Mira: Sharing Resources for Distributed Analytics at Small Timescales

Michael Kaufmann[†‡], Kornilios Kourtis[†], Adrian Schuepbach[†], Martina Zitterbart[‡]
[†]*IBM Research Zurich,* [‡]*Karlsruhe Institute of Technology (KIT)*
[†]Zurich, Switzerland [‡]Karlsruhe, Germany
{kau,kou,dri}@zurich.ibm.com, zitterbart@kit.edu

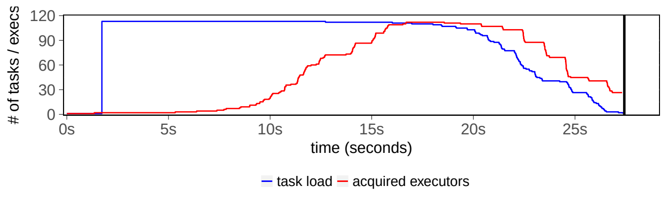
***Abstract*— Modern distributed analytics stacks consist of application frameworks that enable processing of large amounts of data, and a resource manager that allows applications to share computational resources. The initial use case for these systems was running batch jobs with long lifetimes (e.g., a few hours), but, since their inception, new use cases have emerged where users increasingly use them to gain insight interactively, or even online. Efficiently sharing resources under these additional use cases, requires operating at smaller timescales (minutes or even seconds) than the existing systems were designed for and are capable of.**

In this paper, we present Mira, a system for optimized elastic execution of short-running and interactive data-analytics applications with low-latency execution startup, fast resource management and efficient resource utilization on shared clusters. We analyze the resource sharing overheads in a commonly used distributed processing stack (Spark+YARN) and reveal opportunities to accelerate applications in shared environments. Our experiments show, that Mira is able to reduce resource sharing related overheads by more than 400$\times$ and reduce application runtime by up to 4.2$\times$.

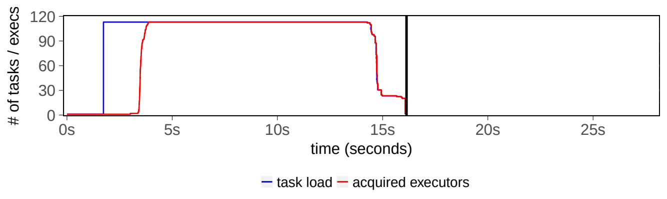
I. Introduction

Today's distributed analytics are written in data-parallel frameworks (e.g., MapReduce [7], Spark [41], Flink [6], etc. [24], [28], [35], [36]), and executed in shared infrastructure using resource managers such as YARN [39] or Mesos [17]. The existing ecosystem was initially developed to serve batch applications that run for hours, typically executed overnight. The wide adoption of the above systems resulted in additional uses: performing interactive exploration of data, as well as online processing [4], [6], [24], [35]. As a result, resource managers are now required to manage resources at smaller timescales than they were originally designed for: minutes or even seconds. This happens for two reasons. First, some applications take a few minutes or less, instead of hours, to complete. Second, applications such as online analytics, exhibit workload fluctuations that quickly change application resource requirements.

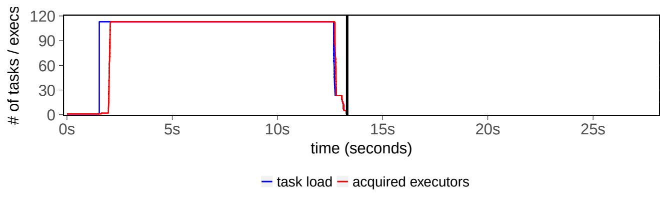
Smaller time granularities pose a significant challenge for existing systems (both resource managers and application frameworks) that were not designed to manage resources within seconds. To illustrate this, we consider a simple Spark



(a) Spark+YARN



(b) Spark+Mira



(c) Spark+Mira (warm: executors are reused)

Fig. 1: Execution of a Spark application that spawns 110 tasks, each of which sleep for 10 seconds.

application running on a YARN-managed cluster. The application initially sleeps and spawns 110 tasks, each of which sleeps for 10 seconds. Fig. 1a shows the demands of the application (blue line) and the resources (task executors) that were allocated to it (red line). Not only there is significant delay in the application acquiring the resources it needs, but there is also delay in releasing the resources after it is done. While long running applications with little change in load can amortize these costs, short running and interactive applications, with highly fluctuating resource demands, cannot. In this case, those costs dominate – or even exceed – the application runtime.

To address the above issues, we present Mira, an efficient execution environment that enables resource management over small timescales (sub-seconds to seconds) for modern distributed applications. Mira includes two parts: a resource

Source code, detailed test descriptions and experimental results will be made available at https://github.com/zrlkau/mira.

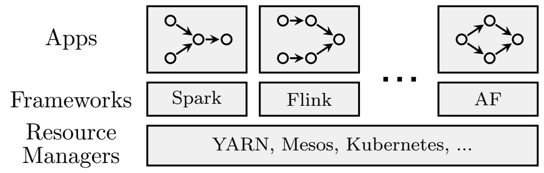


Fig. 2: Applications execute on top of application frameworks (AFs) that operate on a common computing infrastructure managed by a resource manager (RM)

manager (RM) and an application scheduler (AS). The former manages all resources and applications, while the later schedules the tasks of a single application and communicates with the RM. We modified Spark to use Mira's AS, instead of its own scheduler (we believe our work is applicable in other application frameworks, but we focus on Spark in this paper because it is one of the most widely used frameworks).

Mira achieves efficiency in two ways. First, it treats executors not as ephemeral, but as long-lived, shared resources. This allows it to minimize recurring acquisition costs and benefit from warmed-up executors (e.g., JIT, caches), which further improves efficiency as shown in Fig. 1c. Second, as consequence of the minimized resource acquisition cost, Mira is able to use a resource acquisition and release strategy that allows it to respond to workload changes almost instantaneously, thus improving application runtime as well as resource sharing efficiency. Overall, Mira overcomes prohibitive costs of existing systems to quickly react to changes in resource demands.

We summarize our contributions below:

1) Using Spark applications running on top of YARN as an illustrative example, we show and quantify the inability of modern analytics stacks to operate at small timescales.
2) We present Mira, a two-level scheduling system for efficient resource sharing that cuts down the resource acquisition cost, and accelerates application execution via reusing task executors. We integrate our approach into Spark.
3) We evaluate Mira using TPC-DS queries running on top of Spark, where Mira reduces average resource acquisition time by a factor of 5.5× in a multi-application scenario, unassigned and idle resources from 11.1% to 2.9%, and, finally, accelerates 75% of the queries by 2× or more.

The remainder of this paper is structured as follows. First, (II), using Spark+YARN as a representative analytics stack, we analyze and quantify their overheads when managing resources at small timescales. Next, we describe how Mira addresses these issues (III), and perform an experimental evaluation (IV). Finally, we discuss related work (V) and conclude (VI).

