The Design Space of Emergent Scheduling for Distributed Execution Frameworks

Paul Dean
School of Computing and Communications
Lancaster University
Lancaster, UK
p.dean1@lancaster.ac.uk

Barry Porter
School of Computing and Communications
Lancaster University
Lancaster, UK
b.f.porter@lancaster.ac.uk

***Abstract*—Distributed Execution Frameworks (DEFs) such as Apache Spark have become ubiquitous as a solution for the execution of user-defined jobs to process terabytes of data across hundreds of nodes. One of the key costs of DEFs is *scheduling* of which parts of each job are placed on each host; better scheduling decisions provide lower overall execution time for each job, more efficient resource usage, and reduced energy consumption. Existing DEFs use a static approach to scheduling, either with a single generalised scheduler which aims to be a good fit for most workloads, or with a special-purpose scheduler which is tuned to optimise for a particular kind of workload. In both cases the scheduling implementation is fixed at design-time such that the DEF is unable to adjust to the actual characteristics of workloads that arrive at deployment time. In this paper we introduce an *emergent scheduler for Distributed Execution Frameworks*. This scheduler can be composed and re-composed at runtime from a set of different building blocks, allowing the system to dynamically provide the benefits of differing scheduling policies over time depending on the actual properties of incoming workloads – with improved performance and resource usage. In this paper we present the overall design of our emergent scheduler, we discuss the theoretical design space of different scheduling approaches, and we examine a specific research question to determine *the correlation between workload properties and scheduling performance for different scheduler implementations*. Our results are based on a real implementation of our emergent DEF running across multiple hosts in a real datacentre, and our implementation is made available as open-source software.**

I. Introduction

The increasing volume of data created today, combined with increases in processing capabilities and data storage capacity, has led to the emergence of distributed execution frameworks (DEFs) such as Apache Spark [1] to assist with scheduling and processing tasks across hundreds or thousands of compute nodes. Common examples of such tasks include searching or sorting huge databases [2], training deep learning models on very large training sets [3], or processing data science workloads to gain business intelligence [4]. A user submits their overall processing job to one of these frameworks, which then determines how to split up the input data and parallelise and schedule the execution tasks across hundreds of hosts in order to output the data processing result as quickly as possible. DEFs are typically hosted in cloud computing infrastructures, and will usually be processing multiple (tens, or even hundreds) of such jobs from different users concurrently.

As the use of DEFs has increased over the last decade, and the range of different uses for them has increased, this has created a very wide variety of different job types – which makes scheduling very difficult to optimise across the different properties and behaviours of those job types and their combinations. Using the wrong scheduling approach for a given mixture of workloads can be extremely costly in completion time, and in power consumption across the set of servers which are being used to process the workloads [5]–[7].

The state of the art in scheduling design for DEFs offers both different scheduling *architectures* (e.g. centralised or decentralised) and different scheduling *policies* within those architectures (e.g. first-in-first-out). Overall there are three major approaches to date. *General-purpose* schedulers attempt to represent a 'best fit' for most workloads, using a single scheduling architecture and policy, such as that of Apache Spark [1]. *Workload-specific* schedulers, such as that of Ray [8], focus on improving end-to-end performance for a specific workload type, for which a general-purpose scheduler would perform poorly. *Hybrid* (e.g., [9]–[11]) *and adaptive* [2], [12], [13] approaches attempt to gain the best of both by simultaneously employing multiple different scheduling architectures or policies. However, to date they use hand-crafted rules or expect the user to submit detailed specifications with their data processing job to enable selection of the ideal scheduler – or alternatively use offline training of a policy which is then fixed at runtime. Overall, no DEF scheduling approach has examined runtime learning of scheduling architecture and/or policy under uncertainty, based only on real-time observations.

In this paper we introduce an *emergent scheduler*, which composes individual building blocks of behaviour at runtime to form different DEF scheduling architectures or policies. Our approach is able to continuously and seamlessly recompose the scheduler architecture and policy at runtime, based on observations of its deployment environment, to continually drive the system towards an optimal scheduling behaviour based on the actual workloads that it is receiving and the way in which those workloads behave on the set of computational hosts available to the DEF. In principle this allows the scheduler to constantly lower its scheduling overhead and improve decision quality – thereby completing workloads faster and using less server power to do so. Our emergent DEF system in itself provides a new examplar of a compositional self-adaptive system – and in this paper we report on using our emergent DEF to help answer three research questions:

- **RQ1:** What are the dimensions of compositional adaptation in a self-adaptive DEF scheduler?
- **RQ2:** Do different DEF workloads correlate with different ideal scheduler policy compositions, and if so how?
- **RQ3:** If 'yes' to RQ2, can we identify points in a workload at which it is rational to adapt between different scheduler compositions?

To study these questions we have built a fully-functional emergent DEF system, the scheduling element of which can be formed from a variety of different building blocks. We answer RQ1 using a theoretical analysis of the design space. For RQ2 and RQ3 we deploy our emergent DEF in a real data centre environment and provide it with a range of different workloads to examine which workloads operate best under which scheduler compositions.

Our empirical results demonstrate that different workloads *do* strongly correlate with different optimal scheduler compositions, such that no composition is best in all workloads, and that there are some potential indicators in the workload trace of when the system should change between different compositions according to the workload's trajectory. This study indicates that compositional adaptation is promising in a DEF's scheduler system, and serves as a baseline for future work in which we aim to develop online machine learning support to automate the decision making of which composition to use at any point in time. Our system is made available as open-source software, along with detailed instructions of how to recreate our empirical results[1].

In the remainder of this paper we first present related work in Sec. II. We then introduce our emergent DEF design in Sec. III, along with an analysis of its adaptation dimensions. We then present our empirical evaluation in Sec. V, and conclude the paper in Sec. VI.

II. Related work

Distributed Execution Frameworks (DEFs) provide a solution to the increasing demands of large-scale data processing workloads, providing the means for simplified submission, deployment, and distributed execution of submitted jobs. As one of the dominant uses for data centre and cloud systems at present, DEFs have received significant research attention.

