Artifacts Available
acm
V1.1

DayDream: Executing Dynamic Scientific Workflows on Serverless Platforms with Hot Starts

Rohan Basu Roy
Northeastern University
Boston, MA, USA
rohanbasuroy@gmail.com

Tirthak Patel
Northeastern University
Boston, MA, USA
patel.ti@northeastern.edu

Devesh Tiwari
Northeastern University
Boston, MA, USA
devesh.dtiwari@gmail.com

SC22: International Conference for High Performance Computing, Networking, Storage and Analysis | 978-1-6654-5444-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/SC41404.2022.00027

***Abstract*—HPC applications are increasingly being designed as dynamic workflows for the ease of development and scaling. This work demonstrates how the serverless computing model can be leveraged for efficient execution of complex, real-world scientific workflows, although serverless computing was not originally designed for executing scientific workflows. This work characterizes, quantifies, and improves the execution of three real-world, complex, dynamic scientific workflows: ExaFEL (workflow for investigating the molecular structures via X-Ray diffraction), Cosmoscout-VR (workflow for large scale virtual reality simulation), and Core Cosmology Library (a cosmology workflow for investigating dark matter). The proposed technique, DayDream, employs the hot start mechanism for warming up the components of the workflows by decoupling the runtime environment from the component function code to mitigate cold start overhead. DayDream optimizes the service time and service cost jointly to reduce the service time by 45% and service cost by 23% over the state-of-the-art HPC workload manager.**

***Index Terms*—Serverless Computing, HPC Workflows, Cloud Computing**

I. Introduction

Background and Problem. Traditionally, scientific applications have been monolithic and executed on on premise HPC cluster or virtual machine (VM) based clusters on the cloud platforms. However, these applications are usually complex, and often with extensive data and phase dependency among various periods of their execution. For the ease of development, scaling, and updating these applications, implementing scientific applications as workflows in the form of directed acyclic graphs (DAG) is becoming increasingly popular.

However, resource utilization of HPC DAGs varies significantly over their execution. Hence, executing them on traditional HPC clusters with a fixed amount of computing resources leads to over and/or under-provisioning of resources. Hence, HPC DAGs need an execution environment that can provide elastic provisioning of resources [7]. Serverless computing, a new evolving trend in cloud computing, provides the opportunity due to its auto-scaling of resources and pay-as-you-go billing model. *This work demonstrates how serverless computing model, an emerging computing model on cloud computing platforms, can be utilized for efficient execution of complex, real-world scientific workflows, although the serverless computing model were not originally designed for executing scientific workflows.* However, this opportunity also poses unique challenges.

Challenges and Opportunities. Unlike HPC and VM clusters, serverless resources are not pre-allocated and initiated before execution [82, 49]. In serverless, functions are spawned and booted up only when components of a DAG are invoked to be executed. This causes start up delays of serverless function instances, also known as cold start. Cold start time can be a significant fraction of the execution time of components (often 25% to 60%) [66, 82, 81]. This can potentially increase the execution time of workflows, ripping off the benefits of serverless execution.

The traditional approach to mitigating cold start is to predict the invocation of serverless functions and warm up serverless function instances with the function contents and metadata in the memory of the instance, before the function is invoked [66, 70, 52]. This incurs a keep alive cost, but helps to boot up instances faster and start executing the function sooner by undergoing a warm start [43]. A major research challenge in serverless is to effectively predict the invocations of functions and accordingly warm start instances to avoid cold starts while incurring minimum keep alive cost [63, 24]. State-of-the-art serverless works have proposed function prediction schemes for regular production cloud workloads with bursty traffic [66, 22, 24]. Typically, the invocation of the enterprise data center workloads follows a time-series-based temporal pattern, and hence, as expected, warming up of serverless function instances has been shown to be effective [66, 85, 82].

Unfortunately, our experimental characterization reveals HPC DAGs, which are significantly mostly dynamic and irregular in nature [50, 38, 64], vary their execution path for different inputs and operations (Sec. III). The components invoked in their execution path do not follow a time-series friendly pattern for prediction, which makes their invocation challenging to predict and warm up. Therefore, as expected, applying the state-of-the-art serverless technique results in sub-optimal solution for real-world complex scientific workflows since they were designed for different types of workloads and usecase (Sec. III and V).

Contributions. *First, DayDream specifically characterizes and quantifies the execution of three real-world, complex scientific workflows*: **ExaFEL** (used for understanding molecular structures via X-Ray diffraction [51]), **Cosmoscout-VR** (used for exploring planetary data sets and performing large scale virtual reality simulation [27]), and **Core Cosmology**

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00 ©2022 IEEE
Authorized licensed use limited to: Auburn University. Downloaded on February 09,2024 at 19:21:07 UTC from IEEE Xplore. Restrictions apply.

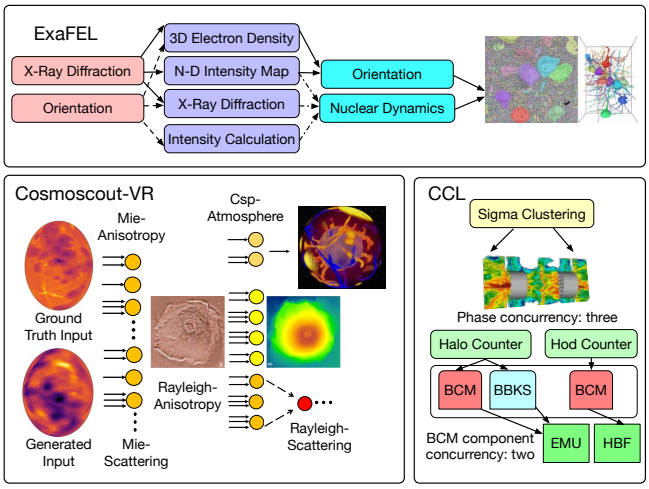

Fig. 1. Dynamic DAGs of ExaFEL, Cosmoscout-VR, and CCL.

Library (a popular cosmology workflow for understanding dark matter [16]).

Second, DayDream is the first work to demonstrate the challenges and opportunities present in leveraging the serverless computing model for efficient execution of real-world dynamic scientific workflows. In particular, this work demonstrates that various execution characteristics are appropriately suited for leveraging elastic serverless platforms for reducing both execution time and cost.

Third, DayDream demonstrates that, unfortunately, existing serverless techniques designed for enterprise workloads are not effective for dynamic scientific workflows. To mitigate this challenge, DayDream employs the hot start mechanism for warming up the components of the workflows – hot start decouples the runtime environment from the component function code. This decoupling allows DayDream the flexibility to only predict the total number of components, instead of the number of components of each type – this design element reduces the cold start overhead significantly and minimizes the keep alive cost. However, extracting the full potential of this design element requires a careful design of the system. In Sec. III and Sec. IV, we describe the design and implementation of DayDream and demonstrate how does DayDream formulate and solve this as an optimization problem.

DayDream's extensive evaluation across multiple serverless cloud providers confirms that DayDream is effective at improving workflow execution time by 45% and cost by 23% over the state-of-the-art HPC workflow manager. DayDream is open-sourced at `https://zenodo.org/record/6941485` for reproducibility and improvements.

II. Background and Definitions

Scientific Workflows. Real-world scientific workflows are dynamic in nature, *i.e.*, for different *inputs* and *operations*, the path of execution down the DAG changes [50, 38, 64, 56]. In this paper, we study three widely used scientific workflows (referred to as HPC DAGs) critical for many scientific missions: (1) ExaFEL [51] belongs to the exascale computing project (ECP) and studies molecular structures via X-Ray diffraction. (2) Cosmoscout-VR [27] is a large-scale virtual reality simulation of the universe. (3) Core Cosmology Library (CCL) [16] studies various properties of dark matter. Before discussing the design of DayDream, next, we define some basic terms related to executing DAGs.

Definitions. Fig. 1 shows some portions of the dynamic HPC DAGs of the three scientific workflows (complete DAGs not shown for brevity). With the help of Fig. 1 we define different parts of a workflow. An unique **run** of a DAG refers to its execution with an unique *operation*, *input* pair. A **component** is the smallest unit of execution in a workflow. For example, the second phase of ExaFEL has the following components: `3D Electron Density`, `N-D Intensity Map`, `X-Ray Diffraction`, and `Intensity Calculation`. Multiple components which can run in parallel, without any data or state dependency between them, form a **phase**. In a phase, a component can have multiple running instances, the sum of which is **component concurrency**. For CCL, the component `BCM` has a concurrency of 2. The *sum of all components concurrences in a phase* constitutes the **phase concurrency**. For example, the phase of CCL indicated with dashed border has a phase concurrency of 3 (2 `BCM` component instances and 1 `BBKS` component). Different phases in a DAG are indexed by a **phase index**.

The execution time of each component is the **component execution time**. Since all the components in a phase start simultaneously and run in parallel, the execution time of the component which runs for the longest in a phase is the **phase execution time**. The sum of the phase execution time of all the phases is the end-to-end execution time of the DAG, also called the **service time**. When executing a DAG in cloud infrastructure, all resources allocated for the execution incur a cost to the cloud provider. The total cost incurred for the execution of an HPC DAG is the **service cost**.

III. DayDream: Motivational Opportunities, Design Challenges, and Technique Description

In a dynamic DAG, the components executed in each phase, the phase concurrency, and also the number of phases change from one run to another. For example from Fig. 1, if the operation `X-Ray Diffraction` is chosen in ExaFEL, the components `3D Electron Density`, `N-D Intensity Map` and `X-Ray Diffraction` are executed in the second phase. However, if the operation of the run is `Orientation`, `Intensity Calculation` is chosen instead of `N-D Intensity Map`. Similarly, in Cosmoscout-VR, when the input is ground truth, the components executed are `Mie-Anisotropy`, `Rayleigh-Anisotropy`, and `CSP-Atmosphere`. However, when a generated input is used, the components executed are `Mie-Anisotropy`, `Rayleigh Anisotropy`, `Rayleigh Scattering`, and then, unlike the ground truth input case, more phases continue till the DAG generates the final output.

Authorized licensed use limited to: Auburn University. Downloaded on February 09,2024 at 19:21:07 UTC from IEEE Xplore. Restrictions apply.

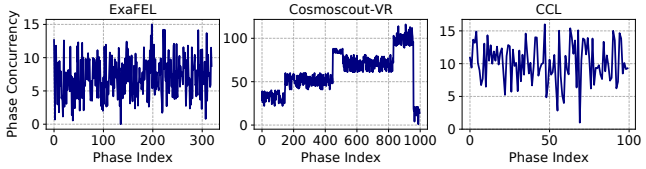

Fig. 2. The degree of parallelism in scientific workflows varies over execution phases (i.e., the number of concurrent components across phases varies).

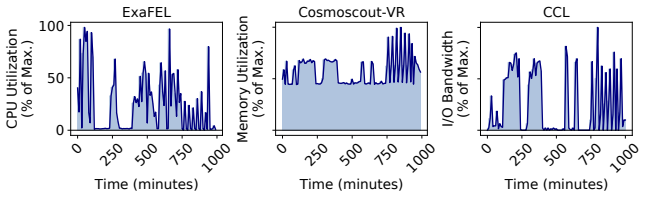

Fig. 3. The resource consumption usage (e.g., CPU utilization, memory utilization, and I/O bandwidth) also varies over time during the execution of scientific workflows.