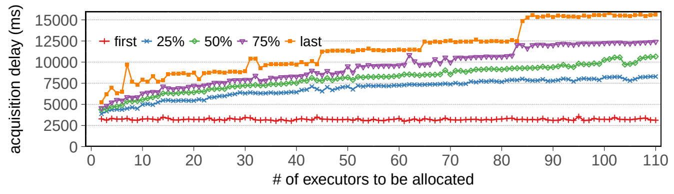
II. Background and Motivation

Most distributed applications are written in application frameworks (AFs) such as Apache Spark [41], Flink [6], and others [1], [29], [36]. Users write their program using high level operations (e.g., map/reduce), while the AF handles distributed execution: data distribution, task scheduling, faults, etc. Commonly, as shown in Fig. 2, these applications are executed on a shared computing infrastructure that is managed by a resource manager (RM) such as YARN [39], Mesos [17], or others [5], [11], [19], [30], [40]. Generally, there are three approaches in how AFs and RMs collaborate, depending on who controls resources, and for how long.

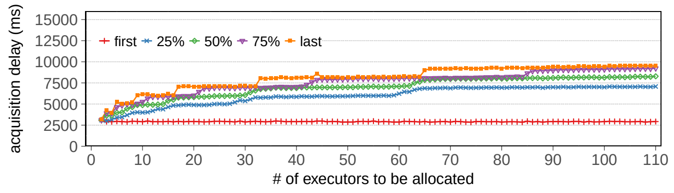
1) In *fine-grained resource allocation*, used in Hadoop MapReduce [16], earlier versions of Spark [41] and others [17], control over resources stays with the RM at all times. For each application task, the RM allocates suitable resources, executes a task on behalf of the AF and releases the resources afterwards. The RM has full insight into the utilization of each resource and can share them among multiple applications. However, this comes at the cost of high task execution overhead, due to resource allocation and release per task, which can be prohibitive for short-running tasks.
2) In *static coarse-grained resource allocation*, used by Spark [41], Flink [6] and others [4], [35], a set of resources is statically allocated to an application and the AF assumes control over them for the entire duration of the application execution. An AF scheduler schedules individual (micro-)tasks onto the allocated resources. This eliminates the high task execution overhead, as resource allocation costs can be amortized across several tasks, thus enabling low-overhead, low-latency task execution. However, as the RM relinquishes control over these resources, it has no insight into their utilization and cannot share them with other applications, even if the owning application does not use them at all times. Consequentially, coarse-grained resource allocation is ill-suited for shared environments.
3) *Dynamic coarse-grained resource allocation* mode allows elastic expansion and contraction of the resources allocated to AF. Many systems support or plan to support this (e.g., Spark [32], Flink [10], and others [29], [34]). As load increases, the AF scheduler may request further resources from the RM and release them once the load has decreased. As in coarse-grained mode, resource allocation costs can be amortized across multiple tasks, thus enabling low overhead execution of tasks *on allocated resources*. While the RM still has no insight into resource utilization, it will regain control earlier, given a cooperative AF or use of preemption techniques, therefore supporting resource sharing.

Architecturally, dynamic coarse-grained resource allocation provides a good trade-off between execution overheads and efficient resource sharing. Nevertheless, because existing systems are not built for small timescales, they are unable to effectively use this mode of scheduling for applications with short durations or fluctuation in resource demands. Specifically, we identify two main issues. First, the fundamental resource allocation operations take a long time, placing an upper bound on resource sharing efficiency. Second, because resource acquisition takes too long, AFs hold on to resources long after load has decreased, further hurting efficiency. Both issues have significant impact on interactive applications with low response time requirements as well as applications with highly fluctuating resource demands as both need to acquire and release resources frequently and quickly. Problems get exacerbated in high-load scenarios with resource shortage. In the next paragraphs, we examine and quantify these issues on common setup, using Spark as the AF and YARN as the RM.

A. Resource Allocation Costs



(a) Spark+YARN



(b) Spark standalone

Fig. 3: Cost allocation of N=2 . . . 110 executors: Delay from task load increase to first/last executor availability, as well as delay for different percentiles on our 14-node test cluster (see IV-A0a).

We use a micro-benchmark to analyze the behavior of Spark using a 14-node cluster (platform and configuration details in IV-A0a) managed by YARN. In each execution, the benchmark spawns a number (N) of tasks to allocate the same number of executors. We execute the benchmark for N=2 up to N=110 executors, repeating the execution for each N five times. Results in Fig. 3a show the average task waiting time for each N, i.e., the time for which the task is ready for execution but no executor is assigned to it for different percentiles of tasks. The results indicate that there is a base cost (*first*) of ≈3.2s, of which ≈1.9s can be attributed to the Spark executor startup, which we break down in Fig. 4, while the remainder is due to overheads in YARN's container startup. In addition to the base cost, there is a variable cost of up to ≈12.6s, resulting in a executor acquisition cost of up to ≈15.8s in total.

To show that these issues are not YARN-specific, we repeat the same benchmark with Spark's standalone mode [33], a lightweight, Spark-specific resource manager included in Spark, using an equivalent configuration. To our knowledge, this is the fastest way of starting Spark applications. The results in Fig. 3b support our assumption. While Spark standalone is ≈23% faster than Spark+YARN, it is also inadequate to operate at small timescales.

To understand the variable cost, we examine the resource acquisition strategy more closely. Fig. 5 shows an excerpt of our benchmark for N=110 executors, and illustrates how

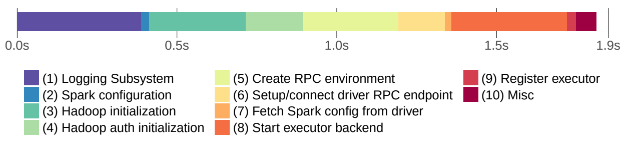


Fig. 4: Breakdown of startup costs of a Spark executor.

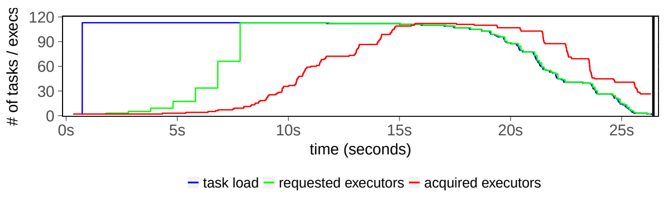


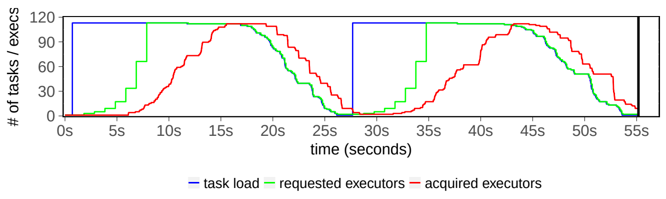
Fig. 5: Resource allocation in Spark+YARN.

Spark uses a slow-start strategy to acquire resources (green line). While this approach might be suitable for long-running tasks where such acquisition costs are negligible, for small timescales, it significantly limits the ability to adapt to workload spikes.

