An Algebra for Robust Workflow Transformations

Nicholas Hazekamp
University of Notre Dame
Notre Dame, Indiana 46556
nhazekam@nd.edu

Douglas Thain
University of Notre Dame
Notre Dame, Indiana 46556
dthain@nd.edu

***Abstract*—Scientific workflows are often designed with a particular compute site in mind. As a user changes sites the workflow needs to adjust. These changes include moving from a cluster to a cloud, updating an operating system, or investigating failures on a new cluster. As a workflow is moved, its tasks do not fundamentally change, but the steps to configure, execute, and evaluate tasks differ. When handling these changes it may be necessary to use a script to analyze execution failure or run a container to use the correct operating system. To improve workflow portability and robustness, it is necessary to have a rigorous method that allows transformations on a workflow. These transformations do not change the tasks, only the way tasks are *invoked*. Using technologies such as containers, resource managers, and scripts to transform workflows allow for portability, but combining these technologies can lead to complications with execution and error handling. We define an algebra to reason about task transformations at the workflow level and express it in a declarative form using JSON. We implemented this algebra in the Makeflow workflow system and demonstrate how transformations can be used for resource monitoring, failure analysis, and software deployment across three sites.**

I. Introduction

Scientific workflows define a set of tasks and their interdependencies to provide performance, reproducibility, and portability. Workflows are used every day in bioinformatics [1]–[6], high energy physics [7], [8], astronomy [9], and many other domains. Workflow management systems provide support for expressing the required resources, environment, and configuration for each task. Correctly specified workflows are explicit about required setup and environments. Often these workflows are designed for a specific site, failing when moved to different sites.

Differences between execution sites makes porting workflows hard and debugging complex. It is common for workflows to assume libraries and programs are available, use applications configured for only one operating system, or rely on unspecified configurations, all of which fail on different sites. Accommodating each site's configuration requires a number of unique transformations to properly execute. The tasks themselves do not change, but the environment, error handling, and configuration may.

A typical use case is the need to deploy the same operating system and software stack on several available compute sites. Unfortunately, each site may have a unique operating system or lack the necessary software. Users need a way to quickly switch between each site, but do not want to rewrite the workflow for each one. The simple answer is to use containers, but how do we easily apply these containers to tasks? This is further complicated when each site may use different container technologies (i.e. Docker [10] and Singularity [11]).

The ability to combine available tools is required to handle unique configurations and environments. Unfortunately, no single tool can address these changes, but multiple tools are needed in conjunction. As the number and variety of tools increases, the complexity of combining them increases as well. For example, if Singularity and a custom script are both applied to a simple task by prepending their commands, characteristics of execution like exit status, provenance of files, and the final executed command become opaque. Properly nesting the container inside of a script allows for differentiating failures, debugging, and consistent execution. Each additional layer must become a more nuanced transformation as nesting technologies, such as containers, resource monitoring, and error handling, becomes necessary. Different combinations of tools are required depending on the site's unique configuration. The variable nature of required tools indicates the importance of only applying tools to a workflow as needed, rather than adding them to the workflow specification at each site.

We define an algebra for workflow transformations to address the complexity of nesting different tools and technologies. Based on the sandbox model of execution, this algebra formalizes the operations for applying transformations to tasks which produce new tasks. These transformations can then be applied in series to produce a task that incorporates all applied transformations. Using formalized task transformations, we are able to precisely apply multiple transformations to a workflow and cleanly map to each task.

This algebra was expressed using JSON so that it is independent of (and therefore portable to) a variety of systems. Using this JSON expression, a driver was written in Makeflow [12] that allows us to apply transformations to a full workflow. We discuss the challenges in applying transformations and how these methods can be applied incorrectly and incompletely. To show the efficacy of this solution we show several case studies. The first uses a Singularity container to provide consistent environments, a resource monitor to give accurate usage stats, and a sandbox to isolate the available files and workspace. The second shows a failure handler that captures a core-dump and converts it into a stack trace, streamlining analysis and lowering data transfer. The final case study executes the same workflow on several sites using an environment builder that dynamically builds required software at each task.

978-1-5386-9156-4/18/$31.00 ©2018 IEEE
DOI 10.1109/eScience.2018.00031

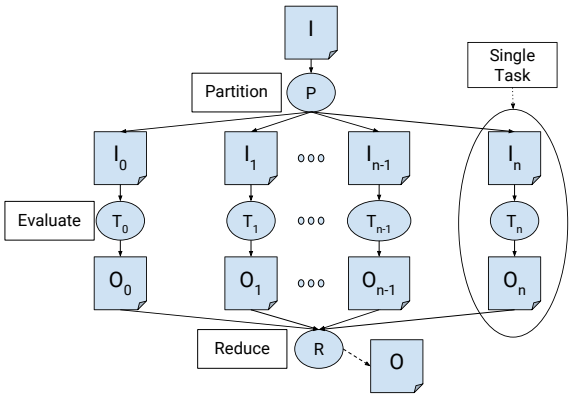

Fig. 1. Example Workflow.
This workflow shows standard split-join behavior. Each circle is a task and and each square is a file. The first task partitions the data, the next set of tasks analyze the individual partitions, and the last task aggregates them. Each task executes independently and tasks are often run on batch execution systems.

II. Background and Challenges

Scientific workflows are a widely used means of organizing a large amount of computational work. A workflow consists of a large number of tasks, typically organized in a graph structure such that the outputs of some tasks can be used as the inputs of other tasks. Each task is a unit of work that can be dispatched to a batch system or cloud facility and can range in scale anywhere from a brief function call lasting a few seconds to a large parallel application running on hundreds of nodes for several hours. Examples of widely used workflow management systems include Pegasus [13], Kepler [14], Swift [15], Makeflow [12], and many others. Figure 1 shows an example of a typical workflow structure.

A workflow primarily describes the researcher's work to run a set of simulations, to analyze a dataset, to produce a visualization, etc. However, like any kind of program, there may be a number of secondary requirements that must be met to complete the work: a particular software environment should to be constructed, resource controls for the batch system will be selected, monitoring and debugging tools should be applied to the task, and so forth. This might involve setting environment variables, providing additional inputs, capturing additional outputs, invoking helper processes, and more.

The first version of a workflow, constructed at a particular computing site, may have all of these aspects intertwined with the definition of the tasks to be done. The application may depend upon software environments installed in fixed paths in a shared filesystem. Environment controls may be set within individual tasks. Resources may be hard coded for a particular batch system The graph structure may reflect the current set of debugging tools enabled. While this may work well at the first site, it may become necessary to move the workflow to another site in order to improve performance, increase scale, or to apply the workflow in a new context. All these site-specific controls are unlikely to work in the new context, and the receiving user is then stuck with the problem of disentangling the core code from the local peculiarities.

