Fine-grained Dynamic Resource Allocation for Big-Data Applications

Luciano Baresi, Sam Guinea, Alberto Leva and Giovanni Quattrocchi

Abstract—Big-data applications are batch applications that exploit dedicated frameworks to perform massively parallel computations across clusters of machines. The time needed to process the entirety of the inputs represents the application's response time, which can be subject to deadlines. Spark, probably the most famous incarnation of these frameworks today, allocates resources to applications statically at the beginning of the execution and deviations are not managed: to meet the applications' deadlines, resources must be allocated carefully. This paper proposes an extension to Spark, called xSpark, that is able to allocate and redistribute resources to applications dynamically to meet deadlines and cope with the execution of unanticipated applications. This work is based on two key enablers: containers, to isolate Spark's parallel executors and allow for the dynamic and fast allocation of resources, and control-theory to govern resource allocation at runtime and obtain the precision and speed that are needed. Our evaluation shows that xSpark can (i) allocate resources efficiently to execute single applications with respect to set deadlines and (ii) reduce deadline violations (w.r.t. Spark) when executing multiple concurrent applications.

Index Terms—Distributed architectures; Control Theory; Quality assurance; Batch processing systems

1 Introduction

The resource management of web-, service-, and cloud-based applications [1]–[6] has been studied for years. In these cases resources are usually provisioned to meet response times and/or throughput thresholds, and their fulfillment typically depends on the intensity and variety of incoming requests [7]. Big-data [8] applications are different: they are batch computations that transform, aggregate, and analyze (extremely) large amounts of data. Special-purpose frameworks [9]–[11] slice these data and carry out their analysis by means of parallel processes executed on a distributed cluster of virtual (or physical) machines. Inputs are provided once and for all at the start of a run, and a single execution may take several minutes or hours. The response time is defined as the time it takes to process the entire set of inputs of a single application run, and resources are provisioned to meet a *deadline*, i.e., a maximum threshold for the response time [12]. Furthermore, big-data applications that exploit the same framework, that is, the same cluster of machines, compete for the resources made available by the framework itself.

Spark [13] is the most widely used framework for these applications. It is more flexible than Hadoop [9] and can support more complex computations. Indeed, a Spark program can embed multiple transformations and actions by organizing them in a direct acyclic graph (DAG). Spark allocates resources to applications statically, at the beginning of their execution, and tends to use all the provisioned resources. The only dynamism managed by Spark refers to switching preallocated executors off or on; for example when they remain idle for a user-defined amount of time, or some tasks have to wait for too long (and idle executors are available). In addition, the resources that are provisioned to executors (e.g., CPU cores) cannot be changed. Scalability is only horizontal and is based on simple time-outs and on the availability of preallocated executors. This means that the resources that are allocated to the applications —to meet their defined deadlines— must be planned carefully, and that runtime deviations are not managed. In addition, Spark cannot (dynamically) redistribute the available resources among the concurrent applications, nor among applications whose concurrent executions were not planned at the beginning.

Literature shows that the static resource allocation problem for big-data applications is a hot research topic. For example, [14], [15] propose solutions for the resource allocation of Hadoop applications, [16], [17] introduce resource optimization models for the a-priori allocation of resources in Spark, [18] presents a performance model of Spark applications. Runtime resource allocation, on the other hand, has received limited attention so far.

The runtime resource allocation problem can be seen as the capability to provision resources as needed, and not just as initially planned. This is not a simple task, since the provisioning will depend on multiple elements. Some will be known beforehand (e.g., the size of the input data set), some will be known after a profiling step (e.g., the nominal performance of the system), while others will only be known during the actual execution (e.g., performance, failures, and the characteristics of other applications that are competing for the cluster's resources).

Dynamic resource provisioning becomes a means to make different applications, which execute concurrently, share resources efficiently. Dynamic provisioning would allow for (i) redistributing resources to the different applications as needed, and (ii) accommodating the execution of new unforeseen applications, even when the resources they need might have already been allocated to others.

This paper presents xSpark (e**x**tended **Spark**), an extended version of Spark that enables the fast and fine-grained,

dynamic provisioning of resources to single or multiple big-data applications running on shared infrastructure. If we focus on a single application, xSpark allows one to dynamically acquire and release resources to meet set deadlines precisely. This way xSpark reduces the resources needed to execute an application without over-provisioning, and helps take into account contingent situations like failures and unexpected delays. If we focus on multiple applications, it redistributes provisioned resources and can also help minimize deadline violations by supporting different scheduling policies — based on actual needs.

xSpark is based on two key enablers. On the one hand, containers —a lightweight virtualization technology [19], [20]— allow for the isolation of processing components (executors) and for the fast and fine-grained provisioning of resources (vertical scalability) [21], [22]. On the other hand, control theory provides the machinery to govern the allocation at run time with the required precision and speed. xSpark starts with an initial approximate resource allocation, and then employs hierarchical, heuristic-based and control theoretical planners to adjust the provisioned resources as the applications execute. Runtime controllers enact fast, fine-grained resource allocations: the CPU time of each Spark executor is allocated with a control period of just 0.5 seconds. Memory is also dynamically provisioned by exploiting off-heap allocation. Our evaluation shows that xSpark can allocate resources dynamically to allow applications to meet deadlines with an error that is always less than 2.5% when dealing with homogeneous data and less than 5.5% with skewed data. It also demonstrates that xSpark can reduce deadline violations with respect to Spark when controlling multiple concurrent applications.

To summarize, the main contributions of this paper are: (i) a container-based extension of Spark, called xSpark, (ii) off-heap-based dynamic memory management, (iii) a hierarchical planner for runtime resource allocation based on a heuristic and a gray-box control-theoretical model, (iv) an additional control-theoretical supervisor that oversees the execution of multiple applications on the same framework, and (v) a thorough evaluation in which we considered both single and multiple applications concurrently.

The rest of the paper is structured as follows. Section 2 introduces Spark and the way it works. Section 3 presents xSpark. Section 4 describes the heuristic used for the preliminary resource allocation, while Section 5 focuses on the control-theoretical solution used to adjust the initial plans at runtime. Section 6 explains how xSpark can allocate resources to control the execution of multiple concurrent applications. Section 7 evaluates the proposed solution and xSpark, Section 8 surveys related approaches, and Section 9 concludes the paper.

2 SPARK

Spark is deployed on a cluster of (virtual) machines and employs a master/worker architecture. The *Driver Program* (i.e., the big-data application to execute) starts by creating a *Spark Context* that interacts with the *Master Node* to manage the parallel computation. The *Master Node* is the manager of the actual computing resources, which are called *Worker Node*s. Each worker node is installed on a dedicated machine and contains an *Executor* that runs for the entire lifetime of the application. The executor performs multiple *tasks* in parallel using a thread pool, and the number of parallelized tasks depends on the number of CPU cores it has been given.

Different applications may share the same cluster. Each would have its own Spark context; master and worker nodes would be shared, but executors would be assigned to their respective applications before execution. Tasks can persist the results of their computations on a distributed storage layer (e.g., HDFS [23]), hosted on the Spark cluster, or on dedicated machines.

To fully comprehend Spark one must understand driver programs. The simple Python program of Figure 1 analyzes a text file in which each line has the following structure $class$:$word$, where $word$ is an English word of a specific $class$: verb, adjective, name, etc. The goal is to identify the words that are not verbs, and that share the same first and last letters. Line 2 creates context sc by providing the URL ($local$, in this case) of the master node and the name of the application ($example$). The statement starting at line 3 comprises 6 Spark operations. Besides reading from file $dataset.txt$ (line 3), it splits each line in the original data set into two parts: word and actual class (lines 4 and 5). It then proceeds to create lists of words that share the same class (line 6), and eliminates the list of $verb$s (line 7). The remaining lists are flattened to create x, that is, the list that contains all the words that are not verbs. The statements at lines 9 and 11 use x to create y and z. They are both lists of tuples of the form $(c, words)$ where c is the first/last character of the $words$. Finally, line 13 computes the $result$ by performing the cartesian product of y and z to obtain a list of tuples of the form $((c_f, words_f), (c_l, words_l))$, where f and l stand for the first and last characters, respectively. It then proceeds to perform the set intersection of $words_f$ and $words_l$ to find all the words that start and end with the same letters. The result is collected at line 15.

Spark operations manipulate *RDD*s (Resilient Distributed Dataset). An RDD is an immutable and fault-tolerant collection of *records* that is split into multiple redundant *partition*s to facilitate parallel computation. Operations can be of two

```
 1  from pyspark import SparkContext
 2  sc = SparkContext('local','example')
 3  x = sc.textFile("dataset.txt")
 4       .map(lambda v: v.split(":"))
 5       .map(lambda v: (v[0], [v[1]]))
 6       .reduceByKey(lambda v1, v2: v1 + v2)
 7       .filter(lambda (k,v): k != "verb")
 8       .flatMap(lambda (k, v): v)
 9  y = x.map(lambda x: (x[0], x))
        .aggregateByKey(list(),
10            lambda v1,v2: v1+[v2],
              lambda v1, v2: v1+v2)
11  z = x.map(lambda x: (x[-1], x))
        .aggregateByKey(list(),
12            lambda v1,v2: v1+[v2],
              lambda v1, v2: v1+v2)
13  result = y.cartesian(z)
              .map(lambda ((k1,v1), (k2, v2)):
              ((k1+k2), list(set(v1) & set(v2))))
14  result = result.collect()
```

Figure 1: Example Spark code.

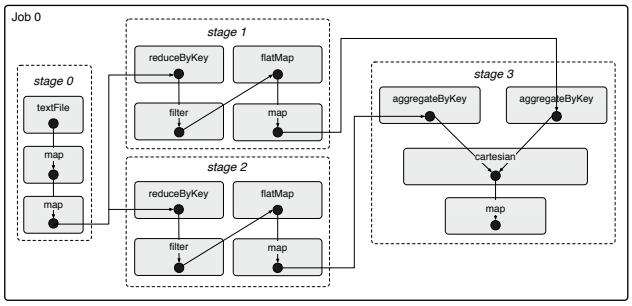


Figure 2: The DAG of our example application.

kinds: *transformation*s create new RDDs (e.g., map, filter, etc.), while *action*s perform computations that generate values (e.g., count, first, collect, etc.). Spark treats the former *lazily*, that is, it chains them together for optimization purposes, and only truly performs them when an action is encountered. This makes Spark particularly efficient when executing iterative algorithms (e.g., machine learning applications or computations on graphs).

Spark starts executing a program by identifying *job*s, delimited by the presence of actions in the code, and *stage*s (within jobs), delimited by operations that require data to be shuffled (i.e., moved among executors), thus breaking locality. Indeed, Spark distinguishes between *narrow* and *wide* transformations specifically for this purpose; the former do not shuffle data (e.g., map, filter, etc.), while the latter do (e.g., reduceByKey, etc.). Spark "identifies" all the operations that are to be executed, up to the first action, and materializes them as a directed acyclic graph (DAG)[1]. The DAG defines the execution order among stages, and defines the extent to which stages can be executed in parallel. Note that a task performs all the operations of a particular stage on a partition of the input RDD of that stage. As previously stated these tasks are executed in parallel, depending on the number of available executors and on the number of CPU cores assigned to each executor.