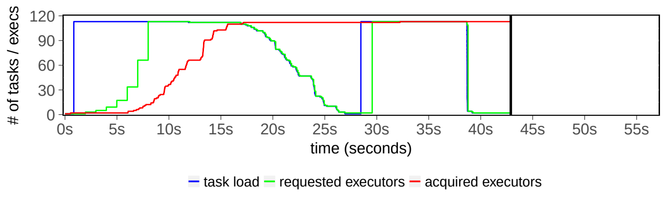
The other source of the variable cost is YARN's response time which increases with the number of concurrent resource requests. This effect is shown in Fig. 5 by the widening gap between request and acquisition curves as the request curve becomes steeper. A potential cause of this behavior is YARN's internal communication (between RM and node manager (NM) as well as application master (AM)), much of which is piggy-backed on periodic heartbeats [39]. While this improves scaling, it may impact the latency of resource acquisition negatively under load, as it is more likely in this situation to miss the response window for the next heartbeat.

B. Resource Reclamation

As a consequence of the high resource allocation costs shown above, AFs can either: i) release resources quickly after use, at the cost of potentially slowing down application execution due to the need to re-acquire them later, or ii) retain resources for a long time after use, at the cost of poor resource utilization. Indeed, Spark can be configured for either approach and by default releases executors after 60s, which – for short applications – corresponds to case ii. We evaluate these cases on Spark+YARN by executing our benchmark for N=110 two consecutive times for each of these two options, summarizing the results in Fig. 6a. As shown in Fig. 6a, where Spark releases executors 1s (minimum possible setting) after they become idle, the number of executors (red) expands and contracts with the number of tasks, resulting in a effective executor utilization of 85% and a runtime of ≈28s for the 2nd wave of tasks. Contrarily, as shown in Fig. 6b, retaining resources leads to a much faster execution of the 2nd wave of tasks, and an effective executor utilization of 63% and a runtime of ≈15s, for the 2nd wave of tasks, or ≈1.9x less than case i. Both of the options are problematic, especially under small timescales.



(a) Release idle resources after 1s



(b) Retain idle resources

Fig. 6: Consecutively allocating N=110 executors on Spark+YARN with and without resource release. The black bar represents the application end.

C. Summary

In the previous paragraphs, we examined in depth the overheads of acquiring and releasing resources when using a common analytics stack (Spark+YARN), and we conclude that these overheads are prohibitive for managing resources at small time granularities. Enabling efficient resource utilization at small timescales requires minimizing these overheads, both, in the AF and the RM.

III. Mira

In this section, we illustrate how Mira addresses the inabilities of systems like Spark and YARN to efficiently share resources at small timescales. Mira is a resource manager (RM) and a per-application task scheduler (AS), which (combined) enable efficient, low-latency resource sharing as well as fast application execution on shared clusters. Mira is built on two main concepts:

1) *Reusing and sharing task executors* (e.g., Spark executors or Docker containers). This minimizes resource acquisition overheads, but also has the potential to accelerate application code execution, e.g., by reducing JIT overheads or benefiting from warm (software or hardware) caches.
2) *Tight integration of RM and AS*. Mira improves the interface (e.g., for resource reclamation) between the RM and AFs to enable better resource utilization and enable high elasticity of applications on small timescales.

Next, we elaborate on the concepts and implementation of Mira as well as on how it addresses the limitations of existing systems as discussed in II.

A. Overview

Fig. 7 shows a high-level overview of Mira's architecture along with the main communication paths. Mira consists of two main components, the resource manager (RM) (III-C) and an application scheduler (AS) (III-D). Multiple AS instances

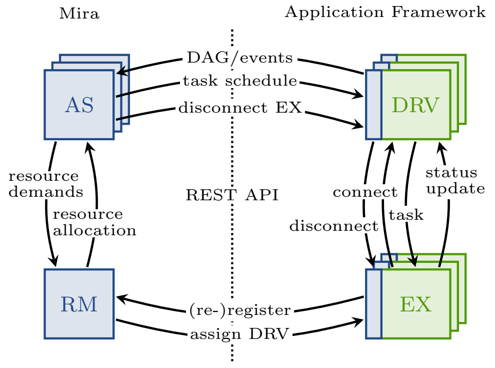


Fig. 7: High-level overview of the Mira architecture.

can coexist, each scheduling a single application. Moreover, Mira must be integrated in two external components. First, to the application framework (AF), where Mira performs directed acyclic graph (DAG) scheduling on behalf of the application. Second, Mira is required to manage the execution environment (EX) to enable reuse and sharing across applications. Component interactions can be divided into external and internal.

a) Mira external API: Communication between AS and AF as well as RM and EX is done via a (language-agnostic) REST API. To integrate Mira into an AF, the AF has to implement the required API. For Spark, we have done this by implementing a connector module inside Spark's application driver (DRV) and executor, that interfaces between Mira and Spark. The driver (DRV) uses the REST API to transmit a (partial) application DAG to the AS as well as any relevant status events, e.g. *stage ready* or *task finished*, but also metrics, such as heap state of executors. In return, the driver receives individual task schedules, i.e. mappings from runnable tasks to executors, from the AS.

The EX registers itself with the RM and receives commands as to which application it has been assigned to, and hence should connect to. Disconnect commands are sent by the AS via the DRV under two circumstances: the RM reduces the resource allocation of the corresponding application, or the AS decides that an EX is no longer needed. Afterwards, the EX re-registers itself with the RM so that it can be reassigned to another application.

b) Mira Internal API: Internally, Mira's components communicate via an intra-process event bus, operated by a central event engine. Calls to the REST API are translated into such events and relayed to the RM or an AS instance. Apart from this, the majority of events concern changes in resource demands from the ASs to the RM as well as changes in resource allocations from the RM to the ASs. All changes are propagates immediately in order to support high application elasticity at small timescales. Within the AF, i.e. between DRV and EX, communication is based on the AF's native, albeit extended protocol.

In the remainder of this section, we describe each component in more detail (III-B to III-D) and close with discussing some implementation aspects (III-E).

B. Execution Environment (EX)

The EX in Mira does not replace the AF's native task execution environment but wraps and extends it such that it enables the executor process to be persistent, avoiding high (as shown in II-A) restart costs. For JVM-based executors, persistence also allows the conservation of the JIT cache so the code can run immediately at native, instead of interpreted, mode. We note that even different applications tend to share large portions of the code (e.g., Spark execution run-time and libraries). Aside from application code execution, this also accelerates the EX control path, reducing the time from assignment to acquisition, as can be seen by comparing the delay between task load increase and executor availability in Fig. 1b and Fig. 1c.

The EX itself is simple and supports two commands. First, it can *connect* to a DRV via instruction from the RM, in which case a new native execution environment is instantiated. Second, it can *disconnect* from a DRV, if ordered by RM, via the corresponding AS, in which case the native execution environment is destroyed, and control is returned to the RM.