The execution path of a dynamic DAG is unknown prior to a run and that is why component scheduling decisions are hard to make. The DAG of a workflow is similar to a tree-like data structure with multiple possible paths of execution at each joint, only one of which is taken during a particular run. The path that will be taken is unknown apriori and is discovered only during execution.

Observation I. *Real-world scientific workflows' execution characteristics are diverse: the type of components, number of components in each phase, and number of phases heavily depend on the operation and input used to invoke the workflow.*

Digging a bit deeper into the execution characteristics, from Fig. 2, we observe that the phase concurrency varies significantly over the execution. Traditionally, these workflows run on HPC or virtual machine (VM) based clusters with fixed computational resources. With this high level of variance in parallelism, the traditional methods of execution can lead to over or under-provisioning of resources. This effect is also visible from Fig. 3, where compute, I/O, and memory requirements of these HPC DAGs are observed to vary significantly. Maintaining a constant amount of computational resources can lead to resource wastage, leading to capital loss due to over-provisioning. It can also lead to resource contention and slowdown of execution due to contention during periods of high resource requirements. More importantly, both of these cases can occur within a run at different points of time. This is why, rather than having fixed resources, elastic provisioning or auto-scaling of resources makes it naturally suitable for DAG execution – as in, for example, serverless computing.

Serverless computing, or *function-as-a-service* (FaaS) is an increasingly popular trend in cloud computing where applications run in **serverless function instances**. These serverless function instances are microVMs. In some ways they are similar to VMs, however, they individually have significantly less computational resources and very low start-up latency.

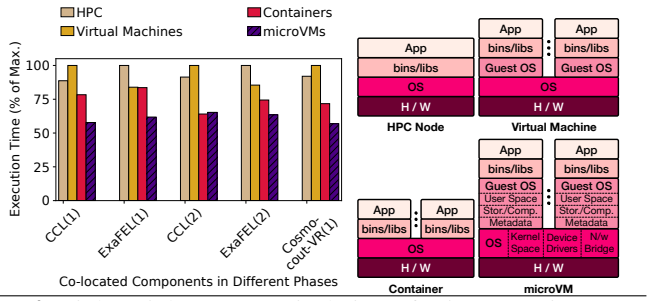

Fig. 4. Lightweight resource isolation of microVMs in serverless reduces the execution time of co-located components (the numbers inside brackets denote the phase index).

Depending upon the load or requirements of an application, a user can elastically spawn up serverless function instances. The fine-grained resource allocation for each serverless function instance makes them ideal for elastic auto-scaling. It helps applications running in serverless to avoid over and under-provisioning of resources. Again in serverless computing, users pay only for the amount of resources they use (pay-as-you-go billing model), unlike VM-based cloud computing where users pay a flat rate regardless of the computational load of their application. Since HPC DAGs also have bursty resource utilization, we investigate the efficacy of serverless computing and expand its capability.

From the resource utilization point of view, serverless makes sense for HPC DAG execution. But typically individual serverless function instances have very low resources than a full HPC or VM-based cluster. Then, *how will the execution time be affected in a serverless execution?*

From Fig. 4 we see that even execution time is lower in serverless. This is because a phase in an HPC DAG has multiple components. These are co-located in an HPC cluster execution, causing resource contention and slowdown. VMs in traditional cloud computing have resource isolation but have high start up latency and overhead of execution as a complete guest OS, along with the software stack needs to be initiated during invocation of a VM. Containers, though resource isolated, share the operating system. This makes them prone to contention. MicroVMs are resource isolated as they individually have separate user space from the guest OS, but they share the kernel space, device drivers, and the network bridge with the host OS. The CPU steal time (which is a measure of resource contention) of individual components is less in serverless MiroVMs by 18% and 11% compared to HPC cluster and containerized execution, respectively. The start up time of components is less in microVMs by 29% compared to VMs.

The lightweight resource isolation of microVMs hits the sweet spot between avoiding contention via resource isolation and low start up latency compared to VMs. This is why, when different components in a phase run in different serverless function instances, the phase execution time is lowest in serverless microVMs. Note that, in Fig. 4, the HPC cluster, VMs, containers, and microVMs have an aggregate of same computational resources. The strict resource isolation among

Authorized licensed use limited to: Auburn University. Downloaded on February 09,2024 at 19:21:07 UTC from IEEE Xplore. Restrictions apply.

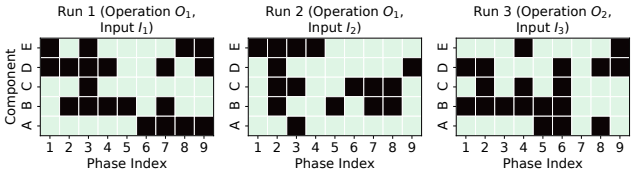

Fig. 5. The invocation of components among separate runs of a scientific workflow do not exhibit an easy-to-identify pattern (the black boxes denote the components invoked in a phase).

components of a phase and low start up latency in microVMs reduces the phase execution time.

Observation II. *Scientific workflows have a varying degree of parallelism and resource consumption over their execution path – making them suitable for serverless computing model.*

However, serverless computing poses its own challenges. In a reserved VM or HPC cluster, all components and their metadata are already present in the memory and when a phase starts, the components can directly start execution. Typically, in practice, all the components are loaded in the memory before execution starts on an HPC cluster instead of on-demand fetch to reduce execution time, but increases resource wastage. However, in serverless, this is not the case. The serverless function instances are invoked and spawned up only when the component execution request is made to reduce resource utilization inefficiency.

Hence, unlike traditional HPC or VMs, the language runtime and application with its metadata need to be loaded into the memory of the function instance before it can start execution. The application and its runtime reside in remote storage and the serverless function has to fetch it over the network. This causes a start up latency, called the **cold start time**. Since each instance runs just one component, the cold start time can be a large fraction of the actual instance execution time. If all the components undergo cold starts, a DAG execution time increases greatly.

A contemporary approach to avoid cold starts is to **keep alive** applications in the memory, *i.e.,* predict the invocation of a function and spawn up serverless function instances with the function metadata and runtime loaded to the memory of the instance before it is actually invoked. Upon invocation, the function undergoes a **warm start**. Keeping functions alive incurs computational resources from the cloud provider's side, which is manifested as the **keep alive cost**. *An effective component invocation prediction mechanism is required to minimize both keep alive cost and avoid cold starts for serverless computing to make it beneficial for dynamic workflows.*

If the components exhibit some pattern in their invocation, then they can benefit from a prediction scheme for warming up function instances. To test this, Fig. 5 shows the occurrence of some components across different phases of Cosmoscout-VR. We note two things – first, within one run, the invocation of a particular component or a group of components does not exhibit any identifiable pattern. Second, the invocation pattern of a component also varies among different runs. Even the

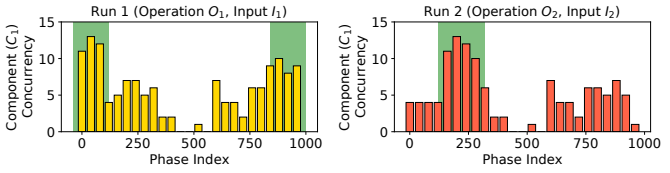

Fig. 6. Component concurrency is challenging to predict over the execution phases of a scientific workflow (the green shaded areas denote the phases when it is most useful to warm up the component).

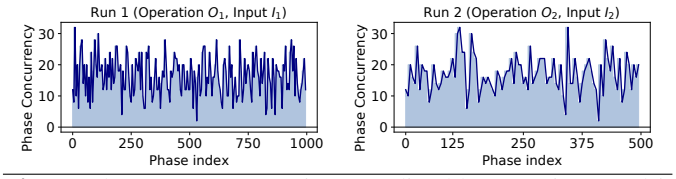

Fig. 7. Phase concurrency is unpredictable over time, and it varies from one run to another of a scientific workflow.

component concurrency of a particular component varies from one run to another (Fig. 6). Fig. 6 also shows the phases in which warming up a component will be most beneficial varies from run to run (shaded regions with the highest component currency).

Finally, Fig. 7 shows that the phase concurrency (*i.e.*, the summation of concurrency of all the components in a phase) does not provide any identifiable temporal pattern. The mean normalized χ^2 error for temporal component concurrency is 0.93, 0.92, 0.94, 0.89, and 0.93 for second-order polynomial fit, third order polynomial fit, fourth order polynomial fit, sinusoidal fit, and logarithmic fit, respectively. Similarly, the temporal phase concurrency has a normalized χ^2 error of 0.88, 0.83, 0.82, 0.81, and 0.88 for second-order polynomial fit, third order polynomial fit, fourth order polynomial fit, sinusoidal fit, and logarithmic fit, respectively. The high values of these test results support that component and phase concurrency do not follow any common mathematical relation.

The lack of predictive patterns in component invocation, component concurrency, and phase concurrency makes it challenging to leverage state-of-the-art serverless techniques for warming up the components. To demonstrate this quantitatively, we apply state-of-the-art production serverless workload prediction scheme (ARIMA-based time series prediction from Serverless in the Wild work [66]) to determine the phase concurrency. Fig. 8 shows that there is a large deviation between the actual phase concurrency and the one predicted by Wild (ARIMA), resulting in a large error (more than 50 components). This is because Wild applies time-series-based regression to predict the current sample based on historical samples. While Wild is effective for the original purpose and workloads it was designed for, we need new methods to make serverless computing attractive for HPC DAGs.

We found that although phase concurrency does not exhibit temporal patterns, a closer observation reveals that these samples are drawn from a predictable distribution. To support this empirically, in Fig. 9, we plot the histogram of the frequency of phase concurrency; the samples follow a predictable shape.

Authorized licensed use limited to: Auburn University. Downloaded on February 09,2024 at 19:21:07 UTC from IEEE Xplore. Restrictions apply.

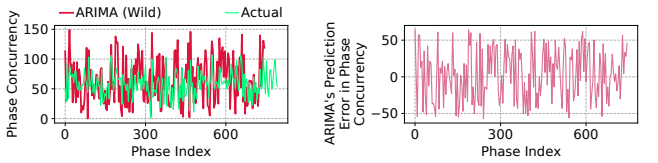

Fig. 8. Traditional methods of phase concurrency prediction are often not suitable for HPC dynamic DAGs.

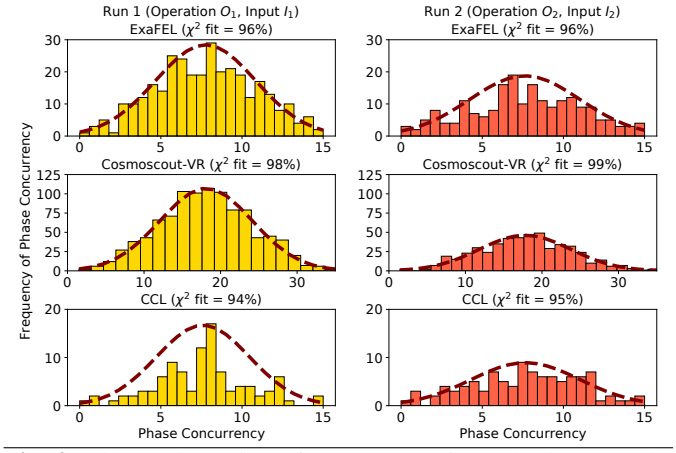

Fig. 9. The total number of components in each phase can be fitted to a statistical distribution: for example, the fitted Weibull distribution remains similar across different runs (alpha = 6 and beta = 3 for ExaFEL, alpha = 10 and beta = 3.2 for Cosmoscout-VR, alpha = 10 and beta = 6 for CCL).