Figure 2 shows the single job that is created from the example code (there is only one action, i.e., the final *collect*) and its four stages. $Stage0$ comprises the operations from line 3 to 5, and ends with the reshuffling operation *reduceByKey* of line 6. Since statements at line 9 and 11 use the RDD generated by this last operation (they both exploit x), and RDDs are immutable, they could evolve in parallel, but the parallelism starts after reshuffling, that is, just after the end of the stage. This is why the operations at lines 6, 7, and 8 are duplicated and are part of both $Stage1$ and $Stage2$, which also comprise the specific *map*s that help create y and z. Since operations *aggregateByKey* at line 9 and 11 require data shuffling, they cannot be part of the parallel stages but they initiate $Stage3$. This last stage aggregates the data from the two preceding stages, computes the cartesian product, and applies the final *map* transformation. The intersection of the two sets is plain Python and thus is not part of the DAG.

1. The DAG is not the control-flow graph of the job's code. It does not contain branches and loops since Spark has already resolved them.

3 xSpark

To extend Spark with *dynamic* resource provisioning, xSpark[2] modifies both its architecture and processing model. xSpark focuses on *stages*. Instead of considering complete applications, xSpark reasons on per-stage deadlines as a means to decompose the overall execution time. Since stages are composed of diverse operations with different degrees of complexity, they must be modeled and controlled individually. xSpark creates dedicated executors for each stage, instead of general-purpose executors that can execute the tasks of any stage, as Spark would normally do. This way, the resources that are (dynamically) provisioned to a given executor will only impact the performance of the stage that it is associated with. This gives xSpark a fine-grained control of the execution of the different stages, and thus of the whole application. Moreover, when a stage is submitted for execution, one executor per worker node is created and bound to that stage. This allows xSpark to equally distribute the computation and the data among the whole cluster.

xSpark uses containers (Docker[3]) to isolate multiple executors running on the same worker node, and to allocate CPU cores and memory (using Linux *cgroups* [24]). In particular, xSpark uses CPU quotas to allocate small fractions of cores[4] to containers/executors. Spark statically preallocates the memory that is associated with each executor, and thus with each application. xSpark, on the other hand, distinguishes between *heap* memory, which can only be allocated statically, and *off-heap* memory, which can be managed dynamically; a small portion of heap memory is reserved for each application, while the remaining memory is assigned dynamically through off-heap memory.

To operate, xSpark requires an appropriately annotated DAG of the entire application. For each stage, the DAG must contain: the execution time, that is, the stage's duration, the number of identified tasks per stage, the number of input (read) records, the number of output (written) records, and the nominal rate, i.e., the number of records that a core processes in a second. These numbers can come from an initial profiling execution, which is often a viable solution since Spark applications are usually not executed only once, and tend to be *long-lasting* assets. Annotated DAGs are stored in the *Annotated Application DAG Repository* and are retrieved as soon as an application is submitted for execution. Note that xSpark does not require that provided annotations be precise with respect to the used input data since they only serve as an initial assumption; xSpark will cope with imprecisions and changes at runtime.

3.1 Architecture

Figure 3 shows xSpark's architecture. Gray boxes represent components that we added to Spark, while dark-gray boxes correspond to control components. White boxes represent the existing components that we modified.

xSpark is based on four controllers. A single, centralized **Memory Manager** (Section 3.2) is deployed on the master node; it manages how memory is shared among running

2. The source code of xSpark is available at https://github.com/deib-polimi/xSpark

3. https://www.docker.com

4. Quotas assign a specific quota of the CPU period to a container.

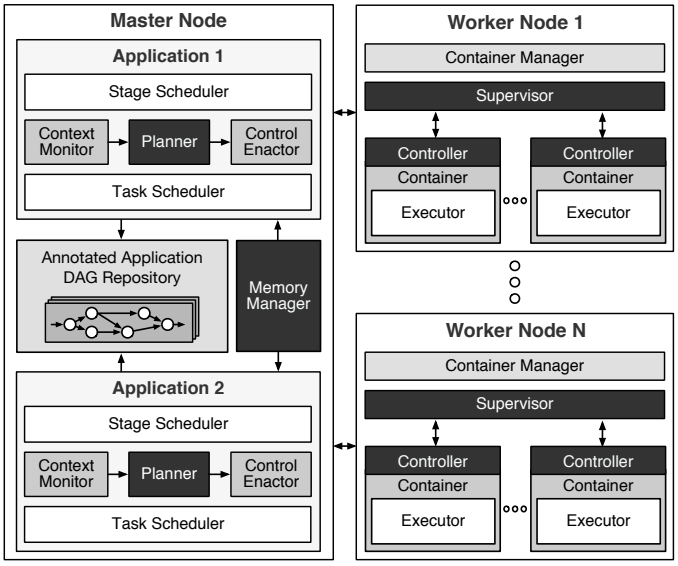


Figure 3: Architecture of xSpark.

applications. As previously stated it considers both heap memory, which cannot be varied at runtime, and off-heap memory, which can be added and removed on demand.

Each application is controlled by a heuristic-based **Planner** (Section 4), implemented within the master node, that oversees the execution of the different stages of the application. For each stage, it computes a deadline, calculates the amount of CPU cores needed to meet it, and assigns them to the executors allocated to the stage. *Context Monitor* oversees the life-cycle of the stages scheduled by *Stage Scheduler*, while *Control Enactor* creates a containerized executor for the submitted stage on each worker node, with the memory limits imposed by *Memory Manager*.

Unfortunately, many factors can influence the actual performance and invalidate the estimation: for example, the quantity of filtered-out records, available memory, number of used nodes, size of storage layer, etc. This is why each executor is equipped with a **Local Controller** (Section 5), based on control theory; it is used to fix these imprecisions by dynamically modifying the amount of CPU cores assigned to the executor. These controllers interpret the estimated deadlines from **Planner** as desired durations or set-points, and aim to allocate just the right amount of resources (i.e., ideally, the minimum amount) to meet the local deadline. xSpark requires that all the executors working on the same stage process the same number of tasks to avoid synchronizing the controllers that are dedicated to the same stage since they work on the same deadline and data quantity. By combining these lightweight controllers and the fast vertical scalability of containers, xSpark is able to achieve control periods of less than a second.

The executors created for different applications can be executed simultaneously and share the resources of the same worker nodes: for example, Figure 3 shows the case of two applications that share the same cluster. Moreover, new applications could be submitted for execution while others are already running and may saturate the cluster. To cope with these issues an additional **Supervisor** (Section 6) is added to each worker node to oversee the local controllers and solve possible resource contentions among the different applications[5] Different policies are supported.

*Supervisor*s also allow xSpark to speed up the computation of stages when the load is low by allocating more resources than the ones strictly needed to satisfy the desired progress rates (e.g., the cores computed by the local controllers). The speed up mechanism is entirely configurable.

3.2 Memory Management

At application-submission time, Spark requires users to specify the amount of memory to dedicate to each executor. When dealing with a single application, choosing this value is simple. In general, picking a larger value will reduce the probability of disk swap and improve performance. However, we still need to pay attention to the system load, because if the heap tries to grow, and there is not enough free memory in the system, the executor (i.e., the JVM) will crash.

Choosing the right amount of memory for each executor when dealing with multiple applications is more complex. Hypothetically, if one could know in advance the number of applications being submitted to the cluster, s/he could partition the available memory among them. Since this is not always feasible, one can either under-provision the running applications to save memory for possible future applications, or allocate the memory *dynamically*.

Unfortunately one cannot resize the heap memory of a JVM at run time: one would need to kill the process and restart it with a new configuration. Knowing that Spark postpones the launch of an application if the requested allocation of memory is not satisfiable, we need to pay attention when choosing the application's memory value. Assigning a high amount of memory would cause application executions to be serialized, while deciding for a lower value would allow for a higher number of applications in parallel, at the price of an increased risk of disk swapping.

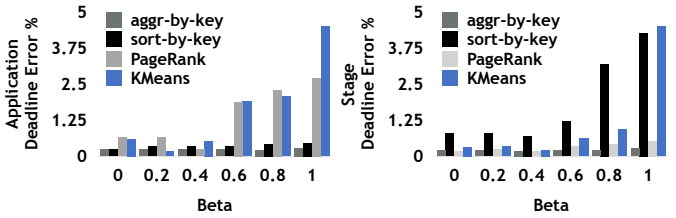
One possible solution is to use off-heap memory to make memory boundaries flexible; even if the best performance is obtained when operating with on-heap memory, Spark can use off-heap allocation both for execution and storage. Off-heap memory refers to objects that are managed directly by the operating system and stored outside the process' heap, that is, they are not processed by the JVM's garbage collector. Accessing off-heap data is slightly slower than accessing on-heap data, but it is faster than reading from and writing on a disk [26].

Since Spark does not provide a way to dynamically resize off-heap memory, xSpark adds the *Memory Manager* component. xSpark launches an application with a relatively small amount of heap memory and dynamically resizes the provisioned off-heap memory according to the number of applications running at any given time. Given a set of running applications A, and the total memory of the cluster M, *Memory Manager* uses the following strategy:

- The quantity of heap memory h allocated to an application is fixed and configurable by the user;
- Given the allocated heap memory $h * |A|$, the remaining portion $O = M - h * |A|$ is equally distributed to

5. Recall that each worker node manages the same applications and thus contentions can be resolved in a distributed way.

the running applications through the off-heap memory mechanism, meaning that each application is set to use at maximum $o = \frac{O}{|A|}$ off-heap memory in addition to h;
- When a new application is submitted for execution, the total off-heap memory becomes $O' = O - h$ and each application reduces its off-heap quota ($o' = \frac{O'}{|A|+1}$);
- When an application terminates, it frees its heap and off-heap memory $O'' = O' + h$, and the other applications increase their off-heap quotas to $o'' = \frac{O''}{|A|-1}$.



(a) Impact on ϵ_a (b) Impact on $avg(\epsilon_s)$

Figure 4: Sensitivity analysis for $beta$.

4 Planner

Each application uses a heuristic-based planner to compute per-stage deadlines. The only input is the foreseen deadline (execution time) for the entire application. When a stage $s \in S$ —where S is the set of the application's stages— is submitted for execution, xSpark computes a preliminary deadline using the following formula:

$$deadline_s = \frac{\alpha \cdot appDeadline - spentTime}{w_s} \quad (1)$$

where $spentTime$ is the time already spent on executing the application, $appDeadline$ is the deadline submitted by the user, and α is a constant, between 0 and 1, that xSpark uses to set the level of conservativeness foreseen to respect the deadline. If $\alpha = 1$, the execution time will be controlled (by the lower control levels) to meet the deadline exactly, while with smaller values for α the control will be more conservative and allow for possible delays due to imprecisions in the control[6]. w_s, the weight of the stage, is computed as follows:

$$w_s = \beta \cdot |R_s| + (1 - \beta) \cdot d_s \quad (2)$$

where R_s is the set of the stages still to be executed, s included, and d_s, the ratio between the sum of the profiled durations of the stages in R_s and the profiled duration of s itself. Hence:

$$d_s = \frac{\sum_r^{R_s} duration_r}{duration_s} \quad (3)$$

Since the performance measured during profiling may be different from the one seen at run time, we mitigate the heuristic by means of constant β. To find a proper value for β, we performed a sensitivity analysis (see Figure 4) using four benchmark applications that we also used to evaluate xSpark (Section 7).

