A Job Sizing Strategy for High-Throughput Scientific Workflows

Benjamin Tovar*, Rafael Ferreira da Silva†, Gideon Juve†,
Ewa Deelman†, William Allcock‡, Douglas Thain*, and Miron Livny§

* University of Notre Dame
Notre Dame, IN
{btovar,dthain}@nd.edu

† University of Southern California
Information Sciences Institute
Marina Del Rey, CA
{rafsilva,deelman}@isi.edu

§ University of Wisconsin
Madison, WI
miron@cs.wisc.edu

‡ Argonne National Laboratory
allcock@alcf.anl.gov

Abstract—The user of a computing facility must make a critical decision when submitting jobs for execution: how many resources (such as cores, memory, and disk) should be requested for each job? If the request is too small, the job may fail due to resource exhaustion; if the request is too large, the job may succeed, but resources will be wasted. This decision is especially important when running hundreds of thousands of jobs in a high throughput workflow, which may exhibit complex, long tailed distributions of resource consumption. In this paper, we present a strategy for solving the job sizing problem: (1) applications are monitored and measured in user-space as they run; (2) the resource usage is collected into an online archive; and (3) jobs are automatically sized according to historical data in order to maximize throughput or minimize waste. We evaluate the solution analytically, and present case studies of applying the technique to high throughput physics and bioinformatics workflows consisting of hundreds of thousands of jobs, demonstrating an increase in throughput of 10-400 percent compared to naive approaches.

Index Terms—high throughput computing (HTC), resource monitoring and enforcement, automatic provision of resources, automatic job sizing, throughput and waste optimization

1 Introduction

High throughput computing (HTC) is an essential component of the scientific enterprise in fields as diverse as biology, physics, economics, and the digital humanities. In HTC, the goal is to run a very large number of independent jobs across a large number of independent nodes, maximizing the number of jobs completed over a long period of time. While early work in HTC assumed that a job occupied an entire node, the advent of large multicore machines now makes it common to run multiple jobs simultaneously on a large machine, while also harnessing multiple machines.

In practice, the end user of an HTC facility has very little direct control over the operation of the underlying facility. System administrators generally control the scheduling order, job placement, and a myriad of other details about how and when jobs run. The user simply submits concurrent jobs and expects that they will return after some time, having succeeded or failed.

However, the user **does** control one detail that can have an enormous effect on throughput: the size of a job, measured in the resources (such as cores, memory, and disk) needed for execution. Typically, a user states these values when a job is submitted and the facility scheduler uses this information to schedule jobs onto the available machines, attempting to meet local policy objectives such as system throughput, machine utilization, fairness between users, and other considerations. If a job should happen to exceed the stated resources, it is typically returned to the user as a failure.

The user of the facility faces a dilemma when selecting the size of a job. If the job size is small, then more jobs can run simultaneously in the available resources, achieving higher throughput. But, the job is more likely to exceed the resource limits at runtime and be returned as a failure. If the job size is large, the job is more likely to successfully run to completion. But, fewer jobs will run in the available resources, reducing throughput relative to the resources consumed. We call this the **job sizing problem**: how should the user select the size of a job so as to maximize the throughput of their workload?

Previous works in HTC has generally made two simplifying assumptions about job sizes. One is that the end user is familiar with the resources needed by each job, and is willing and able to accurately state these needs in advance [1]–[6]. The other is that similar jobs submitted in one batch are likely to have identical resource consumption profiles [7]. In our experience, **neither assumption is generally true** [8]–[12].

There are two reasons underlying this observation:

First, domain experts typically develop a code on their laptop or workstation, and simply know that the code runs effectively within that environment. Given the rapid pace of hardware development and upgrade cycles, they have no reason to know or care (for example) how much memory a single job consumes, or even how much memory is installed on their workstation. They simply take their working code, submit a large number of instances of it to a workflow manager, and expect the system to figure things out.