C. Resource Manager (RM)

The resource manager (RM) determines the resource share of each application, based on a policy, and assigns and revokes them from and to the ASs. Enabled by our previously described optimizations, we designed Mira's RM under the premise that resource acquisition is cheap. Hence, Mira immediately reassigns resources if demands change, and does so frequently, if required, allowing it to be more efficient at small timescales. This is in contrast to other RMs, e.g. YARN, which uses multi-second timeouts before revoking resources, or Mesos, which uses an offer-based approach where an application cannot actively request resources but has to wait for an offer from Mesos, which might not come soon enough when sub-second latency is required.

Currently, Mira uses a weighted fair-share scheduling policy, assigning each running application A_n the number of compute resources R_{A_n} (in form of executor instances) according to the following equation:

$$R_{A_n} = R_{tot} \times \frac{W_{A_n}}{\sum_{m=0}^{|Apps|} min(\hat{W}_{A_m}, W_{A_m})}$$

R_{A_n} are the resources assigned to application A_n, while R_{tot} are the total resources. W_{A_n} and $\hat{W}_{A_m}$ are the fair-share weight and the weight corresponding to the actually used resources by A_m, respectively. If resources remain unallocated, e.g., because some applications do not fully utilize their fair share ($W_{A_m} > \hat{W}_{A_m}$), other applications get resources beyond their weighted fair share. If some applications exceed their fair-share ($W_{A_m} < \hat{W}_{A_m}$), while other applications, that use less than their fair-share, demand more resources, the RM will request the corresponding ASs to immediately release the required amount of resources to satisfy the outstanding demands. It is up to the AS to ensure timely compliance with these requests. Another option would be to use task preemption, but as we show in our evaluation (IV), our approach does not negatively impact performance. The pseudo-code for Mira's RM is shown in Algorithm 1.

As the frequency of resource change demands increases with the number of running applications, the RM has to be able to keep decision latency low. Some of the methods we use to achieve this are:

1) Resource change events are only generated by the AS if they imply a change in resource assignments, e.g. an application releases idle executors.
2) Resource change events are only processed by the RM if they imply a change in resource assignments, e.g. an application requests more executors while unassigned executors remain or its fair share is not yet exhausted.
3) The RM aggregates multiple events when reevaluating resource assignments in order to cope with high load.

In our multi-app benchmarks (IV-D), the median delay of resource assignment reevaluation less than $100\,\mu s$.

Algorithm 1 Handle events and update resource assignments

```
 1: procedure HANDLEEVENTS(event)
 2:     reschedule ← false
 3:     while event new events in queue or timeout do
 4:         if event impacts resource assignment then
 5:             reschedule ← true
 6:         end if
 7:     end while
 8:     if reschedule = true then
 9:         apps ← currently running applications
10:         UPDATERESOURCEASSIGNMENTS(apps)
11:     end if
12: end procedure
13: procedure UPDATERESOURCEASSIGNMENTS(apps)
14:     determine relative share for each app
15:     map relative into absolute share for each app
16:     for all app ← apps that need to release executors do
17:         notify AS to release executor(s)
18:     end for
19:     while unallocated executors exist do
20:         if all apps are satisfied then break
21:         end if
22:         for all app ← apps entitled to more executors do
23:             add one executor to app
24:         end for
25:     end while
26: end procedure
```

D. Application Scheduler (AS)

Mira supports plugable application schedulers with a strict separation of policy and mechanism. The policy, i.e., decision making process is a component in Mira, whereas the mechanism, i.e., the execution of these decisions, is integrated into the AF. In contrast to many other schedulers [11], [26], [38] but similarly to some [14], [15], [29] our AS is DAG-based. That is, it exploits information from an annotated DAG, consisting of *stages* (nodes) and *data dependencies* (edges), to optimize task ordering and placement decisions.

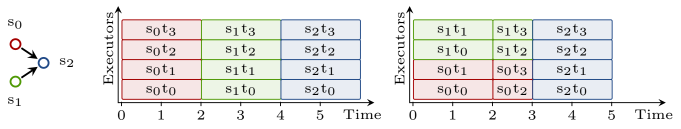


Fig. 8: Conceptual depiction of the schedule of a DAG (left) with 3 stages, 4 tasks each, using back-to-back stage scheduling (middle) and concurrent scheduling (right) as implemented in Mira, where improved JIT and data cache exploitation can accelerate task execution.

While this information offers many opportunities to improve scheduling (e.g., addressing heterogeneity [20]), in the context of this paper we exploit it as follows: Mira schedules independent stages concurrently and concentrates tasks of each stage on a dedicated subset of executors. This improves locality since all tasks of a stage execute the same function and often share data. By maximizing the number of same-stage tasks per executor (without leaving any executor idle), we also maximize the effectiveness of the JIT as well as data caches, which can reduce task runtime, as depicted in Fig. 8, by amortizing costs across a larger number of tasks. As some of those costs do not scale with task runtime, this approach can benefit short tasks in particular, which is a focus of Mira.

E. Implementation

Mira is implemented in approximately 8k lines of C++ code, and will be made available at https://github.com/zrlkau/mira.

IV. Evaluation

In our evaluation we compare Vanilla Spark[1] on YARN with our modified Spark+Mira. We attempt to answer three main questions:

1) *Can we reduce the recurring resource acquisition?* In IV-B and IV-C, we evaluate the impacts of reduced system overheads for executor acquisition and application runtime. As we will show, Mira reduces the acquisition time by up to 434× in our micro-benchmarks. At the same time, the overhead minimization resulted in a runtime reduction of ≈1.5× for 50% of evaluated TPC-DS queries (IV-C0a).
2) *What impact does code execution acceleration by reusing warm executors have?* In IV-C0b we show that reusing warm executors can speedup code execution significantly and reduce application runtime by a factor of 2× or more for 75% of evaluated TPC-DS queries.
3) *Finally, we answer the question of what performance impact Mira can have in a multi-application setting.* In IV-D, we combine all effects of the previous experiments in a scenario with constant background load. In addition we evaluate the impact of immediate executor release on resource usage efficiency. As we will show, Mira is able to achieve an overall performance improvement of up to 4.2×.

[1]We added some annotations to the code to measure certain performance aspects of the scheduler. The same set of annotations are also present in our modified version of Spark.

A. Test Setup

a) Cluster: Our test cluster consists of 14 compute nodes with one additional control node, each equipped with two Intel Xeon E5-2640v3 or E5-2650v2, 160-256 GB RAM, CentOS/Fedora 26 Linux[2]. All nodes are connected with 56 GB IPoIB (Mellanox ConnectX-3) via a single Mellanox SX6036 switch.

