RC25118 (W1102-070) February 28, 2011
Computer Science

IBM Research Report

CIRCUMFLEX: A Scheduling Optimizer for MapReduce Workloads Involving Shared Scans

Joel Wolf[1], Andrey Balmin[2], Deepak Rajan[1], Kirsten Hildrum[1], Rohit Khandekar[1], Sujay Parekh[1], Kun-Lung Wu[1], Rares Vernica[3]

[1]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

[2]IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099
USA

[3]University of California
Irvine, CA 92697

CIRCUMFLEX: A Scheduling Optimizer for MapReduce Workloads Involving Shared Scans

Joel Wolf* Andrey Balmin† Deepak Rajan* Kirsten Hildrum*

Rohit Khandekar* Sujay Parekh* Kun-Lung Wu* Rares Vernica‡

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 will 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 new, novel and highly general method for sharing scans and thus amortizing their costs. This concept, which we will call *cyclic piggybacking*, is an alternative to the more traditional *batching* scheme described in the literature, and seems to have a number of advantages over that scheme. Second, we describe a method for optimizing schedules within the context of this cyclic piggybacking paradigm. This scheme, a significant but natural generalization of the recently introduced FLEX scheduler for MapReduce, can optimize a wide variety of metrics. Such cost functions include average response time, average stretch, and any minimax-type metric. 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, and we describe a practical implementation strategy.

Keywords: MapReduce, Shared Scans, Scheduling, Allocation, Optimization, Amortization

1. INTRODUCTION

Google's MapReduce [1] and its open source implementation Hadoop [2] have become highly popular in recent years. There are many reasons for this success: MapReduce is simple to use, even for users of limited sophistication. It is automatically parallelizable, naturally scalable, and can be implemented on large clusters of commodity hardware. Important built-in features include fault tolerance, communications and scheduling.

We focus on the problem of scheduling MapReduce work in this paper, and more specifically on a specialized but common and important variant first introduced by Agrawal et. al [3]: optimizing the amortized costs of shared scans of Map jobs. Before describing this "shared scan" problem in detail, however, it will help to provide a brief overview of some popular *generic* MapReduce schedulers. Understanding the original MapReduce scheduling problem and its history will motivate our approach to the shared scan special case, and put our solution to that problem in the proper context.

Early MapReduce implementations, including Hadoop, employed *First In First Out* (FIFO) scheduling. But while simple and almost universally applicable, FIFO is known to have problems with job starvation in most environments. A large job can "starve" a small job which arrives even modestly later. Worse, if the large job was a batch submission and the small job was an ad-hoc query, the exact completion time of the large job would not be particularly important, while the completion time of the small job would be.

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 [4–6]. (Slots are the basic unit of resource in a MapReduce environment.) But FAIR does not attempt to actually optimize any specific scheduling metric. And a schedule designed to optimize one metric will generally perform quite differently from a FAIR schedule, or one designed to optimize another metric.

By contrast, the *Flexible Scheduler* (FLEX) [7] can optimize a wide variety of standard scheduling theory metrics while ensuring the same minimum job slot guarantees as in 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. This last includes the number of tardy jobs, tardiness, lateness and also *Service Level Agreements* (SLAs). There are 16 combinatorial choices in all, because the metrics can be either weighted or unweighted, and one can optimize either their average (or, equivalently, from the perspective of optimization, their sum) across all jobs, or the maximum such

*IBM T. J. Watson Research, Hawthorne, NY 10532. {jlwolf,drajan, hildrum,rohitk,sujay,klwu}@us.ibm.com

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

‡University of California, Irvine, CA 92697. (Work done at IBM Almaden.) rares@ics.uci.edu

value. Moreover, FLEX can be regarded as an add-on module that sits on top of FAIR, works synergistically with it, and employs its extensive infrastructure.

But, as pointed out in [3], there is another, more subtle opportunity for scheduling optimization in many common MapReduce environments. This is because the number of *distinct* datasets is often modest relative to the number of MapReduce jobs. Consider a particular MapReduce dataset. It will not be uncommon to have multiple simultaneously active jobs in their Map phases which need to scan this dataset. 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 seems to be considerable leverage in having Map phase jobs *cooperate* in some fashion on dataset scans, thus amortizing the costs of these scans.

In fact, [3] introduced a pair of schedulers designed for MapReduce environments with shared scans. (The two schedulers in question will henceforth be known collectively as AKO, for the authors, Agarwal, Kifer and Olsten.) The AKO schemes determine, for each "popular" dataset, an optimized *batching window*. The idea is that Map jobs associated with this dataset will delay starting work until this window expires, and the scans of all the delayed Map jobs will then be *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, which we will now enumerate.

1. Batching forces the tradeoff of efficiency for latency. In other words, batching a number of scans together causes them to be delayed. A larger batching window is more efficient but causes a longer average delay.

2. The assumption of Poisson arrivals allows the optimization to be performed, but it is restrictive. Jobs do not always arrive according to such a distribution.

3. The assumption that the arrival rates of the jobs can be known in advance is similarly problematic. These estimates will likely be fairly rough approximations, and thus may affect the quality of the optimization solution.

4. The schedule produced is inherently static. The scheme therefore cannot react dynamically to changing conditions. (In fairness, the AKO implementation is more dynamic than the decisions produced by the scheduler itself.)

5. The AKO scheme optimizes two variants of a somewhat unusual scheduling metric known as *perceived wait time (PWT)*. This is defined as the difference between actual response time of a job and its minimum possible response time. (By contrast, the more natural metric *stretch*, to which the authors briefly allude, is the ratio of the two terms. But it is noted that optimizing stretch proved problematic for them.) So AKO optimizes either average and maximum PWT.

6. While average and maximum PWT are metrics which try to philosophically capture the spirit of fairness, the AKO scheme does not specifically deal with minimum slot allocation constraints. These minimum constraints are a key guarantee in both FAIR and FLEX.

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 will be called *cyclic piggybacking*. Because cyclic piggybacking essentially treats the dataset circularly rather than linearly, the advantages of amortization are achieved without the disadvantages of any latency: Map jobs can begin immediately. There is no need for Poisson assumptions. There is no need to have 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, of course, does still require an optimization scheme, and our new scheme for this is a generalization of the FLEX algorithm in that it decomposes each Map job into multiple *subjobs* based on cyclic piggybacking. We then notice a natural chain 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. (There may be heuristics available for the other 5.) 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). All of these can either be weighted or unweighted.

We will call our full shared scan scheduling scheme CIRCUMFLEX. In summary, the CIRCUMFLEX scheme described in this paper involves two main contributions.

First, we provide a novel method called cyclic piggybacking for amortizing the cost of the shared scans of a set of multiple MapReduce jobs involving common datasets. This approach has a number of advantages over the batching scheme described in [3].

