GRASS: Generic Reactive Application-Specific Scheduling

Marcel Blöcher
TU Darmstadt
Germany

Matthias Eichholz
TU Darmstadt
Germany

Pascal Weisenburger
TU Darmstadt
Germany

Patrick Eugster
USI
Switzerland

Mira Mezini
TU Darmstadt
Germany

Guido Salvaneschi
TU Darmstadt
Germany

Abstract

High resource utilization is important to operate compute infrastructures and data centers efficiently. High utilization is achieved by multiplexing several applications over the same physical infrastructure. Yet, with this approach, the different requirements of each application have to be taken into account when scheduling resources.

We propose GRASS, a reactive domain-specific abstraction that allows specifying application-tailored resource scheduling policies. We demonstrate how the declarative approach of GRASS enables extension and composition of scheduling policies. Our evaluation shows the performance benefits of considering application-specific information in a composition of scheduling policies that adapt at run time.

CCS Concepts • **Networks** → *Network resources allocation*; • **Software and its engineering** → *Scheduling*; *Domain specific languages.*

Keywords Data Center Resource Scheduling, Reactive Programming

ACM Reference Format:
Marcel Blöcher, Matthias Eichholz, Pascal Weisenburger, Patrick Eugster, Mira Mezini, and Guido Salvaneschi. 2019. GRASS: Generic Reactive Application-Specific Scheduling. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS '19), October 21, 2019, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3358503.3361274

1 Introduction

Highly virtualized compute infrastructures and data centers promise high resource utilization and hence low operational costs by multiplexing various applications over the shared physical infrastructure. As multiple users can independently run their applications on a common infrastructure, the users' demands for each application are likely to be very heterogeneous. The availability of such flexible compute infrastructures is essential to ensure that services can take advantage of the scalability offered by virtualized data centers still meeting each service quality demands [20].

At the core of a flexible compute infrastructure is the resource management framework (RMF) of a data center, which is responsible for assigning the available resources (i.e., servers, network communication facilities) to users that execute applications by submitting jobs to the cluster. To achieve such assignment, the RMF runs a scheduling policy which selects the best option according to a scheduling goal, e.g., to minimize queuing time of submitted jobs or to maximize successfully served jobs. Typically, such an RMF infrastructure has an event-driven nature. The RMF need to react to different kinds of events, including resource requests sent by the application and status events provided by the underlying infrastructure, for example, when resources become un-/available.

For effective resource scheduling, the RMF should be able to consider all information available, especially application-specific information such as how containers of a requesting application are going to communicate [3]. Furthermore, an RMF with a fixed scheduling policy is likely to be not optimal in every situation [6], which calls for an RMF that supports dynamically configured scheduling policies. The recent trend in programmable networking hardware, which allows running application-specific components on networking devices (e.g., an in-network coordination service [21]), further increases the need for an RMF with flexible scheduling policies [3]. Unfortunately, developing and testing new scheduling policies from scratch is non-trivial even for simple scheduling policies, mainly due to the challenging problem a scheduling policy has to solve (Section 2.1), but also because implementing an RMF with a new scheduling policy requires non-negligible implementation effort.

In this paper, we propose Generic Reactive Application-Specific Scheduling (GRASS), which provides high-level abstractions that foster composability and reusability of scheduling policies, reducing the hurdles of implementing new schedulers by allowing to extend and compose existing ones. Scheduling policies are written in reactive style, which allows schedulers to react to changing application requirements and network conditions. GRASS introduces (I) novel domain-specific abstractions for expressing custom scheduling policies. (II) Scheduling policies can make use of inheritance for easily extending existing policies, or make use of (III) run time compositions to dynamically select parts of exiting scheduling policies. To address the variety of RMF architectures, we show how to (IV) use GRASS to implement RMF architectures of various kind, like queue or graph based schedulers, or schedulers with a distributed architecture. In summary, this paper makes the following contributions:

- We present GRASS, a language-based approach to specify application-specific resource scheduling policies based on reactive programming.
- We show how different scheduling policies can be implemented based on a common reactive scheduling interface.
- We outline how our design enables extension and composition of scheduling policies.
- We evaluate our approach in an event-based simulator, demonstrating how the application-aware combination of different scheduling policies improves performance.

The rest of the paper is organized as follows. Section 2 provides an overview over resource scheduling approaches and introduces reactive programming. Section 3 describes the design of our approach for specifying scheduling policies. Section 4 discusses different scheduler architectures and how our abstractions support them. Section 5 presents the evaluation. Section 6 discusses related work. Section 7 concludes.

2 Background

In this section, we introduce resource scheduling in cluster architectures and we present the fundamental concepts in the reactive programming paradigm.

2.1 Resource Scheduling

A data center resource management framework (RMF) receives jobs (resource requests) submitted by the user and data center resources (e.g., CPU, storage, memory) as an input and finds a valid allocation of resources to jobs. In case there are more jobs awaiting than resources available, some jobs remain in a queued status and are served later. *Preemption* and *migration* are two common operations performed by schedulers that allow one to free up resources without waiting for jobs to complete. Migration allows the scheduler to move running jobs to different resources without affecting the execution. Preemption, on the other hand, terminates the execution of a job completely: A preempted job needs to be rescheduled at a later point in time.