Second, as shown in Fig. 1, a large number of similar jobs submitted in a batch may exhibit complex, long-

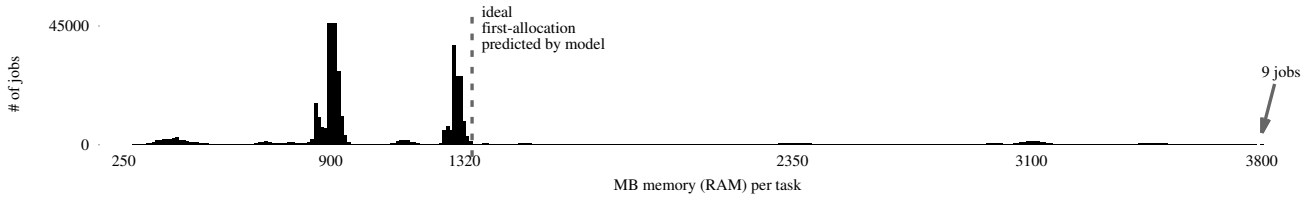


Fig. 1: Histogram for memory usage (resident size) for jobs in a high energy physics (HEP) data analysis workflow. *If we wish to run a new job in the workflow, how should we provision resources given this historical memory usage? Provisioning the maximum 3.8GB ensures no job failure, which it is wasteful for most jobs. Provisioning less than 3.8GB implies that some jobs will have to be retried with a larger allocation. For all these processes we need: a tool to measure the job, infrastructure to record the measurements, and a procedure to compute new provisions based on historical data.*

tailed distributions in resource consumption. These tails arise because even related jobs are not entirely uniform: they may start simulations from different random seeds or process different subsets of heterogeneous data. Measuring the resource consumption of a single job does not trivially produce the expected consumption of future jobs [9], [11], [12]. Moreover, allocating the upper bound of resources consumed by a batch of jobs would result in significant underutilization of the system [10]. A more informed approach is needed.

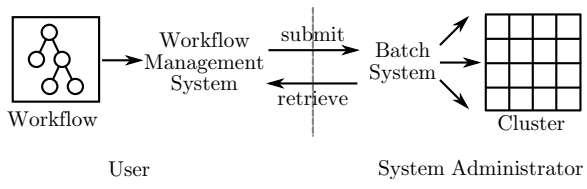
With this in mind, we present a **job sizing strategy** for the high throughput execution of a large number of jobs with complex resource distributions. Section 2 gives an overview of our approach with an example of resource observations from a production workload. Section 3 formalizes our ideas, presenting our resource feedback loop architecture, and our resource accounting model together with formal analysis of first-allocation strategies with straightforward implementations, and considering coarse qualitative information as job categories from the user. Section 4 evaluates this approach using resource consumption data collected from production workflows of thousands of jobs in the domains of high energy physics and bioinformatics. We compare our analytic approach to other strategies based on coarse statistical information and brute-force evaluation. Finally, we show the behavior of the algorithm in a production online workflow. Overall, we demonstrate that our job sizing strategy leads to an overall increase in throughput (from 10% to 400% across different workflows), and decrease in resource waste compared to fixed allocations under the richness of resource variations.

2 The Basic Idea

2.1 System Model

We assume a system composed of the following parts. The user provides a *workflow description* which indicates a set of jobs to be run, and the dependencies between them. A *workflow management system* (WMS) reads the workflow description, identifies which jobs are ready to run, and submits them to a *batch system* running on a cluster, cloud, or grid. The batch system schedules the jobs according to some local policy and places them on machines to run. As jobs complete or fail, they are returned to the WMS, which examines the results and may submit more jobs as dependencies are satisfied (see Fig. 2, below).

Fig. 2: System Components



We assume that the batch system is owned by an outside service provider, and the details of scheduling and placement on the cluster are outside of our knowledge and control. The user's only point of information and control is the WMS, which selects the resources for each job, in terms of a quantity of cores, memory, disk, and possibly other resources. The batch system runs the job within this static allocation of resources. If a running job exceeds any one resource, the batch system will terminate the job and return it to the WMS, indicating the reason for the failure. In this case, the WMS may choose to re-submit the same job again with a different job size.