Second, we notice that one can assume that the various subjobs generated in this manner can be assumed to have a natural chain precedence order. This allows us to formulate a scheduling problem which is a generalization of FLEX. We optimize the scheduling of the subjobs with respect to any of a choice of 11 standard metrics, while respecting minimum slot and chain precedence constraints.

Actually, 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, perhaps within one or more directories. 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, the point is that 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. This is not a *chain* precedence scenario, but at least for the minimax case the CIRCUMFLEX scheduling scheme will work as is. So one can optimize any of 8 separate metrics in this general scenario, including maximum stretch.

While we will focus, for ease of exposition, on the non-overlapping dataset scenario, we will point out the manner in which each portion of CIRCUMFLEX can be extended as we proceed. We should note that notions similar to what we call cyclic piggybacking have been proposed in various other contexts, for example in the broadcast delivery of digital products [8] and in databases [9]. To our knowledge, these have all been for the non-overlapping case: There does not appear to have been any work on the general scenario.

Finally, we examine the performance of CIRCUMFLEX by means of simulation experiments. Since the metrics optimized by AKO and CIRCUMFLEX are disjoint, we choose not to compare them directly. That would seem unfair to one scheme or the other. It would also be unfair to compare CIRCUMFLEX with FAIR or FIFO, because the latter two do not attempt to optimize any particular metric. We therefore compare instead the quality of CIRCUMFLEX with that of FLEX, and similarly 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.

There seeems to be very limited additional work in the area of sharing for MapReduce jobs. Note, however, that [10] examines a variety of sharing alternatives, including shared scans, in a MapReduce framework.

The remainder of this paper is organized as follows. In Section 2 we describe cyclic piggybacking as an alternative employed by CIRCUMFLEX in favor of batching. The CIRCUMFLEX scheduling algorithm, including a discussion of chain precedence, is given in Section 3. In Section 4 we illustrate the excellent performance of CIRCUMFLEX relative to FLEX and BATCH in simulation experiments. Conclusions are in Section 5. We also include two appendices. Appendix A gives some preliminaries involving MapReduce and theoretical scheduling, essentially background information to understand Section 3. Appendix B gives CIRCUMFLEX pseudocode. Appendix C describes a MapReuce implementation of shared scans.

2. CIRCUMFLEX CYCLIC PIGGYBACKING

In this section we describe cyclic piggybacking and how it compares to the batching approach given in [3]. The most instructive way to describe both batching and cyclic piggybacking is via a common, simple example. We will do this before giving a formal definition of each approach.

2.1 Batching

Consider Figure 1. In this example there are two datasets. One is colored *red* and the other *blue*. Different Map jobs scan one or the other of these datasets. The horizontal, black line in the figure represents time. Optimized batching windows for both datasets are computed via a scheme such as [3]. The vertical red lines in the figure depict the (temporal) boundaries of the red batching windows, while the vertical blue lines depict the boundaries of the blue batching windows. The red batching windows are shorter than the blue batching windows. 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. The jobs are numbered, with a plus sign indicating their arrivals. Red jobs 1, 2, 4 and 6 arrive before the end of the first batching window, and they

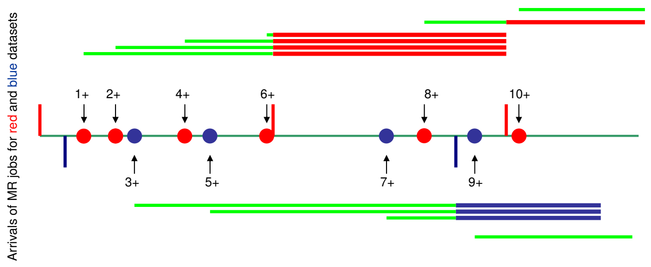

Figure 1: Batching

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. The green lines represent the latency each red job incurs, and the batching itself is indicated via 4 red lines. 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 tradeoff between latency and efficiency is clearly illustrated in Figure 1. Longer batch windows allow for more jobs to be batched together, but the average latency of the jobs increases accordingly. The expected average latency is half the time in a batching window. This is the fundamental weakness of the batching approach.

Formally, the batching schemes described in [3] 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

Our new notion of *cyclic piggybacking* is best understood by considering Figure 2. The example illustrated here is identical to that of Figure 1. So there are again 10 total jobs, and each job must scan one of two datasets. As before, 6 jobs scan the first dataset and 4 jobs scan the second.

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,

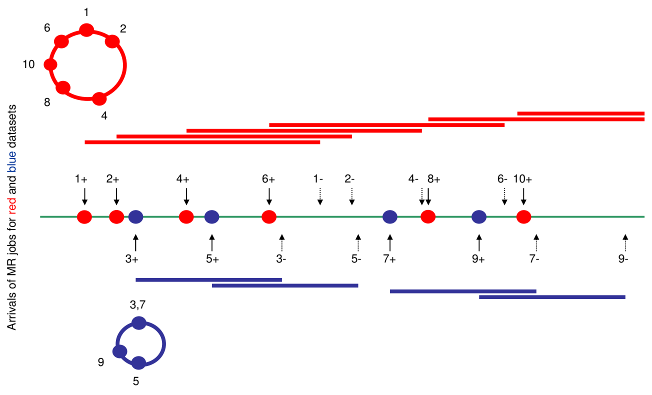

Figure 2: Cyclic Piggybacking

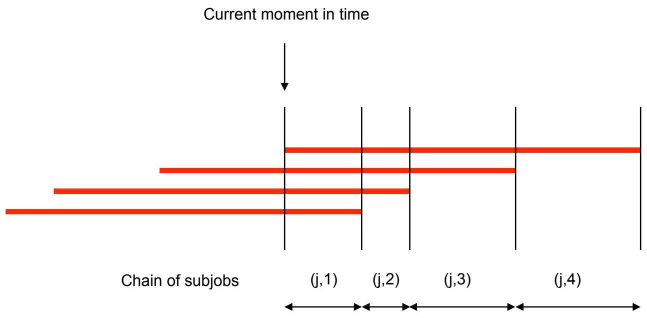

Figure 3: Creation of Chain Precedence Subjobs

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 will indicate 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 will indicate a job arrival, as before, while a minus sign will refer 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. The (aligned) linear view of job 1 is shown above the black central timeline.

Subsequently a second red job arrives at time 2. Again, 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 timeline.

This process continues. The arrival of red job 4 causes a concurency 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.