More importantly, this distribution remains the same for a workflow from one run to another. While different statistical distributions can be fitted, we performed Weibull distribution fitting for demonstration and its simplicity of use due to shape and scale parameter.

We note that the number of phases varies from one run to another (Fig. 9), so the height of the distribution changes, but other parameters remain similar. Empirically, we observed this behavior to be present in different workflows and runs – potentially stemming from the computational steering and workflow parallelism behavior embedded in these workflows [19, 44, 57]. DayDream exploits this idea to make a best-effort prediction of phase concurrency. However, as we show later in design and evaluation, DayDream's effectiveness is not sensitive to phase concurrency exhibiting a specific distribution consistency since DayDream can learn the distribution dynamically.

However, resolving the predictability of phase concurrency is not sufficient for making serverless effective for dynamic HPC DAGs. *This is because each phase may have different types of components and each component may have different concurrency, hence, one can still determine which particular component functions to warm up and how many of each type.* Hence, the traditional idea of a warm start is not useful in the context of dynamic HPC DAGs. This is because even if one correctly predicts the phase concurrency, but warms up the wrong components, it is not useful for mitigating cold start overhead. Also, a wasteful keep alive cost is incurred for

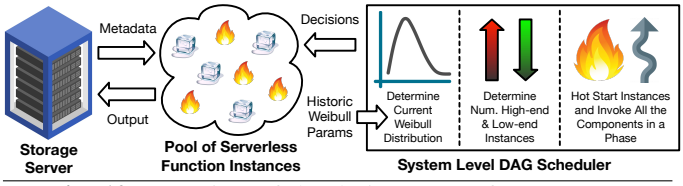

Fig. 10. Overview of the design steps of DayDream.

warming up the wrong components.

To mitigate this challenge, DayDream employs the idea of **hot start**. In this approach, only the OS and possible language runtimes are loaded in the memory of serverless function instances. These instances are booted up and kept alive. The component contents and their metadata are not loaded. Thus, when the components are invoked, hot started function instances can run any component, by loading only the component and its metadata to the instance after invocation. In this way DayDream decouples hot started function instances from component dependency, unlike traditional warm starts.

Observation III. *Unfortunately, employing a serverless computing model as how it has been designed for traditional enterprise workloads, is not suitable for scientific workflows and is significantly sub-optimal. However, there exists an opportunity scope (by decoupling component function code and runtime environment during the warm up stage) to exploit the serverless computing model for scientific workflows.*

DayDream: Design. Motivated by the above observations, we design DayDream. DayDream hot starts and schedules components of an HPC dynamic DAG on a serverless computing platform. Fig. 10 represents the overview of the approach; different sections will be described one by one as we discuss the design.

How does DayDream perform hot starts? Recall that DayDream needs to employ hot start because existing state-of-the-art serverless techniques, using warm start, are not effective for dynamic HPC DAGs (Fig. 8) and this will be further demonstrated in the evaluation (Sec. V). Under the hot start mechanism, DayDream only loads the operating system and language runtime in the memory of a serverless function instance (and not the actual component and its metadata). This makes a hot started serverless function instance suitable for the execution of any component, unlike warm started instances.

Now, the question is *how does DayDream know how many serverless function instances to hot start?* For this, DayDream needs to predict the phase concurrency. As observed earlier, if we plot the histogram of the frequency of the phase concurrency, we observe an identifiable trend – the samples can be fitted to a statistical distribution (Weibull distribution for demonstration in Fig. 9). This motivates the design of DayDream. During the first run of an HPC DAG, DayDream recognizes and fits the Weibull distribution parameters for the frequency of phase concurrency. In all subsequent runs, for each phase, it starts generating sample numbers following this distribution. Typically, HPC workflows are executed multiple times as separate runs with

Authorized licensed use limited to: Auburn University. Downloaded on February 09,2024 at 19:21:07 UTC from IEEE Xplore. Restrictions apply.

different inputs and operations to perform [31]. These generated samples indicate DayDream how many serverless function instances to hot start for each phase. By performing χ^2 histogram fit test on the frequency of phase concurrency, we found that the Weibull distribution results in the best fit (fit percentage shown in Fig. 9). Thus, DayDream uses Weibull distribution to model the data.

DayDream hot starts instances of the next phase when half of the components of the previous phase have finished execution. In our evaluation, we quantitatively demonstrate that distribution-fitting generation is more effective than time-series prediction and results in less error even when the distribution of phase concurrency changes across runs.

It has been mathematically shown that the Weibull distribution provides more flexibility in data modeling than other distributions like Gaussian, Poisson, etc [55]. Since, the phase concurrency distributions of workflows can be significantly different, DayDream uses the Weibull distribution to model it dynamically. Also, dynamic HPC DAGs tend to invoke specific components based on the operation of a run which makes phase concurrency follow a probability distribution.

How does DayDream react when the distribution of phase concurrency changes? Before the start of a run, although DayDream knows the parameters of this distribution, the distribution can change slightly for different runs. In each run, DayDream starts generating the number of hot starts following the parameters of one single historic run's distribution. These parameters are updated dynamically based on the current run's statistics.

For example, let DayDream start a run with the shape and scale parameters of Weibull distribution as β_h and α_h, respectively based on the historic run distribution data. It starts generating samples for the number of instances to hot start ($f(p)$) for a phase p, based on the following distribution:

$$f(p) = \frac{\beta_h}{\alpha_h} \times (\frac{p}{\alpha_h})^{\beta_h - 1} \times e^{-(\frac{p}{\alpha_h})^{\beta_h}} \quad (1)$$

Now, as the phases progress in the current run, after every phase interval p_{int}, the scale and shape parameters of the distribution of frequency of phase concurrency in the current run are re-calculated. This is performed using the χ^2 test, which is a widely used method to determine the goodness of fit of data to a probability distribution [68]. So, after n of such phase intervals, DayDream calculates the shape β_n and scale α_n of Weibull distribution as follows:

$$\beta_n^2, \alpha_n^2 = \underset{\alpha_i \in A, \beta_i \in B}{\operatorname{argmin}} \sum_{i=1}^{n * p_{\text{int}}} \frac{(O^i - E^i_{\alpha_i, \beta_i})^2}{E^i_{\alpha_i, \beta_i}} \quad (2)$$

Here, n phase intervals includes $n * p_{\text{int}}$ phases and hence the summation goes from 1 to $n * p_{\text{int}}$. O^i is the observed values of phase concurrency histogram and $E^i_{\alpha_i, \beta_i}$ is the expected value or the histogram of samples generated from a Weibull distribution with parameters α_i and β_i, following Eq. 1. The optimization solved in Eq. 2 generates the values of scale (out of all possible values in A) and shape (out of all possible values in B) parameters of Weibull distribution which best fits the histogram of the frequency of phase concurrency, seen so far in the current run. However, not only the present best fit values of the Weibull parameters are important, but also the historic values from the first run and the values of these parameters generated in the previous phase intervals are also important to make a robust choice. DayDream calculates the mean values of these Weibull parameters to estimate the optimal parameters (β_n^{opt}, α_n^{opt}) in the current phase interval n.

$$\beta_n^{opt}, \alpha_n^{opt} = \frac{\beta_h + \sum_{i=1}^{n} \beta_i}{n+1}, \frac{\alpha_h + \sum_{i=1}^{n} \alpha_i}{n+1}$$

This is how DayDream dynamically adjusts the parameters of the Weibull distribution of the current run, also taking into account their historic values. This process fine-tunes the values of these parameters. In our experimentation, we found that the optimal values of these parameters do not change by more than 10% among different runs of an HPC DAG. In our evaluation, we chose the phase interval p_{int} to be 25. However, our results are not sensitive to this, and do not change by more than $> 2\%$ when p_{int} is varied from 10 to 100 because the distribution of phase concurrency remains uniform throughout the execution of an HPC DAG. Note that, DayDream's hot starting of serverless function instances based on generating samples from a probability distribution does not precisely predict the phase concurrency in a given phase. It performs a best-effort job of hot starting instances in a given phase by predicting the overall distribution of phase concurrency over multiple phases.

Are all hot started serverless function instances homogeneous? We observed that the computational resource requirements of different components within the same HPC DAG vary significantly. Hence, having all serverless function instances with fixed resources might potentially increase resource wastage and cost. To tackle this challenge, we have two tiers of serverless function instances – high-end instances with higher compute, memory, and I/O resources and low-end function instances with lower resources. We define high-end friendly components as the ones which have more than 20% slowdown upon execution on a low-end serverless instance compared to a high-end one (our results vary by less than 3% when we vary the slowdown percentage from 5 to 30). As we run an HPC DAG for multiple runs, DayDream identifies the high-end-friendly components and keeps that information along with the component metadata. In our analysis of the HPC DAGs, we observed that the fraction of high-end-friendly components remains almost the same (vary by less than 5%) from one phase to the next (the variation over phases that are far from each other can be high). In each of the phases, DayDream identifies this fraction. Following this fraction, in the next phase, DayDream decides the number of low-end and high-end serverless function instances to hot-start. During our evaluation, we confirmed that while this technique is effective, the observed benefits are not sensitive to the chosen thresholds and remain effective over a large range because the distinction between

Authorized licensed use limited to: Auburn University. Downloaded on February 09,2024 at 19:21:07 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1 DayDream's hot start and component execution.

```
βh, αh ← historic values of shape and scale parameters
for each phase with index p do
    Calculate Weibull distribution parameters β_n^opt and α_n^opt
    Generate sample N_f(p) from the Weibull dist. f(p)
    F_p−1 ← high-end friendly fraction in phase p − 1
    Hot start N_f(p) F_p−1 high-end instances
    Hot start N_f(p)(1 − F_p−1) low-end instances
    Execute components on hot started instances
    if N_f(p) < the phase concurrency of p then
        Cold start remaining components
    Terminate instances with no running components
```

low-end and high-end friendly components is based on their execution time, and the placement choice is not varied over time based on the arrival probablity, unlike IceBreaker [63].

What happens when components are invoked in a phase? When a phase starts, DayDream loads the components and their metadata to the memory of the hot started function instances. It then executes the components on these serverless instances. Note that, DayDream identifies the high-end friendly components and accordingly executes them on high-end serverless function instances. If the actual phase concurrency is more than the number of hot started instances, the additional components undergo a cold start. DayDream executes these components on high-end function instances after loading the operating system, language runtime, and also the components with their metadata in the memory of the instances.