In this section we focus specifically on research that considers scheduling in DEFs, which is one of their key operational costs. Improvements to scheduling provides lower end-to-end completion time of workloads [2], [3], [8], higher resource utilization [5], [6] and, lower energy consumption [7]. In existing research there are four main differing approaches to scheduling workloads within DEFs: general purpose schedulers, workload-specific, hybrid, and adaptive.

[1]http://www.projectdana.com/research/seams2021dean

A. General purpose

First, general-purpose approaches, which consist of a single architecture and scheduling policy. General-purpose approaches are able to schedule a large set of workload types applying the same policy to all workloads received [1], [6], [14]–[16]. One prominent example, Apache Spark [1], utilises a fixed scheduling policy ensuring higher quality scheduling decisions through guaranteeing data locality. However, this limits the performance for scheduling a single workload type as the scheduler is accommodating a larger set of workloads the cost of scheduling adversely effects latency sensitive workloads [8], [17].

B. Workload specialists

Second, workload specific approaches, focus on improving the performance of a specific workload type, reducing the performance for workloads not specific to the framework [8], or lose the ability to schedule the majority of workloads [3], [4]. While limiting the performance of a wider set of workloads, workload-specific approaches address the issues of scheduling recent diverging and complex workloads, for example reinforcement learning [8], machine learning [3] and, real-time processing [4], [18]. However, through limiting the number of workloads intended to be scheduled by the framework, the performance of scheduling for a workload is improved. This can be seen with Ray and Apache Spark, Ray provides a reduced scheduling overhead and provides improved performance for latency sensitive tasks in comparison to Spark [8]. However, the ability to make higher quality scheduling decisions are lost by Ray as the scheduler loses the guarantee of data locality and encounters a performance hit when encountering data-intensive workloads a property spark guarantees [1]. In addition, workload specific approaches may require improvements or alternate frameworks to reacquire lost scheduling behaviour and guarantees, for example, accommodate multi-tenant scenarios [5] or improve the efficiency of scheduling decisions [19].

C. Hybrid approaches

Third, Hybrid approaches, attempt to provide the benefits of improved performance offered by workload specific approaches for the larger set of workloads available to general-purpose approaches through the combination of multiple scheduling approaches. The use of multiple scheduling approaches simultaneously provide improved performance for scheduling a larger set of workloads as the policy or architecture may be switched to a set alternative, better suited to schedule the incoming workload [9], [10]. However, hybrid approaches split cluster resources by reserving resources for each approach or add an overhead of switching the scheduling approach and suffer a performance hit if incorrectly selected; this limitation creates difficult scenarios for hybrid approaches to outperform the performance offered by general-purpose approaches, further difficulties arise as users may need to add attributes to the submitted workload to aid in the correct policy is selected [11].

D. Adaptive and learned schedulers

The fourth set of approaches are learning and self-adaptive approaches to scheduling. Learning-based approaches to scheduling are appealing as they offer a significant improvement to reducing the overhead of scheduling in comparison to previous frameworks. The increased performance is achieved through the application of a machine-learning model to increase scheduling performance by lowering the scheduling overhead and rewarding high quality scheduling decisions [2]. However, with a significant increase in performance the approach is limited to a specific workload type and would require the model to be re-trained for unforeseen workloads. Adaptive approaches achieve performance increases through the adaptation of the scheduling policy or architecture at run-time, providing the benefit of an alternate scheduling approach autonomously without the limitations of hybrid approaches requiring two separate approaches simultaneously. However, current adaptive approaches to scheduling are limited by their need for user provided parameters to inform scheduling decisions [12], or the need for user-generated models [13]. The need for domain-specific knowledge and user intervention for informing adaptation limits the approach as scheduling may only be improved and maintained across all encountered workloads with sufficient data and correct models for all. This requires a significant amount of time and knowledge in an environment where workloads are becoming increasingly complex, sporadic and potentially non-repeating.

III. Approach

Every DEF approach has some core conceptual elements in common, illustrated by Fig. 1. Specifically, they are architected such that one distinct software system operates as a cluster manager (Resource Manager) for the collection of host computers (workers/Node Managers) available to the DEF, acting as the overall orchestrator of the system and the place to which jobs are initially submitted by users. A separate but interrelated software system (Application Manager) then operates on one of the available worker nodes to manage delivery and execution of individual tasks on each host (Executors). Besides this basic commonality, there is then significant variation between different DEF approaches in the detail of how scheduling works between the cluster manager and each worker node – with some approaches using a centralised scheduling design where the cluster manager makes all decisions, and others using a decentralised design where worker nodes manage and schedule their own task allocations. Broadly speaking, centralised scheduling approaches may make better decisions (which is good for long-running tasks), but take longer to calculate those decisions (which is bad for short tasks); while decentralised solutions may take much quicker scheduling decisions as they are considering only local states, but may make decisions of poorer quality.

At a high level our emergent DEF is constructed as two different emergent systems representing these two core roles of *cluster manager* and *worker node* agent; in principle this allows us to adjust both the distributed scheduler architecture

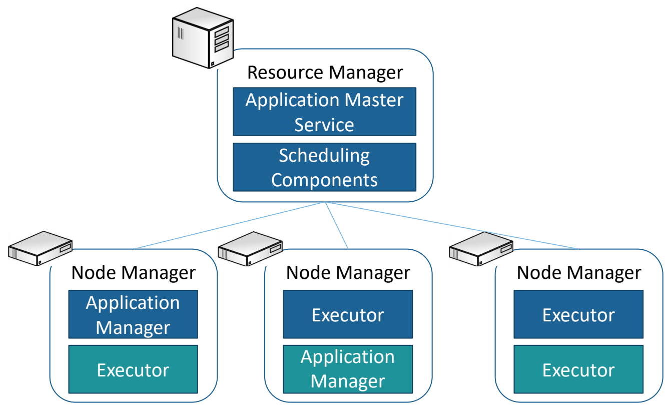

Fig. 1. Overview of a Distributed Execution Framework architecture

and the scheduling policies used within that architecture. In this section we first introduce general background on how emergent software systems work, then present our emergent DEF system as implemented with this paradigm.