We defined the error on the application deadline (a negative value implies a violation) as:

$$\epsilon_a = 100 \cdot \frac{appDeadline - appDuration}{appDeadline}\% \quad (4)$$

and the error on a stage deadline as

$$\epsilon_s = 100 \cdot \left| \frac{stageDeadline - stageDuration}{appDeadline} \right| \% \quad (5)$$

6. Note that when the application deadline is missed, the heuristic computes negative or null stage deadlines, and the estimated number of cores will be equal to the number of available ones.

where $app|stageDeadline$ is the desired execution time of the application/stage, respectively. Similarly, $app|stageDuration$ represents the actual execution time. Note that Equation 4 does not use the absolute value to distinguish between delays and early terminations, and Equation 5 uses the entire application's deadline, and not the stage's deadline itself, as the denominator, to have a more relevant indicator of the control error. We also defined the average and the standard deviation of ϵ_s for all the stages of an execution of an application as $avg(\epsilon_s)$ and $std(\epsilon_s)$.

Figure 4 shows how ϵ_a and $avg(\epsilon_s)$ change with respect to different values of β (ranging between 0 and 1). Even though we could not identify an optimal value for all the applications we used, the values between 0.2 and 0.4 were reasonably adequate, and thus we set β to 0.3. This way we avoid over-fitting profiling data, which is what would happen if we only used d_s. Note that all w_s are computed at run time, since the order of stage execution cannot be known a-priori if there are parallel threads in the DAG. The number of CPU cores to be allocated for executing the stage in a time that is equal to the computed stage deadline (i.e., minimum amount of cores that avoids the deadline violation) is then estimated as:

$$estdCores_s = \lceil \frac{inputRecords_s}{deadline_s \cdot nomRate_s} \rceil \quad (6)$$

where $inputRecords_s$ is the number of records that must be processed by s, and $nomRate_s$ provides the *nominal rate* of s, i.e., the number of records processed by a single core per second (obtained during profiling). $inputRecords$ depends on the sum of the data written by the parent nodes in the DAG (i.e., the sum of records produced by the parents).

The final step computes the initial number of cores and the number tasks to be processed by each different executors. xSpark distributes the load equally amongst all the available workers by creating one executor per stage per worker. This way each executor holds, and thus remotely reads during shuffles the same amount of data meaning that xSpark can compute the same deadline for all the executors (as seen by using function $deadline$). The initial number of cores per executor is computed as follows:

$$intlCoresExec_s = \lceil \frac{estdCores_s}{cq \cdot numExecutors} \rceil \cdot cq \quad (7)$$

where $numExecutors$ is the number of executors (which is always equal to the number of workers), and cq stands for *core quantum*, a constant that defines the quantization applied to the resource allocation. The smaller the value, the

more precise the allocation is. In our experiments we set this constant to 0.05 to allocate cores with a precision of up to 0.05 cores and obtain a low error.

This is only the foreseen, initial core allocation for each local controller, which may continuously change it at runtime. The tasks are then distributed equally (excluding remainders) to the different executors. Since clearly not all deadlines are feasible, given the provided resources and input datasets, xSpark conducts a feasibility check before starting any execution by means of the aforementioned heuristic and warns the user if it is the case. We acknowledge that more sophisticated and precise approaches exist (e.g., [27], [28]), but they are on average too slow for a preliminary check.

5 Local Controllers

Each containerized executor has its own local controller to fulfill a per-stage deadline in the face of exogenous disturbances, by dynamically allocating CPU cores. The centralized heuristic-based control loop determines the desired duration, the maximum and the initial cores to allot for each executor and the number of tasks to be processed prior to the execution of a stage. The local controllers will then continuously adjust the numbers of cores allocated to the executors, according to the work already accomplished and to the desired completion rate.

Executors that are dedicated to different stages are independent, and therefore so are their controllers. Executors that work in parallel on the same stage must complete the same amount of work (i.e., number of tasks) in the same desired time; this means that their local controllers are also independent and do not need to communicate with one another. Given their decentralized nature, we concentrate on the design of a single controller, while the interactions of controllers with their respective *Supervisor*s are extensively detailed in Section 6.

5.1 Controlled System

To derive a model of the controlled system we started with the following two assumptions.

> **Assumption 1:** *at steady-state, with constant resource allocation and disturbances, the progress rate is also constant, and is a function —$f(\cdot)$ to name it— of the allocated resources and of the disturbances. This function is generally non-linear, but regular enough for the values of interest of the involved quantities.*

This assumption is technically required to express the mathematical model. Later in this section we will show how the resulting algorithm can perfectly cope with time-varying disturbances and allocations. For completeness we also notice that progress rates may also depend on available memory. However, memory is either sufficient, thus allocating more is useless, or it is not sufficient. In the latter case there is a performance degradation, but this depends on fine-grained effects connected to cache, swap, and so on, and are best viewed as disturbances at this level. Memory management is especially critical when dealing with multiple applications in parallel. A description on how xSpark dynamically provisions memory is provided in Section 3.2.

> **Assumption 2:** *the output of $f(\cdot)$ exerts its effect through an asymptotically stable, linear, time-invariant dynamic system with unity gain and relative degree.*

This assumption is met if the reaction of the controlled system to a modification in the resource allocation is faster than the control period. In literature the control period is usally set in the order of minutes, but the use of containers, which can be resized in hundreds of milliseconds, allows for a control period of a second or less, while preserving the hypothesis above. In the absence of significant actuation delays thanks to containers, we can safely suppose that the system's relative order is one, but we cannot know anything about its dynamic structure. Therefore, we assume the simplest possible form with one pole and no zeros, that is:

$$\rho(k) = p\rho(k-1) + (1-p)f(c(k-1)) \tag{8}$$

where ρ is the progress rate and p the pole. Having p in the $[0, 1)$ range ensures that the control system is asymptotically stable, as for that $|p| < 1$ would be sufficient; the choice to limit p to positive values further ensures that the reference completion rate will be tracked without oscillations, i.e., with good regularity.

The model is affected by bounded uncertainty, but affine and time-invariant. It belongs to the Hammerstein class, which opens to interesting generalizations. There is a lot of literature on the identification of Hammerstein models, from works like [29] to the time-varying case, which could be useful in the future to generalize our results [30]–[32]. To date, we obtain $f(\cdot)$'s bounds through profiling, while p in (8) is estimated using step response analysis.

5.2 Control Synthesis

The progress set point is chosen based on the desired completion time, which is received from the upper control layer as illustrated in Figure 5. t_{co} is the desired completion time (per-stage *deadline* minus the time at the start of the elaboration) and $\phi \in (0, 1]$ is a configuration parameter that determines the extent to which the control will attempt to complete the execution earlier than the deadline, as a safety measure (as α in Equation 1 but for stage-deadlines).

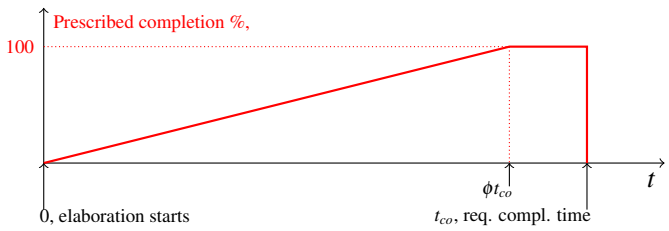


Figure 5: Set point generation for a local controller.

To design the control law, we denote by q the control timestep —i.e., the time between two calculations of $c(k)$— and express the dynamics between $c(k)$ and the accomplished completion percent $a_{\%}(k)$; for the data $e(k)$ elaborated up to step k (the rate between $k-1$ and k is decided at $k-1$) we get:

$$e(k) = e(k-1) + q\rho(k-1), \tag{9}$$

so that indicating with D_o the total amount of data to process, $a_{\%}(k)$ evolves as:

$$a_{\%}(k) = 100\frac{e(k)}{D_o} = a_{\%}(k-1) + \frac{100q}{D_o}\rho(k-1). \quad (10)$$

In z transfer function form (8) and (10) respectively read:

$$\frac{\rho(z)}{u(z)} = \frac{1-p}{z-p}, \quad \frac{a_{\%}(z)}{\rho(z)} = \frac{100q/D_o}{z-1} \quad (11)$$

where $u(k) = f(c(k))$. The transfer function from u to $a_{\%}$, that is the linear part of the dynamics seen by the controller, is thus:

$$\frac{a_{\%}(z)}{u(z)} = \frac{100q(1-p)/Do}{(z-1)(z-p)}. \quad (12)$$

To track the ramp set point showed in Figure 5, the loop transfer function must have two poles in $z = 1$, which can be achieved by a PI (Proportional plus Integral) controller of the form:

$$C(z) = K\frac{z-a}{z-1}. \quad (13)$$

Since specifications in terms of durations are given in time units (e.g., seconds) and not as numbers of sampling periods, it is convenient to re-interpret the control loop in the continuous time. To do that we back-apply the forward difference method and we obtain the s transfer function:

$$P_c(s) = \frac{\mu_P}{s\,(1+s\tau_P)}, \quad \mu_P = \frac{100K_f}{Do}, \; \tau_P = \frac{q}{1-p}, \quad (14)$$

whereas for the controller we have:

$$C_c(s) = \mu_C\frac{(1+s\tau_C)}{s}, \quad \mu_C = \frac{K(1-a)}{q}, \; \tau_C = \frac{q}{1-a}. \quad (15)$$

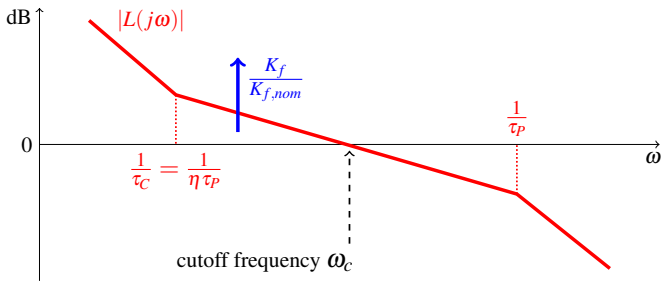


Figure 6: Bode magnitude diagram of the required open-loop frequency response.