An appealing approach to this problem is to define simple modifications that can be individually applied to tasks (transformations) in order to achieve specific local effects. For example, one might have a transformation to run a task in a container environment, another transformation to perform monitoring and troubleshooting, and a final transformation to configure a software environment for the local site. With this approach, the scientific objective of the workflow can be expressed in a portable way. A set of external transformations are used to modify the tasks as needed for the local site. Porting a workflow from one site to another becomes the simple job of adjusting a few transformations rather than rewriting the workflow from scratch. If it is necessary to transform the workflow in a new way, a transformation can be written, shared , and applied to many workflows.

However, our experience is that designing and using transformations is not so easily done. What may seem like a simple and obvious transformation can end up creating complex interactions and incorrect results. As a simple example, suppose that we want to run each task inside a Singularity container named `centos.img`. At first, this sounds as simple as prepending `singularity run centos.img` to each command string then running the task. While this works in limited cases, the general case for workflows with complex task definitions fails. There are several reasons for this:

Substitution semantics. Using basic string substitution to embed one command inside another often complicates execution. Commands that use input/output redirection, consume files, or change the environment collide using basic substitution. Addressing this uncertainty with shell quoting only further complicates the matter and may change the execution.

Workflow modifications. Applying a transformation to a task not only changes the individual task, but may also have an effect on the global structure of the workflow. A command transformed by a container now has an additional input (`i.e. centos.img`) which must be accounted for as a dependency in the workflow. Container images are large and affect the scheduling and resource management of the workflow. In a similar way, the container produces additional outputs which must be collected and managed by the workflow.

Namespace conflicts. Transformations can modify the local filesystem namespace. Log files with fixed names, temporary generated files, files based on task input files, or modifications to the working directory all alter the task and the workflow. These actions blindly modify files outside the workflow or cause race conditions with other concurrent transformations. Since it is not always possible to alter these hard-coded paths into unique filenames, collisions are inevitable.

Troubleshooting complications. The exit semantics of a transformed task are complex as it is not sufficient for a transformation to simply return the task's integer exit status. Each exit status should be differentiated, as transformations may fail separately. For example, preparing the environment may fail because a necessary software dependency is not

present, a container may fail when pulling the container image over the network, or a resource monitor may exit when resources are exhausted. In each of these cases, we must have a means of distinguishing between *transformation failure* and *task failure*. When multiple transformations are applied, the result of the task looks more like a stack trace than a single integer.

To address these challenges, we need a more rigorous way of defining tasks and the transformations on those tasks such that any valid transformation applied to any valid task gives the expected result in a way that can be nested. In short, we need an algebra of workflow transformations in order to make scientific workflows more robust, portable, and usable.

III. An Algebra of Workflow Transformations

We designed a formal abstraction to accommodate the execution behavior of various tools. This formalism isolates each transformation for consistent execution, allowing for organized nesting. In particular, our abstraction describes how to define a transformation for a given tool, as well as aspects of execution to consider. Transformations are based on the sandbox model of execution, which describes all aspects of execution for which a transformation is responsible.

A. Notation

For the purpose of expressing tasks and transformations in a precise way, we use a notation that is based on JavaScript Object Notation (JSON). In addition to the standard JSON elements of atomic values (`true`, `123`, `"hello"`), *dictionaries* `{ name: value }`, and *lists* `[ 10, 20, ... ]`, we add:

- `let X = Y` is used to bind the name `X` to the value `Y`.
- `define F(X) = Y` defines a function `F` that will evaluate to the value `Y` using the bound variable `X`.
- Simple expressions can be built up using standard arithmetic operators and function calls on values and bound variables.
- `eval X` evaluates the expression X and returns its value.

Using this notation, a single task ($T1$) in a workflow is expressed in JSON like this:

```
let T1 = {
 "command": {
   "pre":[ ],
   "cmd": "sim.exe < in.txt > out.txt",
   "post":[ ],
   },
 "inputs"      : [ "sim.exe", "in.txt" ],
 "outputs"     : [ "out.txt" ],
 "environment": {},
 "resources"   :
   {"cores":1, "memory":1G, "disk":10G }
}
```

Note that the schema is fixed. Every task consists of a command with a `pre`, `cmd`, and `post` component, a list of input files, a list of output files, a dictionary of environment variables, and a dictionary of necessary resources (all defined in detail later). Importantly, the formal list of inputs and outputs is distinct from the command-line to be executed, as guessing the precise set of files needed from an arbitrary command-line is difficult. For example, a program might implicitly require a calibration file `calib.dat` and yet not mention that on the command line. The base task's list of inputs and outputs is drawn from the structure of the DAG by the workflow manager.

B. Semantics

Makeflow allows for tasks in this form to be executed on a wide variety of execution platforms, including traditional batch systems (such as SLURM [16], HTCondor [17], and SGE [18]), cluster container managers, and cloud services. Because each of these systems differ in considerable ways, it is necessary to define precise semantics about the execution of the task and the namespace in which it lives. Once these semantics are established, it becomes possible to write transformations that work correctly regardless of the underlying system. To accommodate these varied systems, we introduce the sandbox model of execution.

The **sandbox model** of execution isolates the environment and limits interactions to only specified files. Isolating the task to run only the specified environment allows for higher flexibility about where the task can run as well as increasing the reproducibility of execution. Limiting the locally available files helps prevent undocumented file usage, enforcing accuracy of the file lists.

Applying a sandbox to a task is a multi-step process for ensuring consistent environment creation. It goes as such:

1) Allocate/ensure appropriate space for execution, based on resources.
2) Create sandbox directory.
3) Link/copy inputs to ensure correct in-sandbox name, based on inputs.
4) Enumerate environment variables based on the specified environment.
5) Run task defined command, using *pre*, *cmd*, and *post*.
6) Move/copy outputs outside of sandbox with appropriate out-sandbox name, based on outputs.
7) Exit and destroy sandbox.