A. Emergent Software Systems Background

Emergent software systems use an assembly, perception, and learning framework to automatically construct a working system from a large pool of potential building blocks, and then to continually re-assemble that system while it is running. Continuous re-assembly at runtime is based on real-time observations of the system's performance and deployment environment conditions, connected to real-time learning which explores possible compositions of behaviour to continually move towards a higher-performing composition. To date they have been used in web server systems in both single-host and distributed multi-host environments [20], [21].

Operationally, the assembly part of the framework is provided with a 'main' component, and examines the dependencies (exposed as 'required interfaces') of that component. For each of those required interfaces, the assembly module searches for all available implementing components; many such interfaces will have multiple implementation variants such as different sorting algorithms, buffer management strategies, or cache implementations. The assembly module then examines all of the required interfaces of *those* components, recursively, until it has a set of possible discrete compositions of components which all perform the same functional task but do so in different ways. Probes are then injected into the composed system to read metrics of interest, which are fed to the perception module for collation. A learning algorithm communicates with both the assembly and perception modules to select a composition of interest, read reward information about that composition after an observation period, and then select the next composition. Each composition choice causes the assembly module to compute a delta between its current composition of components and the selected one, and to perform a series of specific runtime adaptations which hot-swap the relevant components to reach the target composition.

The pool of components available to an emergent software system can be dynamically changed at runtime by adding new variants of existing interfaces or removing existing ones. We use the generalised emergent software framework reported in [22], and our emergent DEF is built using the Dana component-oriented programming language which provides seamless runtime hot-swaps with guaranteed soundness [23].

B. An Emergent DEF

Our emergent DEF is composed of two distinct emergent systems: one acting as a cluster manager to which jobs are submitted, and the other as a node manager resident on each worker node that is part of the cluster. This approach in principle allows us to create multiple different scheduling architectures (e.g. centralised / decentralised) by adding the appropriate component building block variants to the pool available to the cluster manager and node manager. For the purpose of the study in this paper, however, we use only a centralised two-tier scheduling architecture and provide multiple scheduling policy implementations within this architecture – such as first-in-first-out or resource-balanced policies.

Fig. 2 shows a subset of the components used by the cluster manager and node manager systems, focusing on the elements for which relevant variation is available for the purpose of our study (in reality both systems are composed of over 30 components). The cluster manager itself is divided into two different systems, a *resource manager* and an *application manager*. The resource manager schedules overall jobs and allocates resources for them, while the application manager schedules individual tasks within the resources allocated to a specific job.

When a data processing task is submitted, for example a Map/Reduce job [24], the description of the job is sent to the resource manager. In the Map/Reduce example, this includes the user-defined scripts representing the map and reduce logic, plus the data source from which input data for map processes will be pulled. As part of the job description, the resource manager may have an estimate of how many CPU cores and how much RAM each map and reduce task is anticipated to need, which may or may not be accurate; even if it is accurate, the resource manager will have no information about how long each map or reduce task may take to complete.

Based on this job description, the DEF then needs to consider both this data processing job and all of the others that have been concurrently submitted or are still in progress, and determine which jobs to schedule, how many resources to allocate in terms of worker nodes, and which specific worker nodes to use based on their relative work profiles and network locations / network path capacities.

In a centralised scheduling architecture, the resource manager is responsible for collating information about the current task load and completion status of each worker, and makes global scheduling decisions on the basis of this. In order to gain this information, worker nodes and application managers exchange regular heartbeat messages with the resource manager, which periodically update the resource manager's view of the cluster status. When a centralised scheduler is making decisions, it is therefore usually doing so on slightly out of date information about the cluster – unless it actively probes every worker node before every decision, which adds significant latency to decision-making in clusters of hundreds or thousands of worker nodes. When the resource manager has made a scheduling decision, resource containers are allocated on each worker node, and an application manager is started to manage to flow of tasks for that job within its allocated resources. From the central resource manager's perspective, its objective is to ensure that resource utilisation in each worker is maximised, while also ensuring that no worker is ever overloaded in terms of its RAM or CPU core usage from its current collection of tasks, since this can cause extreme slowdown in task / job completion.

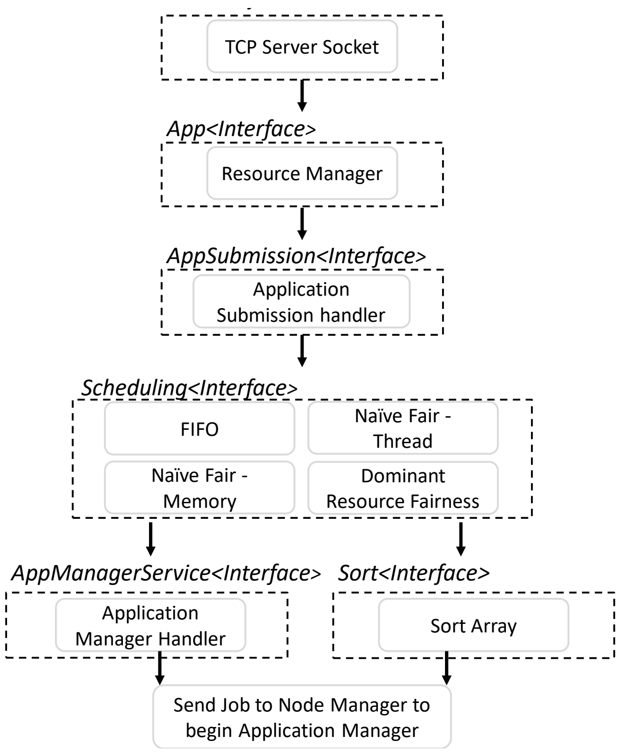
TCP Server Socket
App<Interface>
Resource Manager
AppSubmission<Interface>
Application Submission handler
Scheduling<Interface>
FIFO
Naïve Fair - Thread
Naïve Fair - Memory
Dominant Resource Fairness
AppManagerService<Interface>
Sort<Interface>
Application Manager Handler
Sort Array
Send Job to Node Manager to begin Application Manager

Fig. 2. Example of a composed Resource Manager, with alternative scheduling policy components.