We need to force the magnitude Bode diagram of the open loop frequency response $L(j\omega) = C_c(j\omega)P_c(j\omega)$ to behave like in Figure 6. We substitute $f(\cdot)$ with a (positive, bounded, unknown) gain K_f, for which we assume a nominal value $K_{f,nom}$, and make the controller time constant τ_C proportional by $\eta > 1$ to τ_P, selecting the controller gain so that the cutoff frequency —for $K_f = K_{f,nom}$— is the logarithmic mean of τ_C and τ_P. This provides the nominal cutoff frequency ω_c and phase margin φ_m as:

$$\omega_c = \frac{1}{\eta\sqrt{\tau_P}}, \quad \varphi_m = \arctan\left(\sqrt{\eta}\right) - \arctan\left(\frac{1}{\sqrt{\eta}}\right). \quad (16)$$

Higher values of η yield higher margins but also a lower-frequency zero a in controller (13); this in turn would result in control kicks in the presence of step-like set point variations —which do not occur according to Figure 5— and in longer disturbance recovery times. Omitting lengthy computations, once the continuous-time controller is tuned this way and converted back to discrete time, we get:

$$K = \frac{D_o(1-p)}{100qK_f\sqrt{\eta}}, \quad a = \frac{\eta+p-1}{\eta} \quad (17)$$

to put into (13). With a value of η around ten —corresponding to φ_m around 55°— as a reasonable default, the controller behaves satisfactorily provided that q is small enough with respect to the required completion time, which is plainly a matter of reasonableness. The discrete-time controller in state space form reads:

$$\begin{cases} x_C(k) &= x_C(k-1) + (1-a)\left(a^{\circ}_{\%}(k-1) - a_{\%}(k-1)\right) \\ rc(k) &= Kx_C(k) + K\left(a^{\circ}_{\%}(k) - a_{\%}(k)\right) \end{cases} \quad (18)$$

where $a^{\circ}_{\%}(k)$ is the prescribed progress percentage and $rc(k)$ the computed core allocation at each k control step. As a final remark, in a real application it may transiently happen that the controller computes a negative $rc(k)$, or one exceeding the number of available CPU cores. Denoting by c_{min} and c_{max} the minimum and maximum number of available cores in the worker, $rc(k)$ needs to be clamped within the two, and the state of (18) has to be recomputed to maintain consistency with the input and output. This is done by:

$$x_C(k) = \frac{c(k)}{K} - a^{\circ}_{\%}(k) + a_{\%}(k). \quad (19)$$

Note that rc stands for *requested cores* since this value is read and eventually modified by the worker's *Supervisor* as detailed in Section 6.1.

Finally, we need to assess a couple of important properties of the control system. The **asymptotic stability** refers to nominal conditions, i.e., when the model describes the controlled system exactly. With the controller as defined in Equation 18, phase margin considerations allow us to state that the closed-loop system is guaranteed to be asymptotically stable, provided that the reduction of the said margin due to the sampling and holding is not excessive. To this end, one could set a maximum φ_m reduction $\delta\varphi_m$, and then bound q to the upper limit $2\delta\varphi_m/3\omega_c$, with $\delta\varphi_m$ in radians, as usually done in digital controls.

As for the **nominal performance**, the nominal response time τ_r of the closed loop is the inverse of the cutoff frequency, i.e.,

$$\tau_r = \eta\sqrt{\tau_P}. \quad (20)$$

The time to reach a new set point, or to recover from a step-like disturbance like the abrupt unavailability of a core —provided the goal is still attainable— can be quantified as 5 times τ_r. This is fine because τ_P is the time constant used by the step-by-step resource allocation to act on the processing speed, thus in a well sized system it is small with respect to the control-relevant time scale.

Finally, as for applicability caveats, we need to distinguish between model errors due to *mismatch* and *variability*. In the former case the controlled system does not change its behavior, but the model does not represent it exactly. In the latter, the model may even start out as a perfect *replica* of the

system, but then this can change its behavior. In our case variability should be scarcely relevant, as the typical task consists of elaborating a huge amount of data, but the single operation is simple and self-similar. If this assumption is heavily violated, however, the applicability of our solution may be questioned.

As for mismatch, the main point is whether or not the structural assumptions made on the model are reasonable. Again, this can be assessed by off-line profiling, or through simulation if a reliable enough model of the applications being considered is available. In this respect it is difficult to make general statements on the applicability of our technique, except that once the convenient off-line testing is carried out, no post-deployment issues are to be expected.

6 SUPERVISORS

When dealing with multiple applications running at the same time we need to take into account two possible cases: either we know the workload (i.e., the applications that will be executed in parallel) in advance, or we do not.

In the former case, one can statically allocate resources so that all applications have enough (or at least some) resources. This a-priori allocation could be wrong or sub-optimal, for example, if one application does not really need all the resources that it is given or it needs more. While in the former case we only have a waste of resources, in the latter case we might witness under-performance and missed deadlines. Furthermore, if all resources are pre-allocated there is no room for unplanned applications that may be submitted for execution while the others are already running.

As previously stated, xSpark adds a new hierarchical control layer, implemented by multiple distributed *Supervisor*s to deal with these scenarios. This is needed to oversee how the executors manage concurrent applications when the resources provided by the cluster are not enough to allow each application to meet its deadline.

6.1 Controlling Local Controllers

We deploy a *Supervisor* to each worker node; it is responsible for managing the resource demands of executors (local controllers) running on that machine. Every local controller continues to autonomously determine the CPU cores needed to follow its application's desired progress rate, but the *Supervisor* can decide to modify this value according to the state of the resources.

Local controllers that are deployed to the same machine are synchronized and have the same control period. The *Supervisor* collects resource allocation requests in vector $\bar{rc}$ (requested cores), where rc_i is the specific request made by application i, and produces a new core allocation as vector $\bar{cc}$ (computed cores):

$$\bar{cc} = \gamma * \bar{ac} + (1 - \gamma) * \bar{rc} \tag{21}$$

where $\bar{ac}$ is an allocation vector that uses all available resources. Parameter $\gamma \in [0, 1]$ allows us to boost the execution speed. If $\gamma = 1$, the allocation saturates all resources, and applications will be executed faster than planned. If $\gamma = 0$, the allocation only considers the resources requested by the local controllers. This way we keep the cluster's utilization low and we save resources. The different strategies that can be adopted to compute $\bar{ac}$ are described in Section 6.2.

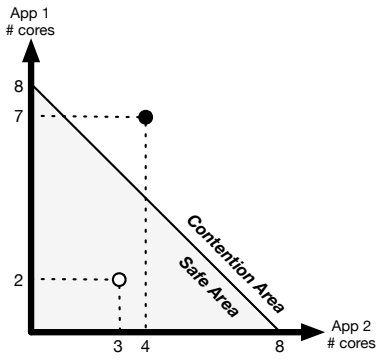


Figure 7: Example of two resource requests made by two applications running in a shared environment; the maximum number of allocatable cores is 8.

Since applications are not aware of one another, the *Supervisor* needs to check whether there is resource contention. If the total amount of requested cores (c_r), that is the sum of all rc_i, is less than (or equal to) the cluster's available resources (c_{max}), there is no contention. If c_r is greater than c_{max}, we have contention and the requests cannot all be satisfied: we must correct the amount of resources to attempt to manage the contention. The two cases can be easily visualized by considering two applications on a two dimensional plane (see Figure 7). On the two axes we have the resources requested by the two applications. In this example, the maximum amount of resources that can be allocated consists of 8 CPU cores. The white dot represents a feasible allocation, because the sum of the requested cores ($3 + 2 = 5$) is lower than 8. The black dot, on the other hand, represents an infeasible allocation, since the sum ($4 + 7 = 11$) is greater than 8. The region above the thick line represents all the combinations of resource requests that cause contention. In this latter case, we need to find a new pair ac_1 and ac_2, such that $ac_1 + ac_2 = c_{max}$, where ac_i is the value in vector $\bar{ac}$ that corresponds to the i^{th} application.

Finally since the number of CPU cores that the executor will acquire might not be what the executor controller calculated, we also need to update the state of the local controllers to maintain consistency otherwise the next control operations would be based on an incorrect previous state.

6.2 Strategies for Core Allocation

Multiple strategies, that take into account the different static and/or dynamic characteristics of the involved applications (e.g., deadlines or nominal rates), can be adopted when distributing the available cores. The *Supervisor* can use these strategies (i.e., how $\bar{ac}$ is computed) to resolve contention and to speed-up the computation (i.e., to set γ). Note that γ is part of the initial configuration, and thus is set before starting any computation (and before knowing of any possible contention).

We advocate that one can make a parallelism between the problem of allocating resources to xSpark executors and the preemptive online scheduling of sporadic tasks, with

arbitrary deadlines, in a real-time multiprocessor system. In this latter case, it has been proven ([33]–[35]) that no optimal on-line scheduling algorithm exists for sporadic task sets with constrained or arbitrary deadlines. Therefore, we decided to focus on sub-optimal approaches.

One of the most popular dynamic-priority planning-based on-line algorithms is called Earliest Deadline First (EDF) [36]; it uses deadlines to determine the priorities of tasks. We can adopt this same strategy and use execution deadlines to determine priorities when allocating resources for multiple applications, but we can also build more sophisticated strategies. Table 1 compares the different strategies we have devised, using three example applications, under the assumption that only 16 cores are available.

Earliest Deadline First "All" (**EDF**$_{all}$). This strategy allocates all the available resources to the application with the nearest upcoming deadline, even if the application requested fewer resources. In general, this approach completes the execution of a single application before allocating resources to the next one, and completes the applications in a certain order, as defined by the proximity of their deadlines. The algorithm requires the maximum number of allocatable cores (c_{max}), and the time to complete (ttc_i) and remaining tasks (rt_i) of each application.

Table 1 shows that the only application that actually acquires resources is the one with the smallest time to complete (deadline). In this case, we have made the assumption that the tasks each application still has to execute can use all the cluster's cores.

Earliest Deadline First "Pure" (**EDF**$_{pure}$). This strategy uses priorities to allocate resources to applications. An application's priority is defined by its remaining time to complete: the shorter the time, the higher the priority. With this strategy applications with low priorities may be paused. Again, the algorithm requires the maximum number of allocatable cores (c_{max}), as well as the time to complete (ttc_i) and the requested cores (rc_i) for each application. Table 1 shows that the application with the shortest time to complete acquires all the cores it asked for; the second application obtains all the remaining cores, which are fewer than the requested ones. The third application is not granted any resource as they are depleted.

Earliest Deadline First "Proportional" (**EDF**$_{prop}$). This strategy assigns a weight to each running application. The weight is related to its remaining time to complete ttc_i and calculated as

$$w_i = 1 - \frac{ttc_i - min(t\bar{t}c) + 1}{\sum_j^I [ttc_j - min(t\bar{t}c) + 1]}$$

where $t\bar{t}c$ is the vector composed by all the ttc_i and $i \in I$ is the set of applications running at that time. If there is not a huge difference in terms of remaining times to complete, all the applications will acquire a portion of the available resources. The algorithm requires the requested number of cores for each application (rc_i) and the maximum number of allocatable cores (c_{max}), and produces the applications' weights (w_i). Requested cores are taken into account to avoid giving an application more resources than actually requested, even when its calculated weight is higher than that of other applications. Table 1 shows all applications receive a certain amount of cores, and none of them is paused.