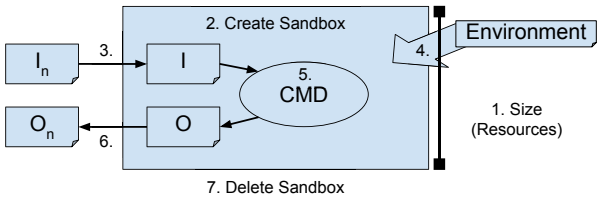

Fig. 2. The sandbox model of task execution. This shows the different steps needed to isolate the task from the underlying workflow environment to prevent side-effect on the environment and filesystem.

```
define Singularity(T)
{
 "command" : {

   "cmd": "singularity run image " +
           T.script + " > log." + T.ID

 }
 "inputs"   : T.inputs +
              ["image", T.script],
 "outputs"  : T.outputs +
              ["log."+T.ID],

 "resources" : {


   "disk"   : T.resources{disk} + 3G
 }
}
```

Fig. 3. Abstract `Singularity` transformation
Describes the Singularity command, added files (such as image and output log), and increases the required disk space. Note, several of the variables are unbound and will be resolved when applied to a task. Unaltered fields are left undefined.

```
eval Singularity(T1) yields
{
 "command": {
   "pre":[ ],
   "cmd": "singularity run image " +
          "t_ID.sh > log.ID"
   "post":[ ],
 },
 "inputs"   : ["sim.exe", "in.txt",
               "image", "t_ID.sh" ],
 "outputs"  : ["out.txt",
               "log.ID"],
 "environment" : {}
 "resources" : {
   "cores"  : 1,
   "memory" : 1G,
   "disk"   : 13G,
  }
}
```

Fig. 4. Resulting task of applying `Singularity` to `T1`.
The transformed task has all of the variables bound. The file lists have combined the previously defined files with the files added by `Singularity`. *The resources are resolved and the required values account for the original task and the transformation.*

C. Transformations as Functions

A transformation is an abstraction of a task and provides the information needed to translate a raw program invocation into a properly defined task. A transformation contains the same fields defined in a task: a command, inputs, outputs, resources, and environment. However, it is an incomplete task with unbound variables that are resolved when applied to a task as a function.

Figure 3 illustrates Singularity written as a transformation. As mentioned above, the generic definition of the transformation contains unbound variables such as `T.cmd`, `T.inputs`, and `T.outputs`. When the transformation is applied to a task, those variables are bound from the task's structure. Singularity requires additional space (3G) to account for the Singularity image. Here, resources are not defined as a static value. Rather they are in addition the to underlying resource. Additionally, the Singularity transformation does not define an environment, so it is left out.

The resulting task of evaluating `Singularity(T1)` can be seen in Figure 4. The previously unbound variables have been resolved, such as `T.inputs` becoming `["sim.exe, "in.txt"]`. The values that were not defined or extended by `Singularity` were resolved from the underlying task, such as `cores` and `memory`. Importantly, to create a valid task even empty fields like *pre*, *post*, and *environment* are still specified, allowing for evaluation and additional transformations to be applied.

If you look carefully at Figure 3 you will notice two variables not bound by the underlying task directly, `T.script` and `T.ID`. As part of the abstraction, the underlying task is emitted as a script that is called in place of the command, creating `T.script`. The ability to treat transformations as functions is achieved by isolating each transformation as a separate process. Isolating a transformation provides several key benefits: clearly defined ordering of transformations, instantiated environments persist only in that process and its children, and exit status can be attributed at each level to track failures. In practice this is achieved by producing a script that defines the task, as seen in Figure 5.

The second variable, `T.ID` is key to this method's success. The ability to uniquely identify each task provides a clear mapping to the workflow. A unique identifier is created using the checksum of the current task, which incorporates the command, input files' names and contents, output files' names, environment, and resources. This identifier is used to identify the output script and can be used by the transformation to uniquely identify files in the workflow. Additionally, as applying a transformation produces a new task, the identifier is updated after each transformation.

D. Applying the Sandbox Model

The creation of a script from a task focuses on isolating just the transformation but relies on finalization of the task sandbox laid out in Section III-B. To consistently apply the sandbox model to a task we define a sandbox procedure to

```
#!\bin\sh
#ID TASK_CHECKSUM

# POST function
POST(){
  # Store exit code for use in analysis.
  EXIT=$?

  # Run post commands.

  # Exit with stored EXIT which may
  # have been updated by post.
  exit $EXIT
}
# Trap on exit and call POST.
trap POST EXIT INT TERM

# Export specified environment.

# Run pre commands.

# Run core command.
sim.exe < in.txt > out.txt
```

Fig. 5. Script created when evaluating `Singularity(T1)`.

produce a script that creates a sandbox, handles files, and runs the command. This procedure is applied to a task prior to execution to isolate the task to a single sandbox directory.

This begins with creating a unique identifier, based on the task checksum. The identifier is used to create the sandbox and script names used in execution. In the script a `POST` function captures the exit status, executes `post` commands, and returns the outputs. This function is set as a trap to also analyze failures. Next, the sandbox is created and inputs are linked into it. The process changes directories, exports the environment, and runs the `pre` commands. After the environment is set up, the task `cmd` can run.

IV. Transformations in Practice

In applying the above algebra, there are design considerations to be made. To maintain the ability to nest several transformations together, it is important to consider the naming conflicts, the importance of differentiating `pre`, `cmd`, and `post`, file management, resource specification, and how the environment of a task is extrapolated.

A. Composability versus Commutability

An important aspect of this algebra is the ability to reason about how the combinations of different transformations interact and if they can be applied to created a valid task. Using the previously defined application of transformations we find that the set of transformations are composable, but not commutable. These transformation are not commutable because the ordering in which they are applied changes the core evaluation of the task. This is, by design, to allow for the differentiation of transformation ordering.

Transformations, in general, are composable. Any transformation can be applied to any task and produce a valid task, with the exception of static name collisions. A static name collision can result when an application uses hard-coded or default names for files, careless naming, or even randomly generated names. Running a single transformation at a time may not cause a collision, but nested transformations and concurrent tasks make collisions inevitable, as is often seen with output logs and files sharing names between tasks.

Naming is resolved at the local level by detecting when applying a transformation creates overlapping names. If collisions are detected, the transformation is not applied and a failure is returned. Though this restricts some combinations, this can be overcome by better understanding the application and using options to produce unique files.

However, if the same restrictions were applied to tasks across the workflow, transformations with static names would be prohibited entirely. As this may be inevitable, static files may be remapped to a unique name in the workflow. As each task is isolated in a sandbox, static files can be renamed when moving to the global namespace using the task identifiers. Remapping of the file relies on a more verbose file specification as a JSON object instead of a string filename. JSON object specification enables the wrapper to specify an *inner_name*, specifying the name inside the sandbox, and the *outer_name*, specifying the name in the workflow context. An example of how this would look with a statically named file can be seen in Figure 6 which defines a resource monitor transformation.

```
define RMonitor(T) {
 "command" : [
  "cmd": "rmonitor -- " + T.script
 ]
 "inputs"  : T.inputs + ["rmonitor",
               T.script]
 "outputs" : T.outputs +
              [{"outer_name="summary."+ID,
               "inner_name"="summary"}]
}
```

Fig. 6. Verbose JSON object file specification.
In this example, the resource monitor uses a statically named default summary, "summary". In this case, the summary file's name is static and will collide in the global workflow context. To avert this collision the file is specified with its static inner_name, and a unique outer_name using the task's ID.