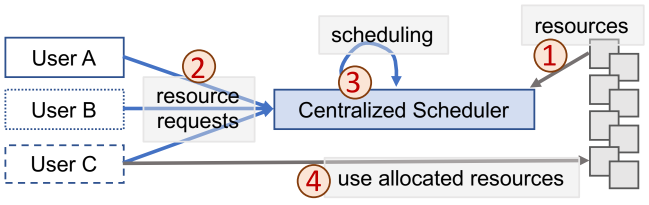
The RMF performs scheduling with objectives like reaching high data center resource utilization, achieving the lowest queuing latency, or maximizing the number of fulfilled resource requests. The data center resource scheduling problem is usually challenging due to the size of modern data centers, which can easily comprise a number of servers in the order of multiple thousands within a large multi-path network fabric. In addition, the RMF has to deal with a mix of jobs with different requirements like priority and expected run time. In order to achieve the target objectives for resource scheduling, an RMF should be aware of such aspects, as well as of the heterogeneity of the resources available in the cluster [20].

2.1.1 Scheduling Architecture

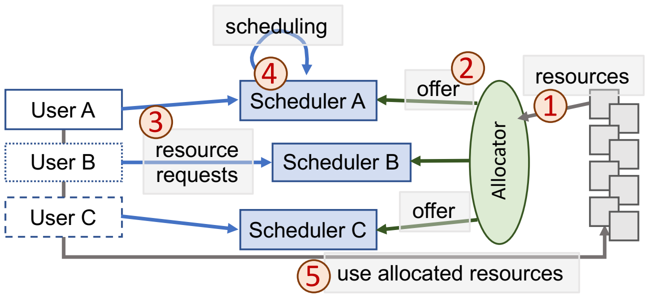
Motivated by the complexity of the scheduling problem, RMFs implement different scheduling architectures to tackle the scheduling problem, mainly centralized, fully distributed, and offer-based using a two-level architecture, or a hybrid of these [10, 17, 31, 33, 34]. Figure 1 shows the high-level interaction of users with the RMF and the resources.

Centralized A centralized RMF consists of a single scheduler instance, which holds all the responsibilities of resource allocation. Before executing the scheduling algorithm, the scheduler collects information about the resources and the current requests for the cluster. When a user receives a resource token from the scheduler, it can use the token to allocate resources for its application. A centralized architecture has the advantage of a consistent state across the data center. Yet, scalability might be an issue even though recent work shows centralized schedulers with good scalability, e.g., Firmament [16].

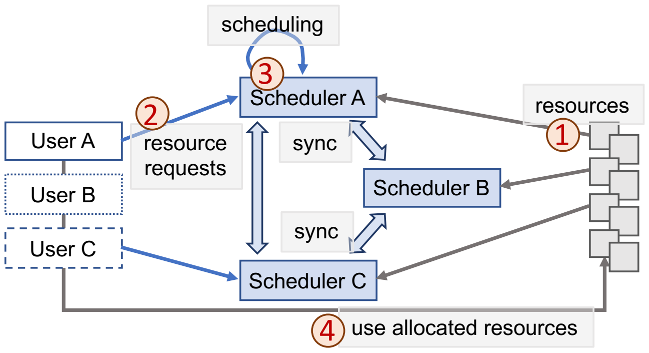
Two-Level Offer-Based A two-level RMF (e.g., [17]) separates the scheduling logic into two phases. *(1-2)* a (central) allocator bundles a set of free resources and offers these resources to one of the many schedulers. *(3-4)* when a scheduler receives requests from users, it uses its resource offers to run the scheduling logic. As a result, the scheduler either returns the offered resources to the allocator (if those cannot be used), or the scheduler hands over resource tokens to the user so that the resources can be used. During the time a scheduler owns a resource offer, the allocator cannot hand over the same resource offer to any other scheduler, i.e., the two-level architecture uses a pessimistic concurrency model.



(a) Centralized



(b) Two-Level Offer-Based



(c) Fully Distributed

Figure 1. Generic Scheduling Architectures

A two-level scheduler has the advantage to support to run different scheduler logic at the same time by using dedicated scheduler instances. A drawback is the strict separation of the running scheduler instances due to the separated resource offers, i.e., the allocator needs to be aware of all schedulers and of their internal state (e.g., how many jobs are queued), in order to make useful resource offers to the scheduler instances.

Fully Distributed A fully distributed scheduler architecture (e.g., [31]) tries to combine the advantage of running different scheduler logic at the same time by using an optimistic concurrency model. All schedulers use their own shadow copy of the data center resources' state. Each time when any of the schedulers finds a valid allocation, the schedulers use a transaction-like protocol to synchronize their state and to agree on a subset of valid allocations. This optimistic concurrency model allows running the scheduling logic simultaneously on the same set of resources, but with a risk of allocation conflicts.

Hybrid Architectures Many schedulers do not strictly belong to any of the previously mentioned architectures, but use a modified version or a combination of multiple architectures. For example, YARN [33] is a centralized scheduler which also uses a lightweight distributed scheduler responsible for requests of low priority or short duration for each node-manager (the instance which controls a server).

Hydra [6] is another hybrid scheduler which extends YARN to split a data center into smaller sub-clusters, each controlled by a dedicated centralized scheduler. If a job spans across multiple sub-clusters, the affected schedulers communicate and share scheduling information.

