The X-Flex Cross-Platform Scheduler: Who's The Fairest Of Them All?

Joel Wolf[1], **Zubair Nabi**[1], **Viswanath Nagarajan**[2], **Robert Saccone**[1], **Rohit Wagle**[1], **Kirsten Hildrum**[1], **Edward Pring**[1], and **Kanthi Sarpatwar**[3]

[1]**IBM Research**
[2]**University of Michigan**
[3]**University of Maryland**

ABSTRACT

We introduce the *X-Flex* cross-platform scheduler. *X-Flex* is intended as an alternative to the *Dominant Resource Fairness (DRF)* scheduler currently employed by both *YARN* and *Mesos*. There are multiple design differences between *X-Flex* and *DRF*. For one thing, *DRF* is based on an instantaneous notion of fairness, while *X-Flex* monitors instantaneous fairness in order to take a long-term view. The definition of instantaneous fairness itself is different among the two schedulers. Furthermore, the packing of containers into processing nodes in *DRF* is done online, while in *X-Flex* it is performed offline in order to improve packing quality. Finally, *DRF* is essentially an extension to multiple dimensions of the *Fair* MapReduce scheduler. As such it makes scheduling decisions at a very low level. *X-Flex*, on the other hand, takes the perspective that some frameworks have sufficient structure to make higher level scheduling decisions. So *X-Flex* allows this, and also gives platforms a great deal of autonomy over the degree of sharing they will permit with other platforms. We describe the technical details of *X-Flex* and provide experiments to show its excellent performance.

Categories and Subject Descriptors

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

General Terms

Algorithms, Experimentation, Performance

Keywords

Cross-platform Scheduling, YARN, DRF

1. INTRODUCTION

The need to analyze disparate datasets and to utilize different processing paradigms has led to a profusion of distributed cluster frameworks in the last few years. To consolidate data center resources, combine various processing paradigms within the same application and enable inter-framework data sharing, a number of cross-platform cluster managers have been designed. These include HPC-style centralized managers [22, 19], centralized two-level managers such as *Mesos* [13] and *YARN* [23], and decentralized managers [18, 16]. Of these, two-level managers have found wide traction due to their ability to match the requirements of popular frameworks (such as MapReduce [5] and *Spark* [29]) that schedule fine-grained *tasks* across processing nodes divided into *slots*. At the top level of this model, the cluster manager allocates resources (typically CPU cores and memory) to frameworks, which in turn distribute these resources across the various jobs and tasks that need to run. Allocation decisions are determined by a scheduling algorithm such as *Dominant Resource Fairness* (*DRF*) [11]. *DRF* aims to equalize the allocation of each framework subject to its most highly demanded resource. This paper is about *X-Flex*, a proposed alternative to *DRF*.

1.1 Why X-Flex?

DRF has many virtues, and can be regarded as the default scheduler in both *YARN* and *Mesos*. But there were several aspects of *DRF* that we felt might better be handled differently, at least in some environments, and these have motivated the *X-Flex* design. We list these motivations below, and in so doing enumerate the key differences between *DRF* and *X-Flex*. Taken together, we note that *X-Flex* is fundamentally, even radically different from *DRF*.

Note that we will use the word *application* generically to denote the entities that share the cluster. These could be platforms, frameworks, departments, users, jobs and so on. We are simply adopting this word to be consistent with the *Application Master (AM)* concept in *YARN*. We will use more specific terms as appropriate. *X-Flex* has been initially implemented in *YARN*, perhaps a more natural fit than *Mesos*. But we see no reason why it could not be implemented in *Mesos* as well.

First, *DRF* is based on an instantaneous notion of fairness. As described in [11], *DRF* keeps track of each application's *dominant resource share (DRS)*, and attempts at each moment to allocate resources to applications in order from lowest to highest *DRS*. We will recall the definition of *DRS* presently, but note first that it depends only on the resource allocations at the *current* time. We believe instead that fairness is a property best measured over time, with knowledge of the past. An application which uses fewer resources earlier should be rewarded with more resources later, and vice versa. *X-Flex* is based on such a long-term notion of fairness, essentially the integral over all previous time of an instantaneous fairness measure. A good analogy might be handling the "sharing" of toys between children. If Alvin has been playing with most of the toys for a while, shouldn't Barbara get her chance? *X-Flex* agrees, but *DRF* does not remember the past. (The analogy between applications and children may actually be at least somewhat apt.)

