIR, and maximum job slot guarantees as well. The desired metric can be chosen from a menu that includes response time, stretch, and any of several metrics which reward or penalize job completion times compared to possible deadlines. FLEX does very well on all metrics, compared to the performance of FAIR and FIFO [8].

As first pointed out in [1], there is another, more subtle opportunity for improved scheduling in many common MapReduce environments: optimizing the amortized costs of shared scans of jobs in their Map phase. (We will use the term *Map jobs* for jobs in their Map phase from here on.) This is because the number of *distinct* datasets is often modest relative to the number of MapReduce jobs. A particular MapReduce dataset may be used simultaneously by multiple Map jobs. Also, for many MapReduce jobs the execution time cost of the Map phase is primarily that of scanning the data. Furthermore, the Map phase is often the most expensive (and sometimes the *only*) phase of a MapReduce job. Given all of

the above, there is considerable leverage in having Map jobs *cooperate* in some fashion on dataset scans, thus amortizing the costs of these scans.

In fact, [1] introduced two schedulers designed for MapReduce environments with shared scans. (The schedulers in question will henceforth be known collectively as AKO, for the authors, Agrawal, Kifer and Olston.) The AKO schemes determine, for each "popular" dataset, an optimized *batching window*. The idea is that Map jobs associated with this dataset delay starting work until this window expires, and the scans of all the delayed Map jobs are then *batched* together, so that a single scan can be performed for all of them. An AKO off-line optimization algorithm assumes Poisson arrivals of known rates for the jobs associated with each dataset, and uses a heuristic scheme based on Lagrange multipliers to find batching windows which minimize either the average or the maximum value of a metric somewhat analogous to stretch. There do seem to be some limitations to the AKO approach. First, batching forces the trade-off of efficiency for latency. Second, the assumption of Poisson arrivals allows the optimization to be performed, but is restrictive. Third, the assumption that the arrival rates of the jobs can be known in advance is problematic. Fourth, the schedule produced is inherently static, and thus cannot react dynamically to changing conditions. Fifth, the AKO scheme optimizes two variants of an unusual scheduling metric known as *perceived wait time (PWT)*. (Optimizing the more natural metric *stretch* was deemed too difficult.) Finally, the AKO scheme does not specifically deal with minimum slot allocation constraints.

Our goal in this paper is to provide an amortizing MapReduce scheduler without these limitations. We eliminate latency due to batching by employing a different shared scan concept which we call *cyclic piggybacking*. Since cyclic piggybacking treats the datasets circularly rather than linearly, the advantages of amortization are achieved without the disadvantages of latency: Map jobs can begin immediately. There is no need for Poisson assumptions, or accurate job arrival rate data. The scheme is dynamic rather than static, simply dealing with Map jobs as they arrive. Indeed, cyclic piggybacking itself does not involve any optimization at all. It can be performed entirely on the fly. Finding high quality slot allocations among the jobs does still require an optimization scheme, and our new scheme for this is a generalization of the FLEX algorithm that decomposes each Map job into multiple *subjobs* based on cyclic piggybacking. We then notice a natural precedence ordering that can be assumed among these subjobs in the optimal solution, and solve a scheduling problem with these constraints. A total of 11 of the 16 original FLEX scheduling metrics can be optimized. The 11 include the *minisum* metrics of average or total response time, stretch, and the *minimax* metrics of tardy jobs, tardiness, lateness and *Service Level Agreements* (SLAs).

Furthermore, the shared scan scenario considered above can be generalized significantly. Consider *semi-shared* scans for jobs in the case where these jobs scan arbitrarily overlapping datasets. Such a scenario would occur quite naturally, for instance, if one job scans a day of data, another scans a week and a third scans a month. (Note that weeks are not necessarily contained within a single month.) Considering the obvious Venn diagram, there is a natural partitioning of the union of the datasets based on the overlapping subsets of various cardinalities. Although the term becomes something of a misnomer in this generalized context, cyclic piggybacking can be easily extended to the case of semi-shared scans. It also turns out, at least for minimax objective functions, that there is a natural precedence order among the overlapping subsets, from more overlapped to less overlapped.