2.1.2 Scheduling Logic

At the core of an RMF's scheduler is its scheduling logic, expressed in the scheduling policy. The existing scheduling policies (Section 6) differ in their algorithmic design of how they solve the resource scheduling problem, and also in the information they consider for the scheduling algorithm.

The algorithmic design either requires the scheduler to consider all resource requests in each scheduling attempt, e.g., if the scheduler is transforming the scheduling problem to a graph problem as in Firmament [16]. Other algorithmic designs require to consider only a subset of a single resource request for each scheduling attempt, e.g., if the scheduler is using a priority queue of pending resource requests.

The most basic set of resource request information is a set of server resources (e.g., CPU and memory). If there is no information available of how long a job will be running, a scheduler needs to do speculative scheduling (e.g., Sparrow [28]). In contrast, if a resource request contains information on how long a job will be running, schedulers can use this information and run a combinatorial optimization problem to (approximately) solve the corresponding mixed integer linear problem (e.g., Firmament [16]). Some schedulers also require knowledge on topological information and on how the containers of a resource request communicate for choosing the set of resources (e.g., CloudMirror [22]).

2.2 Reactive Programming

Functional reactive programming (FRP) [12] overcomes the issues of the Observer design pattern. Observers lack of composability requiring global variables to propagate state changes and add have a negative effect on code comprehension due to inversion of control [24].

FRP allows for data flows to be defined declaratively and more concisely. When declaring a reactive time-changing value v, the reactive system keeps track of all other reactive values on which v depends. The system updates v automatically whenever one of its dependencies changes. Changes are propagated through the reactive system until all transitively dependent values are updated.

In FRP, *signals* (sometimes referred to as *behaviors*) represent continuous time-changing values which are automatically updated by the language runtime. REScala [30] supports *signal expressions* for defining time-changing values. For instance, the signal expression `Signal { s1() + s2() }` depends on the signals `s1` and `s2`. The expression is automatically recomputed whenever `s1` or `s2` changes. Discrete occurrences of values are represented by events. Events support various operators for mapping, filtering or merging event streams and can interoperate with signals. For example, the following code merges two event streams `e1` and `e2`, filters out all negative numbers and folds the discrete stream into a time-changing value, which contains the sum over all event occurrences and is updated automatically whenever one of the events fires:

```
((e1 || e2) filter { v => v >= 0 }).fold(0) { (acc, v) =>
  acc + v
}
```

3 Implementing Schedulers in GRASS

This section introduces the main features of GRASS. In the following, we discuss the scheduling of containers across multiple servers as an example. The proposed abstractions, however, can be easily extended to other kind of resources such as network switches.

We first describe the reactive scheduling interface used by developers to implement scheduling policies (Section 3.1) and provide an insight into how resource allocation is managed internally by the runtime by composing reactive values (Section 3.2). Second, we present examples for two different scheduling policies and show how they can be implemented using the reactive scheduling interface (Section 3.3 and 3.4). Last, we demonstrate how schedulers can be extended and how different schedulers can be composed, reusing existing scheduler policies (Section 3.5).

3.1 Reactive Scheduling Interface

The GRASS domain-specific abstractions for defining scheduling policies is based on a reactive interface to interact with both the application layer and the network controller. Scheduling policies specify resource allocation. Listing 1 shows the representation of resources. A `Container` is represented by a unique identifier and the resources it requires to be executed. A `Resource` can be uniquely identified and provides a specific amount of virtual CPU cores (`cpu`) and memory (`memory` – counted in megabytes).

Applications must be assigned resources prior to being executed. By sending a resource request (`ResourceRequest`) to the scheduler, applications tell the scheduler about the resource requirements (`ResourceRequirement`) they have, i.e., how many CPU cores and memory they need per container.

Listing 1. Resource representation.

```
case class Container(id: ContainerId,
  res: ResourceRequirement)
case class Resource(id: ResourceId, cpu: Int, memory: Int)
case class ResourceRequest(id: ApplicationId,
  requirements: List[ResourceRequirement])
case class ResourceRequirement(cpu: Int, memory: Int)
```

Listing 2. Reactives runtime interface.

```
val allResources: Signal[List[Resource]]
val freeResources: Signal[List[Resource]]
val allocs: Signal[List[(ResourceRequest, List[(Container,
    Option[Resource])])]]
val requestResource: Event[ResourceRequest]
val shutdownContainer: Event[Container]
```

If, for example, an application comprises three containers, the list will contain three instances of `ResourceRequirement`, one for each container.

The scheduler runtime provides a set of time-changing reactive values that can be used to gather information about available resources and resource allocations (Listing 2). The signal `allResources` provides a list of all available resources, while `freeResources` only provides a list of available resources. These signals are constantly updated by the runtime whenever new resources are added to or removed from the system and when resources are allocated or freed. The former allows modeling various real-world scenarios including failing resources or putting resources into maintenance mode. The `allocs` signal provides information about resource allocations. For each resource request, it provides a mapping between each container and the resource this container is allocated to. We use an `Option` for the resource to be able to express that a container has not been allocated yet. Finally, the `requestResource` event is triggered whenever an application requests resources from the scheduler.