Proportional. This strategy represents the rawest way to allocate available cores. In this case, the weights are calculated as

$$w_i = \frac{rc_i}{\sum_j^I rc_j}$$

where I is the set of applications running at a given time, and $i \in I$. This solution creates a fair distribution of resources since no application is preferred over the others. Table 1 shows that the weights are directly proportional to the amount of cores requested by the applications: the final allocation is proportional to the amount of requested cores.

Speed. This strategy takes into account the applications' average nominal rates, that is, the number of input records each application can process per second per core. This value can be obtained, for example, by profiling the application or can be inferred in other ways. An application's average nominal rate is computed as

$$anr_i = \frac{\sum_s^S nomRate_s \cdot w_s}{\sum_s^S w_s}$$

where S is the set of stages of the application, $s \in S$, $nomRate_s$ is the nominal rate of stage s, and w_s is its weight (see Formula 2). An application's weight can then be computed as

$$w_i = \frac{ANR}{anr_i}$$

where $i \in I$, the set of applications running at a given time, and ANR is the average among anr_i. Table 1 shows that the three applications require 30 cores, but only 16 are available, and that ANR is equal to 5, that is, the average among the three anr_i. Since applications A and B have the same nominal rate, they also have the same weight. C has a lower nominal rate —half the one of A and B— and its weight is therefore double that of A and B. As a result, C obtains half the cluster's cores, while A and B receive one quarter each.

The evaluation presented in Section 7.2 highlights that there is no single strategy that can outperform all the others in all possible scenarios. The strategy to use will depend on the requirements one wants to meet. If the goal is to minimize deadline violations (i.e., delays) one of the EDF-based strategies with $\gamma = 1$ will be preferable. If the goal is to minimize errors (i.e., missed deadlines and anticipated ones) and thus minimize resources, one should consider EDF$_{pure}$, EDF$_{prop}$ with $\gamma = 0$, or Proportional with $\gamma = 1$. Finally, Speed might be the best choice if the applications are highly heterogeneous in terms of *nominal rate*s.

Further strategies can considered and easily implemented in xSpark given its modular and hierarchical architecture.

7 Evaluation

This section describes the experiments we conducted to evaluate xSpark. All the experiments used Azure *Standard_D14_v2* VMs with 16 CPUs, 112 GB of memory, and 800 GB of local SSD storage. This kind of VM is optimized for memory usage, with a high memory-to-core ratio. Each machine ran Canonical Ubuntu Server 14.04.5-LTS, Oracle Java 8, Apache Hadoop 2.7.2, Apache Spark 2.0.2 and xSpark. We dedicated five VMs to HDFS and five to Apache Spark and xSpark. The

Application			EDF_{all}	EDF_{pure}	EDF_{prop}		Proportional		Speed		
name	rc_i	ttc_i	ac_i	ac_i	w_i	ac_i	w_i	ac_i	anr_i	w_i	ac_i
A	10	50	16	10	0.97	7.75	0.33	5.28	$6M$	0.83	4
B	8	60	0	6	0.67	5.35	0.27	4.32	$6M$	0.83	4
C	12	70	0	0	0.36	2.90	0.40	6.40	$3M$	1.66	8

Table 1: How the different strategies impact $\bar{ac}$ in a simple example.

datasets were randomly generated with the goal of obtaining homogeneous data; all executions were repeated five times and we show average values.

7.1 Resource Allocation

To assess how xSpark allocates CPU cores dynamically we used eight applications taken from two benchmark suites. *aggr-by-key*, *aggr-by-key-int*, *group-by*, *sort-by-key*, and *sort-by-key-int* stress basic aggregation and sorting capabilities and come from SparkPerf[7]. *KMeans*, *SVM*, and *PageRank* come from SparkBench[8]: the first two are machine learning applications, while the third is a well-known graph processing solution. The first five applications do not contain branches or loops, while the last three are iterative, but the number of iterations are configured at the beginning through parameters. This means that all the executions of each program have the same DAG (see Section 3).

We first used Spark to run each application and set a baseline (*testBase*), that is, to know the shortest execution time given that Spark was configured to use all the resources provided by the cluster: 64 cores in total in our case. The datasets were randomly generated by the benchmark suites, using the application specific parameters reported in Table 2.

App	Parameters
aggr-by-key	$scaleFactor = 5, keys = 5000, tasks = 5000$
aggr-by-key-int	$scaleFactor = 5, keys = 5000, tasks = 5000$
group-by	$scaleFactor = 5, keys = 5000, tasks = 5000$
sort-by-key	$scaleFactor = 50, keys = 5000, tasks = 5000$
sort-by-key-int	$scaleFactor = 60, keys = 5000, tasks = 5000$
KMeans	$iterations = 1, partitions = 1000,$ $dimensions = 20, numClusters = 10,$ $numPoints = 100000000, scaling = 0.6$
PageRank	$iterations = 1, partitions = 1000$ $numVertices = 35000000$
SVM	$iterations = 1, partitions = 1000$ $numExamples = 150000000, features = 10$

Table 2: Benchmarks configuration.

We then used xSpark configured as shown in Table 3 to have a fair comparison against Spark. We executed each application by imposing the same deadlines as the baseline executions (*test0%*), and by relaxing the original deadlines by 20% (*test20%*) and 40% (*test40%*), respectively. Since xSpark works as Spark, deadlines cannot be tighter than the baselines without adding resources. The goal of these experiments was thus to assess the precision with which xSpark can meet deadlines, and how it can optimize the allocation of cores.

7. https://github.com/databricks/spark-perf
8. https://github.com/SparkTC/spark-bench

Param	Value	Range	Description
γ	0	$[0, 1]$	Increment of execution speed (Eq. 21).
α	1	$[0, 1]$	Adherence to app. deadlines (Eq. 1).
ϕ	1	$[0, 1]$	Adherence to stage deadlines (Sec. 5.2).
β	0.3	$[0, 1]$	Divergence from profiling (Eq. 2).
cq	0.05	$(0, \infty)$	Quantization of core allocation (Eq. 7).
q	0.5	sec.	Control period (Eq. 9).

Table 3: xSpark parameters used for the experiments.

To better explain how xSpark works, Figure 8 shows the behavior of a randomly-selected executor in charge of the nine stages of *PageRank* (*test40%*). The black and gray lines refer to the left-hand y-axis and show, respectively, the actual percentage of stage completion ($a_{\%}(k)$ in Equation 10) and the prescribed one ($a^{\circ}_{\%}(k)$ in Equation 18), which is the set point (at each control step k) of the local controller. The blue line refers to the right-hand y-axis and shows the cores allocated to the executor. The E-labeled green vertical lines represent the actual stage ends, while the red dashed vertical lines represent stage deadlines as computed by the planner; the deadline for the last stage is also the deadline of the entire application.

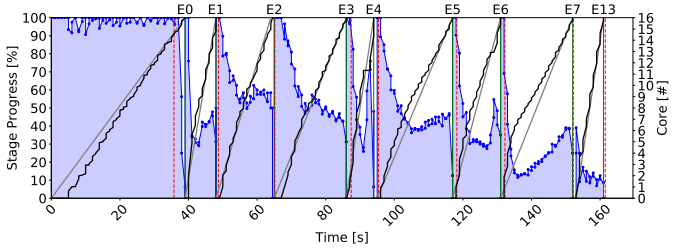


Figure 8: Example of how xSpark executors work.

This chart shows how xSpark fulfills stage deadlines with an error that is close to 0. At runtime, the local controllers foresee the allocation of core fractions to executors: when the actual progress of the stage is lower than the prescribed one, allocated cores are increased, while as soon as it becomes higher, they are quickly decremented. As already said, local controllers exploit a control period of 0.5 seconds and xSpark can thus be quite precise.

Table 4 shows the precision and resource utilization with which xSpark carried out the different experiments. The error in fulfilling set deadlines is always less than 2.5% for

Test	ϵ_a	$avg(\epsilon_s)$	*all*	*util*
aggr-by-key *test0%*	-0.90%	3.00%	60.41	94.40%
aggr-by-key *test20%*	0.78%	1.05%	37.63	58.80%
aggr-by-key *test40%*	0.26%	0.25%	29.15	45.55%
aggr-by-key-int *test0%*	-2.31%	4.81%	62.22	97.22%
aggr-by-key-int *test20%*	0.77%	0.85%	39.21	61.28%
aggr-by-key-int *test40%*	0.77%	0.52%	30.86	48.23%
group-by *test0%*	-1.73%	5.67%	61.57	96.21%
group-by *test20%*	0.28%	0.60%	42.95	67.12%
group-by *test40%*	0.31%	0.36%	34.59	54.05%
sort-by-key *test0%*	-1.31%	0.79%	59.19	92.50%
sort-by-key *test20%*	1.18%	0.41%	45.23	70.68%
sort-by-key *test40%*	0.44%	0.31%	35.44	55.37%
sort-by-key-int *test0%*	-2.48%	1.32%	59.19	92.50%
sort-by-key-int *test20%*	0.32%	0.21%	46.44	72.56%
sort-by-key-int *test40%*	0.42%	0.15%	38.29	59.84%
KMeans *test0%*	-0.92%	1.71%	54.43	85.05%
KMeans *test20%*	0.89%	0.94%	43.08	67.32%
KMeans *test40%*	0.58%	0.72%	36.15	56.49%
PageRank *test0%*	-1.20%	2.70%	58.77	91.83%
PageRank *test20%*	0.71%	1.03%	44.76	69.95%
PageRank *test40%*	0.61%	0.68%	37.93	59.27%
SVM *test0%*	0.71%	2.69%	52.62	82.23%
SVM *test20%*	1.49%	1,58%	41.62	65.03%
SVM *test40%*	1.91%	0.98%	34.39	53.74%

Table 4: Precision and performances of xSpark.

complete applications (ϵ_a), while it can be slightly higher for single stages ($avg(\epsilon_s)$). This is caused by the fact that stages can be very heterogeneous, but the errors compensate each other and the overall error is close to 0%.

xSpark slightly violated the deadline (the negative values of ϵ_a) only when considering as deadline the execution time obtained on Spark (*test0%*) since we had to trade the fine-grained, dynamic allocation capabilities for a bit of performance. If the heuristic computed a stage deadline that is slightly longer than the fastest execution time, xSpark would not allocate all the resources, and the actual execution becomes a bit slower. These violations can easily be avoided by setting α to a reasonable value less than 1, and allow xSpark to consider stricter, more conservative deadlines. This is like asking xSpark to work a bit faster and then be able to meet deadlines even in case of errors or delays. Further experiments suggested that with $\alpha = 0.95$ xSpark never violated set deadlines.

When deadlines are relaxed, xSpark meets them with an error that is always less than 2%. The gain in used resources is significant even when xSpark works with the baseline execution times. Table 4 shows the number of allocated cores per second, where *all* is the average value of allocated cores (i.e., *all* is the ratio between the integral of used cores over the whole execution and the execution duration), and *util* is the percentage of used resources (cores) with respect to the baseline (64 cores).