B. Command Description

Commands express the setup, execution, and post processing of a task. Commands are broken up into three parts, pre, cmd, and $post$ based on the command structure outlined.

Pre is a set of commands that run prior task invocation and setup the task sandbox. This includes setting environment variables, configuring dependencies, and loading modules or software. For example, a Docker transformation would use pre to load or pull images.

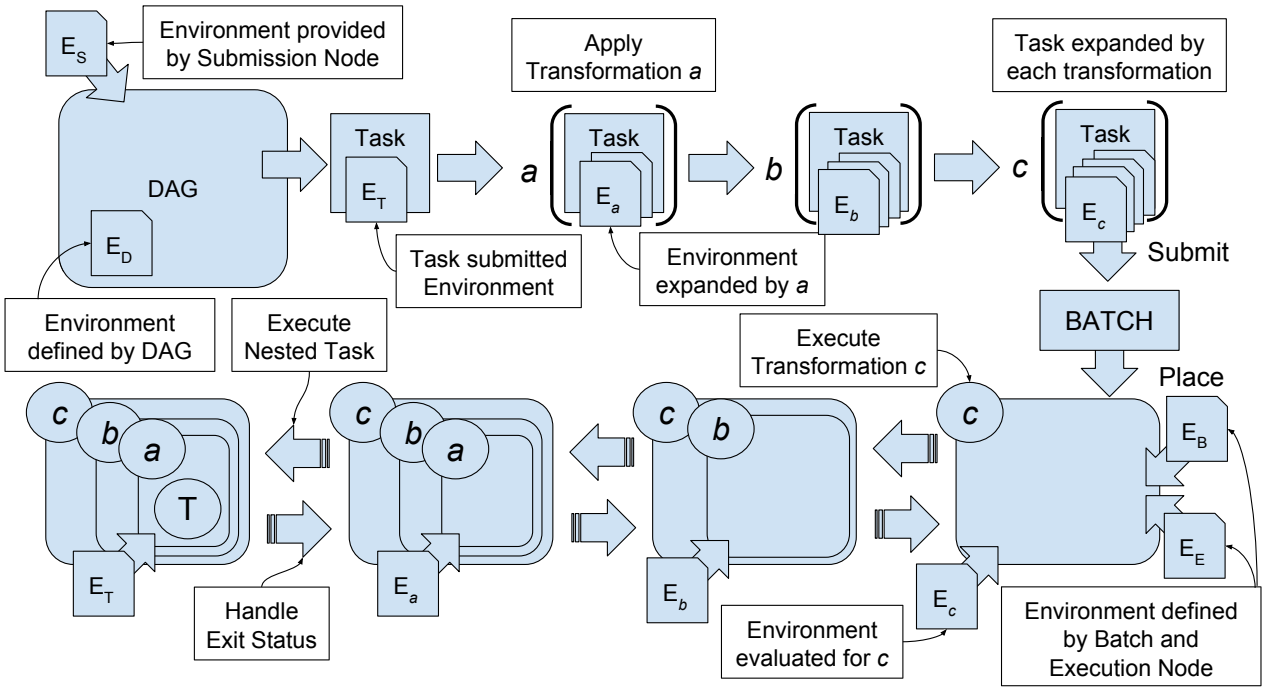

Fig. 7. General approach to Sandbox model of execution

The environment that exists at task execution is derived from several sources. The environment starts at the DAG where variables are resolved internally and from the host machine. These values define the task's initial environment. Transforms are applied to the task which extend the environment (a, b, c are generic transforms), but applied at execution. At the execution site, the environment is defined by the execution node and batch system. A execution each transformation is applied and invokes its environment limited to the transformation's execution.

$post$ is a set of command that run after task invocation and is used to handle failure by interpreting or masking them, create outputs to prevent batch system failures from missing files, or validate correctness of outputs. $Post$ can differentiate docker failing to load an image from task execution failure, allowing more nuanced debugging.

The cmd string outlines the context in which the underlying command is invoked. cmd outlines how the underlying task is called and isolates the effects of the calling transformation.

A benefit of separating the command into these parts is that it allows us to differentiate the failures or problems that result from each part. This is useful when determining that the setup of your container failed so the task should not run or to prevent the failure of $post$ analysis from indicating a task failure falsely. This separation also allows for each transformation to be clearly expressed in a script, enabling simplified debugging.

C. File List Management

As transformations are applied, the list of inputs and outputs grows. It is key for the correct organization of transformations that the set of required files is outlined by the task structure allowing the submitting system to confirm required inputs and verify expected outputs. It is possible for a transformation to rename or mask an existing file in the list. By doing so, the transformation changes the context of the task when evaluated. This can be done to allow for redirecting shared files or when using installed reference material. Maintaining a correct set of files helps prevent task collision. This information can also map a pre or $post$ application onto the files, estimate the space needed for execution, or log these files for later analysis.

D. Resource Provisioning

The resources define the necessary allocation for proper task execution. This value is extended and augmented by transformations as the context and required resources change. Commonly, as transformations are applied, additional disk space is needed to store new files (like container images).

Resource provisioning may not only be additive as the transformations are applied, but also maximal. This is typically the case used for cores. The number of cores does not expand as transformations are added but reflects the largest number of cores needed by any transformation. For example MPI utilizes a static number of cores, and to reflect that the resources specification uses the maximum of the provided value and the previous resource specification. The value of the resources required for a task tracks the largest set of each resource. After the transformations have been applied, the final task contains a single specification reflecting the total expected usage.

E. Environment Elaboration

An important aspect of a task is the execution environment. The environment defines a variety of values that control

execution such as available executables, required libraries and values, and available machines for cluster execution. However, the environment is often overlooked or ignored by the researcher, which causes corruption, errors, and failures. This can be mitigated on a single site, but as more sites are utilized managing these environments becomes unrealistic. It is important to understand how the task environment is defined, when transformations are applied, and how to direct the execution environment.

Figure 7 illustrates many of the locations a task's environment, or expected environment, can be derived and how it changes over execution. The workflow is executed with the submit machine's environment (E_S), defines an internal DAG environment (E_D), and dispatches a task specific environment (E_T). The task environment is defined with values derived from the DAG and submit machine, but crucially should not include variables that reference non-existent programs, libraries, and values at execution.