All scheduler implementations need to adhere to the interface shown in Listing 3. A scheduler is implemented in terms of four events, `assign`, `reject`, `migrate` and `preempt`, which are the most basic events shared among all common scheduling approaches. Based on this common interface, different schedulers – as described in Section 2.1.1 – can be implemented. These events are handled by the scheduler runtime to update the current resource allocations. An `assign` event is fired to instruct the scheduler to allocated a specific resource for a container. The `reject` event allows rejecting a container from being allocated to any resource. The `migrate` event allows migrating a container from one resource to another and `preempt` allows stopping the execution of a container and free its resources.

Listing 3. Scheduler interface.

```
trait Scheduler {
  val assign: Event[(Container, Resource)]
  val reject: Event[Container]
  val migrate: Event[(Container, Resource, Resource)]
  val preempt: Event[(Container, Resource)]
}
```

Listing 4. Runtime implementation of `allocs`.

```
1  type Alloc = (ResourceRequest, List[(Container,
2    Option[Resource])])
3
4  def mapAlloc(allocs: List[Alloc])(container: Container,
5      from: Option[Resource], to: Option[Resource]) =
6    allocs map { case (request, jobAllocs) =>
7      request -> (jobAllocs map {
8        case (`container`, `from`) => container -> to
9        case alloc => alloc
10     })
11   }
12
13 val allocs: Signal[List[Alloc]] =
14   Events.foldAll(List.empty[Alloc])(allocs => Events.Match(
15     requestResource >> { request =>
16       (request -> (request.requirements map { requirement =>
17         Container(requirement, createId()) -> None
18       })) :: allocs
19     },
20     (shutdownContainer || reject) >> { container =>
21       (allocs
22         map { case (request, jobAllocs) =>
23           request -> (jobAllocs collect {
24             case jobAlloc @ (`container`, _) => jobAlloc
25           })
26         }
27         filter { case (_, jobAllocs) => jobAllocs.nonEmpty })
28     },
29     assign >> { case (container, res) =>
30       mapAlloc(allocs)(container, None, Some(res))
31     },
32     migrate >> { case (container, from, to) =>
33       mapAlloc(allocs)(container, Some(from), Some(to))
34     },
35     preempt >> { case (container, res) =>
36       mapAlloc(allocs)(container, Some(res), None)
37     }
38   ))
```

3.2 Scheduler Runtime

The internal implementation of the scheduler runtime that processes the events defined as part of the scheduler policy (`assign`, `reject`, `migrate` and `preempt`) is presented in Listing 4. The `mapAlloc` methods changes the resource associated to `container` from `from` to `to` for the given list of resource allocations (Line 4). The implementation of the `allocs` signal computes the list of all containers and their associated resources for every resource request (Line 1). The signal is implemented as a fold over the event streams from the application (Line 15 and 20) and the scheduler (Line 20, 29, 32 and 35).

Every `requestResource` event adds the new resource request to the allocation list and associates every requested container with `None` (Line 17), i.e., newly requested containers are initially not allocated. The change of the `allocs` signal will automatically propagate to the scheduler, which is responsible for finding an allocation. The `reject` and the `shutdownContainer` event (Line 20) remove containers from the list of allocations. The association of a given container to a given resource is added, changed or removed for the `assign`, `migrate` or `preempt` events, respectively.

Next, we demonstrate how the abstractions provided by GRASS can be used to implement different schedulers with varying complexity.

3.3 Greedy Scheduler

Listing 5 shows an example implementation of a greedy scheduling algorithm. The greedy scheduler tries to allocate the first container that is not yet allocated to some free resource. It will never migrate, preempt or reject for allocating a container. The implementation therefore just implements the `assign` event (Line 6). Whenever the current resource allocations or the set of available resources change (Line 7), we first collect all containers that have not yet been allocated (Line 10). In the next step the containers are sorted by some criteria (Line 11). This step is optional but provides more flexibility. For example, by sorting the containers based on the amount of resources they require, it is possible to schedule the largest or the smallest containers first. For each container, we try to find a resource that has enough capacity to run the container (Lines 12–16). The `find` operation on the `freeResources` signal returns an `Option`, where `None` indicates that no resource is available that fulfills the resource requirements of the respective container. Finally, we select the first container for which a free resource exists (Lines 17–19). For the case that no free resources or containers which have not been allocated exist, the `flatten` operator prevents the `assign` event from being fired (Line 20).

3.4 Preemptive Scheduler

The preemptive scheduler implementation shown in Listing 6 tries to prevent starvation by preempting containers for freeing additional resources if the available resources are not sufficient. The `assign` event is again triggered by a change in the resource allocations or the available resources (Line 6). In this example, the scheduler always tries to schedule the largest container (Line 10), which has not been allocated yet (Line 9). Which criteria is used is omitted from the listing for brevity, but for example, it could be the number of virtual CPU cores required. The scheduler always tries to allocate a container to the largest free resource (Lines 11–17).