As with batching, notions similar to cyclic piggybacking have been proposed in other contexts, including the broadcast delivery of digital products [9] and databases [13]. To our knowledge, these have all been for the non-overlapping case. There seems to be very limited additional work in the area of sharing for MapReduce jobs. Note, however, that [5] examines a variety of sharing alternatives, including shared scans, in a MapReduce framework.

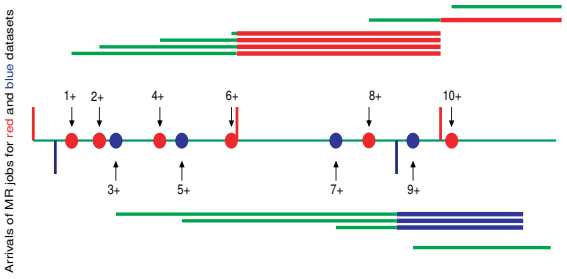
We call our shared scan scheduling scheme CIRCUMFLEX. We focus on the non-overlapping dataset scenario, but each portion of CIRCUMFLEX can be extended. The CIRCUMFLEX scheme involves two main contributions:

1. We adapt the cyclic piggybacking method to MapReduce jobs for amortizing the cost of the shared scans, and generalize it to handle *overlapping* datasets. This approach has a number of advantages over the batching scheme described in [1].
2. We notice that the various subjobs generated in this manner can be assumed to have a natural precedence order. This allows us to formulate and solve a scheduling problem which is a generalization of FLEX, respecting minimum slot and precedence constraints.

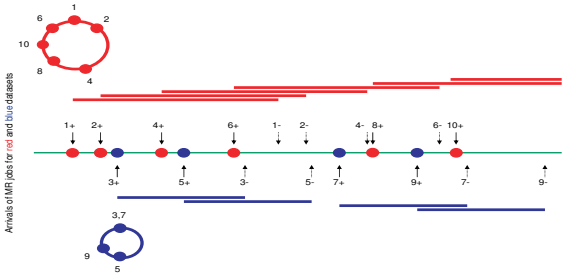
The remainder of this paper is organized as follows. In Section 2 we briefly describe cyclic piggybacking as an alternative employed by CIRCUMFLEX in favor of batching. Next, we present an overview of the CIRCUMFLEX scheduling algorithm in Section 3, though many details are relegated to a technical report [7] in the interests of space. In Section 4 we illustrate the excellent performance of CIRCUMFLEX. Since the metrics optimized by AKO and CIRCUMFLEX are disjoint, we compare the quality of CIRCUMFLEX with that of FLEX, and with a batching scheme of our own devising. (This scheme, called BATCH, actually employs a FLEX optimization algorithm on the batched datasets.) Thus, we are examining the performance benefits of high quality cyclic piggybacking schedules to schedules optimizing the same metric, both with and without batching. In [7], we also present the implementation details along with other supporting information including theoretical preliminaries and pseudo-codes. Conclusions are in Section 5.

2. CIRCUMFLEX

For ease of exposition we employ a simple example with two datasets. See Figure 1. One dataset is colored *red* and the other *blue*. Different Map jobs scan one or the other of these datasets. The horizontal, black line represents time. The figure also depicts the arrival of 10 Map jobs, 6 of which scan the red dataset and 4 of which scan the blue dataset, with a plus sign indicating their arrivals.



(a) Batching



(b) Cyclic Piggybacking

Figure 1: Scheduling schemes

2.1 Batching

Consider Figure 1(a). Optimized batching windows for both datasets are computed via a scheme such as [1]. The vertical red (blue) lines in the figure depict the temporal boundaries of the red (blue) batching windows. The red batching windows are shorter than the blue batching windows. The green lines represent the latency each job incurs, and the batching itself is indicated via red/blue lines. Red jobs 1, 2, 4 and 6 arrive before the end of the first batching window, and they are scanned together, as a batch with concurrency level 4, at the beginning of the second window. This is illustrated above the central horizontal time line. Note that job 1 incurs a latency of nearly the entire red batching window. Red job 8 arrives during the second batching window and is scanned alone at the beginning of the third window. Again, the green line illustrates the latency and the red line illustrates this trivial (concurrency level 1) batch. Red job 10 arrives during the third batching window, and its scan is not shown in the figure. The blue jobs 3, 5 and 7 are illustrated below the central time line and are scanned with concurrency level 3. The green lines indicates latency, as before, and the blue lines indicate scanning. Blue job 9, which arrives in a subsequent batching window, is not scanned in the figure.