After the task is produced, transformations are applied that may append, update, or mask the provided variables. As a transformation is applied, E_T is written out to a script. The transformation can update values set in the task and add values based on needed, such as applications to the $PATH$ or libraries. Applying these transformations produces a stack of environments (E_a, E_b, E_c) that are applied at execution.

Tasks are placed on an execution node, where the environment may vary from the submit machine. The batch system environment (E_B) provides information about the assigned machines, available cores, and location of software modules and may be crucial for applications that use MPI or modules. The execution node environment (E_E) defines information such as local disks and available hardware.

The most basic method of invoking an environment is to apply all variables either at the beginning of execution or just prior to the task invocation. If applied initially, there are likely uninstantiated or unbound dependencies. If just prior to task execution, the context of each layer is evaluated using incorrect values or software. Both ultimately lead to a disconnect between the intended and the actual environment. To prevent this, as tasks are invoked each transformation creates a process that only applies the specified environment, limiting the environment's scope. Some transformations, such as containers, wipe or mask the provided environment. As transformation environments are applied, this should be taken into account, as the order and manner environments are instantiated may not carry through each transformation.

V. Applications of Transformations

We will now look at several example applications of transformations and how they can be used to improve the portability and robustness of workflows.

A. Sandbox Transform

A sandbox transform creates a directory, transfers files, and runs the command of the task. This simple lightweight transformation isolates the execution namespace from the workflow namespace, which allows for file renaming. The sandbox is removed after execution, which eliminates local unspecified files from polluting the namespace and disk quota. The sandbox can be captured for analysis on failure.

B. Container Transform

A container transform wraps tasks in a specified container for isolation and curated environments. A container pre command is used to pull down or unpack containers. This is done separately to differentiate failure of initialization from the invocation. A container cmd invokes the container with the nested command, which creates it own isolated process. Finally, a container $post$ command cleans up container images, reporting exit status.

Calling the nested command from the container isolates the container arguments from the shell script invoked. This prevents issues with differentiating arguments, isolating file redirects, and instantiating an environment inside the container. Containers can also mask the execution environment, which can prevent an environment specified earlier from existing inside the container. Containers often increases the required resources to account for additional files, like container images.

C. Resource Monitoring Transform

A resource monitor transform measures the utilized resources during task execution. If limits are specified, the monitor will stop the task and report if the resources are exceeded. As the resource monitor relies on the expected resources, it can utilize the adjusted specification to adapt as transformations add required resources. Other functionality includes monitoring the files that are accessed and creating a time series of the utilized resources. The resource monitor uses the cmd to track the process. The resource monitor benefits from the sandbox directory as the isolation allows the sandbox to be monitored for disk usage. This does not increase the resource needs but enforces them. In addition to the executable and usage summary, the resource monitor creates additional outputs such as the list of accessed files and resource usage time series, all of which are added as outputs.

D. Environment Transform

Regardless if a submit script, container, or virtual machine is used to run a task, there is often a need for configuration just prior to task execution. This is necessary in cases such as redirecting environment to reference data, configuring variables to include new libraries (such as `LD_PRELOAD`), or specifying a precise version of Java (by setting Java home and library paths). These types of transform rely on the pre command to initialize the environment. This can also be done using the $environment$ dictionary, though these value are directly exported and do not allow for nuanced initialization.

E. Failure Handling Transform

A transform that analyzes and handles errors at the task execution site allows for evaluations of the environment where the error occurred. Running evaluations only on failure limits

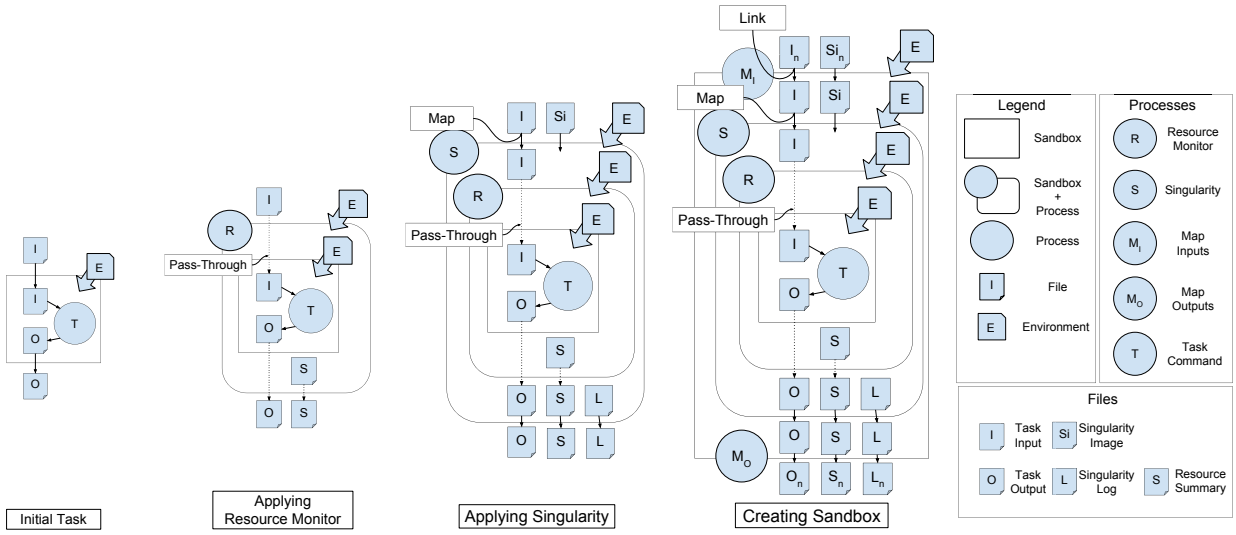

Fig. 8. Evolution of task as transformations are applied.
Starting from the left, we have the initial task with a single input and output. Next, a resource monitor is applied which passes through the original files, but also creates a summary of the resources used. After the resources monitor, a Singularity container is used to provide a consistent operating system, requiring a image to run from and creating a log. Finally, a sandbox is created to isolate execution, limiting file access when singularity maps the current directory. With each additional transformation, we see the complexity of the system increase.

the overhead on normal tasks and lessens the analysis burden of the user. This is used in determining software configuration/version incompatibilities, verifying if failure was due to limited resources, analyzing output files to prevent corrupted output, or process core-dumps into stack traces. Regardless of workflow size, automating error handling helps to handle errors allowing the user to analyze and address problems quickly. The error handling generally relies on the *post* to perform analysis based on the reported exit code or outputs.