What optimization problem does DayDream solve? DayDream solves the joint optimization problem of minimizing *service time* and *service cost*. Service time is the end-to-end execution time of the HPC DAG (sum of the execution time of all the phases), and service cost comprises of keep alive cost of serverless function instances, execution cost of each component, and the wasted keep alive cost of unused hot started instances. If a phase with index p has p_c components, then the service time can be defined as follows:

$$S_t^{\gamma_{k_p},\delta_{k_p}} = \sum_{p=1}^{P_{\max}} \max_{1\le k_p\le p_c} [\gamma_{k_p}\delta_{k_p} t_{k_p}^{HE} + (1-\gamma_{k_p})\delta_{k_p} t_{k_p}^{LE} + \gamma_{k_p}(1-\delta_{k_p})(t_{k_p}^{CS} + t_{k_p}^{HE})]$$

Here, $t_{k_p}^{LE}$ and $t_{k_p}^{HE}$ are the execution time of the k^{th} component in phase p in high-end and low-end serverless function instances, respectively. $t_{k_p}^{CS}$ is the corresponding cold start time. γ_{k_p} is the *tier parameter*, which selects high-end or low-end serverless instance execution of the component. δ_{k_p} is the *hot start parameter*. When it is 1; the component undergoes a hot start, when it is 0; it undergoes a cold start. The *max* function calculates the execution time of phase p, where all components can run in parallel. $P_{\max}$ is the total number of phases in the DAG. Similarly, the service cost is as follows:

$$S_e^{\gamma_{k_p},\delta_{k_p}} = \sum_{p=1}^{P_{\max}} (-p_c + \sum \delta_{k_p}) KA^{LE} + \sum_{k_p=1}^{p_c} [\gamma_{k_p}\delta_{k_p}(e_{k_p}^{HE} + KA^{HE}) + (1-\gamma_{k_p})\delta_{k_p}(e_{k_p}^{LE} + KA^{LE}) + \gamma_{k_p}(1-\delta_{k_p}) e_{k_p}^{HE}]$$

Here, $e_{k_p}^{LE}$ and $e_{k_p}^{HE}$ are the execution cost in high-end and low-end serverless function instances, respectively. KA denotes the keep alive cost of hot started function instances. $(-p_c + \sum \delta_{k_p}) KA^{LE}$ is per phase wasted keep alive cost. Finally, DayDream determines the optimal values of γ_{k_p} and δ_{k_p} which jointly minimizes both service cost and service time.

$$\gamma_{k_p}^{opt}, \delta_{k_p}^{opt} = \underset{\gamma_{k_p}\in[0,1], \delta_{k_p}\in[0,1]}{\operatorname{argmin}} S_t^{\gamma_{k_p},\delta_{k_p}} + S_e^{\gamma_{k_p},\delta_{k_p}}$$

Here, both service time and cost are normalized to their maximum value among all the phases. DayDream gives equal weight in optimizing for service cost and time, but it can be easily modified by providing different weights to S_e and S_t. The overview of DayDream is summarized in Algorithm 1.

IV. Implementation and Methodology

DayDream is implemented using a three-level stack: (1) a system level DAG scheduler, (2) a serverless function instance pool, and (3) a back-end storage server. The system level DAG scheduler hot starts serverless function instances to execute components of a workflow. The application metadata and the output of the components are stored in the back-end server.

System level DAG scheduler. The back-end storage server initially contains the application, the DAG structure, the executables of its different components, the metadata along with the execution environment of the components, and the files that the application accesses during execution. Now, as the application execution starts, the DAG scheduler connects to the back-end storage server and instructs which components corresponding to the current phase should be executed. Also, during the execution of each phase, it predicts how many serverless function instances to hot start for the next phase based on the probability distribution. It also determines the tier of these function instances (low-end or high-end). The DAG scheduler uses *aws lambda invoke* for *synchronous* invocation of serverless instances. It uses *S3 REST API* over *https* to connect to the back-end storage server.

Serverless Function Instance Pool. Depending upon the instructions from the DAG scheduler, function instances for the execution of components of a phase are hot started when half of the components of the previous phase have finished execution. These hot started function instances form the pool. These serverless function instances are placed in microVMs. All the language runtimes used by the different components of the DAG are loaded to the memory of each instance (usually a DAG has only a few different language runtimes). This completes the hot start process. Now, when the components are invoked, the application along with its metadata is loaded into the memory of the hot started instances from the back-end storage to start execution.

New serverless function invocation requests are issued to the cloud provider by the DAG scheduler for the components for which no hot started function instances are left. Then, microVMs spawn up, component language runtimes and application metadata are loaded into the memory of the

Authorized licensed use limited to: Auburn University. Downloaded on February 09,2024 at 19:21:07 UTC from IEEE Xplore. Restrictions apply.

instances, and they undergo a cold start. When more number of function instances than the phase concurrency are hot started, the additional instances are terminated. Serverless function instances use *socket* system calls to retrieve application metadata from back-end storage and uses *execve* to load application contents to the memory.

Back-end storage server. Serverless function instances are stateless and they cannot directly communicate with one another. This is why after the execution of each component, all the output data is transferred from the serverless function instances to the storage server. During the execution of a phase, the back-end storage server notifies the system level DAG scheduler when half of the components of the phase have finished execution (*i.e.*, half of the output files of the phase are available in the storage). This is when the DAG scheduler hot starts function instances. Again, when a phase execution has been completed (all the output files from the phase are present in the back-end server), the back-end storage server notifies the DAG scheduler, which then starts the next phase. Apart from controlling the execution of a DAG, this storage server contains the executables of each of the components, their input files, output file paths, the required libraries, and the DAG structure in form of a tree.

Experimental setup. DayDream uses AWS Lambdas as instances forming the serverless function instance pool. There are two kinds of AWS Lambda instances – high-end (10 GB memory, 6 vCPU cores, and 10 Gb/s I/O bandwidth), and low-end (5 GB memory, 3 vCPU cores, and 5 Gb/s I/O bandwidth). The high-end and low-end function instances incur a cost of $0.0001667/second and $0.0000833/second, respectively. The AWS Lambda platform is configured with a provisioned concurrency of 1000, so that upon invocation of a component there is always a function instance available (hot or cold) to execute the function, and no wait time is incurred. These function instances are microVMs and boot up with Amazon Linux 2.0. The system level DAG scheduler is an Intel Xeon Scalable platform with a 14nm Skylake processor running Ubuntu 18.04 LTS. For the back-end storage, DayDream uses AWS S3 to form a mode of communication between the Lambdas and the DAG scheduler. In determining service cost, DayDream takes into account the cost of maintaining this storage throughout the execution of an HPC DAG; prior works have shown that storage is a critical componnent of serverless execution [39, 62].

All AWS resources are from the same AWS region (us-east-2), to minimize the communication overhead. The DAG scheduler uses *aws-cli* commands and *boto-3* to communicate with S3 and AWS Lambda. DayDream is also tested in Google Cloud Functions and Microsoft Azure Functions, using a similar resource structure as AWS Lambda and it can be easily portable across any serverless platform.

DAG Details. The components of a DAG in a phase are individual programs, which are often multi-threaded, and which do not need to interact with each other while running. When a function instance is invoked a microVM is spawned up extracting a component executable and its metadata from the back-end storage. The metadata of a component of a DAG provided by the application developer and is defined by the component executable, required library files, input files, and output file path. To execute an HPC DAG using DayDream, the user needs to provide the list of components of the DAG, their connectivity tree with each other, and the input and output file paths of the components of the DAG to the DAG scheduler. The executables corresponding to the components are stored in the back-end storage.

Scientific Workflows. We evaluate DayDream with three diverse real-world scientific workflow DAGs with direct science impact [65, 2, 40]. We evaluate DayDream with 50 runs of each of these DAGs, to cover a wide variety of input-operation pairs. **ExaFEL** [51]. ExaFEL is an Exascale Computing Project (ECP) application for near real-time interpretation of molecular structures revealed by X-ray diffraction. ExaFEL has 1,521 components, with an average phase concurrency of 17 over the runs. Each of the runs reads approximately 10 GB of data and writes 27 GB of data. **Cosmoscout-VR** [27]. Cosmoscout-VR is a modular virtual universe developed at the German Aerospace Center (DLR). Cosmoscout-VR has 15,232 components. On average each run has 1,100 phases, a phase concurrency of 90, reads 40 GB of data, and writes 53 GB of data. **Core Cosmology Library (CCL)** [16]. It is a standardized library of routines to calculate basic parameters used in cosmology. CCL has 982 components. On average, each of its runs has 110 phases, reading 22 GB of data and writing 17 GB of data.

Our evaluated workflows largely capture the characteristics of generic scientific workflows. The components of these evaluated workflows are widely used in other major workflows. For example, 81% of the components of ExaFEL are used in other major ECP project workflows like EXAALT, GAMESS, and ExaAM, belonging to domains like material science, biology, and electromagnetics [51]. Also, the compute, I/O, and memory utilization phasal characteristics of the components of these workflows closely match (Pearson correlation coefficient of 0.67 on an average) with components of other static genomic workflows from widely used projects like 1000Genome [72].

Competing techniques. We compare DayDream with state-of-the-art approaches which perform serverless scheduling of production workloads and traditional HPC DAG execution on node clusters. Also, we compare DayDream with an Oracle solution, which is practically infeasible but sets an upper limit on the possible improvement.

Serverless in the Wild (Wild) [66]. This approach uses a histogram and ARIMA-based approach for time series prediction to determine the components to warm up. This technique was primarily designed for warming up enterprise serverless workloads to minimize cold starts and represents the state-of-the-art in serverless computing literature – and is effective for its original purpose. To leverage ideas presented in Wild, we need a cluster of nodes to be allocated for warming

Authorized licensed use limited to: Auburn University. Downloaded on February 09,2024 at 19:21:07 UTC from IEEE Xplore. Restrictions apply.

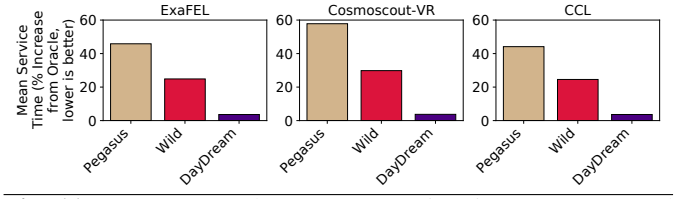

Fig. 11. DayDream reduces mean service time across several runs compared to other state-of-the-art techniques.

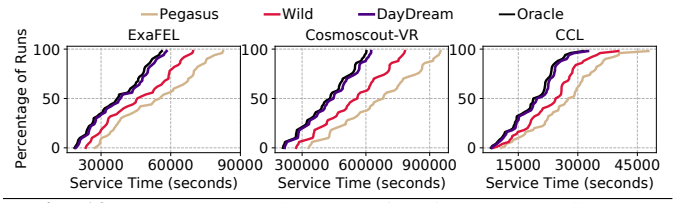

Fig. 12. DayDream reduces service time across all runs.

up serverless function instances. We build this cluster with AWS EC2 m5n nodes, which have the computational resources and costs similar to the high-end AWS Lambda instances used for DayDream. We select the number of nodes as the maximum phase concurrency of an application DAG, so like DayDream, Wild also does not incur wait time.

Pegasus [20]. It is the state-of-the-art HPC workflow manager and is effective for executing scientific workflow, but does not take advantage of serverless platforms. It performs several optimizations to co-locate components on the nodes of a cluster to minimize execution time by avoiding contention. Like Wild, Pegasus also uses AWS EC2 m5n nodes and we select the number of such nodes as the maximum phase concurrency of the application DAG. The cost is calculated as the cost of renting the entire cluster of nodes as unlike Wild and DayDream, the components run as individual executables with maximum achievable parallelization. Thus, at all times all the nodes of the cluster are active.