The trade-off between latency and efficiency is illustrated in Figure 1(a). Longer batch windows allow for more jobs to be batched together, but the average latency of the jobs increases. The expected average latency is half the time in a batching window. This is the fundamental weakness of batching. Formally, the batching schemes described in [1] compute, for each dataset d, an optimized window batching window time T_d. The optimization algorithm assumes Poisson arrivals of jobs, as well as estimates of the rates associated with each dataset. It uses a Lagrange multiplier-based heuristic to find batching windows which minimize either the average or maximum PWT. Assuming a start time of 0, time is then partitioned for any dataset d into multiple windows of the form $[kT_d, (k+1)T_d)$ for each non-negative integer k. Any Map job arriving in window $[kT_d, (k+1)T_d)$ and involving dataset d is then performed using a batched scan starting at time $(k+1)T_d$.

2.2 Cyclic Piggybacking

See Figure 1(b). The example illustrated here is identical to that of Figure 1(a). If one thinks of the dataset as an ordered list of blocks the dataset can be viewed linearly. Though it is something of a simplification, one can think of blocks as being scanned in this line segment from left to right, from first block to last block. However, recognizing that there is no special meaning to the first or last blocks, one can also "glue" these two blocks together and view the dataset *cyclically*. A good analogy here would be a clock, with blocks corresponding to hours. So blocks can be scanned in a clockwise manner, starting with the first block – and as the scan reaches the last block it can simply begin again at the first block. In the figure the top-most point of a circle indicates the boundary between the first and last block. In the clock analogy this point is simply "midnight".

At time 1 the first red job arrives. This is illustrated both via the linear time line and in a cyclic view of the red dataset shown at the top. (A plus sign in the linear view indicates a job arrival, as before, while a minus sign refers to a job departure. In the cyclic view these occur at identical points on the circle.) The red Map job 1 starts to scan data in clockwise fashion from the midnight starting point denoted by 1. Subsequently a second red job arrives at time 2; this arrival is shown in both the linear and cyclic views. Considering the cyclic view, the clockwise arc from point 1 to point 2 involves previously scanned blocks, but job 2 can now *piggyback* its data scan of subsequent common data onto the remaining scan of job 1, amortizing costs. In the linear view one notices that the concurrency level increases to 2 once job 2 starts. When job 1 completes its scan, the remaining blocks of job 2 can be scanned. The concurrency level would be 1 during this portion of the scan, but notice that all of the concurrency levels depend dynamically on potential future arrivals. The subsequent arrival of a third, blue job causes the cyclic view of the blue dataset at the bottom, and the aligned linear view of job 3 below the black time-line. The arrival of red job 4 causes a concurrency level of 3 for the red dataset. The arrival of blue job 5 causes a concurrency level of 2 for the blue dataset. The arrival of red job 6 causes a concurrency level of 4 for the red dataset. Then the departure of job 3 occurs, reducing the concurrency level back to 1 for the blue dataset. Note that the eventual departure of blue job 5 and the subsequent arrival of blue job 7 causes a new single scan of the *first* blocks of the blue dataset again, and so forth.

Cyclic piggybacking suffers from none of the disadvantages noted for AKO in Section 1. The inherent differences in the relative efficiencies of batching and cyclic piggybacking can be measured in terms of the average concurrency levels. (Higher is more efficient.) We will have more to say about these efficiency levels in Section 4.

3. CIRCUMFLEX SCHEDULING

Assuming the MapReduce and theoretical scheduling preliminaries outlined in [7], we now describe the CIRCUMFLEX allocation scheduler. CIRCUMFLEX is an epoch-based malleable allocation layer scheduler for subjobs related by chain precedence. It eliminates the last two disadvantages noted for AKO. Specifically, it optimizes both average and maximum stretch, plus a number of other metrics more natural than those of AKO. (Of the 16 natural combinatorial choices handled by FLEX, the current version of CIRCUMFLEX can handle 11.) Additionally, CIRCUMFLEX handles the minimum constraints that are inherent in both FAIR and FLEX.