VI. Case Studies

A. Resource usage in a Container

Resource monitoring helps to build an understanding of how a task behaves to accurately assign resources. If the task requires a container for execution, the resources utilized may be mischaracterized. As a result, it is useful to be able to separate the resource utilization of the task and the container. To do this, we first applied a resources monitor transformation to the task which will measure the resources used only by the task. Second, we applied the container transformation that allows for the application to run on different platforms.

The definitions of these transformations can be seen earlier in Figure 3 and Figure 6 for Singularity and the resource monitor respectively. Figure 8 illustrates the complexity that occurs when combining these transformations.

```
makeflow bwa.mf –apply rmonitor.jx –apply singularity.jx
```

Using the above `makeflow` call, we executed a workflow that runs BWA [1]. This workflow partitions a large query and runs each chunk concurrently, similar to Figure 1. This workflow was used as the basis to evaluate nesting the resource monitor and Singularity. This workflow was run in four configuration, both Singularity and the resource monitor, just Singularity, just the resource monitor, and the workflow with task sandboxes. We can examine the distribution of task execution time under these different configurations in Figure 9 and see that there is minimal additional overhead for each transformation. For these runs, Singularity utilized an image on a shared filesystem to limit the sandbox creation time, which can also be accomplished using a link. In situations with no shared filesystem the image is transferred and affects performance.

B. Failure Analysis

When moving between sites or changing data it is possible that an application can intermittently fail causing a core-dump. These core-dumps are unwieldy to move around and provide limited insight into the cause of the failure and its environment.

To address this we wrote a transformation that analyzes a core-dump at the execution site, and sends back the resulting stack trace that is produced by GDB (GNU Project Debugger). This transformation provides several keys benefits. The first is that it allows for automated analysis of core-dump failures for the user. Performing this in the execution sandbox provides early resolution about the app that failed and which task created it. Also, core-dumps are bloated and contain all of the memory and stack, which are consolidated considerably in a stack trace. This consolidation limits the amount of data transferred back to the user.

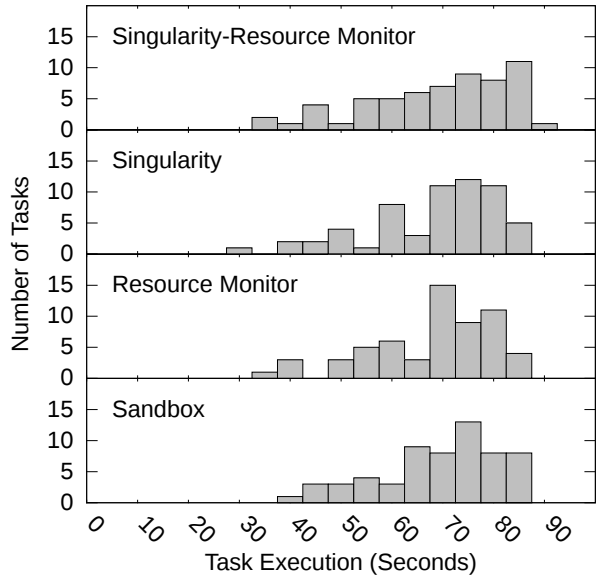

Fig. 9. Histogram of task execution with nested transformations.
Distribution of task execution of BWA workflow grouped by applied transformations, structured like Figure 1. The first configuration runs the resource monitor inside of a Singularity, the second runs just Singularity, the third runs just the resource-monitor, and the last runs the task inside of an application sandbox. We see that the distribution of execution time is consistent between runs, and the amount of transformation overhead is minimal.

Enabling the capture of core-dumps depends on your systems default settings. A common default uses the "core" prefix for core-dumps, but also limits their size. To accommodate this, we set the $ulimit$ to unlimited. After execution, if a core-dump was created we process it using GDB. This creates a stack trace that condenses the program failure. We implemented this transformation as seen in Figure 10.

```
define StackTrace(T) {
 "command" : [
  "pre" : ["ulimit -c unlimited"],
  "cmd" : "./" + T.script,
  "post": ["gdb " + T.command{cmd} +
         "core* -ex bt > stack."+ T.ID ,
           "touch stack." +T.ID]
 ]
 "outputs" : T.outputs+ ["stack."+T.ID]
}
```

Fig. 10. Stack trace transformation
The stack trace transformation allows a user to capture a core-dump of a failed task and convert it into a stack trace. This is done by setting the ulimit to allow the full core-dump, running the script, and then analyzing the core-dump with GDB. The step of touching the stack trace file prevents non-failed tasks from missing output.

To evaluate this, we wrote an application that allocates 1MB of memory and then fails roughly 20 percent of the time, creating a core-dump. We scaled this experiment to see the difference of sending the stack trace instead of the core-dump. As expected, we say differences of several orders of magnitude of transfers, reducing as much as 2.4 GB down to 0.5MB. The table below shows a comparison of workflows from 10 tasks up to 10000. Using this technique on larger memory intensive applications would yield more significant reductions.

Workflow Tasks	Failed Tasks	Total Core-Dump Size	Total Stack Trace Size
10	4	3.7MB	0.8KB
100	24	28.6MB	6.4KB
1000	174	214.9MB	48.8KB
10000	1957	2.4GB	553.1 KB

C. Complex Software Configuration

In scientific computing, researchers often construct and rely on a complex stack of analysis tools which are constructed over months to years of work and configuration. The resulting complexity often prevents researchers from scaling up or sharing their configuration with collaborators. There are several tools and solutions that exist to address this such as containers and build management tools (like Nix [19] or Spack [20]). This problem becomes more complex when the selected solution is not supported on a different platform, such as different container support or required installation permissions (superuser). For this reason we selected VC3-Builder [21], which is a user-level environment specification and construction tool.

As an example of complex software we use MAKER [22], a bioinformatic analysis pipeline which relies on 39 separate packages and a installed size of 4.2G. A MAKER installation requires careful dependency management as several tools rely on hard-coded, installation specific paths and installing by hand can take several hour. As a result, MAKER is often limited to a single carefully configured site. This is addressed with VC3-Builder and we want to leverage this to use MAKER on several sites for one workflow.

The transformation for VC3-Builder often simply invokes `vc3-builder`, specifying the required dependencies, and passing the command. However, because of MAKER's complexity several required packages have restrictive licenses requiring the user pass and unpack the libraries. In Figure 11, we can see a file, `manual-distribution.tar.gz`, which contains the restricted packages. Using pre, we are able to set up the correct directory structure, unpack the manual packages, and prepare for `vc3-builder`.

This workflow was executed using Makeflow and distributed with Work Queue [23], a master-worker execution platform. Workers were created on each site, and tasks were distributed to worker. To show the flexibility of transformations, workers were created on Stampede2, Jetstream, and HTCondor. The table below shows the task execution for each system, all of which were calculated using a single workflow.