The control node runs Mira, YARN RM as well as nameserver for HDFS and does not execute any compute tasks. Input data is read from HDFS, which is backed by 16 GB ramdisks on each of the 14 remaining nodes. The HDFS balancer was used to evenly distribute data across the cluster. We configured Mira and YARN to execute at most 8 concurrent tasks per node, resulting in 112 possible executors. We chose to limit the number of executors to 8 per node since we observed significant slowdown of the execution beyond that point for both, Mira and YARN. We use Spark's client mode for all tests with clients (application drivers) executed on the control node.

b) Software: We use Apache YARN 2.7.3, Apache HDFS 2.8.2, Apache Spark 2.2.2, Oracle HotSpot JVM 1.8.0_144. In order to perform detailed performance evaluations we added and extended annotations within Spark at certain points in its native DAG and task scheduler (e.g. executor request, release and registration, task start and finish) as well as its executor (e.g. startup and shutdown, task start and finish). This slightly modified version was used as basis to integrate Mira into Spark. We omit benchmarks using Spark's standalone resource manager as it only supports FIFO application scheduling [33] but lacks resource reclamation enforcement, e.g. via preemption, which our multi-application benchmarks require.

c) Test applications: We use a simple test program that generates constant background load in form of a large set of short tasks, each running for ≈1 second. The background (BG) application is capable of utilizing all available resources and runs for the entire duration of each test. As foreground (FG) applications, we use a set of 90 TPC-DS [37] queries[3] which read their input data (scaling factor = 100) from HDFS.

d) Settings: For all benchmarks, except where noted otherwise, we use the settings shown in Table I.

B. Micro-Benchmarks

We run the micro-benchmark from II-A, which spawns N=2. . . 110 tasks at a time, to acquire N=2. . . 110 executors in order to evaluate whether Mira can reduce resource acquisition time. Our results represent the average of 10× test runs. Fig. 10 shows the results for Mira.

[2]Including Spectre and Meltdown fixes
[3]Some of our modifications to Spark's lineage graph are incomplete and do support all RDD types yet, which is why we were not able to execute all 100 TPC-DS queries.

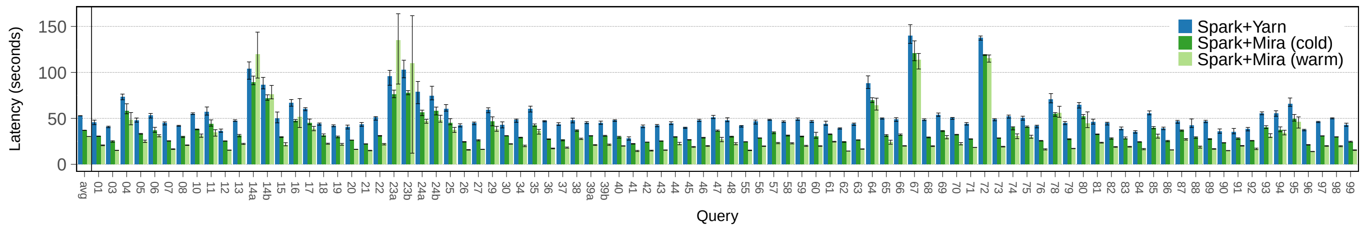


Fig. 9: Comparison of TPC-DS query execution latency between Spark+YARN and Spark+Mira on cold and warm executors in a single application setting using the default 60s executor release (we omit the results for sharing optimized settings due to space constraints).

TABLE I: Relevant non-default settings used throughout our evaluations, unless noted otherwise

Spark setting	Value (default)
spark.dynamicAllocation.enabled	true (false)
spark.shuffle.service.enabled	true (false)
spark.dynamicAllocation.minExecutors	1 (1)
spark.dynamicAllocation.executorIdleTimeout	1s (60s)
YARN setting	**Value (default)**
yarn.nodemanager.resource.cpu-vcores	8 (unlimited)
yarn.resourcemanager.scheduler.class	CapacityScheduler (FairScheduler)
yarn...preemption.total_preemption_per_round	1.0 (0.1)
yarn...preemption.max_wait_before_kill	1s (15s)
yarn...preemption.monitoring_interval	1s (3s)
Java setting	**Value**
-XX:MaxHeapSize (driver)	12g
-XX:MaxHeapSize (executor)	8g
-XX:+UseG1GC	

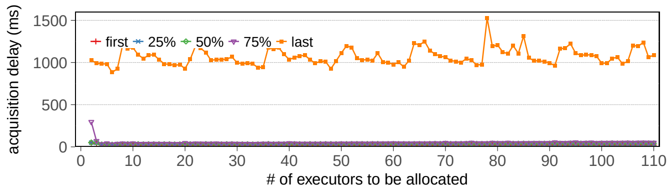


Fig. 10: Allocation cost of N=2 . . . 110 executors of Spark+Mira: Delay from task load increase to first/last executor availability, as well as delay for different percentiles on our test cluster (see IV-A0a). Note that the last executors to register are always cold executors, while all others are always warm. Compare with Fig. 3.

On average, Mira is able to reduce executor acquisition time for the 75th percentile to ≈36ms or a factor of 112× – 434× (4s – 8.6s) faster than Spark+YARN and 85× – 191× compared to Spark standalone, respectively, depending on the total number of executors requested. This represents the share of *warm* executors in each test run. However, in each run we add one previously unused, and therefore *cold*, executor, which register always last, after ≈1.1s. Considering only the last executor to register, Mira is able to reduce delays by 4.9× – 14.6× (5.3s – 15.6s) compared to Spark+YARN and 3× – 9× (3.2s – 9.6s) for Spark standalone, respectively. Furthermore, while the delay in Spark+YARN and Spark standalone increases with the number of executors requested at a time (Fig. 3), it stays constant for Spark+Mira in all tested cases (Fig. 10). This shows that the long-lived EXs (II-A), as well as the exploitation of JIT caches for the control path (III-B), offer significant resource sharing benefits.

C. Single Application Benchmarks

In this section we present the evaluation results of Mira for single applications in order to show the impact of the reduced acquisition overhead as well as code execution acceleration separately. We perform three types of benchmarks.

1) As baseline, we execute each TPC-DS query 5× on Spark+YARN with dynamic resource allocation enabled.
2) In order to evaluate the benefits of the reduced system overhead, we execute each TPC-DS query 5× and compare it to the baseline. In-between each execution we restart Mira to avoid any warm-up effects.
3) In order to evaluate the benefits of code execution acceleration, we execute all TPC-DS queries back to back 5× and compare it to the previous benchmark. This represents the optimal case, because it maximizes the effectiveness of code jitting.

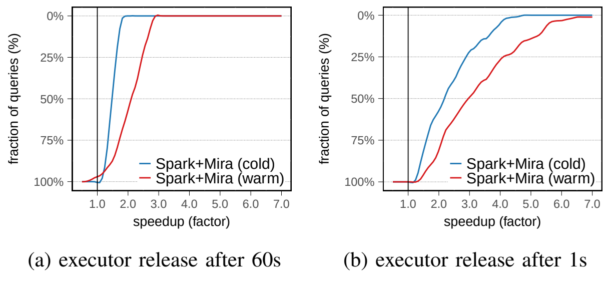


Fig. 11: Comparison of Spark+YARN with default (a) and sharing optimized (b) idle executor release vs. Mira on cold and warm executors.