These numbers help us compare the average resource allocation of xSpark with respect to Spark. Even if Spark provides some rudimentary mechanisms for dynamic resource allocation (see Section 8), in our experiments it always used all 64 cores. xSpark instead allocates the resources according to deadlines, and even with the strictest response times (*test0%*), it was able to use between 2.78% and 17.77% fewer resources than Spark. This is due to the fact that xSpark can immediately release resources when not needed. In particular, the highest saving was with SVM (17.77%), since in some stages the available degree of parallelism was not fully exploited.

Column all shows a significant decrement in used resources when relaxing deadlines, but the experiments witness that there is no "easy" relationship between desired execution times and appropriate amount of resources to achieve them: a manual, experience-based allocation could then be tedious and quite imprecise.

7.1.1 Data Skew

Test	ϵ_a	$avg(\epsilon_s)$	$std(\epsilon_s)$
group-by *test20%* ($s = 1$)	5.44%	2.86%	2.51%
group-by *test20%* ($s = 2$)	-4.31%	2.83 %	1.72%
group-by *test20%* ($s = 3$)	-4.50%	2.46%	2.00%

Table 5: Precision and performance with data skew.

Data skew causes tasks (e.g., key-based operations) to have different durations [37]. Even if managing skewed data is out of the scope of this paper, this problem impacts and degrades both the performance of Spark (by around 20% according to [38]), and the control precision of xSpark. Since xSpark monitors the progress of stage execution (*progress rate*) by comparing the number of executed tasks against the total number of tasks to execute, significantly different durations hamper this estimation.

Before thinking of improving our controllers (e.g., by considering solutions already proposed for MapReduce [39]), we wanted to evaluate how xSpark deals with skewed data. We ran application *group-by* on three different sets of skewed data, generated using Zipf's law [40]: given N values ordered by frequency, the frequency of a value is equal to $\frac{1/k^s}{\sum_1^N 1/n^s}$, where $k \in [1, N]$ is the position of the value in the sequence and $s \in [1, 3]$ identifies the degree of skewness of the data set (the higher the more skewed). We chose application *group-by* since it does not pre-compute any intermediate result/reduction in parallel that may smooth the impact of skewness on the final computation; `reduce-by-key` for example would do that.

Table 5 shows the results of our experiments. Again, we first ran the application on Spark to obtain the baseline deadline, and then performed $test20\%$ with xSpark. The table shows that xSpark can meet the deadline when s is equal to 1 with an error of 5.44%. When s increases to 2 and 3, xSpark slightly violates the deadline (e_a is negative) with an error of 4.40% on average. As mentioned before, the error is higher than when executing applications not (or just slightly) affected by skewed data because of the aforementioned imprecisions of progress monitoring. Moreover, when the data are skewed the control of xSpark is less effective since the duration of a stage is heavily impacted by its long tasks that cannot be parallelized.

Figure 9 shows how skewed data impact the execution of a stage. The figure shows two different behaviors on processing the second stage of application *group-by* imposed by skewed data. The progress rate of the first executor (line A in the figure) moves very quickly to 100%, since

it only receives short tasks that process low-frequency data. In contrast, the second executor (line B) shows a step-like progress rate. The executor is in charge of both short tasks, when the progress increases at a very fast rate (e.g., after 120 and 220 seconds), and three longer ones where progress is constant. In both cases, to complete the execution before the deadline (see the expected progress in light gray), a few cores are needed since the degree of parallelism is limited by data skew. In the first case, xSpark immediately releases allocated resources after completing the stage, while Spark would wait for a predefined amount of time to do it (see Section 8). In the second case, the execution lasts as foreseen, but it only uses a few cores. xSpark releases unused cores as soon as they are not needed anymore, while Spark cannot, since resources are associated with executors and are only released when they terminate. Note that since Spark uses the same executor to process different stages, one cannot allocate resources specifically for a particular stage, but must always consider possible worst cases.

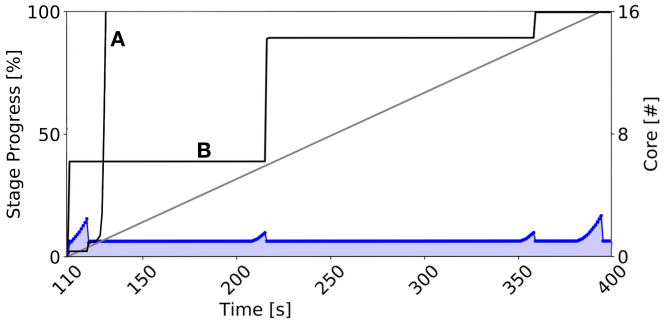


Figure 9: Execution with skewed data.

7.2 Concurrent Applications

To assess the problem of dynamic resource allocation with concurrent applications, we configured xSpark to use its *Memory Manager*, which was disabled with the previous experiments (since single applications should not have memory allocation problems). For these experiments, we used composite benchmarks, that is, sequences of applications separated by time delays. Applications were grouped according to their origins (SparkPerf and SparkBench). We also selected and tested deadlines that would have been feasible if the applications were executed alone, but not necessarily satisfiable when running multiple applications on the same cluster.

The first experiment aimed to assess how xSpark handles the concurrent execution of applications. We used two configurations for xSpark: EDF$_{all}$ with $\gamma = 1$, to maximize the resources allocated to the application with the earliest deadline; and EDF$_{pure}$ with $\gamma = 0$, to ensure that the application with the earliest deadline is given enough resources to complete its execution on time. Each Spark application gets an independent set of executors, which only run tasks and store data for that application. The same happens with xSpark, but the main difference is that Spark, by default, runs all submitted applications in FIFO order, each consuming all available resources, unless the user manually configures the resources allocated to each

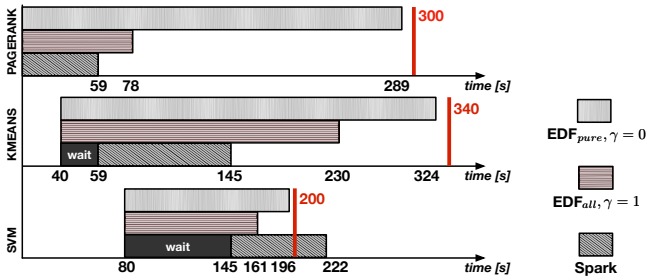


Figure 10: Concurrent execution of Benchmark 1.

application statically at submission time. Instead, xSpark parallelizes the execution of the different applications and allocates the resources to both fulfill deadlines and minimize resource usage.

Table 6 reports the results for the first benchmark, where Δ indicates the amount of time we waited after the beginning of the experiment before submitting an application for execution. Both Δ and $appDeadline$, are defined in seconds, and $appDealine$ is expressed as a duration. Figure 10 shows how the concurrent execution was handled by each system.

Due to the FIFO scheduling of applications in Spark, and the fact that every application allocates all the resources in the cluster, we can see that the deadline requested for SVM cannot be satisfied. Moreover, as shown in Figure 10, the execution of the last application actually begins when its deadline is almost expired, since it needs to wait for the two previous applications to release their resources. Even if Spark was equipped with a non-FIFO application scheduler, one that always selects the application with the closest deadline, the situation would not have changed. When Spark can start KMeans, SVM has yet to be submitted for execution, and thus the only pending application is the one that is started. Instead, xSpark allows us to satisfy all the three proposed deadlines in both configurations.

Configuration	App	Δ	*appDeadline*	ϵ_a
Spark	PageRank	0	300	80.3%
	KMeans	40	300	65.0%
	SVM	80	120	−18.8%
EDF$_{all}$ $\gamma = 1$	PageRank	0	300	74.0%
	KMeans	40	300	36.6%
	SVM	80	120	32.5%
EDF$_{pure}$ $\gamma = 0$	PageRank	0	300	3.6%
	KMeans	40	300	5.3%
	SVM	80	120	3.3%

Table 6: Benchmark 1.

Choosing strategy EDF$_{all}$ with $\gamma = 1$ allows us to satisfy all the deadlines in the benchmark, as we can see in Figure 10. However, due to the nature of this strategy, we have high deadline errors: Table 6 says that *PageRank* completes with a deadline error of 74.0%. In contrast, if we selected EDF$_{pure}$ and $\gamma = 0$ (again, as shown in Figure 10), we increased the execution time of each application with respect to the previous case, but we obtain a smaller deadline error: all applications terminate with a deadline error that is less than 5.3% (see Table 6).

This experiment allowed us to also assess the penalty introduced by our memory management based on off-heap memory. Recall that the management of off-heap memory is slower that the management of on-heap memory, but if one only used on-heap memory, and preallocated all available memory, no new application would be executable because of lack of memory. Our memory management must then be considered as a trade-off between dynamic memory allocation and performance.

Figure 10 says that Spark completes the execution of the three applications in 222 secs, while xSpark $_{all,\gamma=1}$ after 230 secs, with a penalty of less than 4%. Even if this is not a thorough evaluation, given the limitations in managing the heap memory of JVMs, we advocate that xSpark provides a viable solution for memory management at run time when the number of running applications cannot be estimated precisely beforehand.

Benchmark	App	Δ	*appDeadline*	*nomRate*
Bench #2 (SparkBench)	PageRank	0	250	$671K$
	KMeans	40	160	$5142K$
	SVM	80	250	$46K$
Bench #3 (SparkPerf)	aggr-by-key	0	120	$319K$
	group-by	40	200	$486K$
	aggr-by-key-int	80	160	$267K$

Table 7: Benchmarks 2 and 3.

Strategy	γ	#A	#D	$avg(\lvert\epsilon_a\rvert)$	$\epsilon_a A$	$\epsilon_a D$
EDF$_{all}$	0	3	0	12.4%	37.2%	0.0%
EDF$_{all}$	1	3	0	37.2%	111.7%	0.0%
EDF$_{pure}$	0	3	0	6.6%	19.8%	0.0%
EDF$_{pure}$	1	3	0	12.0%	36.1%	0.0%
EDF$_{prop}$	0	3	0	6.7%	20.1%	0.0%
EDF$_{prop}$	1	3	0	17.1%	51.2%	0.0%
Proportional	0	2	1	6.8%	7.5%	-13.0%
Proportional	1	3	0	13.7%	41.2%	0.0%
Speed	0	3	0	5.5%	16.5%	0.0%
Speed	1	3	0	18.1%	54.4%	0.0%

Table 8: Results for Benchmark 2.

We also ran two additional experiments to evaluate the five strategies supported by xSpark with γ equal[9] to both 0 and 1, for a total of 10 options. Table 7 shows the configurations of the two additional benchmarks. Again, the tested deadlines are feasible when the applications execute alone, but cannot be satisfied when running multiple applications together on the same cluster.

Table 8 shows the results for the first benchmark ($Bench\#2$), where $\#A$ indicates the number of applications that completed their execution before the deadline, while $\#D$ is the number of applications that completed with a delay. $avg(|\epsilon_a|)$ is the average of the absolute value of ϵ_a, for the three applications, while $\epsilon_a A$ is the sum of the errors of the applications that finished in advance and $\epsilon_a D$ is the sum of the errors of those that completed with a delay. The table shows that, for most of the configurations, we were able to complete their execution before set deadlines: The value in column #*A* is 3 in 9 out of 10 cases.