Second, what exactly is *DRS*? As described in [11], *DRS* is a maximum (worst case) metric involving multiple normalized resources. Let us assume for concreteness two resources, memory and CPU cores. The *DRS* of any application is the maximum of the normalized fractional share of both. (For example, an application's normalized fractional memory share is the total amount of memory allocated to that application divided by the total amount of memory in the cluster.) Under certain (hypothetical) assumptions [11] shows that *DRS* has a number of very nice theoretical properties. (See also [8, 17].) Specifically these are *sharing incentive*, being *strategy proof* and *envy free*, and *Pareto efficiency*. But while *DRS* has these pleasant features, it also has the disadvantage of being a maximum metric. This means, for example, that an application with a normalized share of 50% of the cores in the system and a normalized share of 1% of the memory has a *DRS* identical to that of an application with normalized shares of 50% and 50%, respectively. Yet the latter application is taking up far more of the cluster resources than the former. So *X-Flex* opts, instead, for an instantaneous fairness metric based on the *sum* rather than the maximum of the normalized fractional shares. The tradeoff is that we gain a seemingly more appropriate metric, and we lose the theoretical guarantees. We thus opt for practical over theoretical.

Third, the notion of multi-resource optimization is currently in vogue, and to the extent that high quality input data is available, that is a good thing. But clusters consist of processing nodes, not monolithic collections of aggregate resources. Resource allocations must by definition respect these node boundaries. Even in a single dimension, the online problem of dynamically selecting and packing tasks with certain resource requirements into a given set of processing nodes so as to maximize the number packed does not admit a good algorithm [2]. For the "dual" problem of *bin packing* all tasks into the minimum number of fixed size processing nodes, there are good online algorithms in a single dimension. But in multiple dimensions, this problem, known as *vector bin packing*, becomes much harder. And the performance of any online algorithm degrades as the number of dimensions increases [1]. (This packing problem is noted, but not discussed further, in [11, 13, 23] even though both YARN and Mesos intend on adding more dimensions to containers beyond the currently supported CPU and memory dimensions.) We take the perspective in *X-Flex* that vector bin packing is best done semi-statically (offline). We actually pack the *YARN containers* in which the tasks will be executed. The advantage is that the problem can be solved much more carefully, and with far less waste. In fact, we optimize both the *size* and the *placement* of our containers, even assigning these containers to nominal application *owners* while factoring in a variety of placement and colocation constraints. The obvious tradeoff is that *X-Flex* is less dynamic, and also requires data that might not be immediately available or sufficiently stable. (To handle these tradeoffs we are, in fact, considering a compromise approach for a future version of *X-Flex*, partly online and partly offline. See [12] for a high quality online packing scheme.)

Finally, it is our contention that in some cases *DRF* makes scheduling decisions at too low a level. The application is typically a job, and these jobs are treated independently. But there do exist frameworks for which intelligent schedulers exist. One such example is MapReduce and the *Flex* family of schedulers [28, 25, 15]. Because it understands the structure of MapReduce, *Flex* can, for instance, schedule to minimize average job response time, stretch, deadline penalties and so on. If a framework can take advantage of its inherent structure to make intelligent scheduling decisions, it seems a shame to cede control to a scheduler such as *DRF* where the only goal might be fairness at the job level. *X-Flex* allows applications to employ an application-specific scheduler, while still handling the sharing details at a lower level. As we shall see, *X-Flex* will also enable applications to share as much or as little as desired. In a similar vein, *DRF* cannot take into account the diverse needs of disparate platforms, such as stream processing systems where the scheduling focuses on low latency and dynamism [26]. On the other hand, *X-Flex* embraces domain specific schedulers. All told, *X-Flex* gives applications a great deal of autonomy and control. In terms of scheduling efficiency there are pluses and minuses. One plus of scheduling at the framework rather than job level would be the need for fewer AMs. The corresponding minus is that there may be more overhead associated with the framework level scheduler.

The bottom line is that *X-Flex* is, indeed, flexible and generic. Advantage 1 above, (longterm rather than instantaneous fairness) can certainly be said to be qualitative rather than quantitative, but we think advantage 2 (a better definition of instantaneous fairness) should result in superior performance. The benefits of advantage 3 (offline vector packing), while less essential to the overall *X-Flex* design, should grow with the number of resource dimensions. Advantage 4 (the ability to employ specialized schedulers for appropriately structured frameworks) may not apply and thus not yield benefits in every cluster environment. But we think it should be very effective in those cases where it does. Basically, *X-Flex* is to *Flex* as *DRF* is to *Fair*.