First, we justify the chain precedence assumption among the subjobs. Then, we describe the two steps of the CIRCUMFLEX scheduler, each of which generalizes FAIR – see [7] for details. The first step is to solve one of two optimization problems, depending on the precise metric chosen. In this step we wish to find a high-quality priority order for the subjobs. Actually, this priority ordering is also a topological order of the subjobs. (Recall that in the "job" context: A *topological order* is an ordering of the jobs which respects the precedence among the jobs. Thus $j_1 < j_2$ whenever $j_1 \prec j_2$.) All the optimization schemes provide as output a so-called *priority order* of the various subjobs, which is then used as input by the second step. The output of the second step is an optimized malleable schedule of the subjobs for the chosen metric in the cyclic piggybacking environment. We have designed a *Ready List Malleable Packing* scheme to solve this second problem.

3.1 Chain Precedence

Suppose, that there are K_d jobs scanning a given dataset d at a particular instant in time. This dataset gives rise to K_d subjobs, namely $\{(d,1),...,(d,K_d)\}$. Cyclic piggybacking has the effect of partitioning the dataset d into K_d+1 disjoint sets of blocks. The first set is relevant to all K_d jobs. The second set is still relevant to K_d-1 jobs, all but the first to arrive. (The first job has already scanned these blocks.) The third set is still relevant to K_d-2 jobs, all but the first two to arrive. Continuing in this nested manner, the K_dth subset is still relevant to 1 job, the last to arrive. The (K_d+1)st subset, which is empty if and only if the last job has *just* arrived, is no longer relevant. In general, subjob (d,k) is relevant to K_d-k+1 jobs.

We claim that the subjobs associated with each dataset d can be assumed to be related by chain precedence. In other words, $(d,1) \prec (d,2) \prec ... \prec (d,K_j-1) \prec (d,K_j)$. A simple interchange argument suffices to see this: No actual job can complete until *all* the blocks associated with its dataset have been scanned. And all of the possible scheduling metrics are functions of this completion time. If $1 \leq k_1 < k_2 \leq K_j$ it can help but cannot hurt the scheduling objective function to perform the scan of a block in subjob (d,k_1) before performing the scan of a block in subjob (d,k_2). This is because all of the original jobs which are relevant to subjob (d,k_2) are also relevant to subjob (d,k_1). After interchanging the block scans into the proper order, the result follows.

3.2 Finding a Priority Ordering

For finding a priority ordering, one of two schemes is employed, depending on the problem variant. The first case handles minisum average response time variants, specifically (weighted) average response time and average stretch. The second case handles all minimax metrics. Either scheme produces an interim schedule that maintains the precedence constraints (optimal for a *single* processor), and the sequencing of the jobs in this interim schedule determines the input ordering to the Ready List Malleable Packing scheme.

Weighted Average Response Time

This case is solved by a generalization of Smith's Rule [6]. (Smith's Rule optimally solves the problem of minimizing weighted average response time for independent jobs j with weight w_j and processing time p_j on a single processor by sequencing the jobs in non-decreasing order of the ratios p_j/w_j.) See [7] for the pseudo-code of this generalization.

Minimax Problems

This case is solved by a Backwards Dynamic Program, see [7] for the pseudo-code. This single processor scheme actually works for arbitrary jobs, any non-decreasing penalty function F_j and any precedence relation $\prec$.

3.3 Ready List Malleable Packing Scheme

The second step again works for arbitrary precedence constraints, so we describe it in generic job terminology. The translation to subjobs and chain precedence is easy.

The scheme inputs one of the output priority orderings from the previous subsection, as appropriate. It then employs *Ready List Malleable Packing scheme*. (A *ready list* is a dynamically maintained list of jobs which are ready to run at any given time. In other words, all precedence constraints must have been satisfied at the time.)