Build Time	Stampede2	Jetstream	HTCondor
(HH:MM)	01:29	00:22	00:30

```
define VC3-Builder(T) {
 "inputs" : T.inputs + ["vc3-builer",
            "manual-distibution.tar.gz"]
 "command" : [
  "pre" : [ "mkdir -p vc3-distfiles",
            "cd vc3-distfiles",
            "cp ../manual* ./",
            "tar xzvf manual*",
            "cd .."],
  "cmd": "./vc3-builder --require maker"
         + T.script, ]
 "resources":{
  "cores" : 4,
  "disk"  : T.resources{"disk"} + 4G,
} }
```

Fig. 11. VC3-Builder transformation
VC3-Builder is typically self-contained, and the specified cmd is sufficient for most software. MAKER, however, relies on several libraries with restricted licenses that must be provided by the user. As a result, the transformation must create the install structure and extract libraries to the correct location for VC3-Builder. We specify cores for build threads and increase the disk for the installation.

VII. Related Work

As we are looking at workflow transformations, it is important to look at some of the major challenges of distributed computing [24], which includes reproducibility, portability, and flexibility, and how other workflow management systems approach them. Swift [15] programming language uses an *app* to define a program's invocation. Nesting apps could be similar, but the level of specification used does not address many of the issue mentioned earlier, such as name collisions, command nesting, and execution evaluation. Instead Swift focuses on compiler-like optimizations for grouping work [25]. Pegasus [13] focuses on planning the execution prior to submission. This allows for more strict adherence to quota limits and attempts to leverage the maximal parallelism, even across multiple sites [26]. Pegasus allows for changes to a workflow, but does so internally by clustering tasks [27]–[29], spawning clean-up tasks to remove files [30], and restructuring the workflow [31].

The similarities between these systems when comparing how a task is defined would allow for clear mapping of the above techniques each system. Importantly for this work, the underlying system continues to manage and submit tasks. The transformation are just applied and nested on top of them.

We also see similar work aimed at supporting containers natively. For these systems to support containers, there was a necessary level of work to abstract their applications, design the interface, and then implement it such that it works on a wide range of tasks [32]–[35]. Though this can deliver consistent execution, the embedded nature of this implementation makes it difficult for users to tailor the execution to their needs/specifications and must be managed. This support must change to address the available containers on a particular site, such as Docker [10], Singularity [11], CharlieCloud [36], or extend to support functionality like Slacker [37].

VIII. Conclusion

In the complex intersection of scientific workflows and unique compute sites, a solution is needed to allow for quick flexible workflow transformations. We introduce an algebra that formalizes task and workflow transformations. In conjunction with the formalization, we also define the sandbox model of execution which allows these transformation to be applied, but stay distinct in execution. We discuss several complications when designing for real workflows, and explain how we addressed them, such as file remapping, differentiating command parts, and clearly handling resources and environment. These methods were evaluated with three case studies. First, we nested a resource monitor in a container with consistent performance. Next, we captured and processed a task core-dump limiting the analysis and file transfer needed. Lastly, we executed a workflow across several sites running a complex application seamlessly. Using workflow transformations we were able to quickly transform workflows to fit unique configurations, without modifying the core work.

IX. Reproducibility Data

In an effort to provide consistent, reproducible results outlined are the resources utilized in this paper and where they can be found. Specific commits are mentioned to provide the exact version that was used.

The first is CCTools, which is an open-source distributed computing tool kit, containing Makeflow and Work Queue (**CCTools**). Second is the source for some of our example workflows, particularly BWA using in the first case study (**Workflows**). The last case study uses VC3-Builder (**VC3-B**). All of the workflows, transformations, and auxiliary programs used to produce this paper, include the tex (**Paper**).

CCTools:github.com/nhazekam/cctools/commit/7ab0645
Workflows:github.com/cooperative-computing-lab/makeflow-examples/commit/600e13b
VC3-B:github.com/vc3-project/vc3-builder/commit/dc285f9
Paper:github.com/cooperative-computing-lab/workflow-algebra-paper/commit/549d4df

As **Paper** is self-referential the final commit may be different.

All of these repositories are open source and contain Makefiles and instructions on how to build and run them.

Acknowledgment

This work was supported by part by National Science Foundation grant ACI-1642409.

REFERENCES

[1] H. Li and R. Durbin, "Fast and accurate long-read alignment with Burrows-Wheeler transform," *Bioinformatics*, vol. 26, no. 5, pp. 589–595, Mar 2010.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *J. Mol. Biol.*, vol. 215, no. 3, pp. 403–410, Oct 1990.

[3] N. Hazekamp and D. Thain, "Makeflow examples repository." [Online]. Available: http://github.com/cooperative-computing-lab/makeflow-examples, 2017.

[4] B. Giardine, C. Riemer, R. C. Hardison, R. Burhans, L. Elnitski, P. Shah, Y. Zhang, D. Blankenberg, I. Albert, J. Taylor, W. C. Miller, W. J. Kent, and A. Nekrutenko, "Galaxy: a platform for interactive large-scale genome analysis," *Genome research*, vol. 15, no. 10, pp. 1451–1455, 2005.

[5] D. Blankenberg, G. V. Kuster, N. Coraor, G. Ananda, R. Lazarus, M. Mangan, A. Nekrutenko, and J. Taylor, "Galaxy: A web-based genome analysis tool for experimentalists," *Current protocols in molecular biology*, pp. 19–10, 2010.

[6] J. Goecks, A. Nekrutenko, J. Taylor, and the Galaxy Team, "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences," *Genome Biol*, vol. 11, no. 8, p. R86, 2010.

[7] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, S. Koranda, A. Lazzarini, G. Mehta, M. A. Papa, and K. Vahi, "Pegasus and the pulsar search: From metadata to execution on the grid," in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, J. Dongarra, M. Paprzycki, and J. Waśniewski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 821–830.

[8] A. Woodard, M. Wolf, C. Mueller, N. Valls, B. Tovar, P. Donnelly, P. Ivie, K. H. Anampa, P. Brenner, D. Thain, K. Lannon, and M. Hildreth, "Scaling Data Intensive Physics Applications to 10k Cores on Non-Dedicated Clusters with Lobster," in *IEEE Conference on Cluster Computing*, 2015.

[9] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny, "Pegasus: Mapping scientific workflows onto the grid," in *Grid Computing*, M. D. Dikaiakos, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 11–20.