Consider Figure 3, which illustrates the state of affairs for the red dataset at precisely the time when the 4th red job j arrives. At this moment in time 3 red jobs are in the process of being scanned. Denote the red dataset by d. The figure shows a decomposition of the current remaining work for the red dataset into 4 subjobs, labeled $(d,1),...(d,4)$. Subjob $(d,1)$ starts at the current time, and completes at the end of the scan of the 1st red job. The concurrency level is 4. Subjob $(d,2)$ starts when subjob $(d,1)$ ends, and completes at the end of the scan of the 2nd red job. The concurrency level is 3. Formally, subjob (d,k) starts when subjob $(d,k-1)$ ends, and completes at the end of the scan of the kth red job. It has concurrency level K_d-k+1, where K_d is the total number of currently active red jobs. (In this case, $K_d=4$.)

A few rather delicate points should be made here. First, note that each subjob belongs to both a Map job and a dataset. The notation clearly references the latter rather than the former. Because calling (d,k) a subjob of job d overuses the term "job", we will avoid this. In other words, we will reserve the word job for the Map jobs themselves. Note also that the notation is quite dynamic – it changes with job arrivals. Finally, there could also many *synonyms* for subjobs, depending on the job under consideration: If jobs j_1 and j_2 represent two succesive scans for the same dataset d, then subjob (d,k) viewed via the first job will be the same as subjob $(d,k-1)$ viewed via the second. To avoid this unsatisfactory notation we will insist on using the subjob terminology associated with the last job to arrive for a particular dataset. (See Figure 3.) We will discover in Section 3, via a simple interchange argument, that the optimal way to complete these subjobs is in sequential order. In the figure this means that subjob $(d,1)$ should be completed before subjob $(d,2)$, and so on.

Defined in this manner, cyclic piggybacking suffers from none of the first four disadvantages noted for AKO in Section 1. There is no built-in latency, since jobs are ready to start their Map phase scans instantly. The arrival distribution and rates are irrelevant. The design is completely dynamic, reacting on the fly to any new job arrivals: No optimization algorithm needs to be executed.

Of course, nothing is ever quite as simple as it might appear at first. The description of cyclic piggybacking above is a bit too simple in several ways. We introduce these issues now, resolving each of them in subsequent sections.

First, by describing the scanning of datasets in either linear or cyclic terms we have essentially implied a discrete block ordering that does not really exist. This simplification was strictly for purposes of exposition. In fact, no real order for the blocks exists, except that inherently implied by the optimality of the subjob sequencing. There is great flexibility built into the MapReduce paradigm: The actual block scan execution order within a subjob will depend on an entirely different layer of the MapReduce scheduler. (This so-called *assignment* layer, which we will describe in Appendix A, considers issues such as data locality when assigning a

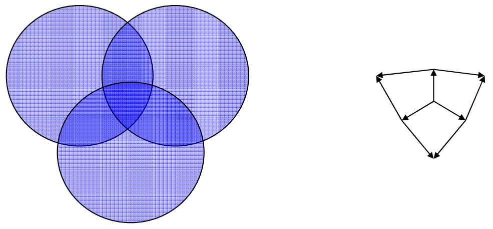

Figure 4: Semi-Shared Scans

Map scan to an available slot on a particular node.) In any reasonable implementation of the cyclic piggybacking scheme there will simply be a bit for each active job and relevant block, which notes whether or not that block has already been scanned for that job. Within a subjob, the scanning order of the blocks is essentially immaterial.

Although we say that the blocks are *ready* immediately for scheduling, we do not necessarily have the resources to schedule that work. We have not yet introduced our MapReduce scheduler, which will attempt to optimally allocate the slot resources to the various jobs and subjobs. The bottom line is that our description of cyclic piggybacking thus far blurs this detail. We will describe MapReduce scheduling preliminaries in Appendix A and give the details of our CIRCUMFLEX scheduler in Section 3.

There are inherent differences in the relative efficiencies of batching and cyclic piggybacking. In both cases these can be measured in terms of the average concurrency levels. (Higher is more efficient.) We will have more to say about the efficiency of cyclic piggybacking in Section 4.

In the case of semi-shared scans the subjobs are modestly more complex. Consider the left-hand side of Figure 4, representing the remaining scans required for three separate but overlapping arrivals at the time of the third arrival. The 6 subjobs created in this example correspond to the subsets in the figure. It will turn out that the optimal way to complete these subjobs is from most overlapped to least overlapped subset, yielding the precedence diagram show in the right-hand side of the figure.

3. CIRCUMFLEX SCHEDULING

Assuming the MapReduce and theoretical scheduling preliminaries outlined in Appendix A, we are in a position to 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 described in Appendix A and 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.

In this section we will first justify the chain precedence assumption among the subjobs. Then we will describe the two steps of the CIRCUMFLEX scheduler. The first step is to solve one of two optimization problems, depending on the precise metric chosen. In one case the optimization problem can be solved by a *Generalized Smith's Rule* scheme. In the other case it can be solved by a *Backwards Dynamic Programming* scheme. Either of these schemes provides as output a so-called *priority order* of the various subjobs, which is then used as input by the second step. We have designed a *Ready List Malleable Packing* scheme to solve this problem. The output of this second step is an optimized malleable schedule of the subjobs for the chosen metric in the cyclic piggybacking environment.

3.1 Chain Precedence

Suppose, as before, that there are K_d jobs scanning a given dataset d at a particular instant in time. We have seen that 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 will be relevant to all K_d jobs. The second set will still be relevant to K_d-1 jobs, all but the first to arrive. (The first job will have already scanned these blocks.) The third set will still be relevant to K_d-2 jobs, all but the first two to arrive. Continuing in this nested manner, the K_dth subset will still be relevant to 1 job, the last to arrive. The (K_d+1)st subset, which will be empty if and only if the last job has *just* arrived, will no longer be relevant. In general, subjob (d,k) is relevant to K_d-k+1 jobs.

We claim that the subjobs associated with each dataset j can be assumed to be related by chain precedence. In other words, $(j,1) \prec (j,2) \prec ... \prec (j,K_j-1) \prec (j,K_j)$. 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 $1 \le k_1 < k_2 \le K_j$ 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. Again, see Figure 3, where we can assume that $(d,1) \prec (d,2) \prec (d,3) \prec (d,4)$.

3.2 Finding a Priority Ordering

In this first step we wish to find a high-quality priority order for the subjobs. Actually, this priority ordering will also be a topological order of the subjobs. (This notion applies to arbitrary precedence constraints rather than just chain precedence, so we recall the definition in the traditional "job" context: A *topological order* is an ordering of the jobs which respects the precedence among the jobs. Thus $j_1 < j_2$ whenever $j_1 \prec j_2$.)

One of two schemes will be employed, depending on the problem variant. The first case handles minisum average response time variants, specifically average response time, weighted average response time and average stretch. The second case handles all minimax metrics. Either will produce an optimal interim (hypothetical) schedule, and the completion times of the jobs in this interim schedule will determine the input ordering to the Ready List Malleable Packing scheme.