Pseudo-code for the Ready List Malleable Packing scheme appears in [7]. 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. 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 are 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 pseudo-code.) Given this set of job allocations, one of the jobs j completes 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 $\mathcal{L}$ is depleted and the output malleable schedule created.

4. SIMULATION EXPERIMENTS

In this section we describe a variety of simulation experiments designed to show the performance of CIRCUMFLEX. In the interests of space, we concentrate on average response time, average stretch and maximum stretch. The CIRCUMFLEX scheme can handle other metrics as well.

We compare FLEX and CIRCUMFLEX with a BATCH variant of our own design. (Recall that the AKO batching schemes described in [1] employed the PWT metric, and are basically off-line algorithms.) To compare batching fairly with FLEX and CIRCUMFLEX, we have devised a scheme which batches every dataset scan at the end of a fixed time window W, and then combined this with a FLEX scheduling algorithm applied to the resulting batches. The start times of the windows are offset evenly, depending on the dataset involved, to space the batch arrivals as equally as possible. This BATCH scheme seems to be in the spirit of AKO. On the negative side, its batching decisions are not as intelligent. On the positive side, it also optimizes the chosen metrics quite well.

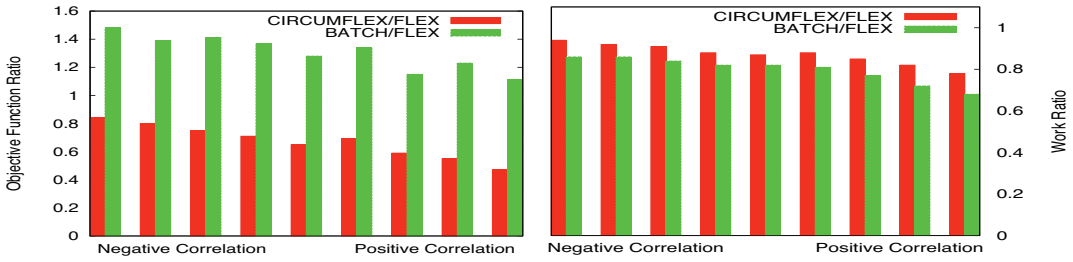
The experimental design is as follows. Each experiment simulated 1000 arrivals for $D = 20$ distinct datasets. The popularity of each dataset is chosen by sampling from the CDF of a Zipf-like distribution with parameter θ_1 equal to either 0, .25, .5, .75 or 1.0. (Zipf-like distributions [4] on a set of size D employ an integer parameter θ between 0 and 1. When $\theta = 1$ the distribution is purely Zipf, and when $\theta = 0$ the distribution is uniform. Zipf-like distributions thus span a wide variety of common skew patterns.) The arrival times themselves are chosen according to a Poisson distribution. The size of each dataset is chosen from a second Zipf-like distribution with parameter θ_2 equal to either 0, .25, .5, .75 or 1.0. The dataset popularity and size Zipf-like distributions are then positively or negatively correlated using a simple scheme. If the correlation is perfectly positive, the largest data sets are also the most popular. If the correlation is perfectly negative, the smallest data sets are most popular. If the correlation is zero, there is no relation between data set size and popularity. We use nine different choices of correlation. Combined with the 5 choices for each of θ_1 and θ_2 this gives us excellent coverage, with 225 parametric alternatives. We assumed a total of $S = 100$ Map slots, corresponding to a cluster of 25 nodes if there were 4 Map slots per node. We ran each experiment for a nominal total of $T = 100$ minutes, though we allowed the Map work to quiesce past this time. Finally, we scaled the dataset sizes so that the total Map times in a non-shared scenario corresponded to 90% utilization. (This utilization is the total time spent by the Map work divided by the product TS.)

In the experiments we computed new schedules for each of the alternative schemes upon each new arrival. These schedules were then followed precisely until the next arrival. Metrics for each arrival were computed, of course, based on the difference between the arrival and completion times.

We ran 10 repetitions of each experiment, taking averages and standard deviations. We recorded the ratio of the objective function value obtained using CIRCUMFLEX to that of FLEX, and similarly that of BATCH to FLEX. We also computed the ratio of the total work for CIRCUMFLEX and BATCH to that of FLEX. For CIRCUMFLEX we averaged the maximum number of concurrent subjobs over all datasets, and for BATCH we averaged the maximum number of jobs batched together over all datasets. We used batch windows per dataset of $W = 2$ minutes (yielding 50 or 51 such windows in 100 minutes) as a base case. Note that this adds an average latency of 1 minute to each arrival.