[10] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[11] B. M. Kurtzer GM, Sochat V, "Singularity: Scientific containers for mobility of compute," *PLoS ONE*, May 2017. [Online]. Available: https://doi.org/10.1371/journal.pone.0177459

[12] M. Albrecht, P. Donnelly, P. Bui, and D. Thain, "Makeflow: A Portable Abstraction for Data Intensive Computing on Clusters, Clouds, and Grids," in *Workshop on Scalable Workflow Enactment Engines and Technologies (SWEET) at ACM SIGMOD*, 2012.

[13] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, B. Berriman, J. Good, A. Laity, J. Jacob, and D. Kat, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming Journal*, vol. 13, no. 3, 200.

[14] L. Bertram, A. Ilkay, B. Chad, H. Dan, J. Efrat, J. Matthew, L. E. A., T. Jing, and Z. Yang, "Scientific workflow management and the kepler system," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1039–1065. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.994

[15] Y. Zhao, J. Dobson, L. Moreau, I. Foster, and M. Wilde, "A notation and system for expressing and executing cleanly typed workflows on messy scientific data," in *SIGMOD*, 2005.

[16] M. A. Jette, A. B. Yoo, and M. Grondona, "Slurm: Simple linux utility for resource management," in *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*. Springer-Verlag, 2002, pp. 44–60.

[17] M. Litzkow, M. Livny, and M. Mutka, "Condor - a hunter of idle workstations," in *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[18] W. Gentzsch, "Sun grid engine: Towards creating a compute power grid," in *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, ser. CCGRID '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 35–. [Online]. Available: http://dl.acm.org/citation.cfm?id=560889.792378

[19] E. Dolstra, M. de Jonge, and E. Visser, "Nix: A safe and policy-free system for software deployment," in *Proceedings of the 18th USENIX Conference on System Administration*, ser. LISA '04. Berkeley, CA, USA: USENIX Association, 2004, pp. 79–92. [Online]. Available: http://dl.acm.org/citation.cfm?id=1052676.1052686

[20] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, "The spack package manager: Bringing order to hpc software chaos," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 40:1–40:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807623

[21] B. Tovar, N. Hazekamp, N. Kremer-Herman, and D. Thain, "Automatic Dependency Management for Scientific Applications on Clusters," in *IEEE International Conference on Cloud Engineering (IC2E)* , 2018.

[22] B. L. Cantarel, I. Korf, S. M. Robb, G. Parra, E. Ross, B. Moore, C. Holt, A. Sanchez Alvarado, and M. Yandell, "MAKER: an easy-to-use annotation pipeline designed for emerging model organism genomes," *Genome Res.*, vol. 18, no. 1, pp. 188–196, Jan 2008.

[23] P. Bui, D. Rajan, B. Abdul-Wahid, J. Izaguirre, and D. Thain, "Work Queue + Python: A Framework For Scalable Scientific Ensemble Applications," in *Workshop on Python for High Performance and Scientific Computing (PyHPC) at the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (Supercomputing)* , 2011.

[24] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers, "Examining the challenges of scientific workflows," *Computer*, vol. 40, no. 12, pp. 24–32, Dec 2007.

[25] T. G. Armstrong, J. M. Wozniak, M. Wilde, and I. T. Foster, "Compiler techniques for massively scalable implicit task parallelism," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 299–310. [Online]. Available: https://doi.org/10.1109/SC.2014.30

[26] W. Chen and E. Deelman, "Partitioning and scheduling workflows across multiple sites with storage constraints," in *9th International Conference on Parallel Processing and Applied Mathmatics*, 2011, funding Acknowledgements: NSF IIS-0905032. [Online]. Available: http://pegasus.isi.edu/publications/2011/WChen-Partitioning_and_Scheduling.pdf

[27] W. Chen, R. Ferreira da Silva, E. Deelman, and R. Sakellariou, "Balanced task clustering in scientific workflows," in *9th IEEE International Conference on e-Science (eScience 2013)*, 2013, funding Acknowledgements: NSF IIS-0905032 and NSF FutureGrid 0910812. [Online]. Available: http://pegasus.isi.edu/publications/2013/chen-clustering-escience2013.pdf

[28] W. Chen, R. Ferreira da Silva, E. Deelman, and T. Fahringer, "Dynamic and fault-tolerant clustering for scientific workflows," *IEEE Transactions on Cloud Computing*, vol. 4, no. 1, pp. 49–62, 2016, funding Acknowledgements: NSF IIS-0905032, NSF ACI SI2-SSI 1148515, and NSF FutureGrid 0910812. [Online]. Available: http://pegasus.isi.edu/publications/2015/chen-tcc-2015.pdf

[29] W. Chen, R. Ferreira da Silva, E. Deelman, and R. Sakellariou, "Using imbalance metrics to optimize task clustering in scientific workflow executions," *Future Generation Computer Systems*, vol. 46, pp. 69–84, 2015, funding Acknowledgements: NSF IIS-0905032 and NSF FutureGrid 0910812. [Online]. Available: http://pegasus.isi.edu/publications/2014/2014-fgcs-chen.pdf

[30] S. Srinivasan, G. Juve, R. F. da Silva, K. Vahi, and E. Deelman, "A cleanup algorithm for implementing storage constraints in scientific workflow executions," *9th Workshop on Workflows in Support of Large-Scale Science (WORKS)*, 2014.

[31] C. K. Gurmeet Singh and E. Deelman, "Optimizing grid-based workflow execution," *Journal of Grid Computing*, vol. 3, no. 3-4, pp. 201–219, December 2005.

[32] C. C. Zheng and D. Thain, "Integrating Containers into Workflows: A Case Study Using Makeflow, Work Queue, and Docker," in *Workshop on Virtualization Technologies in Distributed Computing (VTDC)*, 2015.

[33] K. Sweeney and D. Thain, "Efficient Integration of Containers into Scientific Workflows," in *Science Cloud Workshop at HPDC*, 2018.

[34] W. Gerlach, W. Tang, A. Wilke, D. Olson, and F. Meyer, "Container orchestration for scientific workflows," in *2015 IEEE International Conference on Cloud Engineering*, March 2015, pp. 377–378.

[35] K. Liu, K. Aida, S. Yokoyama, and Y. Masatani, "Flexible container-based computing platform on cloud for scientific workflows," in *2016*

International Conference on Cloud Computing Research and Innovations (ICCCRI), May 2016, pp. 56–63.

[36] R. Priedhorsky and T. Randles, "Charliecloud: Unprivileged containers for user-defined software stacks in hpc," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 36:1–36:10. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126925

[37] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy docker containers," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, 2016, pp. 181–195. [Online]. Available: https://www.usenix.org/conference/fast16/technical-sessions/presentation/harter