Listing 5. Greedy scheduler implementation.

```
class GreedyScheduler extends Scheduler {
  val migrate = Event.empty
  val preempt = Event.empty
  val reject = Event.empty

  val assign: Event[(Container, Resource)] =
    ((allocs.changed || allResources.changed) map { _ =>
      (allocs()
        flatMap { case (_, jobAllocs) => jobAllocs }
        collect { case (container, None) => container }
        sortBy { /* sort to some criteria */ }
        map { container =>
          container -> (freeResources() find { res =>
            res >= container.res
          })
        }
        collectFirst {
          case (container, Some(res)) => container -> res
        })
    }).flatten
}
```

Listing 6. Preemptive scheduler implementation.

```
1  class PreemptiveScheduler extends Scheduler {
2    val migrate = Event.empty
3    val reject = Event.empty
4
5    val assign: Event[(Container, Resource)] =
6      ((allocs.changed || allResources.changed) map { _ =>
7        (allocs()
8          flatMap { case (_, jobAllocs) => jobAllocs }
9          collect { case (container, None) => container }
10         maxByOption { /* sort to some criteria */ }
11         flatMap { container =>
12           val res = freeResources().max
13           if (res >= container.res)
14             Some(container -> res)
15           else
16             None
17         })
18     }).flatten
19
20   val preempt: Event[(Container, Resource)] =
21     ((allocs.changed || allResources.changed) map { _ =>
22       (allocs()
23         flatMap { case (_, jobAllocs) => jobAllocs }
24         collect { case (container, None) => container }
25         maxByOption { /* sort to some criteria */ }
26         flatMap { container =>
27           val res = freeResources().max
28           if (res >= container.res)
29             None
30           else
31             (allocs() flatMap { case (_, jobAllocs) =>
32               jobAllocs collectFirst {
33                 case (container, Some(`res`)) =>
34                   container -> res
35               }
36             }).headOption
37         })
38     }).flatten
39 }
```

The preemption logic is implemented similar to the assignment logic (Lines 20–38). In case there are no resources to allocate the largest container that is not allocated yet, the code picks a container that is currently allocated on the largest free resource and shuts it down preemptively, freeing resources for allocating the largest container. After preemption, the runtime will update the `allocs` signal, which causes the scheduling policy to be re-executed. The scheduler will then either allocate the largest container or preempt another already allocated container to free more resources.

3.5 Composing Schedulers

We can easily extend the functionality of existing schedulers by refining the definition of the event streams of the `Scheduler` interface. For example, Listing 7 shows how we reuse the `PreemptiveScheduler` implementation to prevent starvation, but using a less aggressive from of preemption. Line 2 instantiates the `PreemptiveScheduler`, which was previously described. We define a `usePreemption` signal that changes automatically whenever the number of available resources changes. In the example, we specify a policy where preemption is used only if resource utilization is above 80 % (Line 3), i.e., the `preempt` event is filtered based on the value of the `usePreemption` signal.

Listing 7. Scheduler extension.

```
1  class LessPreemptiveScheduler extends Scheduler {
2    val ps = new PreemptiveScheduler
3    val usePreemption = Signal {
4      freeResources().sum / allResources().sum > 0.8
5    }
6
7    val assign = ps.assign
8    val reject = ps.reject
9    val migrate = ps.migrate
10   val preempt = ps.preempt filter { _ => usePreemption() }
11 }
```

We further support dynamic composition of different schedulers, which allows switching between scheduling policies at run time. Listing 8 shows a scheduler that either uses the greedy or the preemptive policy. The implementation instantiates both the `GreedyScheduler` (Line 2) and the `PreemptiveScheduler` (Line 3). The code defines a metric (left out for brevity) for switching between both scheduler (Line 5) and chooses the scheduler depending on this metric (Line 6). The `assign`, `reject`, `migrate` and `preempt` events are forwarded to the chosen scheduler.

Listing 8. Scheduler composition.

```
class AdaptiveScheduler extends Scheduler {
  val greedyScheduler = new GreedyScheduler
  val preemptiveScheduler = new PreemptiveScheduler

  val switch: Signal[Boolean] = Signal { ... }
  val scheduler: Signal[Scheduler] = Signal {
    if (switch()) greedyScheduler else preemptiveScheduler
  }

  val assign = (Signal { scheduler().assign }).flatten
  val reject = (Signal { scheduler().reject }).flatten
  val migrate = (Signal { scheduler().migrate }).flatten
  val preempt = (Signal { scheduler().preempt }).flatten
}
```

4 Building Resource Management Frameworks with GRASS

RMFs exist with a variety of architectures (e.g., centralized, two-level, distributed), using different internal scheduler designs (e.g., using a queue for pending requests or a graph which holds all requests), and considering a different set of available information about jobs and resources (e.g., with topology awareness of servers or with knowledge about the expected duration of a job at the time of submitting the job to the RMF). In the following, we show how Generic Reactive Application-Specific Scheduling (GRASS) supports building an RMF of these kinds.

4.1 Scheduler Architecture

The most simple architecture is a centralized architecture, where a single scheduler policy is performing the resource scheduling of an RMF (Section 2.1). For a centralized architecture, like the one used by (the very first version of) YARN [33], the GRASS runtime has a data-center-wide view of the resources and jobs and is therefore able to update the signals.