Oracle. This method minimizes both service time and service cost of DAG execution. It hot starts the exact number of serverless function instances as the phase concurrency to avoid any cold starts and cost wastage due to more number of hot starts than the phase concurrency. Note that Oracle is practically infeasible as it requires prior information of HPC DAG run, but it provides the upper bound on performance and cost benefits.

Evaluation Metrics. DayDream optimizes for **Service Time** and **Service Cost**. Service time is the end-to-end execution time (from invocation till the arrival of the final output) of the entire workflow. Service Cost is the cost incurred by the cloud provider for executing the entire DAG. It includes the cost to keep alive the hot started instances and also the execution cost of the components. Note that the keep alive cost of a hot started function instance is the same as the execution cost of the instance per unit time. This is why minimizing wasted hot starts is important.

V. Evaluation and Analysis

Service Time. *DayDream reduces the overall service time by more than 45% over the state-of-the-art workflow manager Pegasus and outperforms the existing state-of-the-art serverless approach (Wild) by more than 22%.*

Fig. 11 shows the mean service time for different workflows normalized to the Oracle solution (lower is better). Note that for each workflow, we report results averaged over 50 runs – these runs are diverse in nature, covering unique operations and inputs. Fig. 12 shows the normalized service time for all such unique runs for each workflow DAG.

From Fig. 11 and 12, we make two major observations. First, DayDream outperforms both competing techniques - the state-of-the-art workflow manager (Pegasus) and the state-of-the-art serverless approach (Wild). Second, DayDream is effective at improving service time across different runs of a given DAG (Fig. 12) – its benefits are consistent across different types of operations and inputs. For example, for Cosmoscout-VR, DayDream reduces service time by a minimum of 41% and a maximum of 47% compared to Pegasus across all runs. DayDream reduces service time by a minimum of 19% and a maximum of 23% compared to Wild across all runs. The reason DayDream outperforms the state-of-the-art workflow manager (Pegasus) is DayDream's ability to hot-start different components prior to their arrival and hence, avoid cold start overhead.

Second, DayDream utilizes microVMs for co-location of components which are better for avoiding more sources of performance interference, as shown earlier in Fig. 4. On average, DayDream's start up (with data fetching) overhead, execution overhead, and output writing overhead is less by 25%, 22%, and 8%, respectively, compared to Pegasus. Since, in Pegasus, the components perform I/O via a parallel file system in a node (versus network-based I/O in DayDream), the I/O overhead is less by 12% compared to DayDream. However, since most I/O is performed after the execution of individual components (to write output), this degrades the overall service time for Pegasus. DayDream outperforms Wild because DayDream employs hot start mechanism instead of Wild's warm up approach, reducing start-up latency by 26%.

The mean execution time of components in the evaluated HPC DAGs is 3.56 seconds. The mean start up overheads of warm started, hot started, and cold started components are 0.85 seconds, 0.93 seconds, and 1.16 seconds, respectively. Since individual components are usually short running in HPC DAGs [34], hot starts significantly improve the service time.

The advantage of the hot start mechanism is that only the operating system and the runtime environment are pre-loaded in the memory in anticipation of the arrival of different component functions, but the actual component function is integrated into the pre-loaded runtime environment only after a component is actually invoked. On average, hot starts reduces component service time by 19% compared to cold starts (warm starts reduce by 26% but are not effective for dynamic HPC DAGs). This mechanism reduces the cold start overhead partially – since the cold start overhead involves loading both the runtime

Authorized licensed use limited to: Auburn University. Downloaded on February 09,2024 at 19:21:07 UTC from IEEE Xplore. Restrictions apply.

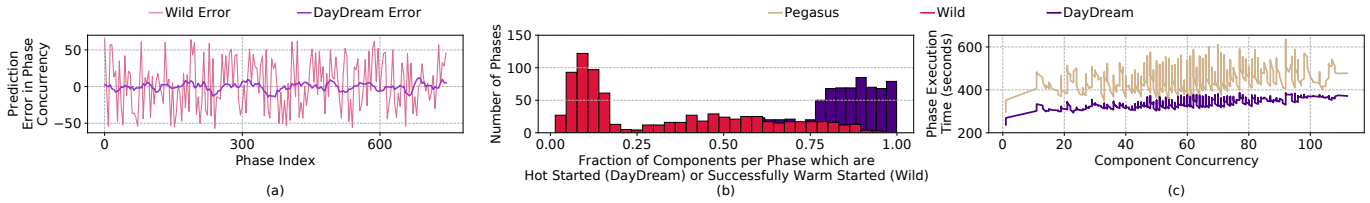

Fig. 13. To improve service time, DayDream (a) improves prediction of the number of hot started components, (b) achieves more successful hot starts than warm starts, and (c) reduces phase execution time as the number of components increase.

environment and component code in the memory.

In contrast, warm start mechanism already pairs up the component code with the runtime environment. While this approach has the potential to eliminate the cold start overhead completely, it requires successful prediction of which component functions will be invoked in the near-future and how many (i.e., the component concurrency for different types of components). For enterprise workloads, function arrivals have high predictability and hence, the warm start approach has been shown to be effective. But for scientific workflow DAGs, component concurrency has poor predictability, but phase concurrency (i.e., the number of component function instances summed up across all different component types in a phase) has more predictability. Hence, decoupling runtime environment and component code during pre-load (i.e., hot start) enable DayDream to reduce the effect of cold start. Next, we discuss why DayDream outperforms competing strategies.

Why does DayDream outperform competing strategies? To understand the reason behind DayDream's effectiveness, we present three key results. First, Fig. 13(a) shows that DayDream is effective at predicting the total number of components (phase concurrency), and hence, the prediction error in the number of hot starts required for different phases is quite low. In contrast, Wild's ARIMA time-series-based prediction is not fully effective at predicting the phase concurrency – even if one ignores the additional challenge of Wild's scheme to predict the component concurrency correctly (i.e., how many instances of each component type need to be pre-loaded in memory).

Previously, in Fig. 8, we showed limited prediction accuracy for Wild's ARIMA-based prediction. DayDream improves upon this prediction capability. ARIMA-based prediction is most useful when a time series follows an identifiable mathematical pattern or when the present phase concurrency is dependent on the past values. However, the phase concurrency in HPC DAGs does not follow such a pattern and instead has a probability distribution. ARIMA's prediction success is dependent upon a temporal correlation of data, which is very limited (Pearson correlation among different temporal windows is less than 0.25) for component and phase concurrency of HPC DAGs. To further demonstrate Wild's limited effectiveness, Fig. 13(b) shows the fraction of components per phase that is successfully pre-loaded. To better understand the trend depicted by this result, we first define the term "successful pre-load" in our context for different schemes.

A successful pre-load for the Wild technique is when the component in the warm started function instance is the same component being invoked. Recall that the Wild technique warm starts multiple instances and each instance may have a different component and runtime environment pairing – some of these instances are wasteful if a particular component is not invoked. In that case, the warmed up container (consisting of component, runtime environment, and operating system) is not used, and it also incurs expense – service cost due to warming up a wasteful function instance.

For DayDream, a pre-load (hot-start with runtime environment and operating system in the memory of function instance) is a successful pre-load if that function instance is needed in that phase. Recall that, under the hot start mechanism, any desired component can be attached to a hot started function instance. A hot started instance pre-load is not successful if it is not needed at all (i.e., the predicted phase concurrency is higher than the actual phase concurrency). Fig. 13(b) shows that Wild has a lower fraction of phase concurrency that is successful, compared to DayDream. This is expected because DayDream does not require to predict which component function to pre-load, instead, it only needs to predict the phase concurrency accurately.

Next, we provide quantitative evidence to support the intuition that pre-loading runtime environments are more effective than traditional state-of-the-art workflow managers (Pegasus). Fig. 13(c) shows the execution time of different phases for DayDream and Pegasus as the number of components per phase increases. We observe DayDream consistently achieves lower execution time for different phases because it pre-loads the runtime environment via the hot start mechanism, whereas Pegasus incurs cold start overhead for both the runtime environment and the component function code. This benefit is particularly desirable when the phase concurrency is large because the cold start overheads add up and then, they delay the start of subsequent dependent phases and components. Also, in Pegasus, multiple components in a phase run in parallel without strict resource isolation like DayDream's serverless approach. These components can contend for the same computational resource and increase the phase execution time.

Service Cost. *DayDream reduces the overall service cost by 23% and 12% over the state-of-the-art workflow manager Pegasus and state-of-the-art serverless approach Wild, respectively.*

Recall that the service cost has two components: cost incurred during the execution of different components in function instances and the cost of keeping a hot or warm

Authorized licensed use limited to: Auburn University. Downloaded on February 09,2024 at 19:21:07 UTC from IEEE Xplore. Restrictions apply.

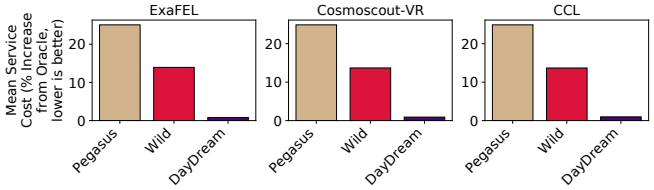

Fig. 14. DayDream's mean service cost is close to the Oracle.

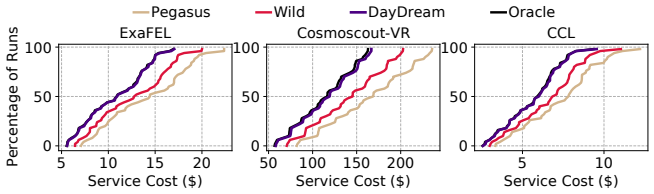

Fig. 15. DayDream reduces service cost across all different runs significantly more than the competing techniques.

started container alive in memory in the anticipation of an invocation. It is trivial to reduce the service time of workflows by simply pre-loading an excessively high number of instances for different components and keeping them alive in memory at all times. However, this naive approach is cost prohibitive. This is why we present results to confirm that DayDream is also effective at reducing cost, in addition to reducing service time (Fig. 14).

Similar to service time results, Fig. 14 shows the mean service cost for different workflows normalized to the practically-infeasible oracle solution. Fig. 15 shows the normalized service cost for all 50 unique runs for each workflow. We note that DayDream outperforms both competing techniques - the state-of-the-art workflow manager (Pegasus) and the state-of-the-art serverless approach (Wild).

Compared to Pegasus, the serverless approach used in DayDream executes components faster due to more resource isolation and avoiding cold starts. This reduction in execution time reduces the service cost. Also, as seen from Fig. 16(a)-(c), DayDream increases the computational resource utilization compared to Pegasus, as each individual serverless function instance in DayDream has much lower computational resources than the cluster of nodes in Pegasus. Reducing resource wastage minimizes the service cost in DayDream.

DayDream reduces the service time due to lesser resource wastage compared to the state-of-the-art serverless approach (Wild) (Fig. 16(a)-(c)). Though Wild also uses serverless function instances for the execution of the components, DayDream's usage of two-tier serverless function instances (low-end and high-end) helps it to identify the components which require even lesser computational resources than the majority of other components. The introduction of low-end function instances in DayDream further reduce the service cost.