Many distributed data processing jobs, including our map/reduce example, also have *dependencies* in their job between different sub-tasks; for this reason it is useful to separate out the operation of each job to its own application manager, which can analyse the dependency graph for that job and determine which tasks should go into which allocated resources in the cluster for that job. In our map/reduce example these dependencies exist between map and reduce tasks: reduce tasks rely on the data transformation of map tasks, and so reducer tasks only begin to be started up once enough map tasks have finished. This also creates a *data locality* question for the application master in deciding which tasks to place into which allocated worker resources, since reducers are fed data from appropriate map tasks; to avoid networking bottlenecks, reducer locations are ideally arranged in such a way that this data does not need to travel far in the cluster (for example by placing reducers tasks on the same host at which at least some of the data from relevant map tasks is already stored).

Overall this creates a very complex scheduling problem even in a static or single-use system. The co-location of many processing jobs on the same cluster, and uncertainty over the precise resource requirements and run time of each task in a job, also makes it a highly *dynamic* problem.

C. Scheduler Architecture

Our current scheduler architecture is restricted to a centralised approach, which reflects the dominant architecture of systems like Apache Spark [1]. Although our emergent DEF design allows alternative scheduling architectures to be added to the pool of alternatives, including fully decentralised ones, for this study we are focused on different scheduling policies within the centralised scheduling approach.

We specifically study four different policies, representing four different compositions of components in the resource manager which schedules entire jobs and allocates resources on worker nodes in which the tasks of those jobs can execute.

The first scheduler is first-in-first-out (*FIFO*), a common approach implemented by a range of existing DEF frameworks [1], [18]. FIFO handles incoming job requests sequentially without considering the relative resource profiles of each job. This means that a very resource-hungry job may be the only thing scheduled in the cluster due to it being first in the queue, even if the next four jobs in the queue could have been executed in parallel as they have lighter resource requirements. More commonly, it also means that if the *third* job in the queue could in theory have been executed alongside the *first* job, but the second job would *not* fit with the first job, then only the first job is run by itself to completion rather than running it in parallel with the third job in the queue. In general, a FIFO scheduling approach tends to be good in cases where the job queue is relatively homogeneous in resource usage, but bad in cases where it is not.

Our second and third schedulers are implementations of *Naive Fair* [25], which schedules jobs dependent on their resource usage and attempts to ensure resource utilisation fairness across submitted jobs. This approach is also used across a range of existing DEF frameworks (e.g., [14], [19]), and works by allocating resources on the basis of evenly sharing a particular resource (such as memory) between all jobs which request that resource. If five jobs are submitted to the resource manager at the same time, and all request the same amount of memory overall, the resource manager will deploy all five jobs into the cluster but give them an equal share of the available cluster's memory (such that each job may get less memory than it would like). We implement two different versions of this scheduler; one which prioritises *CPU core* usage as its fairness objective, and the other which prioritises memory usage. In general, Naive Fair scheduling tends to be good when the job list is heterogeneous and most jobs submitted to the DEF have the same dominant resource (i.e., all are memory-intensive), but is bad in cases where different jobs have a different dominant resource (so some require memory and some require CPU cores). In this case a Naive Fair CPU-based scheduler for example will not establish fairness of memory, allowing a memory-intensive job to dominate the cluster at the potential expense of all others.

Our final scheduler implementation is Dominant Resource Fairness (DRF), a scheduling policy designed to improve fairness among applications through comparing the applications' relative dominantly-reserved-resource rather than using a system-wide policy of either checking memory or thread usage [26]. DRF otherwise works in a similar way to Naive Fair, and tends to be good when both the job list and dominant resource are hetergeneous. If the job list and resource usage tend to be homogeneous, however, DRF will spend longer taking potentially equivalent decisions to a simple approach like FIFO. In addition, all of the 'fair' schedulers can have a tendency to over-share resources which can slow down completion times for all jobs in high-concurrency settings – and in extreme cases can consequently result in longer cumulative job completion times compared to a simpler approach like FIFO.

IV. Adaptation Analysis

Our specific study in this paper focuses on the runtime adaptation of scheduling policies. However, our overall emergent DEF supports a wider range of compositional adaptation. Based on our broader implementation work to date, in this section we provide an analysis of each compositional adaptation dimension and the likely effects and tradeoffs of adaptation in those dimensions.

These dimensions are the scheduler architecture; application master scheduling policy; and fault tolerance strategy. We also examine the key challenges for successful self-adaptive systems in this domain, which are workload analysis, mixed workloads, and action latency.

A. Adaptation Dimensions

a) Scheduler architecture: The top-level architecture of the scheduler can range from fully centralised to fully decentralised, or a hybrid of the two. Centralised architectures maintain a central list of resources and current task allocations, though this list is often slightly out of date as it relies on periodic notifications from worker nodes; when the list is up to date the scheduler is able to make high quality decisions, but ones that can often take time to compute. Decentralised schedulers operate by worker nodes scheduling tasks initially based on their local compute availability; these decisions are fast but uncoordinated [18]. Hybrid scheduling architectures concurrently provide or separate cluster resources for a combination of scheduling approaches and schedule tasks via one approach typically determined by their estimated runtime; incorrect decisions on the scheduling approach are costly and the extended decision process adds additional overhead [11]. Centralised architectures tend to work well in data-intensive workloads, but poorly in latency sensitive workloads where even millisecond delays can greatly impact performance [8]. Decentralised schedulers work well with latency sensitive workloads as overheads for scheduling are reduced, but are poor for workloads requiring data locality guarantees or "bin-packing" scenarios [18]. Hybrid schedulers work well for both coarse and fine grained workloads, but add latency and may perform poorly with unexpected workloads as they balance trading scheduling overhead and execution guarantees [9]. One adaptation dimension is therefore overall the architecture of the scheduler to further reduce scheduling overheads or improve the quality of scheduling decisions, suppressing the limitations of a single architecture and the cost of concurrently maintaining multiple.