We note in passing *Quasar* [6], which performs cluster management in the context of a cloud environment. Among other things, they observe via a production Twitter example that *Mesos* under *DRF* typically has low cluster utilization. Their argument is that users do not always understand resource requirements, and accordingly *Quasar* uses QoS requirements to drive resource allocation. In this sense their approach is orthogonal to ours.

The remainder of this paper is organized as follows. §2 describes the *X-Flex* algorithms as implemented in *YARN*. These algorithms include two off-line schemes (*X-Size* and *X-Select*) and two on-line schemes (*X-Schedule* and *X-Sight*). §3 describes performance results comparing *DRF* and *X-Flex* from several perspectives. We evaluate average response times as well as cluster utilizations for MapReduce jobs. Finally, §4 lists conclusions.

2. X-FLEX ALGORITHMS

We begin with an overview of the *X-Flex* offline and online

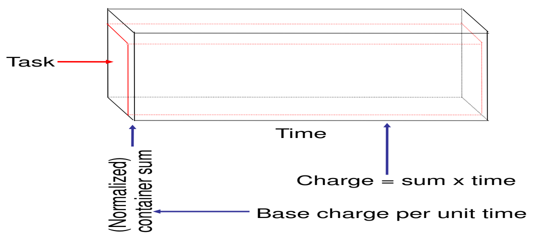

Figure 1: *X-Flex* Charging Mechanism

components and a description of some key *X-Flex* concepts. Tasks in *YARN* are executed in *containers*. In *X-Flex* we pre-pack these containers into processing nodes in an essentially offline manner. The goal of offline *X-Flex* is twofold.

First, we decide on the *dimensions* of these containers. These dimensions typically pertain to CPU cores, memory and possibly other resources. Every container must fit within the dimensions of at least one processing node. We create a *limited number* of container dimensions by an optimization algorithm called *X-Size*, the goal being to minimize the amount of resources utilized when actually executing tasks on containers.

Second, we vector pack containers of these dimensions into the processing nodes. Each packed container is also assigned an application *owner* whose resource requirements are appropriate for the container dimensions, and the aggregate dimensions of all the containers assigned to each application approximately match the share of the cluster allocated to that application. This is performed by an optimization algorithm called *X-Select*.

Now *X-Flex* allows applications to use each other's containers according to explicit sharing guidelines. So one application may (temporarily) execute tasks on a container owned by another application. To understand *X-Flex* sharing we need to describe the *charging mechanism* it employs. And this is quite simple. See Figure 1. If application A uses a container owned by application B for time t, it is charged as the product of the normalized container "size" and t. So if the container uses an amount r_d of resource in dimension d and the aggregate cluster resources in that dimension is R_d, the instantaneous charge is $\sum_d r_d/R_d$, while the total charge is $(\sum_d r_d/R_d) \cdot t$. Note that *X-Flex* charges by the container rather than the task resource requirements. But it also attempts to place tasks into containers which do not dramatically exceed the task requirements.

One can think of the borrowing of a container as a "rental", and in this context the charging mechanism simply describes the unit of currency – the cost of that rental.

X-Flex gives applications a great deal of autonomy over the extent to which they will share containers with other applications. At one extreme, an *X-Flex* application may simply indicate that it (like Garbo [10]) "wants to be alone". In that case, the containers assigned to it by *X-Select* will only be used by that application, and the application will never use containers owned by another application. Effectively, such Garbo applications will be given a fixed partition of the cluster, though that partition may not respect processing node boundaries. We naturally hope that there will be few such applications, but *X-Flex* does support them.

For the remaining applications, *X-Flex* creates an environment allowing as much or as little sharing as desired. Specifically, each such application A will provide a *sharing bound* δ_{AB} (in units of *currency*) with respect to any other

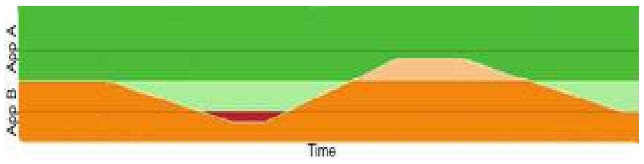

Figure 2: *X-Sight* View of Sharing Between Two Applications

application B. (Application A may simply provide a universal sharing bound δ_A, in which case δ_{AB} will be set to δ_A for all other applications B.) Clearly, the sharing bounds between applications A and B should be symmetric. So the final sharing bounds $\Delta_{AB} = \Delta_{BA}$ are set to $\min(\delta_{AB}, \delta_{BA})$.