Weighted Average Response Time: This case is solved by a generalized version of Smith's Rule [12]. (Smith's Rule optimally solves the generic problem of minimizing weighted average response time for independent jobs j with weight w_j and processing time p_j by sequencing the jobs in order of the ratios p_j/w_j.) The pseudo-code for the Generalized Smith's Rule is given in Appendix B.1. It will be clear that this does, indeed, represent a generalization of Smith's rule to the case of chain precedence subjobs. See line 7, in particular, where the partial ratios of sums replace the traditional Smith Rule ratios. The proper ordering is achieved via line 8.

Since chain precedence is maintained in the resulting sequence of subjobs, the ordering of the completion times of the subjobs is a topological ordering as well, and this priority ordering is input to the second step.

Minimax Problems: This case is solved by a Backwards Dynamic Program. The pseudo-code for this case is given in Appendix B.2. This scheme actually works for arbitrary jobs, any non-decreasing penalty function F_j and any precedence relation $\prec$. *Chain* precedence is not required. So we have described the pseudocode more generically.

Since precedence is maintained in the resulting sequence of subjobs, the ordering of the completion times is again a topological ordering, and this priority ordering is input to the second step.

3.3 Ready List Malleable Packing Scheme

The second step again works for arbitrary precedence constraints, so we will 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.)

The pseudo-code for the Ready List Malleable Packing scheme appears in Appendix B.3. 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 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 pseudo-code.) 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 output malleable schedule created.

In the semi-shared case the interchange argument yields a more general precedence relationship between the subjobs created. As before, no job can complete until all the blocks associated with it have been scanned. Considering Figure 4, it can help but cannot hurt to scan the three-way intersection before two-way intersections, and the two-way before the one-way. The resulting precedence graph is shown. (The arrows indicate precedence.) Of the three crucial algorithms of CIRCUMFLEX only the Generalized Smith's Rule breaks down. It requires chain precedence. But the Backwards Dynamic Programming and Ready List Malleable Packing schemes carry over unchanged. So in the semi-shared scan case we can handle minimax metrics, but not weighted average response time metrics.

4. SIMULATION EXPERIMENTS

In this section we describe a variety of simulation experiments designed to show the performance of CIRCUMFLEX. We will concentrate on three metrics, namely average response time, average stretch and maximum stretch. The CIRCUMFLEX scheme can, as noted, handle all other minimax metrics as well. In the interests of space we have chosen maximum stretch as representative and most important of the minimax metrics, but maximum tardiness or lateness runs yield comparable results.

We will compare FLEX and CIRCUMFLEX with a BATCH variant of our own design. (Recall that the AKO batching scheme described in [3] employed the maximum PWT metric, and was basically an off-line algorithm.) 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 were 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 was as follows. Each experiment simulated 1000 arrivals for a total of $D = 20$ distinct datasets. The popularity of each dataset was 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 [15] 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 were chosen according to a Poisson distribution. The size of each dataset was 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 were then positively or negatively correlated using a simple scheme, as follows. For positive correlations, we employ a parameter κ between 1 and the number of datasets D. Assuming the datasets are indexed in order of decreasing popularity, we choose a random number κ_1 between 1 and κ, and assign the most popular dataset to be the κ_1th largest dataset. We then choose another random number κ_2 between 1 and $\kappa + 1$, with κ_1 excluded, and assign the 2nd most popular dataset to be the κ_2th largest dataset. Continuing in this manner for a total of D steps we determine the complete relationship between dataset popularity and size. If κ is 1, the correlation is perfectly positive. If κ is D, the correlation is fully random. If

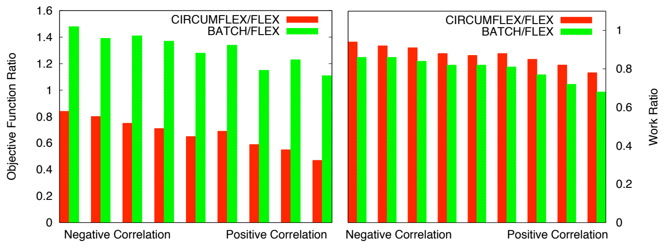

Figure 5: Average Response Time (a) Value and (b) Work Ratios, $\theta_1 = \theta_2 = 1$

we start instead with the datsets indexed in order of increasing popularity we range from perfectly negatively correlated to fully random. In our experiments we increment κ in steps of 5 for both positive and negative correlations yielding a total of 9 parametric choices from perfectly negatively correlated (labeled 0) through randomly correlated (labeled 4) to perfectly positively correlated (labeled 8). 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. The scheme in [3] 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 assumed, more generally, a parameter f between 0 and 1 which determines the fraction of Map phase work which is due to the scans. If $f = 1$ we have the pure scan scenario assumed by [3]. Finally, we scaled the dataset sizes so that the total Map times in a non-shared scenario corresponded to a fixed utilization ρ. (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 simulation experiment, taking averages and standard deviations. We recorded the ratio of the objective function value obtained using CIRCUMFLEX to that of FLEX, and similarly the objective function ratio 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 the parameters $\rho = .9$, $f = 1$ and a 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 5(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. (Compare also the performance of FLEX with that of FAIR [7]: FLEX does very well on all metrics.) On the other hand, BATCH always performs worse than FLEX, because of the tradeoffs 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) very modest for CIRCUMFLEX, relatively less so for BATCH. This is an indication of the robustness of the CIRCUMFLEX scheme. Figure 5(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.

Next consider Figure 6(a), which 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. Rather than display this as a 3-dimensional graph, which is difficult to see, we have chosen to arrange these in 5 "planes" of 5 values each. So 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 5(a) is shown in summary form in the right-most pair of bars in Figure 6(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 BATCH: The extra average latency of 1 minute is too great an impediment for BATCH to overcome.

Figures 6(b) and 6(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 was for average response time. A small dataset scan which arrives early in a batching window will cause 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.