b) Application master scheduling policy: Our current study focuses on the scheduling policy of the top-level resource manager. However, in scheduling architectures which use an application manager to govern task execution of a particular job, the way in which that application master schedules tasks within its allocated resources is an obvious point of implementation variation. Classic strategies like first-in-first-out, or more nuanced approaches which weight execution order by location in the dependency graph, are possible in the application manager, as well as approaches which consider data locality in different ways between dependent tasks in the job. Because variation at the application manager level would create two inter-dependent autonomous tiers, some form of coordination or planning/knowledge exchange is likely to be necessary between the resource manager's adaptation strategy and that of application managers.

c) Fault tolerance strategy: Finally, the way in which DEFs handle fault tolerance can make a big impact on their completion times. In real deployments host failures and common, and the workload characteristics can make a significant difference to the best recovery strategy. At a task level, for example, long-running tasks can be periodically checkpointed so that they can be quickly resumed elsewhere in their current host worker node fails; while for short-running tasks this approach is unlikely to be worthwhile since re-executing a failed task from start is relatively fast. The same applies to the overall job level, where options include checkpointing the application manager's state of the current job progress, or running multiple resource manager copies which mirror each others' state to maintain knowledge of application manager resource reservations and results.

B. Self-Adaptation Challenges

As discussed above, while there are interesting adaptation dimensions in the DEF domain, enabling self-adaptive behaviours comes with specific challenges – in particular workload analysis, mixed workloads, and potentially extreme action-effect latency.

a) Workload behaviour analysis: One of the most intensively studied aspects of self-adaptation in DEFs is workload behaviour analysis (though not in the context of compositional adaptation for the DEF's implementation itself). One of the key challenges to making accurate scheduling decisions is correctly estimating the characteristics of each job, including how many CPU cores it needs, how much RAM it needs per task, and how long each task is likely to execute for. The more accurately these properties can be estimated, the better the decision-making can be at the scheduler. A wide range of research has therefore examined how to use various proxy measurements of a submitted job and its associated data input to attempt to predict the job's runtime properties [9], [10], [15], [27]. This research is complementary to our own, and would fit into our framework prior to the initial scheduling step.

b) Mixed workloads and action-effect latency: For systems that can continually adapt their behaviour, a particular challenge in DEFs is that workloads observed in a real deployment will tend to be mixtures of each individual anticipated workload, where any given mixture is untested – and each different mixture creates a new state about which to learn at runtime. It may therefore be useful to abstract over details of specific workloads, and transfer learned information about individual workloads into mixtures of those workloads.

In this context it is also unclear how best to measure performance of a self-adaptive DEF at runtime: whether average task execution time is sufficient, or whether the average completion time of entire job sets is needed to gain an accurate measure of scheduling effectiveness. Entire jobs can take days to complete in some application settings, so this has significant implications for the speed with which adaptation can take place – and the speed at which traditional explore/exploit reinforcement learning can occur.

Finally, there is likely to be relatively high latency in the effect of taking an action (i.e., adapting to a different composition). When changing from one scheduling policy to another, or from one architecture to another, a large number of in-progress tasks are likely to exist which were scheduled by the previously-used DEF policy or architecture, and it will take time – potentially a long time – for the set of all active tasks to shift to an allocation that is entirely the design of the newly-chosen scheduling policy. This again makes decision-making potentially slow, unless factors such as gradually improving task execution times can be taken as evidence that the system is moving in the right direction and so can 'score' the chosen DEF composition early and continue exploring others.

V. Evaluation

Our second and third research questions are studied using an empirical evaluation. These questions are:

- Do different DEF workloads correlate with different ideal scheduler policy compositions, and if so how?
- If 'yes', can we identify points in a workload at which it is rational to adapt between scheduler compositions?

The answers to these questions are informative towards future application of machine learning approaches to automate the self-adaptation of our emergent DEF scheduling framework. Our experiments use the four different scheduling policy alternatives presented in Sec. III-C: FIFO, Naïve Fair Thread, Naïve Fair Memory, and DRF. Each scheduling alternative

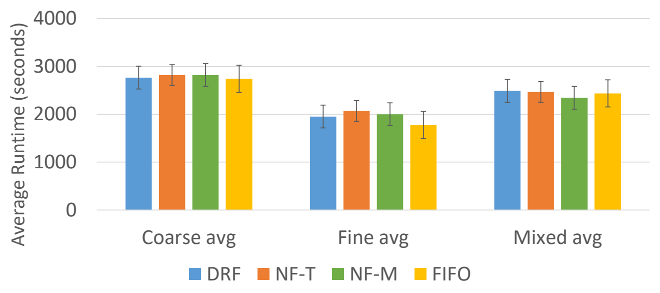

Fig. 3. Average total run-time of each workload class, by granularity

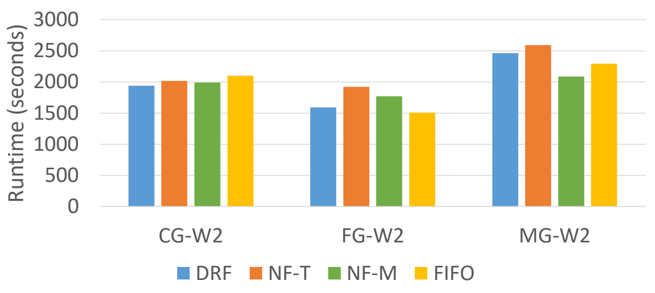

Fig. 4. Example total run-time of a specific workload for each granularity

was deployed on a real data centre cluster and issued with a set 15 synthetic workloads of differing granularity. In each experiment, for each scheduling alternative, we record the time to complete each job within a workload and the overall completion time of each workload. All of our source code, including workloads and instructions to repeat our evaluation, is made available online[2].