Fig. 11 summarizes our results. Comparing Spark+YARN with Spark+Mira using the default 60s idle executor release setting for Spark and *cold* executors for Mira, we see a runtime reduction of 1.5× or more for 50% of all queries (Fig. 11a) due to reduced executor acquisition overhead (III-B) as well as our acquisition strategy (III-D). However, the default setting of Spark w.r.t. executor release does not enable efficient resource sharing on small timescales. In order to evaluate Spark's behavior in such a scenario, we reduced the idle executor release timeout for executors without any cached data to 1s (Fig. 11b). Here, Spark+Mira achieves a runtime reduction of ≈2.1× for 50% of all queries. The difference can be explained by the additional cost that Spark+YARN has to pay for executor reacquisition (see II-B). In both settings, we suspect that runtime improvements are, in part, also due to the same-stage task concentration on executors (see III-D) that further optimizes JIT and data cache effectiveness.

On *warm* executors, Mira achieves an acceleration by a factor of 2.1 or more for 50% or all queries, using default executor release settings and ≈3 for the sharing optimized settings, respectively, due to a more effective JIT exploitation. The complete results for all 3 benchmarks are shown in Fig. 9.

In the following we are going to break down the results and analyze the impact of Mira's executor acquisition strategy as well as the reuse of warm executors separately.

a) Reduced System Overheads: Mira reduces the response time to workload changes as well as the executor acquisition time. Fig. 12 exemplifies both aspects. As one can see in Fig. 12a, the maximum parallelism and the number of executors requested lag by several seconds in the case of Spark+YARN, due to Spark's slow-start resource acquisition strategy. In total, it can take up to 15s to acquire all executors, which is in line with the delay we measured in II-A, and poses a significant overhead for short-running applications and workload spikes. Fig. 12b shows the same query executed on Spark+Mira. Here, one can see that Mira requests executors immediately (see III-C), which leads to a much faster ramp-up. Cold executors are acquired with a delay of ≈1s after the load increase, which also corresponds to our micro-benchmark results in IV-B.

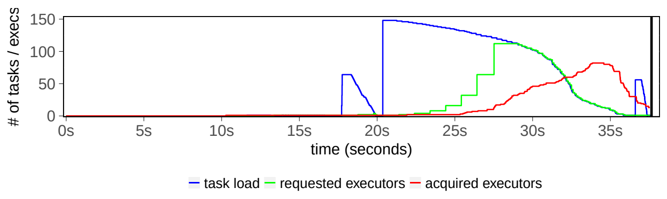
Fig. 12c as well as in the 2nd wave of tasks in Fig. 12b show that, once executors are warm, the acquisition time is, in line with our micro-benchmarks in IV-B, reduced even further.

b) Code Execution Acceleration: Mira reuses executors within and across applications in order to exploit benefits stemming from JVM warm-up, i.e. code jitting, and the resulting execution of native code instead interpreted byte code. Therefore, applications will be able to run on warm executors most of the time. Warm executors can accelerate two aspects. First, the framework functions can be optimized more efficiently, which is the reason for faster executor acquisition, as shown Fig. 10. Secondly, application code itself, especially parts that heavily rely on function calls to often used libraries, can benefits from more efficient execution. Fig. 12 shows both aspects. Comparing Fig. 12b and Fig. 12c, one can see the much narrower spikes in maximum application parallelism, which indicates that application code is executed faster and tasks finish in a shorter period of time.

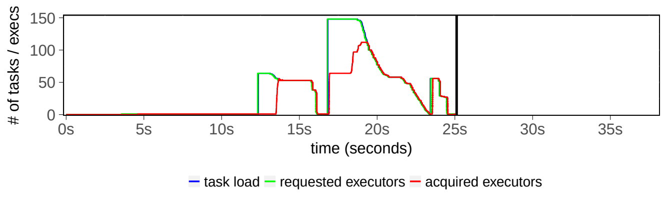
Lastly, while YARN takes about ≈10s from application submission to admission and the start of the very first task, Mira only requires ≈4s.

D. Multi-Application Experiments

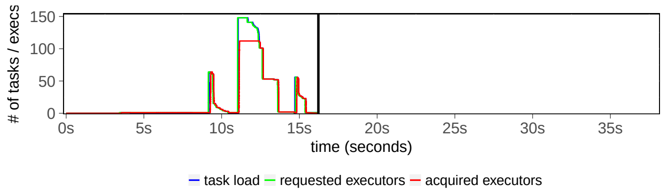
In this section we present the evaluation results for Spark+YARN and Spark+Mira with multiple concurrently running applications in order to show benefits of the aspects



(a) Spark+YARN



(b) Spark+Mira (cold)



(c) Spark+Mira (warm)

Fig. 12: Plot of TPC-DS query 7 (complete application execution) on 112 executors. In (b) and (c), the blue and green line coincide. The vertical black bar indicates the end of the application.

evaluated in IV-C0a combined with the tight coupling of RM and AS of Mira in a shared environment.

For this benchmark, we execute two applications concurrently. A background (BG) application, consisting of an infinite loop of stages, each with 8192 1s-tasks. The purpose of the BG application is to generate a constant task load on the cluster in order to force the RM to actively balance resource requests from multiple applications and to show the effects of loose vs. tight coupling of RM and AS in Spark+YARN (II-B) and Spark+Mira (III-C), respectively. As before we use TPC-DS queries as foreground (FG) applications.

We configured FG and BG queues in YARN's capacity scheduler, both with a guaranteed minimum share of 50% of all resources and a maximum of 99%[4]. For Mira we use comparable settings.

As YARN's default settings can lead to significant delays in resource reassignment, and therefore inflated runtimes, we modified the default settings according to Table I to allow YARN to react more quickly to changing resource demands. While those were the best settings for YARN we found, improving our baseline by a factor of ≈1.14, we did not do a full parameter space exploration. We repeat each test 5×.

The complete results are shown in Fig. 13 and a summary CDF is shown in Fig. 14. Spark+Mira is able to achieve a runtime reduction of ≈2.4× for 50% of all queries. At the

[4]A maximum queue share of 100% would effectively block another application from being started due to lack of resources. As workaround, our settings ensure that no single application can occupy all resources.

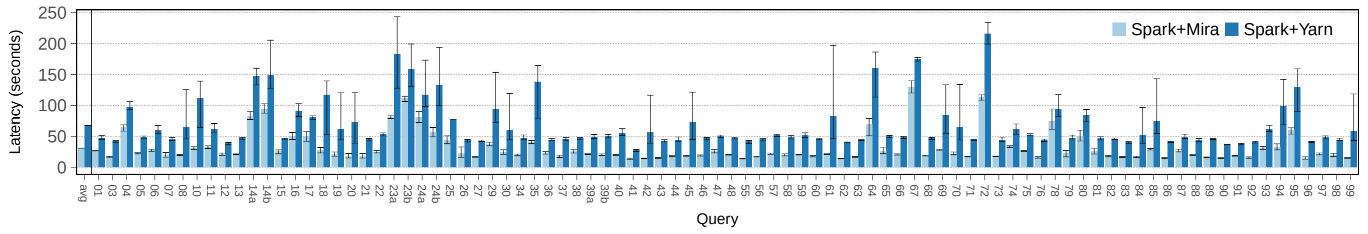


Fig. 13: Comparison of query total execution latency between Spark+YARN and Spark+Mira with constant BG load. The first column is the average across all runs.