We have considered a base case of $\rho = .9$, but note that, for BATCH and CIRCUMFLEX, the scan portion of the work in the Map phase is largely unaffected by values of ρ even much bigger than 1. If there are sufficient resources in the cluster to batch every dataset during every window, or to scan each dataset continually in the CIRCUMFLEX cyclic piggybacking

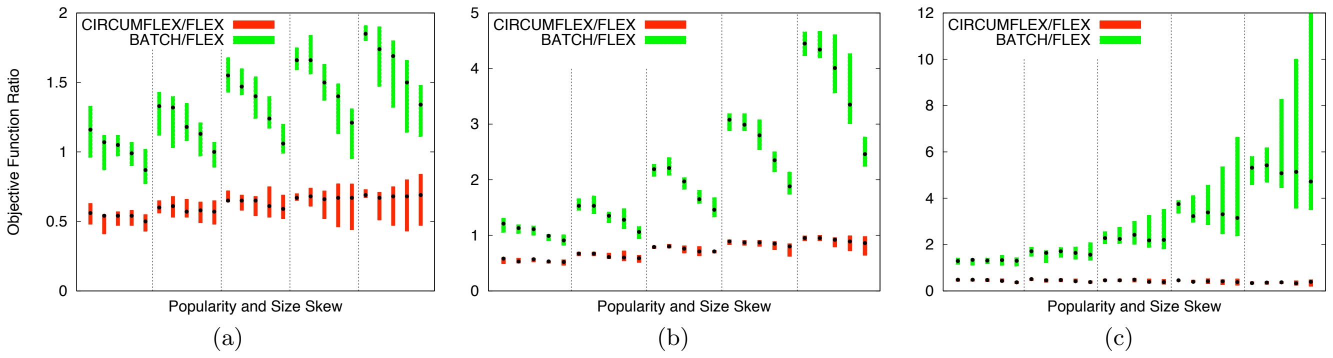

Figure 6: (a) Average Response Time, (b) Average Stretch and (c) Maximum Stretch Ratio Summaries

case, there is actually no theoretical limit to the value of ρ if $f = 1$. The scan work simply does not grow. (The non-scan Map work and the Reduce work certainly *do* grow.) Experiments back up this claim. The value $\rho = .9$ is the highest value for which FLEX would run consistently, but both CIRCUMFLEX and BATCH experiments go on satisfactorily as far as we have tested them, with essentially unchanging work averages. (Ratios to FLEX obviously become irrelevant for high ρ values.)

We have noted that choosing $f = 1$ as a base case results in the cleanest possible expression of the benefits of CIRCUMFLEX, eliminating the effects of some Map and all Reduce work. But this is also not realistic in practice. As f decreases the benefits of either batching or CIRCUMFLEX clearly become less important overall, and at $f = 0$ disappear completely. But our simulation experiments with varying values of f behave in the entirely expected manner, so we do not show them.

5. CONCLUSIONS

In this paper we have introduced a new CIRCUMFLEX 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 natural and effective technique for amortizing the costs of shared scans. Jobs are decomposed into a number of subjobs, which are related by chain precedence constraints. In the second stage, the resulting chain precedence scheduling problem is solved in a two step process for any of a variety of metrics, including average response time, average stretch and maximum stretch.

Of course, CIRCUMFLEX works best in an environment in which many closely arriving jobs scan the same dataset or datasets. On the other hand, the scheme will work entirely satisfactorily if all jobs scan separate datasets. In particular, such a scenario will not cause any significant additional overheads relative to the original Map Reduce scheduling paradigm, and still provide some performance gains. If employed in the environment for which it was designed, the benefits can be large. Our experimental comparisions between FLEX,CIRCUMFLEX and BATCH support this. Moreover, CIRCUMFLEX works well in a general overlapping dataset environment, resulting in a number of subjobs related by more arbitrary precedence constraints. In this scenario CIRCUMFLEX can optimize any minimax metric, including maximum stretch.

6. REFERENCES

[1] Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. ACM Transactions on Computer Systems **51(1)** (2008) 107–113
[2] Hadoop: http://hadoop.apache.org
[3] Agrawal, P., Kifer, D., Olston, C.: Scheduling shared scans of large data files. In: Proceedings of VLDB. (2008)
[4] 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)
[5] Zaharia, M.: Hadoop fair scheduler design document, http://svn.apache.org/repos/asf/hadoop/mapreduce /trunk/src/contrib/fairscheduler/designdoc /fair_scheduler_design_doc.pdf
[6] 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: Proceedings of EuroSys. (2010)
[7] Wolf, J., Rajan, D., Hildrum, K., Khandekar, R., Kumar, V., Parekh, S., Wu, K.-L., Balmin, A., FLEX: A slot allocation scheduling optimizer for MapReduce Workloads. In: Proceedings of Middleware, Bangalore, India (2010.)
[8] Wolf, J., Squillante, M., Turek, J., Yu, P: Scheduling Algorithms for Broadcast Delivery of Digital Products. IEEE Transactions on Knowledge and Data Engineering **13(5)** (2001) 721 – 741
[9] Zukowski, M., Heman, S., Nes, N., Broncz, P.: Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In: Proceedings of VLDB (2007)
[10] Nykiel, T., Potamias, M., Mishra, C., Kollios, G., Koudas, N.: MRShare: Sharing across multiple queries in MapReduce. In: Proceedings of VLDB (2010)
[11] Ibaraki, T., Katoh, N.: Resource Allocation Problems. MIT Press (1988)
[12] Pinedo, M.: Scheduling: Theory, Algorithms and Systems. Prentice Hall (1995)
[13] Blazewicz, J., Ecker, K., Schmidt, G., Weglarz, J.: Scheduling in Computer and Manufacturing Systems. Springer-Verlag (1993)
[14] J. Leung, E.: Handbook of Scheduling. Chapman and Hall / CRC (2004)
[15] Knuth, D.: The Art of Computer Programming. Addison-Wesley (2011)
[16] Apache ZooKeeper: http://hadoop.apache.org/zookeeper
[17] Hunt, P., Konar, M., Junqueira, F., Reed, B.: ZooKeeper: wait-free coordination for internet-scale systems. In: Proceedings of USENIX (2010)

APPENDIX

A. SCHEDULING PRELIMINARIES

In this section we will give brief overviews of a number of MapReduce and scheduling theory concepts. Understanding these will simplify our exposition of the CIRCUMFLEX scheduler in Section 3. We will outline the two MapReduce scheduling layers, the scheduling metrics we consider and their usefulness, the theoretical notion of malleable scheduling, and, finally, the concept of epoch-based scheduling.