Now the actual sharing imbalance between applications A and B may change over time, due to the borrowing of containers of one by the other. The key idea is that this imbalance is compared with the bound Δ_{AB}. If application A is in "debt" to application B by Δ_{AB} *or more*, application B will be allowed to preempt it with new container request(s).

See, for example, Figure 2. This is an actual (compressed) *X-Sight* (details of *X-Sight* are given below) view of the pairwise sharing over time between two applications A and B. The horizontal axis is time, while the vertical axis shows the degree of sharing imbalance between the two applications. Specifically, the white line segments illustrate the changing sharing imbalance over time. Application A is represented by green, and application B by orange. The horizontal center line indicates perfect balance, while the symmetrical lines above and below correspond to $\pm\Delta_{AB}$. Initially the two applications are in perfect balance, but eventually application A requests an idle container of application B, and this is granted. The sharing imbalance then shifts towards application B, *favoring* application A. The pale green shading extends to the sharing bound $-\Delta_A B$, and then the graph turns red. The red zone corresponds to a situation in which application B can preempt containers in return, and one can see this happening. And the process continues indefinitely. Applications have the opportunity to borrow containers, but they are forced to share responsibly.

There is an open-ended spectrum of sharing degrees between applications. Note that even applications with sharing bounds of 0 can borrow containers at times. They simply have to give the containers back on demand. So, for example, MapReduce frameworks using *Flex* might have a sharing bound of 0, but use containers of others to perform preemptable, *best effort* work.

We now give further details on *X-Flex*, focusing on the online components. The mathematical details of the two offline components are interesting, but due to lack of space we will only give overviews of these here. (See [27] for a more complete exposition). We discuss the online *X-Flex* components first.

2.1 Online X-Flex Components

X-Schedule is the key online component of *X-Flex*. It runs as the *YARN* scheduler inside the Resource Manager, replacing *DRF*. It is the component through which *YARN* applications request and receive container allocations. *X-Schedule* uses the container assignment configurations generated via periodic *X-Size* and *X-Select* runs. The container assignment configuration contains entries describing container definitions (memory size, CPU cores, processing node) as well as the application *owner*. Using this information, *X-Schedule* maintains for each application the set of containers it owns. It also tracks which of those containers have been assigned by the scheduler, along with the identity of the application to which they have been assigned.

X-Schedule also uses a second set of configurations which define the type of application, the degree of resource sharing that each application allows, and the current sharing status. Those (Garbo) applications that indicate they will not share any of their containers are scheduled in the obvious manner, and need not be considered further. So we will concentrate on applications that are willing to share. These applications maintain their pairwise (symmetric) *sharing bounds* here. Three pieces of additional data are updated each time a scheduling decision is made involving a container that has been shared by the pair. These are the sharing imbalance *lastShare* at the time the calculation was made, the current slope *lastSlope* describing the trend in sharing between the two applications, and the time *lastCalcTime* of the calculation. The *lastShare* value may be positive, negative or zero. It indicates the degree of imbalance between the two – which application (if either) was benefiting more from resource sharing at the time *lastTime*. A *lastShare* value of zero indicates the two applications are in perfect balance. The value of *lastSlope* may also be positive, negative or zero. It indicates the trend towards future imbalance, and is calculated as the sum of all the instantaneous charges for containers of one application which are in use by the other, with the obvious plus and minus signs. A *lastSlope* value of zero indicates the platforms are in steady state. All three values are initially set to zero. The point of all this, of course, is to allow *X-Schedule* to extrapolate the sharing imbalance between the two applications at the current time *curTime*, and thus determine whether or not this imbalance equals or exceeds the sharing bound.

Applications submit allocation requests to *X-Schedule* in order to obtain the containers needed to execute their tasks. These allocation requests specify the requirements (memory, number of CPU cores and so on) and number, rack level or host level locality constraints, request priority, and preemption priority. When *X-Schedule* attempts to fulfill allocation requests for an application it will satisfy requests in request priority order, as specified by the application, from highest to lowest. Unique to *X-Schedule* is the ability for an application to also specify the type of container that should be used to satisfy the request – *OwnedOnly, OwnedFirst* and *NonOwned*.