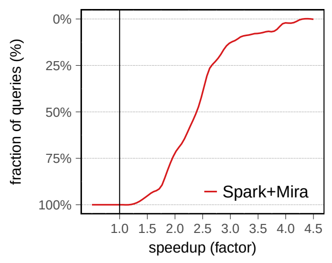


Fig. 14: Average speedup of Spark+Mira compared to Spark+YARN with constant BG load.

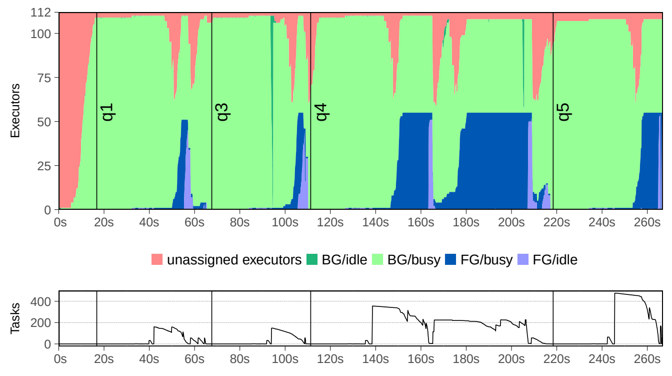


Fig. 15: Excerpt of Spark+YARN multi-app benchmark run showing the resource sharing between BG and FG applications (top) and the corresponding task load (maximal parallelism) of the FG application (bottom). FG application submissions are indicated by the labelled vertical black lines. Compare with figure 16.

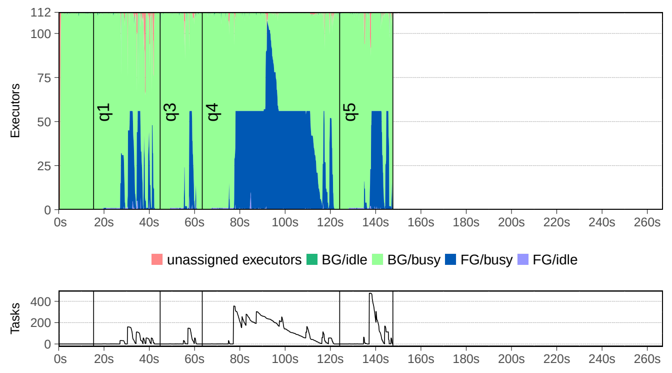


Fig. 16: Excerpt of Spark+Mira multi-app benchmark run showing the resource sharing between BG and FG applications (top) and the corresponding task load (maximal parallelism) of the FG application (bottom). FG application submissions are indicated by the labelled vertical black lines. Compare with figure 15.

same time, Spark+Mira achieved a BG task throughput of 91 tasks per second vs. 87 for Spark+YARN[5]. In the following we are going to analyze the results more closely.

Fig. 15 and Fig. 16 show the allocation of resources to the BG and FG applications in a benchmark run on Spark+YARN and Spark+Mira, along with the task load at any point in time. Areas in green (blue) represent executors allocated to the BG (FG) application. Red areas represent executors that the RM has not assigned to any application or are in-between assignments.

A comparison of both figures highlights multiple points: First, Spark+Mira is able to execute the same number of queries in 148s instead of 267s. This is due to the lower latency executor acquisition by the FG application (compare delay from query start to executor increase) as well as the overall shorter task runtime (compare size of blue areas). Moreover, there are fewer unassigned executors during the transition period as shown by the virtually non-existent red spikes in Fig. 16. We expect that with more applications and more executor reassignments, the number of unassigned executors will increase further for Spark+YARN. Finally, the high cost of resource reacquisition for Spark+YARN (see also Fig. 6) can be seen during the execution of query 4 where Spark releases executors due to a short dip in task load (at ≈165s) just to reacquire them a few seconds later with a multi-second delay.

The efficiency metrics in table II show the differences more concretely. Spark+YARN need 914 ES[6] on average to assign all entitled resources to the FG application after their task load increased whereas Spark+Mira only requires 131 ES or 6.97× less. In 43.5% of cases – which correspond to short *task load* (see task load graph in 15 for example) – the reaction time of Spark+YARN is longer than the load persists, whereas this only happens in 0.9% of cases for Spark+Mira. Finally, Spark+Mira achieves a higher resource utilization rate, both in terms of number of resources unassigned as well as in numbers of assigned, but idle resources.

[5]Note that the background tasks had a constant runtime of 1s, thus are not able to benefit from warm-up effects.

[6]ES = executor seconds (like CPU seconds), i.e. a number of executors being in a state for a number of seconds, e.g. 2 executors being busy for 5 seconds results in an busy value of 10 ES.

TABLE II: Efficiency metrics (average over 5 TPC-DS runs). The unit ES refers to *executor seconds* (like CPU seconds), i.e. a number of executors $\times$ a number of seconds.

	Spark+YARN	Spark+Mira
Resource assignment delay (FG)	914 ES	131 ES
Missed resource increases (FG)	43.5%	0.9%
Total unassigned and idle resources	11.1%	2.9%

E. Conclusion

Our evaluation shows that Spark+Mira can – due to it's reduced resource acquisition overhead as well as code execution acceleration – reduce runtime of workloads significantly while utilizing resources more efficiently. While we found this to be true for all tested TPC-DS queries in a shared environment, shorter queries benefit more than longer queries as not all overheads scale with the overall execution time.

V. Related Work

A. Resource Managers and Schedulers

YARN [39] and Mesos [17] are two of the most commonly used resource schedulers, both operating on a two-level structure (AFs/RM), but they are incapable to deal with small timescales. We illustrate YARN's issues in II. Mesos does not allow applications to request resources but makes periodic offers which applications can accept or reject. This makes low-latency resource acquisition difficult as the application has no control over the timing and frequency of resource offers.

There are various other approaches to cluster resource management. Kubernetes [5] is a container orchestration system based upon Google's internal Borg [40] and Omega [30] focusing on reliable, scalable and elastic deployment of service oriented workloads. Resources managers have been used to solve various optimization problems, such as workload-specific co-scheduling [8], [9], resource packing [13] and near-term capacity planning [38] and overall optimization of scheduling decisions at scale [11], [19], [26]. These works are largely orthogonal to Mira, which focuses on minimizing recurring resource acquisition costs and maximizing sharing efficiency.

B. Application Frameworks and Schedulers

Tez [29] is a meta-framework that focuses on providing common functionality (e.g. a DAG scheduler), similarly to Mira, for application frameworks, such as Hive [36] and Pig [28]. In contrast to Mira, Tez requires YARN as resource manager.

Storm [35], Flink [6] and Apex [4] are predominantly stream processing frameworks, though the latter two can also be used for batch processing. We believe that they can benefit from the low latency resource scheduling Mira provides in order to quickly adapt to changes in load.