A more complex architecture is the two-level architecture, like the one used by Mesos [17]. In such an RMF, the GRASS runtime serves multiple scheduling policies. The two-level architecture is based on a pessimistic concurrency model, hence before a scheduling policy is able to use resources for scheduling purpose, an allocator component (Section 2.1) sends resource offers to a scheduling policy (making these resources available for scheduling purpose). GRASS supports two ways of realizing a two-level architecture. A straightforward implementation uses the GRASS runtime to act as the allocator, i.e., making the decision of which policy gets which resources in its `allResources` and `freeResources` signal. A drawback of this approach is that none of the scheduling policies could influence the decision of which resources are going to which scheduling policy. GRASS, however, also allows to delegate the allocator component to a composition of scheduler policies. This brings the benefit that `allResources` and `freeResources` contain all data-center-wide resources, allowing the composition of scheduler policies to consider policy-specific objectives when performing the allocation logic.

The distributed architecture, like the architecture used by Omega [31], implements an optimistic concurrency model. Similar to the two-level architecture, multiple scheduling policies run at the same time. However, instead of using an allocator component (two-level architecture) which sends resource offers, each scheduling policy considers (always) all resources when performing scheduling. This could lead to potential conflicts of scheduling decisions (when scheduling policies compete for the same resources). GRASS supports two approaches of resolving scheduling conflicts. A simple solution is to delegate the resolution logic to the GRASS runtime, which hides it from the scheduling policies. A better approach would leave the resolve logic to the composiion of scheduling policies, allowing to consider more policy-specific objectives when resolving allocation conflicts.

4.2 Scheduler Designs

The resource scheduling problem itself, i.e., considering a set of available resource and making the allocation decisions of which job gets which resource share, is a computationally expensive problem due to several factors like the number of involved jobs and resources (Section 2.1). This is why RMFs use scheduler designs which transform the problem to a problem of less complexity. In general, there are two approaches how the scheduling logic solves the scheduling problem. Either the scheduler considers all jobs when performing a scheduling attempt, or the scheduler considers only a small subset of all jobs or a single job for each scheduling attempt.

Firmament [16] transforms the resource scheduling problem into a graph problem where it solves a min-cost max-flow problem, i.e., Firmament needs to consider all jobs for each scheduling attempt. Sparrow [28] shows the opposite and considers only a single job for each scheduling attempt, simply by using a queue which orders jobs for processing.

The GRASS runtime provides the list of allocations `allocs`, which also holds entries for each newly arrived resource request (Section 3.2), which allows a scheduling policy to build up the internally required representation of requesting jobs to run its scheduling logic, like building a queue of demanding jobs or updating a graph each time a job arrives.

4.3 Extending Resources and Requests

A key aspect of GRASS is the possibility to design application-specific scheduling policies. Application-specific policies bring the advantage to come up with scheduling policies that consider special requirements or preferences when selecting resources specifically for each individual resource request. The GRASS runtime provides the resource requirements of each request using a `ResourceRequirement`, and the details about all data center resources using `Resource` (Section 3).

This interface can be easily extended to support for example topology-aware scheduling policies like Kraken [15] or CloudMirror [22].

To support topology-aware scheduling policies, we extend `Resource` with the resource location in the data center, which could be a reference to a graph representation. Typical use case for using topological information of resources are to check the distance between two resources or to check which resource belongs to which availability zone [16].

For scheduling policies which require the application to include a topology in a resource request, we further extend `ResourceRequirement` with a notion of how the requested containers want to communicate. Typical representations use a virtual cluster abstraction, which gives the requesting application the illusion of connecting containers to a single centralized switch with guaranteed bandwidth capacity [15]. A more flexible representation uses a directed graph of container groups, which allows for expressing more fine-grained communication patterns of the requested containers [22]. The GRASS runtime is not restricted to any of these representations by simply extending `ResourceRequirement` with the required information.

5 Evaluation

The objective of the evaluation is to assess how implementing different scheduling policies according to the application requirements and changing between these policies at run time influences the resource scheduling performance. We evaluated different resource schedulers implemented with GRASS in a discrete event-based simulator, similar to the Omega simulator [31]. The simulator simulates the data center resources and the users which send jobs, i.e., resource requests, and runs the scheduler providing the scheduling logic. The workload generator uses a trace of a Google data center [29] comprising 12 500 servers over about a month-long period. The trace contains information about the time when a resource request occurred and various details about the resource request itself, e.g., which resources have been requested. For this paper, we scaled the trace down to a cluster with 500 machines and ran a simulation of 48 hours. When a scheduler performs a scheduling attempt, we set the think time of the scheduler as a function of the number of containers a job (resource request) contains. This is a common scheme to evaluate the performance of data center resource schedulers [16, 31].