A.1 MapReduce Scheduling Layers

MapReduce scheduling in Hadoop actually consists of two decoupled layers. The lower layer attempts to implement the decisions of the upper layer, while taking into consideration a variety of real-world issues not known to the upper layer.
Allocation Layer. It is assumed that each host is capable of simultaneously handling some maximum number of Map phase tasks. These are called Map slots. (A similar statement holds for Reduce phase tasks, but we are not concerned with those in this paper.) Aggregating these Map slots over all the hosts in the cluster, one computes the total number S of Map slots. The role of the allocation layer scheme is to apportion the Map slots among the active Map jobs in some intelligent manner. The CIRCUMFLEX scheme is actually an allocation layer scheduler. CIRCUMFLEX is *fair* in the same sense as FAIR and FLEX: Specifically, given a minimum number m_j of Map slots for job j, the scheme will allocate a number of slots $s_j \geq m_j$, thereby preventing job starvation. (CIRCUMFLEX also respects maximum slot constraints: Given a maximum number M_j of Map slots for job j, the scheme will enforce $s_j \leq M_j$ for each job j. FLEX respects maxima as well.) The *resource* constraint [11] is also respected: $\sum_j s_j \leq S$. (Because CIRCUMFLEX schedules at the subjob level with chain precedence constraints, we note that all of the above carries over naturally. The minimum for a subjob, for example, is the maximum, over all relevant jobs, of the job minima. The maximum for a subjob is the minimum, over all relevant jobs, of the job maxima. Because of chain precedence the number of slots s_j is unambiguously defined.)
Assignment Layer. It is this layer that makes the actual assignments of Map job tasks (blocks) to slots, attempting to honor the decisions made at the allocation layer to the extent that this is "reasonable". Host *slaves* report any task completions at *heartbeat* intervals, typically 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. Bookkeeping then yields an effective ordering of the jobs, from most relatively underallocated to most relatively overallocated. For each currently unassigned slot, the assignment model then finds an "appropriate" task from the most relatively underallocated job that has one. The notion of what is appropriate varies with the version of Hadoop. One example is the locality of the block to the host. In some assignment layer implementations a non-local block from the best job may be passed over for some period of time in favor of a less appropriate job which has a local block left to scan. This is known as *delay* scheduling [6]. Other affinity issues come into play in different assignment layer implementations.

A.2 Metrics

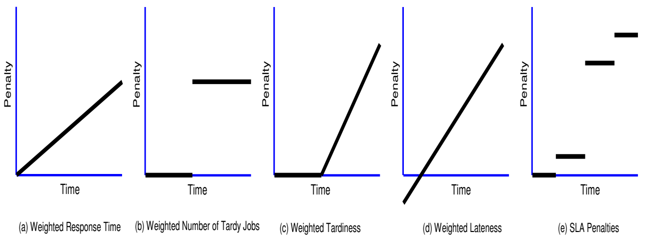

Figure 7: Per Job Metrics

A *penalty function* measures the "cost" of completing that job at a particular time. The subfigures in Figure 7 describe the five most common categories of per job penalty functions in the scheduling theory literature [12–14]. Several combinatorial alternatives exist within most of the categories. For example, we will see below that each of the first four categories can be weighted or not. (In some cases specific weight choices will have special meanings. In other cases they simply define the relative importance of each job.) Also, one might choose to either minimize the sum of the per job penalty functions (a minisum problem), or minimize the maximum of the per job penalty functions (a minimax problem). Optimizing each of these alternatives serves a different but useful purpose.
Response Time. The metric illustrated in Figure 7(a) is probably the most commonly employed in computer science. (The weight is the slope of the linear function.) There are several natural examples. Solving the minisum problem effectively minimizes the average response time, weighted or otherwise, because the sum differs from the average by a multiplicative constant. In the unweighted case, solving the minimax problem minimizes 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, equivalently, the minimum response time required to perform job j in isolation) is W_j. As have noted, the completion time of a job divided by W_j is known as the *stretch* of the job. It 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 more commonly used than either average or maximum PWT, the fairness metrics in AKO.
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 Figure 7(b). The unweighted case counts the number of jobs that miss their deadlines, clearly a useful metric. The weighted case counts some jobs more than others.
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 Figure 7(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. As before, the

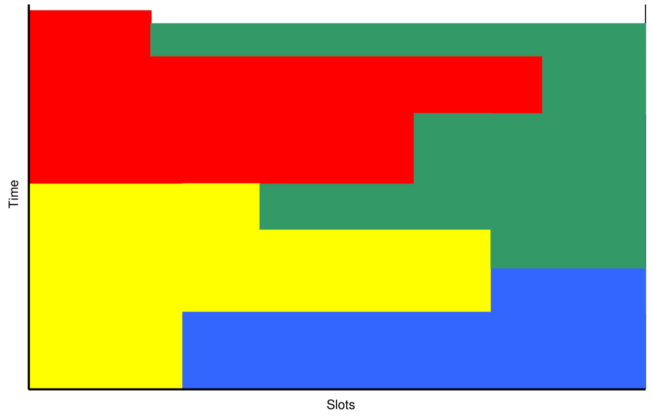

Figure 8: Chain Precedence Malleable Schedule

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 Figure 7(d).

SLA Costs. In this metric each job j has potentially multiple pseudo-deadlines $D_{j,k}$ which increase with k. And the penalties $p_{j,k}$ increase with k also. This yields Figure 7(e), 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.

A.3 Malleable Scheduling

From a scheduling perspective a key feature of the Map 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. The CIRCUMFLEX scheme takes advantage of this.

One can build a schedule in which each Map job is assigned a fixed allocation of slots for the duration of the job. This is known as *moldable* scheduling. *Malleable* scheduling is more general: Instead of making a static decision about the per job slot allocations, one can create multiple intervals. Different intervals will involve different allocations of slots. Each interval then contributes a portion of the total work required to perform the job. And this can be done in the context of subjobs with precedence constraints as well. (See [14] for further details.) Figure 8 illustrates a potential malleable schedule of 2 jobs with a total of 4 subjobs. The yellow and red subjobs are part of one job, related by precedence constraints. The blue and green subjobs are part of another job, also related by precedence constraints.

A.4 Epoch-Based Scheduling

The CIRCUMFLEX scheme is an example of an *epoch*-based allocation scheduler. This means that time is partitioned 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 for the eth epoch must be completed by the start time eT of that epoch.

The CIRCUMFLEX scheme receives input describing the total number of Map slots in the system, the number of active Map jobs, their subjobs, the minimum and maximum number of slots per job, the chain precedence constraints and estimates of the remaining processing times required for each of the subjobs. Then the algorithm outputs a high quality malleable schedule consisting of allocations of slots to subjobs in some number of intervals. Allocations for the eth epoch will likely extend beyond the start time of the $(e+1)$st epoch. But all allocation decisions will be superseded by the decisions of the newest epoch. 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. This means that generally only the first interval in the output will actually be enforced by the assignment model during each epoch.

An advantage of an epoch-based scheme is its resilience over time. Epoch by epoch, the CIRCUMFLEX scheme automatically corrects its solution in light of more current work estimates, newly arrived or departed jobs, and cluster state changes.