C. Execution Environment Optimizations

Lion et. al [22] have done an extensive study of the impact and sources of JVM cold start costs across several distributed frameworks and applications, such as Spark, Hive and HDFS, and conclude that the warm-up time is a common bottleneck for short-running jobs. They present a transparent client/server JVM replacement that enables a persistent, warmed-up JVM process to be reused repeatedly and offers a large speedup for certain workloads. Mira applies the same optimizations, but it achieves it by reusing executors. Similar optimizations exist for other environments than JVM. Oakes et. al [25] have studied overheads of Python and containers in serverless frameworks and unveil significant costs for cold-starting Docker containers and the Python runtime and they present approaches at how to minimize both, e.g. by forking new Python runtimes from existing ones. A similar approach is used in Sand [2].

D. Serverless Execution

The serverless execution model, offered by many cloud providers [3], [12], [18], [23], provides high scalability and automatic elasticity for applications that process independent events. These properties are also desirable for data analytics workloads [21], resulting in serverless offerings for Spark [27], [31]. We believe Mira is an attractive option for building such offerings, due to its low latency resource management.

VI. Conclusion and Future Work

We presented Mira, a system that enables efficient resource sharing for analytics applications that operate on small timescales and achieves significant performance and efficiency improvements over existing techniques, accelerating analytics workloads by up to 4.2$\times$ in a shared environment.

As a continuation of our work, we plan to integrate Mira into other applications frameworks such as Hive or Flink, as well as explore other use-cases, such as graph processing and elastic machine learning. Mira uses an one-size-fits-all executor model which might not always be the best option (e.g., incompatible libraries or security issues). We plan to investigate, using multiple executor pools with different access policies and configurations, in future work. Mira's source and further information will be made available at https://github.com/zrlkau/mira.

References

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *OSDI* (2016), vol. 16, pp. 265–283.

[2] Akkus, I. E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P., and Hilt, V. Sand: Towards high-performance serverless computing. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)* (2018).

[3] Amazon. https://aws.amazon.com/lambda/, accessed 2018/06/22.

[4] Apex, A. https://apex.apache.org, accessed 2018/08/18.

[5] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. Borg, omega, and kubernetes. *Queue 14*, 1 (2016), 10.

[6] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 36*, 4 (2015).

[7] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 10–10.

[8] DELIMITROU, C., AND KOZYRAKIS, C. Paragon: Qos-aware scheduling for heterogeneous datacenters. *SIGPLAN Not. 48*, 4 (Mar. 2013), 77–88.

[9] DELIMITROU, C., AND KOZYRAKIS, C. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2014), ASPLOS '14, ACM, pp. 127–144.

[10] FLINK. https://issues.apache.org/jira/browse/FLINK-4319, accessed 2018/08/06.

[11] GOG, I., SCHWARZKOPF, M., GLEAVE, A., WATSON, R. N. M., AND HAND, S. Firmament: Fast, centralized cluster scheduling at scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, Nov. 2016), USENIX Association, pp. 99–115.

[12] GOOGLE. https://cloud.google.com/functions/, accessed 2018/06/22.

[13] GRANDL, R., ANANTHANARAYANAN, G., KANDULA, S., RAO, S., AND AKELLA, A. Multi-resource packing for cluster schedulers. In *ACM SIGCOMM Computer Communication Review* (2014), vol. 44, ACM, pp. 455–466.

[14] GRANDL, R., CHOWDHURY, M., AKELLA, A., AND ANANTHANARAYANAN, G. Altruistic scheduling in multi-resource clusters. In *Proceedings of OSDI16: 12th USENIX Symposium on Operating Systems Design and Implementation* (2016), p. 65.

[15] GRANDL, R., KANDULA, S., RAO, S., AKELLA, A., AND KULKARNI, J. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, Nov. 2016), USENIX Association, pp. 81–97.

[16] HADOOP, A. https://hadoop.apache.org/, accessed 2018/06/23.

[17] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R. H., SHENKER, S., AND STOICA, I. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI* (2011), vol. 11, pp. 22–22.

[18] IBM. https://www.ibm.com/cloud/functions, accessed 2018/06/22.

[19] ISARD, M., PRABHAKARAN, V., CURREY, J., WIEDER, U., TALWAR, K., AND GOLDBERG, A. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 261–276.

[20] KAUFMANN, M., AND KOURTIS, K. The HCl Scheduler: Going all-in on Heterogeneity. In *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17)* (Santa Clara, CA, 2017), USENIX Association.

[21] KLIMOVIC, A., WANG, Y., KOZYRAKIS, C., STUEDI, P., PFEFFERLE, J., AND TRIVEDI, A. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, USENIX Association.

[22] LION, D., CHIU, A., SUN, H., ZHUANG, X., GRCEVSKI, N., AND YUAN, D. Don't get caught in the cold, warm-up your jvm: Understand and eliminate jvm warm-up overhead in data-parallel systems. In *OSDI* (2016), pp. 383–400.

[23] MICROSOFT. https://azure.microsoft.com/en-us/services/functions/, 2018.

[24] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 439–455.

[25] OAKES, E., YANG, L., ZHOU, D., HOUCK, K., HARTER, T., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, 2018), USENIX Association.

[26] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 69–84.

[27] OWEN, G., LIANG, E., CHOCKALINGAM, P., AND SHANKAR, S. Databricks Serverless: Next Generation Resource Management for Apache Spark, June 2017.

[28] PIG. https://pig.apache.org, accessed 2018/08/16.

[29] SAHA, B., SHAH, H., SETH, S., VIJAYARAGHAVAN, G., MURTHY, A., AND CURINO, C. Apache TEZ: A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD international conference on Management of Data* (2015), ACM, pp. 1357–1369.

[30] SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 351–364.

[31] SOWRIRAJAN, V., BHUSHAN, B., AND AHUJA, M. Qubole announces apache spark on aws lambda, Nov. 2017.

[32] SPARK. https://spark.apache.org/docs/latest/job-scheduling.html, accessed 2018/08/06.

[33] SPARK. https://spark.apache.org/docs/latest/spark-standalone.html, accessed 2018/08/18.

[34] STORM. https://issues.apache.org/jira/browse/STORM-2284, accessed 2018/08/06.

[35] STORM. http://storm.apache.org, accessed 2018/08/06.

[36] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow. 2*, 2 (Aug. 2009), 1626–1629.

[37] TRIVEDI, A. https://github.com/zrlio/sql-benchmarks, accessed 2018/08/16.

[38] TUMANOV, A., ZHU, T., PARK, J. W., KOZUCH, M. A., HARCHOL-BALTER, M., AND GANGER, G. R. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2016), EuroSys '16, ACM, pp. 35:1–35:16.

[39] VAVILAPALLI, V. K., MURTHY, A. C., DOUGLAS, C., AGARWAL, S., KONAR, M., EVANS, R., GRAVES, T., LOWE, J., SHAH, H., SETH, S., SAHA, B., CURINO, C., O'MALLEY, O., RADIA, S., REED, B., AND BALDESCHWIELER, E. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (New York, NY, USA, 2013), SOCC '13, ACM, pp. 5:1–5:16.

[40] VERMA, A., PEDROSA, L., KORUPOLU, M., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys '15, ACM, pp. 18:1–18:17.

[41] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* (Berkeley, CA, USA, 2010), HotCloud'10, USENIX Association, pp. 10–10.