An *OwnedOnly* request type tells *X-Schedule* that it should try to satisfy the allocation request using only containers owned by the application. It examines each free, owned container and keeps a numerical score indicating how well the attributes of the candidate container satisfy the requirements of the request. Certain attribute mismatches will eliminate the container from consideration altogether. For example, a request specifying a particular rack or host will be eliminated if the candidate container is not on that rack or host. A container whose resource dimensions are not all at least those of the request will also be eliminated. In the other direction, containers whose aggregate normalized dimensions are more than a prespecified *fitness value* times the aggregate normalized dimensions of the request are also eliminated. (The default fitness value is 2.) This guards against assigning very large containers to small requests and thus attempts to reduce wasted resources. After all free containers have been considered, the one with the highest score is allocated to the application. The container is inserted into the *in use* list of the application in *preemption priority* order (lowest to highest). If there are no free containers available but the application owns containers in use by other applications, *X-Schedule* may attempt to satisfy the request by preempting one of those. This depends on the comparison described above between the extrapolated sharing imbalance and the sharing bounds. We will discuss the selection of a container to be preempted below.

An *OwnedFirst* request tells *X-Schedule* that it should try first to satisfy the request from the containers an application owns, using the algorithm described above. If no suitable containers are available, it will next try to fulfill the request from the unused containers of other sharing applications. The free containers of each application are enumerated and subjected to a scoring mechanism similar to the one described above, but with an additional scoring component based on the degree of sharing between the two applications. Using the *sharing context* data mentioned earlier, new calculations are made to reflect what these values would be if the container were to actually be allocated. First a *newShareProjection* is calculated taking the *lastShare* and adding to it the *lastSlope* multiplied by the difference in time since the last calculation. Next a *newSlopeProjection* is calculated by taking the *lastSlope* and adding to it the container size to estimate how the slope of the trend line would be affected by making the allocation. Finally, a *Time To Live (TTL)* estimate is calculated by taking the sharing bound and subtracting the *newShareProjection*. This result is then divided by the *newSlopeProjection*. The *TTL* projection is then weighted and added into the score. Containers that have small *TTL* projections are more likely to be preempted or taken back sooner, and thus have a smaller effect on the score value than those that have larger *TTL* projections. After enumerating all the applications and their free containers, the one with the highest score is chosen and allocated to the requesting application. The *sharing context* for the requesting application and the owning application pair is updated with the results of the new share calculations mentioned above. If this process fails, *X-Schedule* will attempt to fulfill the request using preemption, as described below.

A *NonOwned* request tells *X-Schedule* that it should attempt to satisfy the request using only containers that the requesting application does not own. It uses an algorithm identical to the second step of *OwnedFirst*, trying to satisfy a request using free containers from applications other than the requesting application. If none are available, *X-Schedule* may again attempt to satisfy the request by preempting a suitable container in use by another application.

We note that there are use cases for each of these request types. *OwnedFirst* is, as one would expect, the most common *X-Flex* type. For applications with small or zero sharing bounds, however, one might issue an *OwnedOnly* request when doing mission critical work, and a *NonOwned* request when doing best effort work.

Finally, we describe *preemption*. This is the strategy *X-Schedule* employs when there are no free containers of the requested type. There are two types of preemptions that can occur. The first type occurs when an *OwnedOnly* or *OwnedFirst* request is made and there are no free containers owned by the requesting application. In this case *X-Schedule* will examine (in *preemption priority* order) all the in-use containers owned by the requesting application which have been loaned to other applications. For each candidate container it calculates a score as described earlier, with an additional test to see if the candidate container can indeed be preempted. A container is eligible for preemption if the application currently using that container has a *newShareProjection* greater than or equal to the pairwise sharing bound. Any container that cannot be preempted is eliminated. After examining all the candidate containers, the one with highest score is chosen, if any.

The second type of preemption occurs in cases of an *Owned-First* or *NonOwned* request types. Containers owned by other applications are examined in preemption priority order, using the same scoring system. (If the candidate container is already in use by the requesting application it is, of course, skipped.) The candidate container with the highest score, if any, is chosen.

In either type of preemption the application losing the container is notified, and has a configurable amount of time to release the container on its own. Once the grace period has expired, the container is forcibly killed and the reassignment to the requesting application occurs.

It is worth mentioning that while there is an overhead associated with the various online calculations incurred by *X-Schedule* (such as updating the sharing bound), it is negligible considering the heartbeat based container allocation model employed by YARN. Allocation cycles in this model are typically on the order of seconds.

The second online component is the real-time visualizer, *X-Sight*. *X-Sight* allows an administrator to see three separate views. The first is the overall cluster utilization over time, partitioned by application. The second, illustrated in Figure 2, is the sharing bounds and imbalance over time for any pair of applications. The third shows the vector packing of containers into the processing nodes, the owners and the current users of those containers.

2.2 Offline X-Flex Components

Now we will give very brief overviews of the two mathematical components of *X-Flex*. These are interesting problems in their own right, but space precludes a full exposition here. Complete details can be found in [27].