From Fig. 13(a) we have observed that the phase concurrency prediction accuracy of Wild is often lower than DayDream. Also, unlike DayDream, the identification of the correct component to warm up is important in Wild, as otherwise the warm up of a function instance is not useful and it adds up to wasted keep alive cost. By more effective prediction of

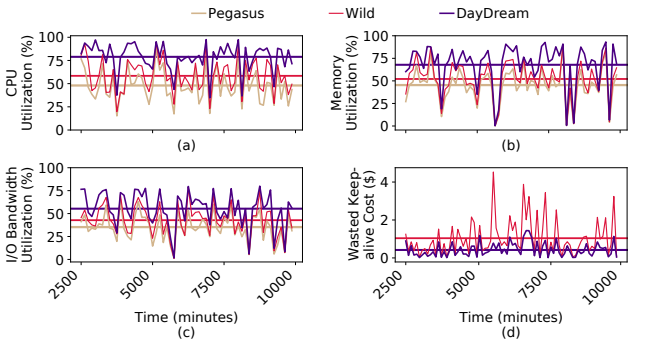

Fig. 16. DayDream reduces service cost by increasing (a) CPU utilization, (b) memory utilization, (c) I/O bandwidth utilization, and (d) reducing the wasted keep-alive cost (the horizontal lines represent the respective mean values).

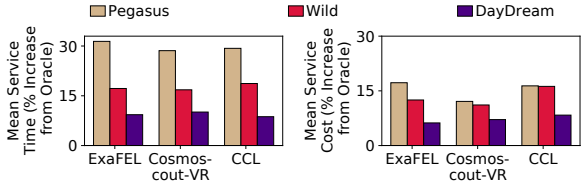

Fig. 17. DayDream reduces key metrics even for runs with hard-to-predict component concurrency across the phases.

the phase concurrency and hot start of components, DayDream reduces the wasted keep alive cost (Fig. 16(d)) which, in turn, reduces the service cost.

Limitation. DayDream's service cost benefits may be limited if a workflow has multiple different language runtimes for its various components. In such a case, all of these runtimes need to be compressed and stored in every hot started function instance. However, in a practical scenario, such a case is rare as most real-world workflows need only one or two language runtimes [69]. A mitigation strategy is to spend development effort on limiting runtime heterogeneity to three or less. There is also an opportunity to potentially combine Wild and DayDream's prediction technique to further improve the component prediction accuracy, more than what each technique can achieve individually in isolation.

Overhead. *DayDream incurs minimal overhead and is suitable for practical deployment on today's serverless platforms.*

All techniques incur overhead because they need to determine which components to pre-load and terminate. In DayDream, the main source of the overhead is determining the phase concurrency and hot starting instances. However, this overhead is typically negligible for all approaches. On average, this overhead is 0.043%, 0.036%, and 0.028% of a component execution time for Wild, Pegasus, and DayDream, respectively.

Effectiveness for hard-to-predict phase concurrency runs. *DayDream is effective even for workflows which have hard-to-predict phase concurrency during their runs.*

On average, 6% of the runs do not exhibit a pattern that follows an easy-to-infer probability distribution of phase

Authorized licensed use limited to: Auburn University. Downloaded on February 09,2024 at 19:21:07 UTC from IEEE Xplore. Restrictions apply.

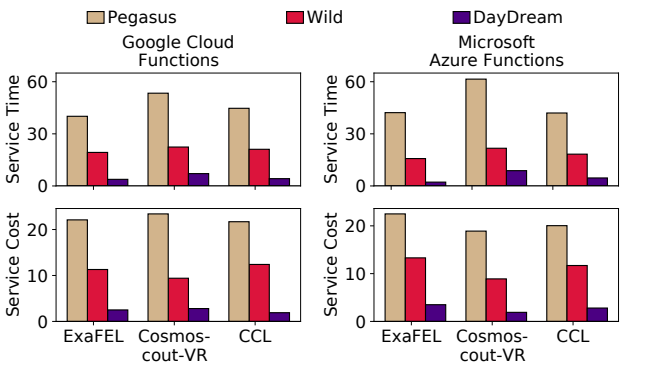

Fig. 18. DayDream reduces key metrics across cloud vendors.

concurrency, and in such cases, the distribution changes over time. Hence, it is particularly important to ensure that the DayDream is effective for these runs with challenging phase concurrency. Among all the runs, the top 10% runs where DayDream's prediction of phase concurrency has the highest error rate are defined as the hard-to-predict runs. These runs denote the worst-case performance of DayDream. Even for these runs, DayDream outperforms Wild by more than 8% and 7% in terms of service time and service cost, respectively (Fig. 17).

DayDream is portable across multiple cloud vendors. *DayDream's benefits in terms of performance and cost is present across multiple cloud vendors including Google Cloud, Amazon Web Services, and Microsoft Cloud.*

Our results show that DayDream's benefits are portable across serverless providers (Fig. 18). DayDream requires minimal command line argument changes (for example, the *cli* commands to run serverless function instances, invoke the system level DAG scheduler and back-end storage, depending upon the chosen provider) to make it easily portable across these cloud platforms. The design of DayDream is based on widely-available characteristics present on many serverless platforms like cold start and resource isolation. On average, DayDream reduces service time by 14% and 43% and service cost by 9% and 17% compared to Wild and Pegasus, respectively.

VI. Related Works

Scientific Workflows and Workflow Managers. Due to their modularity, scalability, and ease of maintenance, designing HPC applications in form of workflows is becoming increasingly popular [37, 86, 42, 59, 60, 28, 25, 15]. They have been a part of major HPC projects and discoveries [84, 73, 50]. This is why workflow managers like Pegasus, DAGman, and Kepler are becoming important to manage the data and state dependencies [20, 5, 21, 17, 80]. They optimize the execution by various means like optimizing the data flow path and properly scheduling the components to minimize contention [18, 41, 89, 17, 20, 10, 10, 76, 6]. However, these works do not account for the elastic resource needs of HPC workflows [11, 23, 31, 45, 83, 32, 78, 77, 91, 33, 26, 90, 79], which is why we design DayDream to reduce resource wastage and speed up execution.

High-Performance Serverless Computing. Serverless computing is traditionally used for short-running cloud workloads due to their stateless nature [54, 9, 1, 8, 53, 29, 30, 88, 4, 87, 35, 74, 82]. As serverless provides high parallelism, federated platforms like FuncX have been developed to ease serverless execution on HPC clusters [14]. Techniques like Nightcore [36], SAND [3], Cloudburst [75], SOCK [54], and Catalyzer [22] perform optimizations for fast start up, low inter function communication, and fast scaling. But these works are designed for applications or individual components of a serverless DAG [58, 67, 46, 12, 66]. But unlike DayDream, they do not perform optimizations to improve the execution of the entire workflow, by taking into account serverless specific characteristics of DAGs. These techniques are orthogonal but compatible with DayDream as their optimizations can be further added to DayDream without changing its functionality. Archipelago [71] and Wukong [13] submit their jobs as regular DAGs which are simplistic and static in nature (components and phase execution characteristics are static) and hence, not suitable for complex and dynamic HPC DAGs – the focus of DayDream's design.

Mashup [61] is the first work to specifically explore the serverless computing model for improving the execution efficiency of HPC workflows. Mashup designed a hybrid execution model (serverless and VM-based cluster) to improve the performance of HPC workflows. However, Mashup was not designed for dynamic scientific workflow whose execution characteristics and invocation path changes significantly during a job run – the focus of DayDream. Also, DayDream design trade-offs and optimization problems are different due to hot start mechanism and irregular nature of considered scientific workflows. Recent concurrent works [48, 47] have also explored serverless execution for enterprise DAG of functions via fusing multiple functions together, bundling and right-sizing the VMs. However, these enterprise DAGs have lower parallelism and dynamic execution characteristics (change in phase and component concurrency), compared to the highly irregular and complex scientific workflows.

VII. Conclusion

To bridge the gap between HPC workflows and serverless execution, we designed DayDream to serve complex HPC DAGs. We designed novel methods to improve the execution of three real-world, complex, dynamic scientific workflows on serverless platforms. DayDream, employs the hot start mechanism for warming up the components of the workflows by decoupling the runtime environment from the component function code to mitigate cold start overhead. DayDream optimizes the service time and service cost jointly to reduce the service time by 45% and service cost by 23% over the state-of-the-art HPC workload manager.

Acknowledgement. We thank the anonymous reviewers for their constructive feedback. This work was also supported by Northeastern University, NSF Award 1910601 and 2124897.

Authorized licensed use limited to: Auburn University. Downloaded on February 09,2024 at 19:21:07 UTC from IEEE Xplore. Restrictions apply.

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 419–434, 2020.

[2] M Aguena, C Avestruz, C Combet, S Fu, R Herbonnet, AI Malz, M Penna-Lima, M Ricci, SDP Vitenti, L Baumont, et al. Clmm: a lsst-desc cluster weak lensing mass modeling library for cosmology. *Monthly Notices of the Royal Astronomical Society*, 508(4):6092–6110, 2021.

[3] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. {SAND}: Towards high-performance serverless computing. In *2018 {Usenix} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 923–935, 2018.

[4] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 972–986. IEEE Computer Society, 2020.

[5] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludascher, and Steve Mock. Kepler: an extensible system for design and execution of scientific workflows. In *International Conference on Scientific and Statistical Database Management.*, pages 423–424. IEEE, 2004.

[6] Kaizar Amin, Gregor Von Laszewski, Mihael Hategan, Nestor J Zaluzec, Shawn Hampton, and Albert Rossi. Gridant: A client-controllable grid workflow system. In *37th Annual Hawaii International Conference on System Sciences.*, pages 10–pp. IEEE, 2004.

[7] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M Wozniak, Ian Foster, et al. Parsl: Pervasive parallel programming in python. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 25–36, 2019.

[8] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference*, pages 41–54, 2019.

[9] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[10] Junwei Cao, Stephen A Jarvis, Subhash Saini, and Graham R Nudd. Gridflow: Workflow management for grid computing. In *CCGrid 2003. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2003. Proceedings.*, pages 198–205. IEEE, 2003.

[11] Adam G Carlyle, Stephen L Harrell, and Preston M Smith. Cost-effective hpc: The community or the cloud? In *Second International Conference on Cloud Computing Technology and Science*, pages 169–176. IEEE, 2010.

[12] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 13–24, 2019.

[13] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. Wukong: a scalable and locality-enhanced framework for serverless parallel computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 1–15, 2020.

[14] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. Funcx: A federated function serving fabric for science. In *High-Performance Parallel and Distributed Computing*, 2020.

[15] Nauman Riaz Chaudhry, Anastasia Anagnostou, and Simon JE Taylor. A workflow architecture for cloud-based distributed simulation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 32(2):1–26, 2022.

[16] Nora Elisa Chisari, David Alonso, Elisabeth Krause, C Danielle Leonard, Philip Bull, Jérémy Neveu, Antonio Villarreal, Sukhdeep Singh, Thomas McClintock, John Ellison, et al. Core cosmology library: Precision cosmological predictions for lsst. *The Astrophysical Journal Supplement Series*, 242(1):2, 2019.

[17] Rafael Ferreira da Silva, Rosa Filgueira, Ilia Pietri, Ming Jiang, Rizos Sakellariou, and Ewa Deelman. A characterization of workflow management systems for extreme-scale applications. *Future Generation Computer Systems*, 75:228–238, 2017.