A. Methodology

Our evaluation uses synthetic workloads which are specifically designed to support a methodical examination of the design space for DEF scheduling. Our 15 workloads consist of 5 coarse-grained containing higher demand for system resources, 5 fine-grained workloads containing lower resource demand and higher task counts, and 5 mixed-grain workloads containing a mixture of fine and course grained workloads. The specific content of each workload, including thread and memory requirements and task counts, is generated with random variation within a certain threshold (such that coarse-grained workloads required thread counts within a certain range). Each workload features 20 distinct data processing jobs, which are submitting to the DEF with a regular 300ms interval (such that the cluster is often processing multiple overlapping jobs in parallel, as is typical in DEF services). Each individual job in a workload contains up to 40 tasks which can be deployed with varying levels of parallelism across the cluster.

All experiments performed were within a real data centre across a five-node cluster, consisting of one Resource Manager and four Node Managers. The Resource Manager has a Xeon CPU E5-2630v4 (20 threads x 2.20GHz), 16GB memory. The remaining four Node Managers each have a Xeon CPU E3-1280v2 (16 threads x 3.60GHz), 16 GB of memory. The nodes are connected via gigabit Ethernet. Experiment workloads and scheduling policies differ; however, the underlying hardware remains unchanged throughout the experiments.

B. Results for workload granularity

In this section we present our results on the performance of scheduling policies Naïve-Fair Memory (NF-M), Naïve-Fair Thread (NF-T), FIFO, and Dominant Resource Fairness (DRF) when subjected to the set of workloads described above.

[2]http://www.projectdana.com/research/seams2021dean

Fig. 3 shows the average total running time results from each workload class, on each of our four scheduling alternatives. Overall we see that the coarse-grained workload is completed most efficiently by the FIFO scheduler, which is also the most effective scheduler for the fine-grained workload. The mixed workload, by comparison, is completed most efficiently by the NF-M scheduler. This results demonstrates that there is not a single ideal scheduler alternative across different workload patterns, and confirms the results of various workload-specialist DEF research studies. If we consider the time difference between the best and worst scheduler choices in these scenarios, on average FIFO completes its fine-grained workload in 20% less time than NF-T, which is a significant amount of computation power and energy consumption.

These overall results indicate that suitable divergence exists between scheduler implementations for compositional self-adaptation to yield positive results. However, there is also significant variation within these averages; as an example, Fig. 4 shows the same data for just one of the workloads from each class, on each scheduling alternative. Here we see that there can be significantly more variation in overall workload completion time between scheduler alternatives than the average picture shows, and in particular we see that DRF performed best for the coarse-grained workload variant within this particular experiment, versus FIFO being best for this workload on average.

We next examine the detailed execution of each workload, which reveals a much more complex story than the average results suggest – and shows that different scheduling alternatives appear to be better at different *phases* of a single workload. We next examine these detailed execution results.

For each class of workload we present one detailed execution analysis as an exemplar of that class. Our coarse-grained workload class example is shown in Fig. 5. This graph shows each individual job in the workload on the x-axis, in the order in which they were submitted to the DEF (with each job submitted 300ms apart). We overlay the timing results from each scheduling alternative on the same graph, even though each scheduling alternative was executed on this workload in isolation. The graph therefore shows how quickly each scheduler alternative completes each successive job in its workload, relative to the other scheduling alternatives, which helps to explain the above variation in average performance.