2.2 The Job Sizing Problem

What job size (measured in system resources) should the WMS select for each job that it submits to the batch system?

Broadly speaking, if the initial job size selected is too small, it is more likely that the job will fail and be returned, thus wasting resources on a failed run that must be retried. On the other hand, if the initial job size selected is too large, the job will succeed on the first try, but waste resources that go unused inside the job's allocation. If the waste is large enough, throughput will be reduced because those resources could have been used to run another job.

If the resources consumed by a collection of jobs were constant, then the solution would be easy: run one job at a large size, measure its consumption, and then use that smaller size for the remainder of the jobs. However, our experience is that real jobs have non-trivial distributions. For example, Fig. 1 shows the histogram of memory consumption for a set of jobs in a high energy physics workflow run on an HTCondor [13] batch system at the University of Notre Dame. (We describe this workflow later in the paper.) Note that the histogram shows large peaks at approximately 900MB and 1300MB, but there are small number of outliers both above and below those values.

What memory size should we select for this workload? If we pick 3.8GB RAM for all jobs, then every job will succeed, but then most jobs would end up wasting several GB of memory that could be used to run other jobs. On the other hand, we could try running each job with a smaller value, wait to see which ones succeed or fail, and then increase the allocation gradually until reaching 3.8GB.

But precisely what smaller value should be used for the first attempt? The dotted line in Fig. 1 shows the value chosen by the method described in this paper, achieved by balancing the potential waste of over-allocation against that of under-allocation.

2.3 Our Approach

We present a comprehensive solution to the job sizing problem. Our solution consists of a user-level *resource monitor* to observe, record, and enforce resource limits on jobs in a facility-independent way. These observations are stored in a *historical archive* so that online systems can accumulate information across multiple runs. We develop a *resource feedback loop* that uses historical information to compute a recommended *first allocation* for the job sizing problem. (The dotted line in Fig. 1 shows the first allocation selected by this algorithm.) If the attempt to run the job fails, the resource archive is updated, and the job size is increased.

We consider two objectives in selecting a first allocation. If the user's objective is to *minimize waste*, then our algorithm balances the weighted probability of waste from a job size that is too small against the waste of a job size that is too large. If the user's objective is to *maximize throughput*, then our algorithm maximizes the number of jobs completed relative to the resources consumed. Finally, we show that these objectives, while not identical, only result in different allocations for a reduced set of resource usage and execution times values.

Finally, we evaluate these techniques on a large high energy physics workload of over 500K jobs and demonstrate that both throughput and waste can be improved by a factor of two over naive approaches. Our approach is external to the HTC facility in use, which means it can be used entirely from the user side by making modifications to the WMS, and no changes to the facility itself. That said, these techniques could be used by schedulers that need job resource usage information to place jobs efficiently [14]–[17].

3 Resource Feedback Loop

We consider the resources related to a job in three stages: (1) how many resources the job used in previous executions, (2) how many resources the job is using during the current execution, and (3) how many resources we predict the job will use in future executions. As shown in Fig. 3, these stages are arranged in two nested loops. The outer, or *main* loop, going from top-to-bottom and returning with a dashed line deals with the querying and recording of historical data. It communicates job information to the inner, or *execution* loop, which allocates, executes, and monitors jobs. We think of resources as real-valued functions of time per job. Managing and communicating these detailed job resources records as time-series across the different stages of execution becomes unwieldy as the number of jobs, and their duration, increases. Thus, we focus on conservatively describing the job using its peak resource usage values. We refer to the records of peak resource usage values as resource summaries.

3.1 User-Level Monitoring Tool

To collect job resource consumption in a consistent way across varying batch systems, we deploy a user-level monitoring tool along with each job. (The tool is fully described and evaluated in an earlier paper [18].) The monitoring tool generates resource summaries describing the execution of a job, and enforces allocations by keeping the job within given limits. Our interest is in measuring the job as a whole, considering its entire process tree, rather than recording individual processes. By necessity, the design of the monitoring tool is highly dependent on the host operating system, and we focus on jobs that run on Linux. The tool is designed to be independent of the batch-system, and it is dispatched as a wrapper for the job, running as a regular user process with no administrator privileges required.