B. PSEUDO-CODE

B.1 Generalized Smith's Rule

```
1: for $d = 1$ to $D$ do
2:    Create ordered chain of subjobs $\mathcal{L}_d = \{(d,1),...,(d,K_d^1)\}$ for dataset $d$, where $(d,K_d^1)$ represents the last subjob for dataset $d$, and where subjob $(d,k)$ has processing time $p_{d,k}$ and weight $w_{d,k}$.
3:    Set $K_d^0 = 1$
4: end for
5: Set $\mathcal{L} = \cup_{d=1}^{D} \mathcal{L}_d$
6: while $\mathcal{L} \neq \emptyset$ do
7:    Compute for each subjob $(d,\kappa)$ with $\kappa$ between $K_d^0$ and $K_d^1$ the partial ratio of sums $\sum_{k=K_d^0}^{\kappa} p_{d,k} / \sum_{k=K_d^0}^{\kappa} w_{d,k}$
8:    Find subjob $(\bar{d},\bar{\kappa})$ for which the partial ratio of sums is minimized
9:    Schedule subjobs $(\bar{d},K_{\bar{d}}^0),...,(\bar{d},\bar{\kappa})$
10:   Set $\mathcal{L} = \mathcal{L} - \{(\bar{d},K_{\bar{d}}^0),...,(\bar{d},\bar{\kappa})\}$
11: end while
```

B.2 Backwards Dynamic Programming

```
1: Set $\mathcal{J} = \{1,...,J\}$
2: Compute the subset $\mathcal{J}' = \{j \in \mathcal{J} | \nexists j' \in \mathcal{J}, j' \prec j\}$
3: while $\mathcal{J}' \neq \emptyset$ do
4:    Compute $j^* = arg\min_{j \in \mathcal{J}'} F_j(\sum_{j \in \mathcal{J}} p_j)$
5:    Remove $j^*$ from $\mathcal{J}$
6:    Recompute $\mathcal{J}'$
7: end while
```

B.3 Ready List Malleable Packing

```
1: Set time $T_0 = 0$
2: Create list $\mathcal{L} = \{1,\ldots,J\}$ of jobs, ordered by priority and respecting precedence
3: Create sublist $L_1$ of ready jobs
4: for $i = 1$ to $J$ do
5:    Allocate $m_j$ slots to each job $j \in L_1$
6:    Set $L_2 = L_1$, with implied ordering
7:    Compute slack $s = S - \sum_{j \in L} m_j$
```

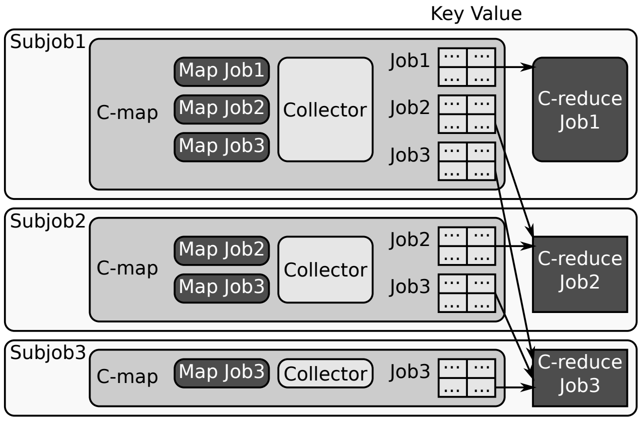

Figure 9: Shared Scans in MapReduce

```
 8:     while $s > 0$ do
 9:         Allocate $\min(s, M_j - m_j)$ additional slots to highest priority job $j \in L_2$
10:         Set $L_2 = L_2 \setminus \{j\}$
11:         Set $s = s - \min(s, M_j - m_j)$
12:     end while
13:     Find first job $j \in L_1$ to complete under given allocations
14:     Set $T_i$ to be the completion time of job $j$
15:     Set $L_1 = L_1 \setminus \{j\}$
16:     Add all immediate successor jobs to $j$ in ready list $L_1$
17:     Compute remaining work for jobs in $\mathcal{L}$ after time $T_i$
18: end for
```

C. SHARED SCANS IMPLEMENTATION

We support circular scans in MapReduce by translating each subjob into a MapReduce job that uses modified map and reduce tasks, called C-mapper and C-reducers respectively. C-mappers identify all map functions that need to be executed by the subjob and proceed to call every function on every input record. They prefix every output record with the job ID of the map function that produced it. The outputs are partitioned by this job ID first. Within each partition a C-mapper uses the comparison and partitioning functions of the original jobs. C-reducers are modified to read not only the results of the C-mappers of this subjob, but also of all the previous subjobs of the MapReduce job in question.

For example, `Subjob1` in Figure 9 runs map functions of three jobs. It produces all output partitions of `Job1` followed by partitions `Job2` and then `Job3`. C-reducers of `Subjob1` read only the ouput partitions of `Job1`. C-reducers of `Subjob1` read partitions of `Job2` from both `Subjob1` and `Subjob2`. Finally, C-reducers of `Subjob3` read ouput partitions of `Job3` from all three subjobs.

In Hadoop, the dominant open-source MapReduce implementation today, C-mappers can be implemented utilizing the existing APIs and, without modifying the Hadoop framework itself. C-reducers, in contrast, require modifications to Hadoop reducer code. A special care must be taken to preserve fault-toulerance of Hadoop, as C-reducers break a key assumption of Hadoop by introducing dependencies between Hadoop jobs.

In order to facilitate communication between job clients, C-mappers, and C-reducers of multiple subjobs, we utilize Apache ZooKeeper an open source distributed coordination service. In the remainder of this section we, first, provide some background about Hadoop and ZooKeeper, and then describe the architecture of C-mappers and C-reducers.

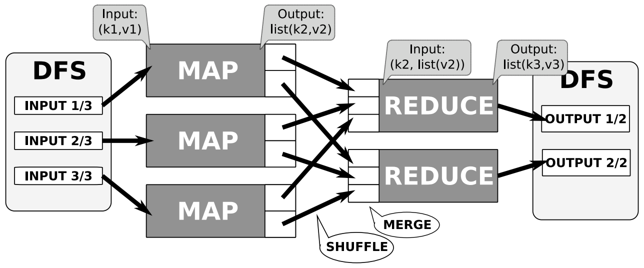

Figure 10: Data Flow in a MapReduce Computation

C.1 MapReduce and Hadoop

MapReduce [1] is a popular paradigm for data-intensive parallel computation in shared-nothing clusters. Example applications for the MapReduce paradigm include processing crawled documents, Web request logs, etc. In the open-source community, Hadoop [2] is a popular implementation of this paradigm. In MapReduce, data is initially partitioned across the nodes of a cluster and stored in a distributed file system (DFS). Data is represented as `(key, value)` pairs. The computation is expressed using two functions:

```
map    (k1,v1)        → list(k2,v2);
reduce (k2,list(v2))  → list(k3,v3).
```

Figure 10 shows the data flow in a MapReduce computation. The computation starts with a map phase in which the `map` functions are applied in parallel on different partitions of the input data, called splits. A map task, or mapper, is started for every split, and it iterates over all the input `(key, value)` pairs applying the map function. The `(key, value)` pairs output by each mapper are assigned a partition number based on the `key`, and sorted by their partition number and the `key` using a fixed-size memory buffer. At each receiving node, a reduce task, or reducer, fetches all of its sorted partitions during the shuffle phase, and merges them into a single sorted stream. All the pair values that share a certain key are passed to a single reduce call. The output of each `reduce` function is written to a distributed file in the DFS.