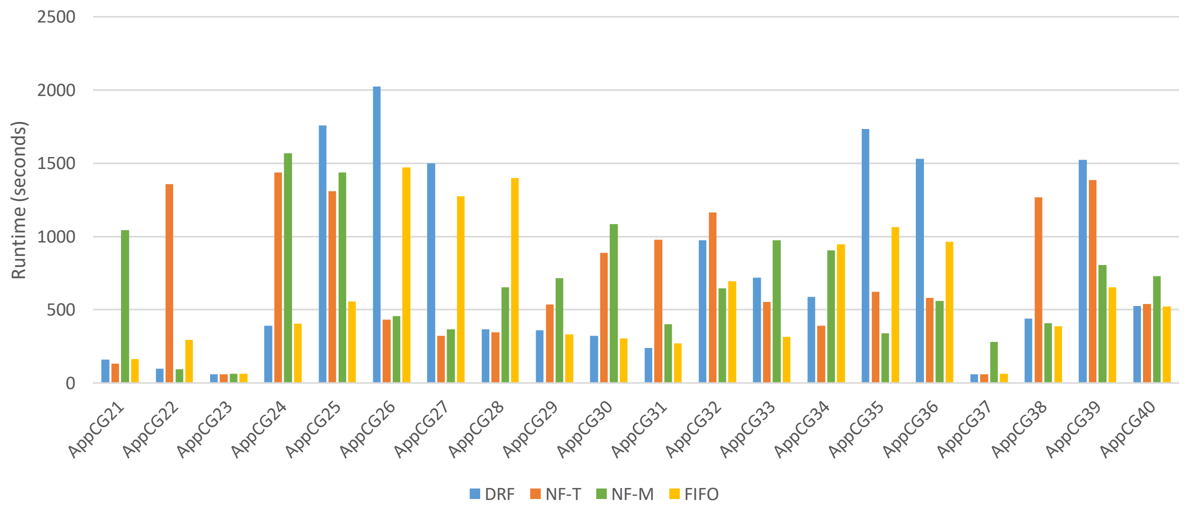

Fig. 5. Individual Job runtime for workload CG-W3

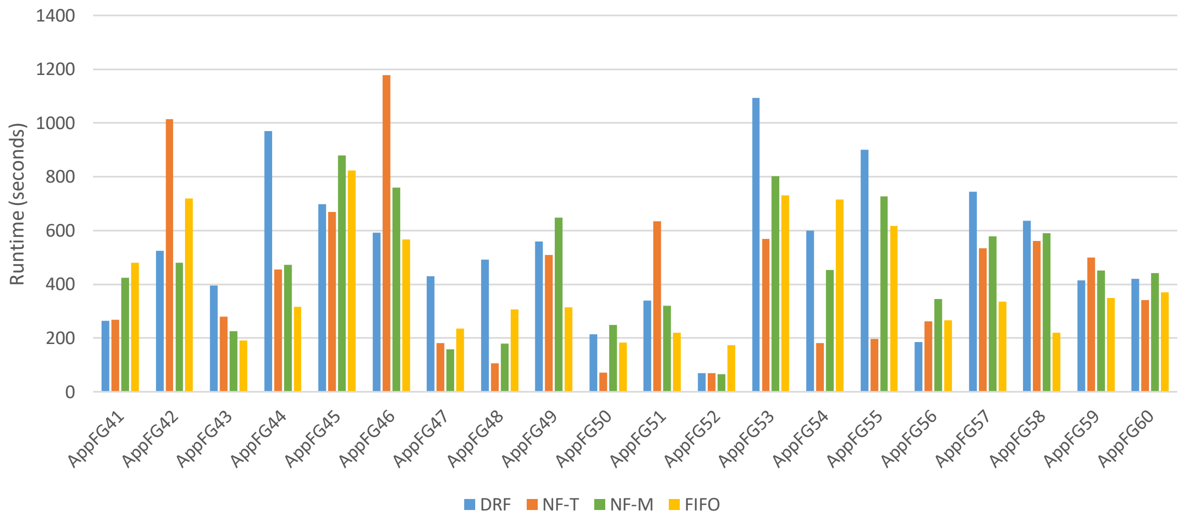

Fig. 6. Individual Job runtime for workload FG-W3

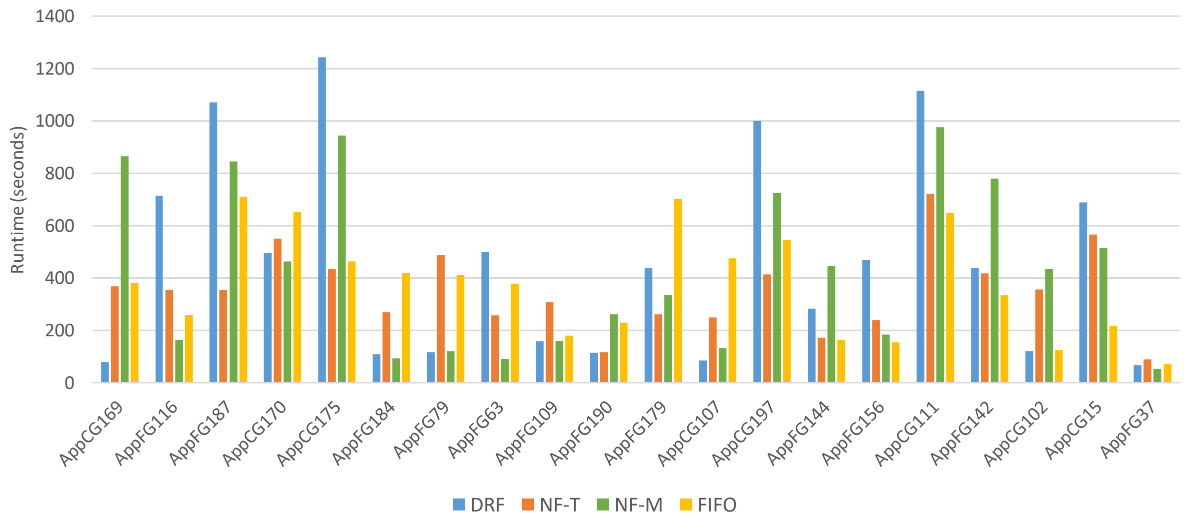

Fig. 7. Individual Job runtime for workload MG-W3

Examining our coarse-grained workload results on Fig. 5 in detail, we make three observations. First, out of the 20 jobs in the workload, FIFO completes 7 of them faster than any other scheduler alternative and is close to the fastest in another 6 jobs. This helps to explain why FIFO is better on average for the coarse-grained workload class. Second, there are therefore 7 jobs for which FIFO performs notably worse than alternative schedulers, and in the majority of these cases it performs *much* worse – for example in AppCG26 and AppGC35. Third, it is notable that FIFO performs worse across *successive* jobs in the workload for a sustained period of time, seen between AppCG26–28, and AppCG34–36. In both of these time periods, the NF-T scheduler is consistently a better choice, indicating a potential detectable change point at which mid-workload adaptation could yield positive results. However, it is also worth noting that the higher performance of NF-T at these points in the workload is at least partly dependent on the previous decisions made by NF-T in earlier parts of the workload; disentangling these temporal dependencies may be non-trivial.

The detailed view of our fine-grained workload is shown in Fig. 6, for which our average data suggests that FIFO is also the better choice overall. Our detailed view for this specific fine-grained workload demonstrates that FIFO was better than other scheduling alternatives for only 8 out of the 20 jobs in the workload, with the NF-T scheduling alternative being better than FIFO for 7 out of the 20 jobs in the workload. Despite this detailed result, FIFO still completed the overall workload marginally faster than NF-T. This lends further credence to the potential gain from adapting between scheduler alternatives in the middle of a set of processing jobs.

Finally, an example detailed view of our mixed-grain workload is shown in Fig. 7, for which our average data suggests that NF-M is the better scheduler choice overall for the fastest workload completion time. Mixed-grain workloads are generally the most difficult for any one scheduler to deal with, with expected average workload granularity being one of the key targets for specialist schedulers. Our mixed workload uses a mixture of long-running jobs and short-running jobs, again submitted at the same frequency of 300ms between job submissions. The x-axis of Fig. 7 notes which jobs are long or short, using *CG* to indicate long ones and *FG* to indicate short ones. In this particular example, NF-T (not NF-M) completes the overall workload fastest; however, NF-T is fastest at completing just 6 out of 20 of the specific jobs in this workload compared to other schedulers. Despite this low level of specific job performance across the workload, no other scheduler has a higher number of specific jobs at which it was fastest. As with the above experiments, we again see significant periods during the workload over which other scheduler alternatives were better than NF-T, with NF-M in particular being better during the sustained set of fine-grained jobs FG-184–109. This in particular may suggest sustained dominant job granularity as a useful discriminator for self-adaptation between scheduler alternatives in mixed DEF workloads.