The smallest deadline errors —and no violations— in this experiment were achieved with strategies EDF$_{pure}$ ($avg(|\epsilon_a|) = 6.6$) and Speed ($avg(|\epsilon_a|) = 5.5$), both with $\gamma = 0$. Strategy Proportional (with $\gamma = 0$) also shows a small $avg(|\epsilon_a|)$ but one, out of three, applications ended with a delay ($\#D = 1$). This might not be a problem if we want to ensure a fair distribution of resources across the applications, at the price of (slightly) violating some of the deadlines. Furthermore, strategy Proportional with $\gamma = 1$ increased the number of applications that end in advance. Choosing $\gamma = 1$ is not always the best choice; it is simply a way to speed up the computation if future contention is expected. As a result, if we consider EDF$_{all}$, we move from $avg(|\epsilon_a|) = 12.4$ with $\gamma = 0$ to $avg(|\epsilon_a|) = 37.2$ with $\gamma = 1$, which is about three times greater.

Table 9 shows the results for the second benchmark ($Bench\#3$). The workload of the three applications is too high for the allocated resources, resulting in at least one violation with any strategy. Since we were not sure we could eliminate the delays, we decided to examine how Spark would behave if it had an EDF-like task scheduler for its applications. In particular, we tried to give all the resources to a single application, since we wanted to mimic the behavior of EDF$_{all}$ with $\gamma = 1$, but with the advantage of knowing the workload a-priori. We called this approach Clairvoyance EDF: we used the logs produced by Spark to know exactly all the applications to execute, along with the duration of all their tasks. Our analyses with Clairvoyance EDF showed that only two out of three applications can complete their executions by the designated deadlines ($\#A = 2$ and $\#D = 1$). This is the same result obtained by EDF$_{all}$ (without knowing the applications to execute and their duration in advance).

If this workload is run in a situation in which we have a "strict" deadline, we need to minimize the number of violations. As a result, we need to choose the strategy with the smallest #*D* ($\#D = 1$). This ends up being either EDF$_{all}$ or EDF$_{pure}$ with $\gamma = 1$. On the other hand, if the deadline is considered to be "soft", we may want to minimize the value of $\epsilon_a D$ by choosing EDF$_{all}$ with $\gamma = 1$ ($\epsilon_a D = 16.6$). If paying for more resources is as costly as violating the deadline (or even preferred), we may want to minimize the average deadline error $avg(|\epsilon_a|)$. In this composite benchmark the best choice would be to use either EDF$_{pure}$ or EDF$_{prop}$ with $\gamma = 0$.

These experiments show that there is no single strategy that is always better than the others. To generalize, EDF $_{all}$ with $\gamma = 0$ is the best strategy when it comes to avoiding deadline violations, while EDF $_{pure}$ with $\gamma = 0$ is preferable when not violating deadlines is as important as saving resources. Additional, custom strategies could also be conceived and added to xSpark without modifying its local controllers and heuristic-based planners.

7.3 Threats to Validity

The experiments were conducted using eight different applications and the datasets were generated randomly using

9. We decided to focus on the two extreme values, but any value in-between could be applicable.

Strategy	γ	#A	#D	$avg(\lvert\epsilon_a\rvert)$	$\epsilon_a A$	$\epsilon_a D$
EDF_{all}	0	2	1	20.2%	39.5%	21.2%
EDF_{all}	1	2	1	24.4%	56.7%	16.6%
EDF_{pure}	0	1	2	10.8%	5.8%	26.7%
EDF_{pure}	1	2	1	11.5%	12.1%	22.5%
EDF_{prop}	0	1	2	11.0%	5.8%	27.3%
EDF_{prop}	1	1	2	10.8%	6.1%	26.4%
Proportional	0	1	2	12.6%	3.8%	34.0%
Proportional	1	1	2	11.7%	5.0%	30.2%
Speed	0	1	2	13.6%	5.5%	35.5%
Speed	1	1	2	11.5%	5.8%	28.6%

Table 9: Results for Benchmark 3.

well-known benchmarks. Even if xSpark improves Spark with respect to different metrics, we must highlight threats that may hamper the validity of our experiments ([42]):

Internal Threats. Each application was profiled using Spark and then executed using xSpark on the same dataset. If the profiling datasets are (significantly) different in terms of data distribution from the one used to execute the applications, the precision of the heuristic in computing stage deadlines decrease, but the local controllers could still cope with these imprecisions. We conducted some additional experiments (*group-by test20%*) to asses that xSpark is still able to effectively allocate resources dynamically in this scenario. In particular, we obtained an error (ϵ_a) on average equal to -3.30 when controlling a dataset generated with $s = 0.5$, but using the profiling of a uniform dataset. On contrary when executing an application with uniform dataset controlled with a profiling of skewed data ($s = 0.5$) we obtained an error that is on average equal to 0.86%.

More experiments are needed to correlate deadlines and the optimal amount of resources to fulfill them. However, this was not the goal of our work since we allocate resource at runtime with our fast and fine-grained control. Not only does a static (and manual) approach have to find out the optimal allocation, using a comprehensive model or past experience, without dynamic allocation it might be impossible to optimize resources since resource demand is non-constant. This is even truer when dealing with multiple concurrent applications (the scheduling of which is often not in control of the system administrator), that could contend for resources at any time during their execution.

External Threats. Some of our assumptions may limit the generalization of the experiments/solution. Even if Spark was conceived to exploit in-memory processing, it relies on a storage layer (usually HDFS), which may act as a bottleneck. The first assumption is that HDFS must be sized properly: in real world scenarios this is not always the case; to avoid the problem we decided to dedicate the same number of VMs to HDFS as the ones used to run the executors. Moreover, some of the stages of our test applications were conceived to carry out a high number of parallel read/write operations towards the storage layer to stress its performance, and we observed no significant delays.

The second assumption is that memory is considered to be enough. The more data are processed in parallel, that is, the more cores a VM offers, the more data must be loaded in memory. This is a general requirement of any big-data solution and we see it as part of configuring the VMs properly, rather than a possible threat to the generalization of our experiments.

The last assumption refers to the type of Spark applications we considered. xSpark takes into account the core API and the graph and machine learning libraries of Spark. We also conducted some preliminary experiments with another well-known benchmark suite called TPC-H[10] that exploits Spark SQL to implement business oriented queries. The results are similar to those presented and show that xSpark can successfully control this type of application —both individually and in parallel. In contrast, xSpark cannot deal with stream-based applications: their processing model is different and one should think of special-purpose qualities of service, rather than setting a deadline for completing a given execution.

Construct and Conclusion Threats. The experiments demonstrate the validity of our claims, i.e., that the fine-grained and fast resource allocation capabilities provided by xSpark can provision resources precisely and efficiently to multiple applications and have them meet set deadlines.

Obtained results are statistically robust, and only show a small variance (see Table 4). We also used skewed data and obtained similar results.

8 Related Work

The work presented in this paper must be compared with the results obtained in different research areas.

First of all, Ousterhout et al. [38] provide a comprehensive analysis of the performance of various tools for data analytics. As for Spark, they show that CPU allocation is often the bottleneck. They quantify that network optimization reduces execution time by 2%, while if one optimizes disk usage, there is a gain of 19%. They also identify the Java garbage collector and I/O transfers as significant speed limitations for big-data applications.

Spark itself provides limited capabilities to adjust the resources (executors) dynamically allocated to tasks. The *External Shuffle Service* allows Spark to save resources by switching off the executors that remain idle for a user-defined amount of time; they can then be switched-on again if tasks remain idle for too long. This dynamism however is only limited to considering the executors preallocated to an application (it cannot borrow additional executors from the system), and works at executor level, that is, it cannot manage CPU cores.

As for additional resource management solutions, Spark can be paired with external resource managers —such as Mesos [43] and YARN [44]. Mesos sends resource offers (*push-based* scheduler) to its clients and manages both CPU cores and memory, while YARN waits for resource requests (*pull-based* scheduling) and only considers memory. These systems do not support any application-specific policy for resource management. One could think of using our controllers on top of these systems for this. They both support containers to launch executors, but they do not offer any form of vertical

10. http://www.tpc.org/tpch/

scalability. Mesos also offers an optional fine-grained mode, where each task is containerized, but the runtime overhead is heavy: this is why the use of fine-grained resource allocation is deprecated in Spark 2.0.

Lakew et al. [46] and Barna et al. [47] use containers as means to allocate resources dynamically. Similarly to our past work [3], Lakew et al. [46] exploit containers and model-predictive control and system identification to support vertical elasticity [1] and target different KPIs and resource dimensions. Barna et al. [47] target autonomic, containerized multi-tier applications. They exploit layered queuing networks to create self-tuning controllers for applications composed of heterogeneous components such as web services, databases, and big data elements. These solutions could be used to manage the resources allocated to a complete Spark instantiation, but they cannot manage the resources allocated to the different applications since they have no visibility of them. xSpark can do that since besides working on dynamic resource management, we have also changed the architecture and processing model behind Spark to work at a lower granularity level.

Moving to complementing the execution of big-data applications with deadlines, we can only mention a few works. AROMA [15] is a deadline-aware tool for resource inference and allocation of MapReduce applications on the cloud. It uses a two-phase machine learning and optimization framework. Adaptation matches resource utilization with previously executed jobs and makes provisioning decisions accordingly. Cura [49] is another tool based on a new MapReduce cloud model for data analytics in the cloud. It leverages MapReduce profiling to automatically create the best cluster configuration and to optimize global resource consumption. Malekimajd et al. [50] provide upper and lower bounds for MapReduce job execution times in Hadoop based on a linear programming model.

As for Spark, Gibilisco et al. [18] propose a performance model for DAG-based applications that allows one to have an accurate prediction of how the application will behave given a specific data size and certain configuration settings; they do not provide dynamic resource management. Marconi et al. [28] describe a formal model to verify the feasibility of set deadlines given the DAG of the application, the input dataset, and the resources available. Sidhanta et al. [17] introduce OptEx, an optimization model to configure a Spark cluster according to time and cost constraints. Their approach is static, they support VM-only clusters and do not consider data skew.

Islam et al. [16] propose another static resource allocation system called dSpark. It solves an optimization problem to compute different possible resource allocation schemas, with different costs and resources required; the user then selects the one s/he prefers. dSpark cannot manage multiple applications, runtime contentions, and data skew, and the allocation is limited to VMs. Even if these works have some limitations, they could be complementary to our solution. Currently, we use our fast heuristic to preallocate resources, but more sophisticated solutions could be adopted.

The same complementarity applies to the works that monitor the execution of big data applications [39], [51]–[54]. For example, Morton et al. [39] propose ParaTimer, a progress indicator for MapReduce DAGs. They estimate the progress of complex MapReduce applications that are translated into a DAG of jobs. They use a critical path algorithm to find the longest sequence of tasks to be processed. They also handle data skew and failures by providing a set of estimations that consider different scenarios.