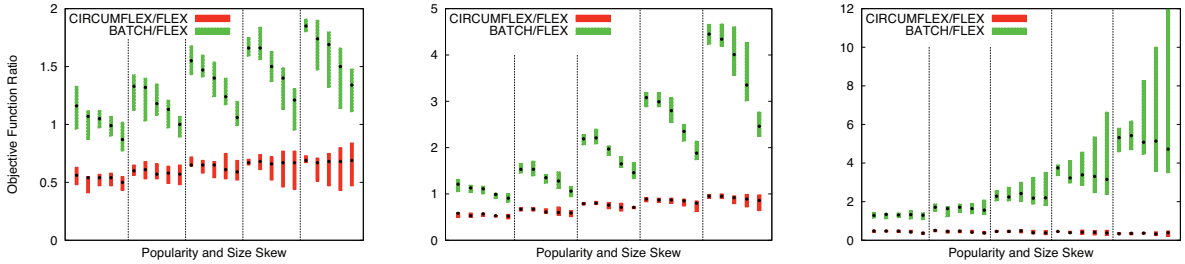
Figure 2(a) shows the average response time ratios of CIRCUMFLEX and BATCH to FLEX for the highly skewed case $\theta_1 = \theta_2 = 1$. Note that the performance of CIRCUMFLEX improves as the correlation ranges from perfectly negative to perfectly correlated. Compared with FLEX, which is optimizing average response time as well, CIRCUMFLEX average response times decrease from 81% down to 47%. There is reason to believe that dataset popularity and size might sometimes be negatively correlated, because, for example, many MapReduce jobs might involve the most recent day of data. But even in this case, CIRCUMFLEX performs 19% better than FLEX. (In the absence of shared scans, FLEX does very well on all metrics, compared to the performance of FAIR and FIFO [8].) On the other hand, BATCH always performs worse than FLEX, because of the trade-offs involved. In the case of perfectly negative correlation, BATCH is 48% worse than FLEX. In the perfectly positive correlation case, BATCH is still 11% worse. Standard deviations of these ratios (not shown) are very modest for CIRCUMFLEX, relatively less so for BATCH. This is an indication of the robustness of the CIRCUMFLEX scheme. Figure 2(b) shows the comparable work ratios for this same example. Both ratios generally decrease from left to right, as one would expect. By definition, CIRCUMFLEX does more work than BATCH but less work than FLEX. The maximum number of concurrent CIRCUMFLEX subjobs averaged 2.8 in the most negatively correlated case, and 4.3 in the most positively correlated case. The maximum number of jobs batched together ranged from 4.8 to 5.0, also indicative of the greater efficiency of BATCH.

Figure 3(a) shows the ranges of the CIRCUMFLEX to FLEX ratio and the BATCH to FLEX ratio for all 25 parametric choices of θ_1 and θ_2. These are arranged in 5 "planes" of 5 values each. The left-most group of 5 all correspond to $\theta_1 = 0$, and individually to $\theta_2 = .25 * \tau$, where τ ranges from 0 to 4. (Thus the detailed data in Figure 2(a) is shown in summary form in the right-most pair of bars in Figure 3(a): The minimum, median and maximum value across all 9 correlation parameters is shown for each objective function ratio, for both CIRCUMFLEX and BATCH. Note that this is possible to show *because* the ratios never overlap. The worst CIRCUMFLEX ratio is always better than the best BATCH ratio for any particular choice of θ_1 and θ_2. Indeed, the worst CIRCUMFLEX to FLEX ratio (.98) in the entire graph is essentially identical to the best BATCH to FLEX ratio overall (.97). It is also clear that the ratios for CIRCUMFLEX are much more consistent and tightly clustered across the parametric choices than the ratios for BATCH. CIRCUMFLEX always performs better than FLEX, and BATCH nearly always performs worse than FLEX: The extra average latency of 1 minute is too great an impediment for BATCH to overcome.