The two schemes *X-Size* and *X-Select* are executed in that order when *X-Flex* is initialized. After that, either *X-Size* and *X-Select* or possibly just the latter will be repeated periodically (and presumably infrequently), when the input data changes sufficiently or *X-Flex* performance degrades beyond some predefined threshold.

The primary input to *X-Size* is a profile of the various resource requests made by the applications using the cluster, weighted by frequency. The number K of container shapes allowed is also input: The idea is that we want to create only a relatively modest number of container shapes. (A similar problem exists in cloud environments, though presumably with a different objective function.) The output is a set of K different container dimensions so that every request "fits" into at least one, optimized to minimize the total resource used when assigning these requests to their *best fitting* containers. Here, the resource usage of a request is the sum of the normalized dimensions of the container to which it is assigned. We note in passing that if the notion of fit were based on the maximum (as in *DRF*) rather than the sum, a very simple dynamic program would work well. In our context the problem is harder, and we create a *polynomial time approximation scheme (PTAS)* [24] to solve it. This means that for any $\epsilon > 0$ we have a polynomial time algorithm whose performance is within $1 + \epsilon$ of optimal. Assume initially that there are two dimensions, say CPU cores and memory. The loss of an ϵ factor comes from considering only solutions on one of $\pi/\epsilon - 1$ equi-angled rays in the first quadrant emanating from the origin. For solutions on these rays, the scheme, a more elaborate dynamic program on K, provides an exact solution. Higher dimension are handled inductively. This scheme is then repeated for various decreasing values of ϵ until a predetermined amount of execution time has elapsed.

Next we describe *X-Select*. The input here is the set of processing nodes, the applications, the container sizes from *X-Size*, and the forecasted mix of required containers and their applications. There may also be constraints on these containers, including resource matching, colocation and/or exlocation of pairs of containers. The output is a valid vector packing of containers (together with application owners) into processing nodes which optimizes the overall number of containers that are packed, while giving each application its share of containers. This output is precisely what is needed by *X-Schedule*. When *X-Flex* is initialized the *X-Select* algorithm attempts to maximize a multiplier λ: It essentially employs a bracket and bisection algorithm to find the largest value such that containers corresponding to λ times the required mix can be vector packed into the existing processing nodes. Any given λ corresponds to a fixed set of containers to pack, and a greedy algorithm that vector packs containers into one processing node at a time is known to be a 2-approximation [3, 4]. An iterative improvement heuristic is then employed to further optimize the vector packing, and simultaneously determine whether or not the packing is feasible. In subsequent *X-Select* runs only the iterative improvement heuristic is employed, with the additional incremental constraint that the packing on only a prespecified fraction of the processing nodes may be changed.

3. EXPERIMENTS

In this section we focus on experiments which compare *X-Flex* with *DRF*. We have designed and implemented an *Application Master (AM)* for MapReduce with plug-in schedulers for *Flex, Fair* and *FIFO*. We have also written an AM [21, 14] for *IBM InfoSphere Streams* [20]. We expect that this streaming application will commonly be run in Garbo mode, since its work is long running. One can imagine other sharing decisions as well, but we will accordingly not discuss this application further here.

The question of how *fairness* should be defined remains qualitative, essentially unquantifiable. We naturally believe that taking our definition of instantaneous fairness and our longer term view makes good sense.

	Flex	Fair	FIFO	DRF
Small ART	76.2	124.7	268.3	376.8
Medium ART	270.1	333.5	275.8	364.0
Large ART	539.0	539.6	188.6	154.3
Overall ART	122.4	171.8	267.0	367.5
Makespan	544.2	544.6	563.0	713.4