Different approaches exist for scheduling multiple big-data applications (mostly MapReduce applications) on a shared cluster. Kc et al. [55] present an approach that schedules Hadoop jobs to meet QoS deadlines. They model the execution cost of each task, taking into account both processing and data transfer times, and then estimate the number of Map and Reduce jobs required to satisfy the deadline. Polo et al. [56] also provide a solution for the performance-driven co-scheduling of the tasks of diverse MapReduce applications. They introduce a new task scheduler that can dynamically estimate the cost of parallel task executions, and dynamically reallocates resources without distinguishing between Map and Reduce jobs. [57] is another scheduling technique that dynamically builds performance models of the workloads and uses them to inform the adaptive scheduler when numerous applications are competing for shared resources. While these solutions address the problem of scheduling applications given the executors (and thus the resources reserved to them), xSpark considers resources first and then computes a feasible scheduling given the policy adopted for allocating resources at runtime.

9 Conclusions and Future Work

The paper proposes a solution for enriching Spark with fine-grained dynamic resource allocation, and presents xSpark as a supporting prototype infrastructure. The proposed solution considers CPU cores and memory and allows for their optimized allocation when the framework is used to execute both single and multiple applications. The assessment we conducted witnesses both a better utilization of resources and a reduced number of violated deadlines.

As for future work, we would like to extend our approach to consider disk and network usage. We will also keep studying how to combine containers and control theory to manage an infrastructure that hosts heterogeneous applications (e.g., web services and big-data frameworks).

Acknowledgments

This work was supported with grant by project *EEB - Edifici A Zero Consumo Energetico In Distretti Urbani Intelligenti (Italian Technology Cluster For Smart Communities) - CTN01_00034_594053* and by the GAUSS national research project (MIUR, PRIN 2015, Contract 2015KWREMX).

References

[1] S. Dustdar, Y. Guo, B. Satzger, and H.-L. Truong, "Principles of Elastic Processes," *IEEE Internet Computing*, vol. 15, 2011.

[2] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem, "Adaptive Control of Virtualized Resources in Utility Computing Environments," in *Proc. of the 2nd ACM European Conference on Computer Systems*. ACM, 2007.

[3] L. Baresi, S. Guinea, A. Leva, and G. Quattrocchi, "A Discrete-time Feedback Controller for Containerized Cloud Applications," in *Proc. of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016.

[4] N. Roy *et al.*, "Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting," in *Proc. of the 4th IEEE International Conference on Cloud Computing*. IEEE, 2011.

[5] D. Ardagna, B. Panicucci, and M. Passacantando, "A Game Theoretic Formulation of The Service Provisioning Problem in Cloud Systems," in *Proc. of the 20th international conference on World wide web*. ACM, 2011.

[6] C. Klein *et al.*, "Brownout: Building More Robust Cloud Applications," in *Proc. of the 36th International Conference on Software Engineering*. ACM, 2014.

[7] D. A. Menascé, "QoS Issues in Web Services," *IEEE Internet Computing*, vol. 6, 2002.

[8] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing As the 5th Utility," *Future Generation of Computing Systems*, vol. 25, 2009.

[9] "Apache Hadoop," http://hadoop.apache.org, 2017.

[10] "IBM InfoSphere," https://www-01.ibm.com/software/data/infosphere/, 2017.

[11] M. Isard *et al.*, "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks," in *Proc. of the 2nd ACM European Conference on Computer Systems*, 2007.

[12] A. Verma, L. Cherkasova, V. S. Kumar, and R. H. Campbell, "Deadline-based Workload Management for MapReduce Environments: Pieces of the Performance Puzzle," in *Proc. of the 22th IEEE Network Operations and Management Symposium*. IEEE, 2012.

[13] "Apache Spark," http://spark.apache.org, 2017.

[14] A. Verma, L. Cherkasova, and R. H. Campbell, "ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments," in *Proc. of the 8th ACM International Conference on Autonomic Computing*. ACM, 2011.

[15] P. Lama *et al.*, "AROMA: Automated Resource Allocation and Configuration of MapReduce Environment in the Cloud," in *Proc. of the 9th International Conf. on Autonomic Computing*. ACM, 2012.

[16] M. Islam *et al.*, "dSpark: Deadline-based Resource Allocation for Big Data Applications in Apache Spark," in *Proc. of the 13th IEEE International Conference on eScience*, 2017.

[17] S. Sidhanta, W. Golab, and S. Mukhopadhyay, "OptEx: A Deadline-Aware Cost Optimization Model for Spark," in *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2016.

[18] G. P. Gibilisco *et al.*, "Stage Aware Performance Modeling of DAG Based in Memory Analytic Platforms," in *Proc. of IEEE 9th International Conference on Cloud Computing*. IEEE, 2016.

[19] "Docker," http://docker.com, 2017.

[20] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux Journal*, 2014.

[21] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An Updated Performance Comparison of Virtual Machines and Linux Containers," in *Proc. of the 16th IEEE International Symposium on Performance Analysis of Systems and Software*, 2015.

[22] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors," in *Proc. of the 2nd ACM European Conference on Computer Systems*, vol. 41. ACM, 2007.

[23] "HDFS Users Guide," https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html, 2017.

[24] "Linux Manual: cgroups," http://man7.org/linux/man-pages/man7/cgroups.7.html, 2016.

[25] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," *Mobile Networks and Applications*, vol. 19, 2014.

[26] N. Kozlowski, "Spark Memory Management Part 1: Push It to the Limits," https://www.pgs-soft.com/spark-memory-management-part-1-push-it-to-the-limits, 2017.

[27] E. Gianniti, A. M. Rizzi, E. Barbierato, M. Gribaudo, and D. Ardagna, "Fluid Petri Nets for the Performance Evaluation of MapReduce and Spark Applications," *SIGMETRICS Performance Evaluation Review*, vol. 44, 2017.

[28] F. Marconi, G. Quattrocchi, L. Baresi, M. Bersani, and M. Rossi, "On the Timed Analysis of Big-Data Applications," in *Proc. of the 10th NASA Formal Methods Symposium*. Springer, 2018.

[29] K. Narendra *et al.*, "An Iterative Method for the Identification of Nonlinear Systems Using a Hammerstein Model," *IEEE Transactions on Automatic Control*, vol. 11, 1966.

[30] A. Nordsjo and L. Zetterberg, "Identification of Certain Time-varying Nonlinear Wiener and Hammerstein Systems," *IEEE Transactions on Signal Processing*, vol. 49, 2001.

[31] J. Voros, "Identification of Hammerstein Systems with Time-varying Piecewise-linear Characteristics," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 52, 2005.

[32] D. Guarin and R. Kearney, "An Instrumental Variable Approach for the Identification of Time-varying, Hammerstein Systems," *IFAC-PapersOnLine*, vol. 48, 2015.

[33] K. S. Hong *et al.*, "On-line Scheduling of Real-time Tasks," in *Proc. of the 9th Real-Time Systems Symposium*. IEEE, 1988.

[34] M. L. Dertouzos *et al.*, "Multiprocessor Online Scheduling of Hard-real-time Tasks," *IEEE Tran. on Software Engineering*, vol. 15, 1989.

[35] N. W. Fisher, *The Multiprocessor Real-time Scheduling of General Task Systems*. The University of North Carolina at Chapel Hill, 2007.

[36] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo, *Deadline Scheduling for Real-time Systems: EDF and Related Algorithms*. Springer Science & Business Media, 2012, vol. 460.

[37] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "A Study of Skew in MapReduce Applications," *Open Cirrus Summit*, vol. 11, 2011.

[38] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making Sense of Performance in Data Analytics Frameworks," in *Proc. of the 12th USENIX Conference on Networked Systems Design and Implementation*. USENIX, 2015.

[39] K. Morton, M. Balazinska, and D. Grossman, "ParaTimer: A Progress Indicator for MapReduce DAGs," in *Proc. of the 36th ACM International Conference on Management of Data*. ACM, 2010.

[40] J. Lin, "The Curse of Zipf and Limits to Parallelization: A Look at the Stragglers Problem in MapReduce," in *Proc. of the 7th Workshop on Large-Scale Distributed Systems for Information Retrieval*, 2009.

[41] K. G. Shin and P. Ramanathan, "Real-time Computing: A New Discipline of Computer Science and Engineering," *Proceedings of the IEEE*, vol. 82, 1994.

[42] C. Wohlin *et al.*, "Empirical research methods in web and software engineering," *Web Engineering*, 2006.

[43] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center," in *Proc. of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI. USENIX, 2011.

[44] V. K. Vavilapalli *et al.*, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proc. of the 4th Annual Symposium on Cloud Computing*. ACM, 2013.

[45] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types," in *Proc. of the 8th USENIX Conference on Networked Systems Design and Implementation*, 2011.

[46] E. B. Lakew *et al.*, "Kpi-agnostic Control for Fine-grained Vertical Elasticity," in *Proc. of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2017.

[47] C. Barna, H. Khazaei, M. Fokaefs, and M. Litoiu, "Delivering Elastic Containerized Cloud Applications to Enable DevOps," in *Proc. of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 2017.

[48] L. Baresi, S. Guinea, G. Quattrocchi, and D. A. Tamburri, "MicroCloud: A Container-Based Solution for Efficient Resource Management in the Cloud," *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, 2016.

[49] B. Palanisamy *et al.*, "Cura: A Cost-Optimized Model for MapReduce in a Cloud," in *Proc. of the IEEE 27th International Symposium on Parallel and Distributed Processing*, 2013.

[50] M. Malekimajd *et al.*, "Optimal MapReduce Job Capacity Allocation in Cloud Systems," *SIGMETRICS Perform. Eval. Rev.*, vol. 42, 2015.

[51] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data," in *Proc. of the 8th ACM European Conference on Computer Systems*. ACM, 2013.

[52] S. Chaudhuri *et al.*, "When Can We Trust Progress Estimators for SQL Queries?" in *Proc. of the 31th ACM International Conference on Management of Data*. ACM, 2005.

[53] S. Chaudhuri, V. Narasayya, and R. Ramamurthy, "Estimating Progress of Execution for SQL Queries," in *Proc. of the 30th ACM International Conference on Management of Data*. ACM, 2004.

[54] K. Lee *et al.*, "Operator and Query Progress Estimation in Microsoft SQL Server Live Query Statistics," in *Proc. of the 42th ACM International Conference on Management of Data*. ACM, 2016.

[55] K. Kc and K. Anyanwu, "Scheduling Hadoop Jobs to Meet Deadlines," in *Proc. of the IEEE 2nd International Conference on Cloud Computing Technology and Science*. IEEE, 2010.

[56] J. Polo *et al.*, "Performance-driven Task Co-scheduling for MapReduce Environments," in *Proc. of the 20th IEEE Network Operations and Management Symposium*. IEEE, 2010.
[57] J. Polo, Y. Becerra, D. Carrera, M. Steinder, I. Whalley, J. Torres, and E. Ayguade, "Deadline-Based MapReduce Workload Management," *IEEE Transactions on Network and Service Management*, vol. 10, 2013.