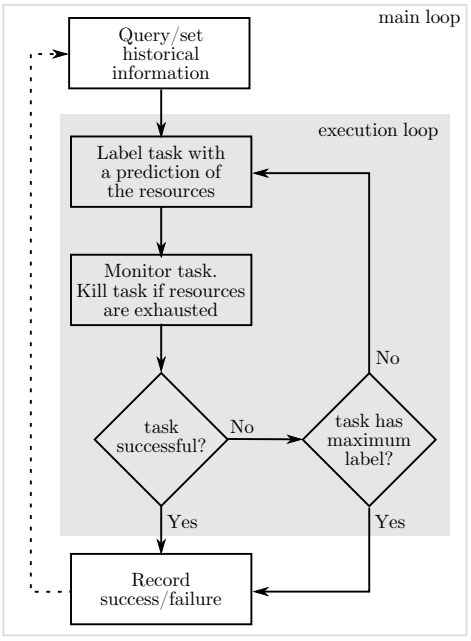


Fig. 3: Resource feedback loop.

The system has two nested feedback loops: the outer main *loop queries historical resource usage information, executes the workflow in the inner* execution *loop, and updates the historical data with new measurements collected. The execution loop computes resource allocations to provision and run the workflow's jobs, and retries jobs with larger allocations when resources are exhausted.*

We note that many other tools provide whole-system resource usage, such as load average, or swap memory used, but this information is not appropriate to characterize single jobs running on a machine. The objective is to observe peak resource values as closely as possible. Peak information can be accessed through the `/proc` filesystem, but this information is only available while a process is active. Thus, the monitor needs to run concurrently to (as the parent process of) the job, to track the appropriate processes on `fork` calls, and to retrieve peak information just before `exit/wait` events complete. Such control can be implemented by overloading `libc` functions using LD_PRELOAD [19], or through the `ptrace` interface [20]. Our monitoring tool can also produce a time series of the resources used, but as we discuss in Section 3.3, some formal conservative assumptions can be made such that only peak information is necessary.

Regarding the measurement of cores, our monitoring tool provides two statistics: average cores, and peak cores used. The average number of cores used is computed by `cpu_time/wall_time`. For peak cores, we similarly use

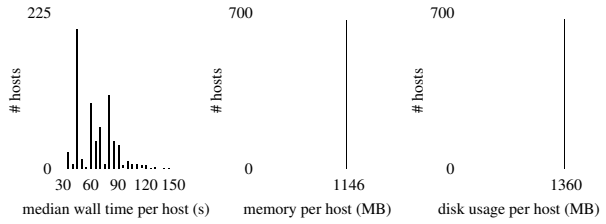


Fig. 4: Effect from the underlying resources pool.
For these histograms, the same job was dispatched to 670 different opportunistic hosts, and ran a total of 100000 times. The measurement for memory and disk in all executions was a unique value, while wall-time showed some variability.

cpu/wall-time, but using 20 second windows.

We have integrated the monitor with the Makeflow [21] workflow system, such that a user may simply activate monitoring through a command line switch, or an API call[1], after which jobs are automatically run with the monitor, and resource summaries are recorded. The resource information becomes immediately available to make decisions about running further jobs.

It is reasonable to question whether variations in resource measurement (such as those in Fig. 1) are due to the properties of jobs themselves, or due to unexpected variations between machines. To eliminate the latter, we designed an artificial job that computes cosines for about 60 seconds in one of our workstations. The values are also stored in memory (1.1GB), and written to a file (1.3GB). This job was dispatched 100000 times to our local campus batch system on which, for this particular run, landed in 670 different machines. The machines are highly heterogeneous, having been purchased from different vendors at different times, and vary from 4 to 64 cores, 1GB to 1TB of memory, and have a variety of X86-64 compatible CPU architectures.