Setup We evaluated three different schedulers. The *greedy scheduler* without any preemption (Section 3.3), an *preemptive scheduler* that always preempts running jobs if the currently considered job cannot be scheduled (Section 3.4), and an *adaptive scheduler* that preempts other jobs depending on the overall data center resource utilization with different upper and lower utilization thresholds (Section 3.5). The latter

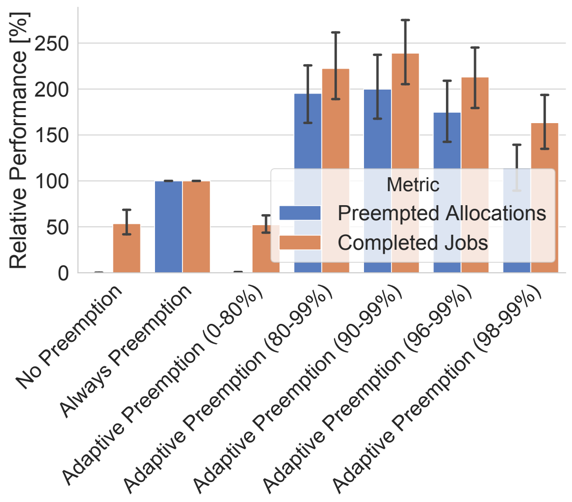


Figure 2. Comparison of different scheduling policies.

was evaluated using five different configurations. The scheduler switched to preemptive scheduling if the resource utilization u was 1. $u \leq 80\%$ 2. $80\% \leq u \leq 99\%$ 3. $90\% \leq u \leq 99\%$ 4. $96\% \leq u \leq 99\%$ and 5. $98\% \leq u \leq 99\%$.

In each experiment, we run the exact same workload (resource requests) using a different configuration of the scheduler that is responsible for doing resource scheduling. We repeated each experiment 11 times, each time using a different seed, where the seed determines which subset of the overall trace is used. As a metric, we take the number of *completed jobs*, i.e., jobs that got all requested resources and finished their container workload, and the number of *preempted allocations*, i.e., the number of allocations that the scheduler preempted during scheduling. Depending on the objectives of the RMF, a scheduling policy might optimize for a lower number of preempted allocations (since preempted containers might loose processing progress), and/or a higher number of completed jobs. This paper proposes not a specific scheduler logic, but rather a generic way of expressing composites of different scheduling logic using GRASS. Hence, we show how a combination of scheduling policies changes the performance characteristics of the involved scheduling policies.

Evaluaton Results Figure 2 shows both, the number of preempted allocations as well as the number of completed jobs for the different scheduling policies. All results are normalized to the experiment when the preemptive scheduler is running. For adaptive preemption, the numbers indicate the minimum and maximum data center resource utilization for which preemption is performed (if necessary).

The preemptive scheduler achieves two times more completed jobs, at the cost of more preemptions. The adaptive preemption scheduler with an upper threshold of 80 % shows the same performance as the greedy scheduler, hence preemption does not get triggered in this experiment when data center utilization is below 80 %.

The adaptive preemption schedulers that start preemption if data center utilization is above 80 % show more preemptions than the other schedulers, but also achieve higher numbers in completed jobs. Finding the best configuration of the adaptive scheduler should be a run time task, in order to be able to consider the ongoing performance characteristics and constraints given by the resource requests. GRASS allows such a design to build an RMF tailored to the application-specific requirements with a composition of scheduling policies that adapt at run time.

6 Related Work

This section discusses related work both in the field of data center resource scheduling and reactive programming.

6.1 Data Center Resource Scheduling

Data center resource scheduling is a very active research area over the last years [20]. Different scheduler architectures have been proposed, including two-level [17], centralized [33] and distributed schedulers [4, 6, 8, 31, 34]. Another line of research focuses on the information on which schedulers rely to make scheduling decisions, (e.g., run time estimates [16, 19, 32], speculative resource assignment [28]) or the optimization strategy (e.g., guaranteeing network bandwidth among allocated containers [2, 15]). Some schedulers combine multiple aspects of the before-mentioned aspects [9, 10]. GRASS unifies all these different aspects of resources schedulers in a single language.

A recent distributed RMF, Hydra [6], allows for changing the used scheduling policy at run time using an interceptor design pattern, but is restricted to run only one scheduling policy at the same time for each sub-cluster of a data center. The compositionality of GRASS allows for capturing both dimensions in the scheduler implementation.

6.2 (Functional) Reactive Programming

FRP – introduced by Elliott and Hudak [12] to define visual animations in a declarative style – has been applied to various areas including robotics [18], wireless sensor networks [26] and network switch programming [14]. While FRP is traditionally defined over continuous time and is given denotational semantics [1, 27], some languages deviate from the purely functional approach, e.g., resulting in an FRP-like abstractions defined over discrete time [30]. User interfaces are an especially popular field of application for FRP. Flapjax [24] is a JavaScript-based language for implementing client-side user interfaces of web applications. Other approaches like Elm [7], a purely functional language, compile to JavaScript. Reactive Extensions offer abstractions for event streams and are available for a number of languages, e.g., RxScala, RxJava or RxJS.

Similar to our approach of specifying resource scheduling policies by leveraging reactive programming, FRP has been applied to compute the operator placement for complex event processing operators [36]. Recent research on RP focuses on issues such as concurrency [11], fault tolerance [25], distribution [35, 37], different levels of consistency [23] and the application to areas such as autonomous vehicles [13] and IoT and edge computing [5].

7 Conclusion

In this paper, we presented GRASS, a programmable resource management framework that allows the definition of application-specific resource scheduling policies via a domain-specific abstractions. The abstractions are based on reactive programming, enabling the declarative specification of scheduling policies which react to changing application requirements as well as changing network conditions.

We demonstrated how our language design supports extension and composition of scheduling policies, simplifying the implementation of application-tailored schedulers and improving their performance.

Acknowledgments

This work has been co-funded by ERC grant FP7-617805 and the German Research Foundation (DFG) as part of the Collaborative Research Center (CRC) 1053 MAKI.