Table 1: Average Response Time and Makespan in Seconds

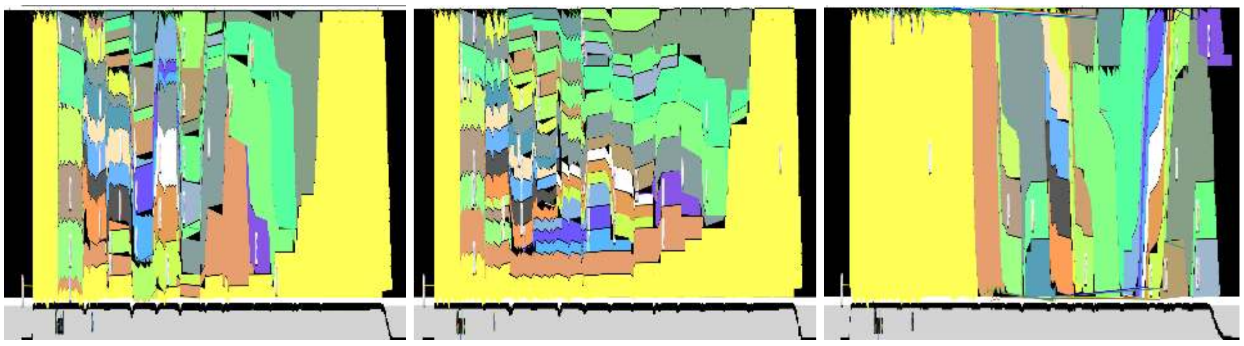

Figure 3: Flex, Fair and FIFO

Perhaps the most important difference between the two schedulers is the ability within *X-Flex* to employ higher level scheduling appropriate to the application. *Flex* for MapReduce is one such scheduler. It is highly effective in MapReduce applications because of the inherent structure there. For each (one second) scheduling interval it produces a hypothetical *malleable* schedule [9] for the current set of jobs and the particular performance metric (such as average response time) being optimized. It then proposes allocations of containers to jobs in the immediate future, and these decisions are (approximately) instantiated by the AM. The entire process repeats every scheduling interval. *But* in order to do this, *Flex* needs at least a temporarily constant view of the available cluster resources. It cannot do this within the *DRF* context, because there are no such guarantees: *DRF* considers only the current instance in time. On the other hand, *Flex* will work well with *X-Flex* if one assumes a sharing bound of zero. It can perform mission critical work in its share of the cluster, and best effort work elsewhere.

Accordingly we designed a set of experiments to test *Flex* performance together with *Fair* and *FIFO*, using 3 corresponding AMs within *X-Flex*. Each AM used one container. We used an additional 75 containers allocated to the 3 Map Reduce variants, with 6 processing nodes on one rack. Each AM was given ownership of an equal share of the cluster, 25 containers in all. We gave *Flex* a sharing bound of zero. We created three types of MapReduce jobs, for simplicity only using Map tasks. The small jobs had 5 tasks, and there were 25 of them per MapReduce variant. The 5 medium jobs per variant had 25 tasks, and the single large job per variant had 125 tasks. This sort of approximately Zipf-like distribution in job size and frequency is quite typical of real workloads. The corresponding *DRF* experiments used 78 containers, and had 75 small jobs, 15 medium and 3 large jobs. Since *DRF* schedules at the job level, more containers were needed for the AMs.

Table 1 summarizes results averaging 5 separate runs each of this experimental setup. The average response times are broken out by job type, and the overall average is listed as well. For the 3 columns associated with the *X-Flex* setup we see significantly better performance than that of the last column, *DRF*. The average response time for *Flex* is 33% of *DRF*. And within *X-Flex*, *Flex* average response time is 71% of *Fair* and 46% of *FIFO*. Makespans for the 3 *X-Flex* applications are comparable, as one would expect. There is a fixed amount of work. But the makespan for *DRF* is significantly higher, due to the overhead of the extra AMs.

X-Flex had an average CPU utilization of 92% across all 6 processing nodes, while the corresponding *DRF* utilization was 80%. Memory utilization was 73% for *X-Flex* and 66% for *DRF*. We attribute the better numbers in part to the additional AM overhead associated with *DRF*.

These response time numbers are in keeping with past results for *Flex* and to the particular experimental setup. And there are simple reasons. Consider Figure 3, which shows a *FlexSight* [7] view of one such experiment for each of *Flex, Fair* and *FIFO*. *Flex*, when optimizing average response time, essentially attempts to schedule jobs by size, small to large. (A *scout* is executed quickly in order to estimate this size, which is then continually extrapolated.) Jobs are elongated in the container dimension but are shrunk in the time dimension, also shrinking the response time. *Fair*, in an effort to actually be fair, does the reverse. *FIFO* elongates jobs in the right dimension, but it orders its jobs based only on arrival time. See, for example, the large (yellow) job or the medium (brown) job towards the bottom of the figure.

The moral is that a cross-platform scheduler like *X-Flex* is required if one wants to obtain the benefits of a more intelligent application scheduler.

This experimental setup emphasized one aspect of performance of *X-Flex* compared to *DRF*. But it kept the container design very simple in an effort to isolate the packing effect. We separately experimented with 2-dimensional vector packing problems. This experiment is a bit delicate, because the offline component of *X-Flex* does depend on a reasonably accurate forecast of application request dimensions (in this case, CPU cores and memory). Factoring that out of the experiment, we produced an offline *X-Select* solution for *X-Flex* and compared it to what would occur in *DRF*. The same realtime workload in *X-Flex* produced 32% more successful container requests than *DRF*.