[18] Daniel De Oliveira, Eduardo Ogasawara, Fernanda Baião, and Marta Mattoso. Scicumulus: A lightweight cloud middleware to explore many task computing paradigm in scientific workflows. In *3rd International Conference on Cloud Computing*, pages 378–385. IEEE, 2010.

[19] Ewa Deelman, Tom Peterka, Ilkay Altintas, Christopher D Carothers, Kerstin Kleese van Dam, Kenneth Moreland, Manish Parashar, Lavanya Ramakrishnan, Michela Taufer, and Jeffrey Vetter. The future of scientific workflows. *The International Journal of High Performance Computing Applications*, 32(1):159–175, 2018.

[20] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira Da Silva, Miron Livny, et al. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35, 2015.

[21] Ewa Deelman, Karan Vahi, Mats Rynge, Rajiv Mayani, Rafael Ferreira da Silva, George Papadimitriou, and Miron Livny. The evolution of the pegasus workflow management software. *Computing in Science & Engineering*, 21(4):22–36, 2019.

[22] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Cat-

Authorized licensed use limited to: Auburn University. Downloaded on February 09,2024 at 19:21:07 UTC from IEEE Xplore. Restrictions apply.

alyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 467–481, 2020.

[23] Pradeep Fernando, Ada Gavrilovska, Sudarsun Kannan, and Greg Eisenhauer. Nvstream: Accelerating hpc workflows with nvram-based transport for streaming objects. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 231–242, 2018.

[24] Alexander Fuerst and Prateek Sharma. Faascache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 386–400, 2021.

[25] Richard Gerber, William Allcock, Chris Beggio, Stuart Campbell, Andrew Cherry, Shreyas Cholia, Eli Dart, Clay England, Tim Fahey, Fernanda Foertter, et al. Doe high performance computing operational review (hpcor): Enabling data-driven scientific discovery at hpc facilities. 2014.

[26] Wolfgang Gerlach, Wei Tang, Kevin Keegan, Travis Harrison, Andreas Wilke, Jared Bischof, Mark DSouza, Scott Devoid, Daniel Murphy-Olson, Narayan Desai, et al. Skyport-container-based execution environment management for multi-cloud scientific workflows. In *2014 5th International Workshop on Data-Intensive Computing in the Clouds*, pages 25–32. IEEE, 2014.

[27] Andreas Gerndt. Cosmoscout vr: Interactivity and immersion for space data exploration and mission planning. *AR/VR for European Space Programmes*, 2019.

[28] Nicholas Hazekamp, Joseph Sarro, Olivia Choudhury, Sandra Gesing, Scott Emrich, and Douglas Thain. Scaling up bioinformatics workflows with dynamic job expansion: A case study using galaxy and makeflow. In *2015 IEEE 11th International Conference on e-Science*, pages 332–341. IEEE, 2015.

[29] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[30] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with openlambda. In *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[31] Valerie Hendrix, James Fox, Devarshi Ghoshal, and Lavanya Ramakrishnan. Tigres workflow library: Supporting scientific pipelines on hpc systems. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 146–155. IEEE, 2016.

[32] Muhammad H Hilman, Maria A Rodriguez, and Rajkumar Buyya. Multiple workflows scheduling in multi-tenant distributed systems: A taxonomy and future directions. *ACM Computing Surveys (CSUR)*, 53(1):1–39, 2020.

[33] Christina Hoffa, Gaurang Mehta, Tim Freeman, Ewa Deelman, Kate Keahey, Bruce Berriman, and John Good. On the use of cloud computing for scientific workflows. In *2008 IEEE fourth international conference on eScience*, pages 640–645. IEEE, 2008.

[34] Zahaf Houssam-Eddine, Nicola Capodieci, Roberto Cavicchioli, Giuseppe Lipari, and Marko Bertogna. The hpc-dag task model for heterogeneous real-time systems. *IEEE Transactions on Computers*, 2020.

[35] Jananie Jarachanthan, Li Chen, Fei Xu, and Bo Li. Amps-inf: Automatic model partitioning for serverless inference with cost efficiency. In *50th International Conference on Parallel Processing*, pages 1–12, 2021.

[36] Zhipeng Jia and Emmett Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 152–166, 2021.

[37] Gideon Juve, Ewa Deelman, Karan Vahi, Gaurang Mehta, Bruce Berriman, Benjamin P Berman, and Phil Maechling. Scientific workflow applications on amazon ec2. In *2009 5th IEEE international conference on e-science workshops*, pages 59–66. IEEE, 2009.

[38] Sanjay Kadam et al. Bio-inspired workflow scheduling on hpc platforms. *Tehnički glasnik*, 15(1):60–68, 2021.

[39] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, 2018.

[40] Douglas Kothe, Stephen Lee, and Irene Qualters. Exascale computing in the united states. *Computing in Science & Engineering*, 21(1):17–29, 2018.

[41] Cui Lin, Shiyong Lu, Xubo Fei, Artem Chebotko, Darshan Pai, Zhaoqiang Lai, Farshad Fotouhi, and Jing Hua. A reference architecture for scientific workflow management systems and the view soa solution. *IEEE Transactions on Services Computing*, 2(1):79–92, 2009.

[42] Xiangyu Lin and Chase Qishi Wu. On scientific workflow scheduling in clouds under budget constraint. In *2013 42nd International Conference on Parallel Processing*, pages 90–99. IEEE, 2013.

[43] Wes Lloyd, Minh Vu, Baojia Zhang, Olaf David, and George Leavesley. Improving application migration to serverless computing platforms: Latency mitigation with keep-alive workloads. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 195–200. IEEE, 2018.

[44] Jakob Lüttgau, Shane Snyder, Philip Carns, Justin M Wozniak, Julian Kunkel, and Thomas Ludwig. Toward understanding i/o behavior in hpc workflows. In *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*, pages 64–75. IEEE, 2018.

Authorized licensed use limited to: Auburn University. Downloaded on February 09,2024 at 19:21:07 UTC from IEEE Xplore. Restrictions apply.

[45] Ketan Maheshwari, Eun-Sung Jung, Jiayuan Meng, Venkatram Vishwanath, and Rajkumar Kettimuthu. Improving multisite workflow performance using model-based scheduling. In *2014 43rd International Conference on Parallel Processing*, pages 131–140. IEEE, 2014.

[46] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. {SONIC}: Application-aware data passing for chained serverless applications. In *2021 {USENIX} Annual Technical Conference ({USENIX}{ATC} 21)*, pages 285–301, 2021.

[47] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. {ORION} and the three rights: Sizing, bundling, and prewarming for serverless {DAGs}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 303–320, 2022.

[48] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. Wisefuse: Workload characterization and dag transformation for serverless workflows. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(2):1–28, 2022.

[49] Anupama Mampage, Shanika Karunasekera, and Rajkumar Buyya. A holistic view on resource management in serverless computing environments: Taxonomy and future directions. *ACM Computing Surveys (CSUR)*, 2021.

[50] Marta Mattoso, Kary Ocana, Felipe Horta, Jonas Dias, Eduardo Ogasawara, Vitor Silva, Daniel de Oliveira, Flavio Costa, and Igor Araújo. User-steering of hpc workflows: state-of-the-art and future directions. In *Proceedings of the 2nd ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, pages 1–6, 2013.

[51] Paul Messina. The exascale computing project. *Computing in Science & Engineering*, 19(3):63–67, 2017.

[52] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[53] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 561–577, 2018.

[54] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. {SOCK}: Rapid task provisioning with serverless-optimized containers. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 57–70, 2018.

[55] PE Oguntunde, OS Balogun, HI Okagbue, and SA Bishop. The weibull-exponential distribution: Its properties and applications. *Journal of Applied Sciences*, 15(11):1305–1311, 2015.

[56] J Luc Peterson, K Athey, PT Bremer, V Castillo, F Di Natale, JE Field, D Fox, J Gaffney, D Hysom, SA Jacobs, et al. Merlin: enabling machine learning-ready hpc ensembles. Technical report, Lawrence Livermore National Lab.(LLNL), 2019.

[57] Marcin Płóciennik, Tomasz Żok, Ilkay Altintas, Jianwu Wang, Daniel Crawl, David Abramson, Frederic Imbeaux, Bernard Guillerminet, Marcos Lopez-Caniego, Isabel Campos Plasencia, et al. Approaches to distributed execution of scientific workflows in kepler. *Fundamenta Informaticae*, 128(3):281–302, 2013.

[58] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 193–206, 2019.

[59] Ioan Raicu, Ian T Foster, and Yong Zhao. Many-task computing for grids and supercomputers. In *2008 workshop on many-task computing on grids and supercomputers*, pages 1–11. IEEE, 2008.

[60] Gonzalo P Rodrigo, Erik Elmroth, Per-Olov Östberg, and Lavanya Ramakrishnan. Enabling workflow-aware scheduling on hpc systems. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 3–14, 2017.

[61] Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, and Devesh Tiwari. Mashup: making serverless computing useful for hpc workflows via hybrid execution. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 46–60, 2022.

[62] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Characterizing and mitigating the i/o scalability challenges for serverless applications. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*, pages 74–86. IEEE, 2021.

[63] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 753–767, 2022.

[64] Michael A Salim, Thomas D Uram, J Taylor Childers, Prasanna Balaprakash, Venkatram Vishwanath, and Michael E Papka. Balsam: Automated scheduling and execution of dynamic, data-intensive hpc workflows. *arXiv preprint arXiv:1909.08704*, 2019.

[65] Simon Schneegans and Markus Flatken. The solar system within arm's reach. *DLR Magazine*, (161):18–22, 2019.

[66] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 {USENIX} Annual Technical Confer-*

Authorized licensed use limited to: Auburn University. Downloaded on February 09,2024 at 19:21:07 UTC from IEEE Xplore. Restrictions apply.

ence ({USENIX}{ATC} 20), pages 205–218, 2020.

[67] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. Serverless linear algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 281–295, 2020.

[68] Donald Sharpe. Chi-square test is statistically significant: Now what? *Practical Assessment, Research, and Evaluation*, 20(1):8, 2015.

[69] Galen M Shipman. Programming models in hpc. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2016.

[70] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. Prebaking functions to warm the serverless cold start. In *Proceedings of the 21st International Middleware Conference*, pages 1–13, 2020.

[71] Arjun Singhvi, Kevin Houck, Arjun Balasubramanian, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Archipelago: A scalable low-latency serverless platform. *arXiv preprint:1911.09849*, 2019.

[72] Nayanah Siva. 1000 genomes project. *Nature biotechnology*, 26(3):256–257, 2008.

[73] Ola Spjuth, Erik Bongcam-Rudloff, Guillermo Carrasco Hernández, Lukas Forer, Mario Giovacchini, Roman Valls Guimera, Aleksi Kallio, Eija Korpelainen, Maciej M Kańduła, Milko Krachunov, et al. Experiences with workflows for automating data-intensive bioinformatics. *Biology direct*, 10(1):1–12, 2015.

[74] Vikram Sreekanti, Harikaran Subbaraj, Chenggang Wu, Joseph E Gonzalez, and Joseph M Hellerstein. Optimizing prediction serving on low-latency serverless dataflow. *arXiv preprint arXiv:2007.05832*, 2020.