The machines are managed by the HTCondor [23] batch system, which is able to harness computational resources which otherwise would remain idle because their owners are not using them. As soon as the owner starts using the machine, all HTCondor jobs are evicted. HTCondor is also able to partition hosts, such that multicore machines may serve several jobs, even from different users, at the same time. Given the opportunistic nature of our campus cluster, a priori, we have no precise knowledge about the hosts available to execute a workflow. In fact, as this small test shows, on average, the job ran for 73.73 seconds, with median 63.84s, and median absolute deviation of 25.15s. In Fig. 4, we show histograms for the median per host for wall-time, and the unique values recorded for memory and disk. The wall-time median per host was 64.78s, with a median absolute deviation of 26.32s.

These results indicate that conclusions about resource requirements such as cores, memory usage, and disk space can easily be used across different pools of resources, while one must exercise some care with regards to wall-time and cpu-time. For example, when dealing with wall-time our developments in Section 3.3 will use ratios and averages. This leads to a great practical reduction in time dependence, as presented in Section 4.

1. All of our software is available at [22].

3.2 Historical Archive

The resource summaries of executed jobs reported by the monitor are recorded in a database, which we refer to as the *resource archive*. Users of the archive submit JSON-encoded resource summaries with a structured schema. Additionally, using recent advances in PostgreSQL[2], we keep the raw JSON document submitted. We found this to be of great benefit, as we discovered that users are more likely to query the database by custom labels given to jobs, rather than by the numeric value of resources recorded. We formalize user custom labels by adding the field `category` to our structured schema. Categories are subsets of jobs that the user determines having the same purpose (e.g., *merge*, *analysis*, or *parameter-X*). In our experience, users can more easily label the type of job, compared to give ballpark figures of resources consumed; jobs with the same category label are suspected to have similar resource consumption, but this is not guaranteed.

Queries to the archive can be done directly through a website[3], or they can be made automatically by a workflow manager using a REST interface. For efficiency, the archive is contacted twice per workflow run. First, to bootstrap the allocation sizes based on historical data, and later to update the archive with the new jobs measured when the workflow finishes. The result of a query is a list of JSON objects encoding the jobs matched. SQL-based queries can be made according to user, category, command-lines, executable names, input/output files, user-defined fields, or resource ranges. A query can also be made by workflow type, for workflows that have the same description[4].

In Fig. 5, we show an example of historical information from a High Energy Physics (HEP) workflow. The figure shows the resource histograms for a CMS[5] experiment data analysis production workflow run with Lobster [24], a system for deploying high-throughput applications on opportunistic resources built on top of our tools, and designed with HEP jobs in mind. The workflow consisted of 538078 jobs that were run on approximately 25,000 different machines in our campus batch system over the course of a month. The workflow was labeled by the user with four categories, which correspond to five different steps: generating initial input data and simulation (LHEGS), digitization (DIGI), reconstruction (RECO), final reductions (mAOD) and merging (MERGE). All steps lend to high parallelization with bags of jobs, and can be run as pipeline where no step has to be completely finished before the next can start processing. We show the historical histograms for wall-time, cpu time, peak cores, memory, and disk, split by user-provided categories and as a whole. As it can be seen, the distributions cannot be easily characterized, as they may have many modes, with non-trivial tails.

3.3 Resource Allocations

We think of allocations per resource as establishing two-dimensional *boxes* where the jobs run. One dimension of the

2. As of PostgreSQL version 9.2.

3. `http://dvdt.crc.nd.edu`

4. That is, the description of the workflow (e.g., a file describing graph dependencies) has the same checksum. The archive does not try to interpret such descriptions, as it is workflow engine agnostic.