4. CONCLUSIONS

In this paper we have presented a new and novel cross-platform scheduling scheme known as *X-Flex*. It is currently implemented within *YARN*. While still in a relatively early stage of its development cycle, it appears to have several qualitative and quantitative advantages over *DRF*. Among these are a long term view of fairness, a seemingly more suitable definition of instantaneous fairness, a mathematically sophisticated offline vector packing scheme to create containers and their owners, the flexibility, if desired, to work with framework-specific schedulers which can take advantage of inherent structure, and finally the ability of applications to share as much or as little as they desire and/or require.

5. REFERENCES

[1] Y. Azar, I. Cohen, S. Kamara and B. Shepherd. Tight Bounds for Online V Bin Packing. In *Proceedings of STOC*, 2013.

[2] J. Boyar, K. Larsen and M. Nielsen. The Accommodating Function - A Generalization of the Competitive Ratio. In *Proceedings of IADS*, 1999.

[3] C. Chekura and S. Khanna A PTAS for the Multiple Knapsack Problem. In *Proceedings of SODA*, 2000.

[4] R. Cohen, L. Katzir and D. Raz. An Efficient Approximation for the Generalized Assignment Problem. *Information Processing Letters*, 100(4): 162-166, 2006.

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

[6] C. Delimitrou and C. Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of ASPLOS*, 2014.

[7] W. De Pauw, J. Wolf and A. Balmin. Visualizing Jobs with Shared Resources in Distributed Environments. In *IEEE Working Conference on Software Visualization*, 2013.

[8] D. Dolev, D. Feitelson, J. Halpern, R. Kupferman and N. Linial. No Justified Complaints: Fair Sharing of Multiple Resources. In *Proceedings of ITCS*, 2012.

[9] M. Drozdowski. Scheduling for Parallel Processing. *Springer.*

[10] *www.wikipedia.org/wiki/Greta_Garbo*

[11] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker and I. Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of NSDI*, 2011.

[12] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao and A. Akella. Multi-Resource Packing for Cluster Schedulers. In *Proceedings of ACM SIGCOMM*, 2014.

[13] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of NSDI*, 2011.

[14] Z. Nabi, R. Wagle and E. Bouillet. The Best of Two Worlds: Integrating IBM InfoSphere Streams with Apache YARN. In *Proceedings of IEEE Big Data*, 2014.

[15] V. Nagarajan, J. Wolf, A. Balmin and K. Hildrum. FlowFlex: Malleable Scheduling for Flows of MapReduce Jobs. In *Proceedings of Middleware*, 2013

[16] K. Ousterhout, P. Wendell, M. Zaharia and I. Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of SOSP*, 2013.

[17] D. Parkes, A. Procaccia and N. Shah. Beyond Dominant Resource Fairness: Extensions, Limitations, and Indivisibilities. In *Proceedings of EC*, 2012.

[18] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek and J. Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of EuroSys*, 2013.

[19] G. Staples. TORQUE Resource Manager. In *Proceedings of Supercomputing*, 2006.

[20] IBM Infosphere Streams *www.ibm.com/software/products/en/infosphere-streams.*

[21] IBM Infosphere Streams/Resource Managers Project *https://github.com/IBMStreams/resourceManagers.*

[22] D. Thain, T. Tannenbaum, Todd and M. Livny. Distributed Computing in Practice: The Condor Experience: Research Articles. *Concurrency and Computation: Practice & Experience*, 17:(2-4), 323-356, 2005.

[23] V. Vavilapalli, A. Murthy, C. Douglis, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed and E. Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of SoCC*, 2013.

[24] V. Vazirani. Approximation Algorithms. *Springer.*

[25] J. Wolf, A. Balmin, D. Rajan, K. Hildrum, R. Khandekar, S. Parekh, K.-L. Wu and R. Vernica. On the Optimization of Schedules for MapReduce Workloads in the Presence of Shared Scans. *VLDB Journal*, 21(5): 589-609, 2012.

[26] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu and L. Fleischer. SODA: An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computing Systems. In *Proceedings of Middleware*, 2008.

[27] J. Wolf, Z. Nabi, V. Nagarajan, R. Saccone, R. Wagle, K. Hildrum, E. Pring and K. Sarpatwar. The X-Flex Cross-Platform Scheduler. *IBM RC*, 2014.

[28] J. 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.

[29] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of USENIX – Networked Systems Design and Implementation*, 2012.