[75] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *Proceedings of the VLDB Endowment*, 13(11).

[76] Young-Kyoon Suh, Hoon Ryu, Hangi Kim, and Kum Won Cho. Edison: a web-based hpc simulation execution framework for large-scale scientific computing software. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 608–612. IEEE, 2016.

[77] Kyle MD Sweeney and Douglas Thain. Early experience using amazon batch for scientific workflows. In *Proceedings of the 9th Workshop on Scientific Cloud Computing*, pages 1–8, 2018.

[78] Kyle MD Sweeney and Douglas Thain. Efficient integration of containers into scientific workflows. In *Proceedings of the 9th Workshop on Scientific Cloud Computing*, pages 1–6, 2018.

[79] Claudia Szabo, Quan Z Sheng, Trent Kroeger, Yihong Zhang, and Jian Yu. Science in the cloud: Allocation and execution of data-intensive scientific workflows. *Journal of Grid Computing*, 12(2):245–264, 2014.

[80] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency and computation: practice and experience*, 17(2-4):323–356, 2005.

[81] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. {FaaSNet}: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 443–457, 2021.

[82] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 133–146, 2018.

[83] Teng Wang, Suren Byna, Bin Dong, and Houjun Tang. Univistor: Integrated hierarchical and distributed storage for hpc. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 134–144. IEEE, 2018.

[84] David Woollard, Nenad Medvidovic, Yolanda Gil, and Chris A Mattmann. Scientific software as workflows: From discovery to distribution. *IEEE software*, 25(4):37–43, 2008.

[85] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 30–44, 2020.

[86] Dong Yuan, Yun Yang, Xiao Liu, and Jinjun Chen. A cost-effective strategy for intermediate data storage in scientific cloud workflow systems. In *2010 IEEE international symposium on parallel & distributed processing (IPDPS)*, pages 1–12. IEEE, 2010.

[87] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 1049–1062, 2019.

[88] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–12, 2019.

[89] Yong Zhao, Youfu Li, Ioan Raicu, Shiyong Lu, Cui Lin, Yanzhe Zhang, Wenhong Tian, and Ruini Xue. A service framework for scientific workflow management in the cloud. *IEEE Transactions on Services Computing*, 8(6):930–944, 2014.

[90] Yong Zhao, Youfu Li, Ioan Raicu, Shiyong Lu, Wenhong Tian, and Heng Liu. Enabling scalable scientific workflow management in the cloud. *Future Generation Computer Systems*, 46:3–16, 2015.

[91] C. Zheng, B. Tovar, and D. Thain. Deploying high throughput scientific workflows on container schedulers with makeflow and mesos. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 130–139, 2017.

Authorized licensed use limited to: Auburn University. Downloaded on February 09,2024 at 19:21:07 UTC from IEEE Xplore. Restrictions apply.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

Artifact Link. The artifact has the DOI:10.5281/zenodo.6941485. It is available at: https://zenodo.org/record/6941485

Description. Daydream uses serverless computing to execute dynamic scientific workflows, expressed as directed acyclic graphs (DAGs). DayDream learns about invocation and execution characteristics of the components in a dynamic DAG. This helps DayDream to optimally schedule the components of a DAG to jointly minimize both execution time and cost. DayDream introduces a novel concept called *hot start* of serverless function instances using a probabilistic model to speed up DAG execution on a two-tier (high-end and low-end) serverless platform.

Evaluated Applications. DayDream is evaluated with the scientific workflows: ExaFEL, Cosmoscout-VR, and CCL. These workflows are widely used by the HPC community and are parts of major HPC projects. They represent significant variation in terms of compute characteristics and have all significant components of scientific workflows.

Implementation Details. DayDream is implemented in Python3.6 using AWS Lambda in Amazon Cloud as the serverless platform. However, it can be easily ported across other commercial serverless platforms, like Google Cloud Functions in Google Cloud and Azure Functions in Microsoft Cloud. It utilizes properties of commercial serverless platforms such as cold starts. DayDream has three major components in its design: (1) A system level DAG scheduler, (2) a back-end storage server, and (3) a serverless function pool. The system level DAG scheduler hot starts serverless function instances to execute components of a DAG. The back-end storage server is used by the different serverless functions to exchange messages during their execution. Now, as the application execution starts, the DAG scheduler connects to the back-end storage server and instructs which components corresponding to the current phase should be executed. Depending upon the instructions from the DAG scheduler, function instances for the execution of components of a phase are hot started.

Nomenclature Used. DayDream executes applications in form of dynamic DAGs. In a DAG, individual units of execution are called components. The components which can run in parallel form a phase. Each component is executed by a serverless function, i.e., a AWS Lambda. Phase time is the time to complete each phase of the DAG. The sum of all phase times is the total execution time of the DAG. Service Time of a serverless function is the total end-to-end time of execution of a function which includes the compute and I/O times, as well as the cold start time or hot start time and the wait time (if applicable). Each DAG can run using different component and phase structures, each of which is referred to as a run. DayDream is evaluated with 50 runs of each of the three applications.

Setup DayDream. DayDream executes the components of DAGs on AWS Lambda. However, the controller of DayDream runs on a local computer, which invokes functions, hot starts functions, and performs all other scheduling decisions. This local computer can be any Linux based system which can run python subprocess, python threading, and can install aws command line interface (CLI). In our experimentation we used a 20-core machine with Ubuntu 18.04 LTS operating system. On this local computer, perform the following actions to setup DayDream:
(1) *Install the required libraries and AWS CLI:*

```
pip3 install awscli
pip3 install scipy
pip3 install scipy
```

(2) *Configure AWS CLI:*

```
aws configure
```

Then follow the prompts to enter your AWS credentials to link the local computer to your AWS account.
(3) *Clone the artifact repository in the local computer:*
The directory contains three folders CCL-daydream, cctx-workflow-daydream, and cosmoscout-vr-daydream. Each of these folders contain scripts, executables and experimental data of the application DAGs.
(4) *Set up AWS execution role:*
Login to AWS Web portal. Set up an execution role from AWS dashboard with full access of Lambda to S3. This execution role id should start with *arn:aws:iam:...* followed by numericals. Copy this execution role id.

From the root directory of the repository go to *cd CCL-daydream/build/*. Open the *create.py* python file and the find two lines which says *command = aws lambda create-function....* In these two lines replace the —role parameter with the execution role id which you just created before. Perform the same action, by changing the *create.py* files for both the other applications by going into their respective build directories (*cd cosmoscout-vr-daydream/build/*, and *cd cctbx-workflow-daydream/build*).

This completes the setup of setup of DayDream.

Run DayDream.
(1) *Run CCL:*

```
cd ./CCL-daydream/build
python3 main.py
```

This is the one and only line required to execute all 50 runs of CCL with DayDream as the controller. It sets up CCL in AWS Lambda, sends its executables to AWS, hot starts functions, executes functions on high-end (AWS Lambda with 10 GB memory) and low-end (AWS Lambda with 5 GB memory) AWS Lambdas, and does all other scheduling decisions that DayDream performs. It takes approximately 158 minutes to complete all the 50 runs. The *run-1* to *run-50* folders are inside *./CCL-daydream/* directory. After the completion

Authorized licensed use limited to: Auburn University. Downloaded on February 09,2024 at 19:21:07 UTC from IEEE Xplore. Restrictions apply.

of each run, 3 output files are generated in the corresponding *run* directories: (a) *phase_time.txt* (time to complete each phase of the DAG, the sum of which is the total execution time of the DAG). (b) *function_service_time.txt* (contains the service time of all individual components in all the phases). (c) *execution_cost.txt* (the cost incurred to the cloud provider for each individual components, the sum of which is the total execution cost of the DAG).

(2) *Run cctbx-workflow:*

```
cd ./cctbx-workflow-daydream/build
python3 main.py
```

This is the one and only line required to execute all 50 runs of cctbx-workflow with DayDream as the controller. It sets up cctbx-workflow in AWS Lambda, sends its executables to AWS, hot starts functions, executes functions on high-end (AWS Lambda with 10 GB memory) and low-end (AWS Lambda with 5 GB memory) AWS Lambdas, and does all other scheduling decisions that DayDream performs. It takes approximately 531 minutes to complete all the 50 runs. The *run-1* to *run-50* folders are inside *./cctbx-workflow-daydream/* directory. After the completion of each run, all the 3 output files are generated inside the corresponding run directories, similar to CCL.

(3) *Run cosmoscout-vr:*

```
cd ./cosmoscout-vr-daydream/build
python3 main.py
```

This is the one and only line required to execute all 50 runs of cosmoscout-vr with DayDream as the controller. It sets up cosmoscout-vr in AWS Lambda, sends its executables to AWS, hot starts functions, executes functions on high-end (AWS Lambda with 10 GB memory) and low-end (AWS Lambda with 5 GB memory) AWS Lambdas, and does all other scheduling decisions that DayDream performs. It takes approximately 585 minutes to complete all the 50 runs. The *run-1* to *run-50* folders are inside *./cosmoscout-vr-daydream/* directory. After the completion of each run, all the 3 output files are generated inside the corresponding run directories, similar to CCL.

Code Structure. The *main.py* script under *./application-name-daydream/build/* is the main controller of DayDream. It has the following main functions: (a) *define_application* – it generates the DAG for the particular run and maps the executables (.exe files) with the different components. (b) *predict_hot_start* – it predicts the number of serverless functions to hot start in each phase based on a Weibull distribution fit. (c) *hot_start* – it actually invokes and hot starts Lambdas in AWS (d) *execute* – when a component is invoked, this function hot starts them if functions are available or cold starts them if hot started functions are not available. (e) *delete_container* – it deletes serverless functions after the completion of the run. (f) *generate_output* – it generates the output files of each run.

The *main.py* access three other python scripts: *create.py*, *invoke.py*, and *delete.py* to create, invoke, and delete serverless functions, respectively.

The *test.zip* folder inside *./application-name-daydream/build/* contains the package with executables and AWS Lambda functions which is shipped to AWS by *main.py* to create serverless Lambda functions.

The directory *./application_name-daydream/my_test/* records the profiling data of each components in a DAG. It includes the concurrency per phase, cpu utilization, memory utilization, and I/O bandwidth utilization.

Evaluation Metrics. We compare the performance of DayDream in terms of DAG execution time (summation of all phase times) and execution cost.

AUTHOR-CREATED OR MODIFIED ARTIFACTS:

Artifact 1

Persistent ID: `https://zenodo.org/record/6941485##`

Artifact name: DayDream: Executing Dynamic Scientific Workflows on Serverless Platforms with Hot Starts

Reproduction of the artifact with container: Each *run* folder under *./application-name-daydream/* has three baseline files (*execution_cost-baseline.txt*, *function_service_time-baseline.txt*, and *phase_time-baseline.txt*). These baseline files are the data generated during the experimentation of DayDream and are used in the paper. If the data in the corresponding generated files (*execution_cost.txt*, *function_service_time.txt*, and *phase_time.txt*) matches with the data from the baseline files, with a less than 10% error bound, then the results of DayDream are successfully reproduced.

Authorized licensed use limited to: Auburn University. Downloaded on February 09,2024 at 19:21:07 UTC from IEEE Xplore. Restrictions apply.