5. CMS stands for Compact Muon Solenoid, a general purpose particle physics detectors built on the Large Hadron Collider (LHC) at the Conseil Européen pour la Recherche Nucléaire (CERN)

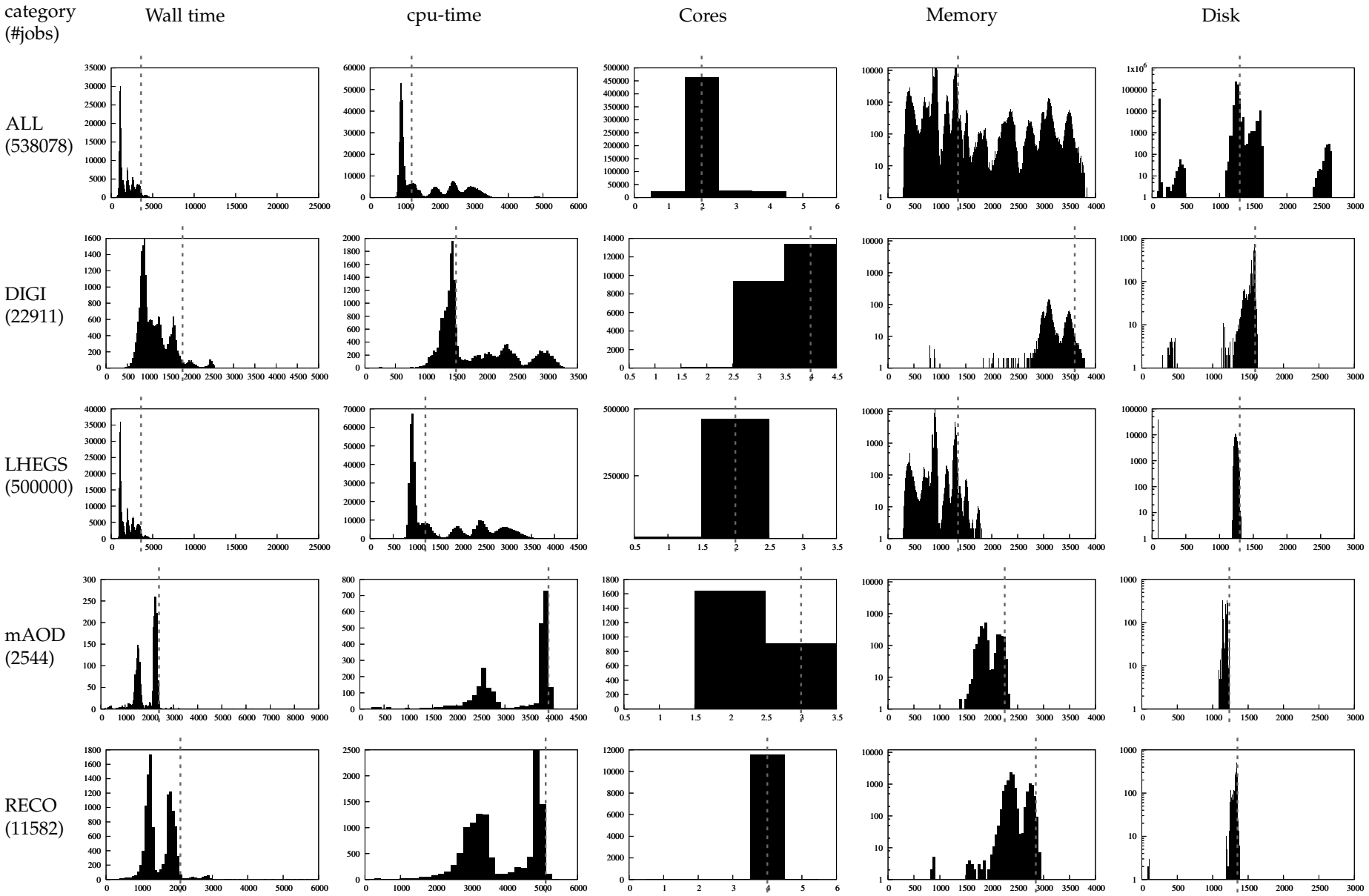


Fig. 5: Peak resource usage histograms for a CMS data analysis workflow.

Historical histograms for wall-time, cpu time, peak cores, memory usage, and disk space. (Other resources, such as average cores, omitted given space considerations.) Histograms are shown for the workflow as a whole (ALL), and split by user-provided categories (DIGI, LHEGS, mAOD, and RECO). A single distribution family cannot be used to describe the resources of a category, as the histograms have in general different shapes, are multi-modal, and have non-trivial tails. The dashed line shows the first-allocation for maximum throughput, to be introduced in Section 3.3, Equation 3.