Finally, the framework also allows the user to provide initialization and tear-down function for each MapReduce function and customize hashing and comparison functions to be used when partitioning and sorting the keys.

C.2 ZooKeeper

CIRCUMFLEX needs to synchronize multiple Hadoop clients that independently submit jobs for the same dataset, into a single circular scan. To this end we need a distributed metadata store that can perform efficient distributed reads and writes of small amounts of data in a transactional manner. The Hadoop project includes just such a tool a distributed coordination service called Apache ZooKeeper [16, 17]. ZooKeeper is highly available, if configured with three or more servers, and fault tolerant. Data is organized in a hierarchical structure similar to a file system, except that each node can contain both data and sub-nodes. A node's

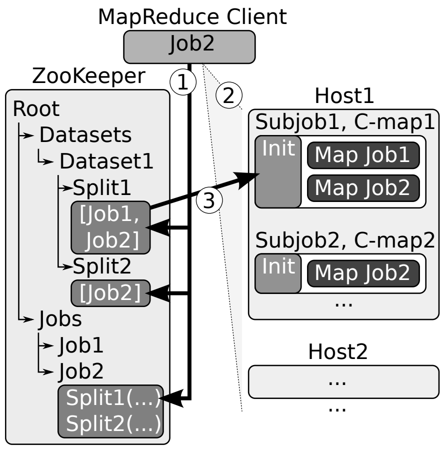

Figure 11: ZooKeeper Data Structure

content is a sequence of bytes and has a version number attached to it. A ZooKeeper server keeps the entire structure and the associated data cached in memory. Reads are extremely fast, but writes are slightly slower because the data needs to be serialized to disk and agreed upon by the majority of the servers. Transactions are supported by versioning the data. The service provides a basic set of primitives, like `create`, `delete`, `exists`, `get` and `set`, which can be easily used to build more complex services such as synchronization and leader election. Clients can connect to any of the servers and, in case the server fails, they can reconnect to any other server while sequential consistency is preserved. Moreover, clients can set watches on certain ZooKeeper nodes and they get a notification if there are any changes to those nodes.

C.3 C-mappers and C-reducers

The C-map and C-reduce tasks get setup by job clients that all connect to a single ZooKeeper service. ZooKeeper data structure contains a node per dataset. When the client sets up a job j that reads a dataset d, it locates this datasets's node Z_d in ZooKeeper. First, it creates a 'lock' subnode of Z_d, and does not proceed until this write succeeds. This ensures that all job clients that want to read the dataset are serialized during the following critical section.

For every data split d_i of the dataset d, the client reads the corresponding split data structure from ZooKeeper node Z_{d_i}, a child of Z_d. If node Z_{d_i} exists in ZooKeeper, the client appends j's ID to the job list in the node – these splits will be executed by the already existing subjobs. They take a ride on the bus that is already scheduled to leave the station, so we refer to them as *rider* splits. If Z_{d_i} does not exist, the client creates a new node Z_{d_i}, with only j's ID in the job list – these splits form the new subjob for j, and we call them *drivers*. This critical section is fast, usually subsecond, so performance impact of serialization is negligible.

For example, Figure 11 shows a job client of `Job2` that needs to read `Dataset1`, which consists of two splits. The client, reads the ZooKeeper structure for `Split1` and appends `Job2`. For `Split2` the structure does not exist, so the client creates it. Thus, for `Job2`, `Split1` is a rider and `Split2` is a driver.

For every rider split the job client computes its reduce partition offset – the sum of the number of partitions for all the jobs before j in that split's job list. Recall that a C-mapper produces output partitions of all the jobs in its job list, in order. For example, `Subjob1` in Figure 9 produces all output partitions of `Job1`, followed by partitions `Job2`, and then `Job3`. Thus, partition numbers of `Job1` are unchanged. Partition numbers of `Job2` are shifted by a fixed offset – the number of partitions of `Job1`. These output partitions will be read by C-reducres of `Subjob2`, not `Subjob1`. For `Job3` the offset is the sum of partition counts of `Job1` and `Job2`.

The offset information is needed by both C-mappers, for their partitioner functions, and C-reducers, to know which partitions of previous subjobs to read. To store this information, the job client creates a single node in ZooKeeper that corresponds to job j. This node contains a table of splits with their offset for partitions of job j, and the job ID of the driver, i.e. the first job in the job list. This *offset table* is stored in a compact form, since all splits in a subjob share the job lists and the offsets.

The client submits a job to MapReduce that has a C-map task for every driver split. If there are no driver splits, the job contains a single NOOP map task needed to start the reducers. When a C-mapper starts it reads the ZooKeeper node that corresponds to its input split, and deletes this node making sure that it deletes the same version that it just read (otherwise it reads again). This atomic read-and-delete marks the cutoff when the bus leaves the station, so no more jobs can catch a ride. The C-mapper takes the job list from its ZooKeeper node and proceeds to read the job configurations of all these jobs, and combine all their map tasks into one.

The high-level architecture of a C-mapper is shown in Figure 9. A C-mapper runs initialization functions of its map tasks. Then it reads the input records, for each record sequentially invoking the map function of every map job. Each map function is given a different output collector, which acts as a wrapper over the real output collector and prefixes every output key with the job ID of the map function producing the output. Similarly, the C-mapper contains a custom comparator which unwraps the output key and calls the comparator of the corresponding job. The C-mapper partitioner function unwraps the key, calls the corresponding partitioner function, and also reads from ZooKeeper the job's partition offset within the split. The final partition number is obtained by adding this offset and the partition number returned by the job's partitioner. Finally, the C-mapper runs the tear-down functions of its map tasks. C-reducers start by reading their job's offset table from the ZooKeeper. For every split the table contains the driver job ID J_D and a partition offset O. Normally, a reducer number N of a job J, reads N's partition from the ouptut of every map task of J. C-reducers also read partition $O + N$ of J_D's output, for every split in the offset table.

In case of a system failure, this output partition may not exist, and since job J has already finished, Hadoop cannot rexecute its map tasks as it normally does in case of failures. To work around this problem C-reducer reports to the framework that one C-mapper in its own job has failed and needs to be re-executed, and puts the information about which split needs to be rerun in ZooKeeper. Once the C-mapper is restarted, it will find the split metadata in ZooKeeper and read that split instead of its assigned one.

Once all C-reduce tasks are complete, our customized clean-up task removes the job's offset table and other job-related structures from the ZooKeeper.