References

[1] Heinrich Apfelmus. 2011. Reactive-banana. http://hackage.haskell.org/package/reactive-banana. Accessed 2019-09-11.

[2] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2011. Towards Predictable Datacenter Networks. In *ACM SIGCOMM Computer Communication Review*, Vol. 41. ACM, 242–253.

[3] Theophilus A. Benson. 2019. In-Network Compute: Considered Armed and Dangerous. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*. ACM, New York, NY, USA, 216–224.

[4] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and Coordinated Scheduling for Cloud-scale Computing. In *Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI)*.

[5] Ben Calus, Bob Reynders, Dominique Devriese, Job Noorman, and Frank Piessens. 2017. FRP IoT Modules As a Scala DSL. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS 2017)*. ACM, New York, NY, USA, 15–20.

[6] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, Roni Burd, Sarvesh Sakalanaga, Chris Douglas, Bill Ramsey, and Raghu Ramakrishnan. 2019. Hydra: A Federated Resource Manager for Data-center Scale Analytics. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 177–192.

[7] Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 411–422.

[8] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. 2016. Job-aware Scheduling in Eagle: Divide and Stick to Your Probes. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*.

[9] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. 2018. Kairos: Preemptive Data Center Scheduling Without Runtime Estimates. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*. 135–148.

[10] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. 2015. Hawk: Hybrid Datacenter Scheduling. In *2015 USENIX Annual Technical Conference*. USENIX Association, 499–510.

[11] Joscha Drechsler, Ragnar Mogk, Guido Salvaneschi, and Mira Mezini. 2018. Thread-safe Reactive Programming. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 107 (Oct. 2018), 30 pages.

[12] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*. ACM, New York, NY, USA, 263–273.

[13] Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. 2017. Vehicle Platooning Simulations with Functional Reactive Programming. In *Proceedings of the 1st International Workshop on Safe Control of Connected and Autonomous Vehicles (SCAV'17)*. ACM, New York, NY, USA, 43–47.

[14] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: A Network Programming Language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, New York, NY, USA, 279–291.

[15] Carlo Fuerst, Stefan Schmid, Lalith Suresh, and Paolo Costa. 2018. Kraken: Online and Elastic Resource Reservations for Cloud Datacenters. *IEEE/ACM Transactions on Networking (TON)* 26, 1 (2018), 422–435.

[16] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. 2016. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *Proceedings of OSDI'16: 12th USENIX Symposium on Operating Systems Design and Implementation*. 99.

[17] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.. In *NSDI*, Vol. 11. 22–22.

[18] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2003. Arrows, Robots, and Functional Reactive Programming. In *Summer School on Advanced Functional Programming 2002, Oxford University (Lecture Notes in Computer Science)*, Vol. 2638. Springer-Verlag.

[19] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 261–276.

[20] Brendan Jennings and Rolf Stadler. 2014. Resource Management in Clouds: Survey and Research Challenges. *Journal of Network and Systems Management* (2014), 1–53.

[21] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 35–49.

[22] Jeongkeun Lee, Yoshio Turner, Myungjin Lee, Lucian Popa, Sujata Banerjee, Joon-Myung Kang, and Puneet Sharma. 2014. Application-driven Bandwidth Guarantees in Datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM, 467–478.

[23] A. Margara and G. Salvaneschi. 2018. On the Semantics of Distributed Reactive Programming: The Cost of Consistency. *IEEE Transactions on Software Engineering* 44, 7 (July 2018), 689–711.

[24] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: A Programming Language for Ajax Applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 1–20.

[25] Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. 2018. Fault-tolerant Distributed Reactive Programming. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs))*, Todd Millstein (Ed.), Vol. 109. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 1:1–1:26.

[26] Ryan Newton, Greg Morrisett, and Matt Welsh. 2007. The Regiment Macroprogramming System. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN '07)*. ACM, New York, NY, USA, 489–498.

[27] Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, New York, NY, USA, 51–64.

[28] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 69–84.

[29] Charles Reiss, John Wilkes, and Joseph L. Hellerstein. 2011. *Google Cluster-usage Traces: Format + Schema*. Technical Report. Google Inc., Mountain View, CA, USA. Revised 2014-11-17 for version 2.1. Posted at https://github.com/google/cluster-data.

[30] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications. In *Proceedings of the 13th International Conference on Modularity (MODULARITY '14)*. ACM, New York, NY, USA, 25–36.

[31] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 351–364.

[32] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. 2016. TetriSched: Global Rescheduling with Adaptive Plan-ahead in Dynamic Heterogeneous Clusters. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*. 35:1–35:16.

[33] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 5.

[34] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 18.

[35] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed System Development with ScalaLoci. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 129 (Oct. 2018), 30 pages.

[36] Pascal Weisenburger, Manisha Luthra, Boris Koldehofe, and Guido Salvaneschi. 2017. Quality-aware Runtime Adaptation in Complex Event Processing. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '17)*. IEEE Press, Piscataway, NJ, USA, 140–151.

[37] Pascal Weisenburger and Guido Salvaneschi. 2019. Multitier Modules. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, Alastair F. Donaldson (Ed.), Vol. 134. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 3:1–3:29.