C. Discussion

The above experiments highlight both the potential value of adapting at runtime between different DEF scheduler architectures, in terms of significantly faster job completion times, but also show the difficulty in understanding how to efficiently schedule based on a workload alone. Our data indicates two possible alternatives in detecting and potentially learning which scheduler to use at runtime. One point is at the resource manager, which observes the submission of new jobs to the system – and has some ability to determine the data size of those jobs and potential properties of the computation profile that a job may have (e.g., memory or CPU-intensive). This high-level detection may indicate whether the current mixture of jobs tends towards something uniform, or is a diverse mixture, which may inform overall scheduling policy decisions. The second possible detection point is in the detailed execution performance of jobs in the cluster; here we see multiple instances in which a particular scheduler gets progressively worse at job completion time despite the fact that the job characteristics are relatively constant over time; this may act as a second point of feedback to adjust the scheduling policy to an alternative. In either case, a self-adaptive controller is complicated by the fact that prior scheduling decisions have a 'long tail' into the performance of a newly-selected scheduler, such that it takes time for the set of currently-scheduled jobs to be such that each job was scheduled by the currently-chosen scheduler. This effect may also cause a form of hysteresis whereby the mixture of scheduling decisions made by two different schedulers is actually the ideal.

VI. Conclusion

The increasing amount of data available today, combined with large volumes of readily-available compute power, has led to distributed data execution frameworks becoming a key feature of global computation. The scheduling policies used by these DEF frameworks have a significant impact on the completion time of data processing tasks, and on the amount of energy consumed by the underlying compute resources. In this paper we have presented a platform with which to study self-adaptive behaviours in DEFs, with our emergent DEF system.

Using this system we have also presented an initial study of compositional adaptation for scheduling policies in DEFs. Using a set of synthetic workloads generated to represent different points in the DEF design space, while also providing a high level of reproducibility, this study demonstrates that there are notably different ideal scheduler policies for different kinds of DEF workload. When we examine the details of these workload executions, we also see clear evidence that a particular scheduler which is best on average experiences significant periods during a workload in which it performs very poorly compared to alternative schedulers.

In future work we will examine self-adaptive control strategies which are able to take advantage of these divergence points to learn when to adapt to each scheduler alternative. In addition, we will examine the complexities of multi-tier scheduling, coordinating the adaption of task and resource manager scheduling policies concurrently. We will also examine changes to overall scheduler architecture, to move between centralised and decentralised approaches, and how this more extensive kind of adaptation affects scheduling performance.

VII. Acknowledgements

This work was partly supported by the UK Leverhulme Trust via the *Self-Aware Datacentre* project, grant RPG-2017-166.

References

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[2] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 270–288.

[3] A. Sergeev and M. D. Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *ArXiv*, vol. abs/1802.05799, 2018.

[4] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 147–156.

[5] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 261–276.

[6] H. Tariq, H. Al-Sahaf, and I. Welch, "Modelling and prediction of resource utilization of hadoop clusters: A machine learning approach," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, ser. UCC'19. New York, NY, USA: Association for Computing Machinery, 2019, p. 93–100.

[7] M. E. Haque, Y. He, S. Elnikety, T. D. Nguyen, R. Bianchini, and K. S. McKinley, "Exploiting heterogeneity for tail latency and energy efficiency," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 625–638.

[8] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan *et al.*, "Ray: A distributed framework for emerging AI applications," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 561–577.

[9] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga, "Mercury: Hybrid centralized and distributed scheduling in large shared clusters," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 485–497.

[10] Y. Xia, R. Ren, H. Cai, A. V. Vasilakos, and Z. Lv, "Daphne: A flexible and hybrid scheduling framework in multi-tenant clusters," *IEEE Transactions on Network and Service Management*, vol. 15, no. 1, pp. 330–343, 2018.

[11] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel, "Hawk: Hybrid Datacenter Scheduling," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, 2015, pp. 499–510.

[12] J. Polo, C. Castillo, D. Carrera, Y. Becerra, I. Whalley, M. Steinder, J. Torres, and E. Ayguadé, "Resource-aware Adaptive Scheduling for MapReduce Clusters," Tech. Rep.

[13] Z. Niu, S. Tang, and B. He, "An adaptive efficiency-fairness meta-scheduler for data-intensive computing," *IEEE Transactions on Services Computing*, vol. 12, no. 6, pp. 865–879, 2019.

[14] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:16.

[15] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," *SIGPLAN Not.*, vol. 48, no. 4, pp. 77–88, Mar. 2013.

[16] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[17] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia, "Structured streaming: A declarative api for real-time applications in apache spark," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: ACM, 2018, pp. 601–613.

[18] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 69–84.

[19] I. Gog, M. Schwarzkopf, A. Gleave, R. N. Watson, and S. Hand, "Firmament: Fast, centralized cluster scheduling at scale," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 99–115.

[20] B. Porter and R. Rodrigues Filho, "Losing control: The case for emergent software systems using autonomous assembly, perception and learning," in *International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE, September 2016, pp. 40–49.

[21] B. Porter and R. Rodrigues Filho, "Distributed emergent software: Assembling, perceiving and learning systems at scale," in *International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE, June 2019.

[22] B. Porter, M. Grieves, R. Rodrigues Filho, and D. Leslie, "Rex: A development platform and online learning approach for runtime emergent software systems," in *Symposium on Operating Systems Design and Implementation*. USENIX, November 2016, pp. 333–348.

[23] B. Porter, "Runtime modularity in complex structures: A component model for fine grained runtime adaptation," in *Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering*, ser. CBSE '14. New York, NY, USA: ACM, 2014, pp. 29–34.

[24] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, ser. OSDI'04. USA: USENIX Association, 2004, p. 10.

[25] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 265–278.

[26] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. USA: USENIX Association, 2011, p. 323–336.

[27] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: scalable and coordinated scheduling for cloud-scale computing," in *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2014, pp. 285–300.