(a) Value (b) Work

Figure 2: Average Response Time Ratios, $\theta_1 = \theta_2 = 1$



(a) Average Response Time (b) Average Stretch (c) Maximum Stretch

Figure 3: Ratio Summaries

Figures 3(b) and 3(c) illustrate the corresponding results for average and maximum stretch, respectively. (For maximum stretch the metric is calculated as the largest value over all 1000 arrivals.) In both of these metrics the relative performance of CIRCUMFLEX is even stronger than it is for average response time. A small dataset scan which arrives early in a batching window causes a high average stretch value and a *very* high maximum stretch value. Both the average and maximum stretch ratios vary widely, depending on the parametric choices. They are never nearly as good as FLEX. On the other hand, the performance of CIRCUMFLEX in both cases is extremely predictable, and always much better than that of FLEX. In the case of maximum stretch the performance differences are dramatic.

The scheme in [1] made the reasonable simplifying assumption that nearly all the work of the Map phase is in the scanning, so that the subsequent computational work can be ignored. We choose the same as a base case for the cleanest possible exposition of the benefits of CIRCUMFLEX. Our simulation experiments with varying the the fraction of Map phase work which is due to the scans behave entirely in the expected manner, so we do not show them.

5. CONCLUSIONS

In this paper we have introduced CIRCUMFLEX, a new scheduler for Map phase jobs in MapReduce environments. This scheme has major advantages over the previous AKO scheme. CIRCUMFLEX is a two stage approach. In the first stage, *cyclic piggybacking* provides a natural and effective technique for amortizing the costs of shared scans. Jobs are decomposed into a number of subjobs, which are related by natural precedence constraints. In the second stage, the resulting precedence scheduling problem is solved for any of a variety of metrics, including average response time, average stretch and maximum stretch.

Our experiments comparing CIRCUMFLEX with FLEX and BATCH show that the benefits of CIRCUMFLEX can be large. Moreover, CIRCUMFLEX works well in a general overlapping dataset environment, resulting in a number of subjobs related by more arbitrary precedence constraints.

6. ACKNOWLEDGEMENTS

Rares Vernica is partially supported by NSF grant 0910989.

7. REFERENCES

[1] P. Agrawal, D. Kifer, and C. Olston. Scheduling shared scans of large data files. In *Proceedings of VLDB*, 2008.

[2] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *ACM Transactions on Computer Systems*, 51(1):107–113, 2008.

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

[4] D. Knuth. *The Art of Computer Programming*. Addison-Wesley, 1998.

[5] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Mrshare: sharing across multiple queries in mapreduce. *Proc. VLDB Endow.*, 3(1-2):494–505, 2010.

[6] M. Pinedo. *Scheduling: Theory, Algorithms and Systems*. Prentice Hall, 1995.

[7] J. Wolf, A. Balmin, D. Rajan, K. Hildrum, R. Khandekar, S. Parekh, K.-L. Wu, and R. Vernica. Circumflex: A scheduling optimizer for mapreduce workloads involving shared scans. Technical Report RC25118, IBM Technical Report, 2011.

[8] J. L. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, and A. Balmin. Flex: A slot allocation scheduling optimizer for mapreduce workloads. In *Proceedings of Middleware*, 2010.

[9] J. L. Wolf, M. S. Squillante, J. J. Turek, P. S. Yu, and J. Sethuraman. Scheduling algorithms for the broadcast delivery of digital products. *IEEE Trans. on Knowl. and Data Eng.*, 13(5):721–741, 2001.

[10] M. Zaharia. Hadoop fair scheduler design document, http://svn.apache.org/repos/asf/hadoop/mapreduce/trunk/src/contrib/fairscheduler/designdoc/fair_scheduler_design_doc.pdf.

[11] M. Zaharia, D. Borthakur, J. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user mapreduce clusters. Technical Report EECS-2009-55, UC Berkeley Technical Report, 2009.

[12] M. Zaharia, D. Borthakur, J. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of EuroSys*, 2010.

[13] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative scans: dynamic bandwidth sharing in a dbms. In *Proceedings of VLDB*, 2007.