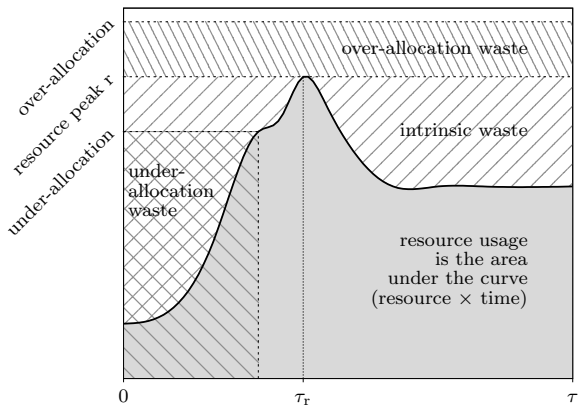


Fig. 6: Resource accounting model.

Resource values (thick black curve) are modeled as functions of time. Exact resource usage is found by the integral of the resource value curve, and has units of resource units × time. For simplicity, we do not consider exact resource usage, but instead focus on usage as a function of the peak value. This generates three kinds of waste: (1) intrinsic waste occurs for fixed allocation as the difference of the resource current usage to its peak. The more the average usage of the job approaches its peak, the less intrinsic waste. (2) over-allocation waste occurs when the allocation is larger than the job peak. (3) underallocation waste considers the resources wasted when the job fails given resource exhaustion.

box corresponds to the maximum resource peak allowed, while the other corresponds to execution time (i.e., wall-time). Allocations have three main purposes: (1) to homogenize the resources where jobs are executed (i.e., define *resource boxes* where jobs run), (2) to serve as input to schedulers to request appropriate allocations (i.e., create the resource boxes), and (3) to serve as input to the monitoring tool to terminate jobs when resources are exhausted (enforce the resource boxes). Note that these allocation computations occur *before* jobs are dispatched to the batch system.

As we mentioned before, precise requirements for a job's resource needs are generally unavailable or inaccurate. As seen in Fig. 6, we confront two situations: either resources are *overallocated*, wasting unused resources; or resources are *underallocated*, with all the resources allocated wasted given that the job fails with resource exhaustion, and further resources wasted as the job needs to be retried.

When allocating resources for a job, we assume that all dependencies of the job have been satisfied by the WMS, and it is ready to run. In the most general case, we can assume that all jobs belong to the same bag-of-jobs and share the same general properties. However, we have found that resource allocation can be improved if the user performs a coarse grouping of jobs into *categories* (e.g., simulation, analysis, visualization, etc.) that are known to have similar properties. Our further developments do not assume this category labeling, but they can take significant advantage of it, as we show in Section 4.

In practical terms, completely eliminating wasted resources is not possible, even if perfect information is available. Resource usage per job varies with time, and successful jobs executed on practical, static allocations, will almost surely have resources overallocated (referred to as *intrinsic waste* in Fig. 6). In general, there are two options: we can allocate the maximum resources available so that no job needs to be retried, or we can make smaller allocations initially, letting some jobs be retried (perhaps several times).

Next, we present a set of conservative simplifying assumptions that we call the *slow-peaks* model, which enables us to optimize job retries for minimum expected waste, or maximum expected throughput. The assumptions of the slow-peaks model are: (1) jobs do not modify their behavior according to the available resources; (2) resource exhaustion results in failure (e.g., no migration, checkpointing, nor resource expansion occurs); (3) if a job takes τ time to complete successfully with allocation r, and it exhausts its resources with allocation $\mathrm{r}' < \mathrm{r}$, the resource exhaustion is detected in the worst case at τ time. It is this last point that gives the name to the model, as we work under the worst-case assumption that resource peaks do not occur *fast*, but at the end of execution. As for the batch system, we assume that any resources not consumed by the job's resource allocation remain available to the underlying batch to be used by other jobs. In Section 3.4.4, we describe how this assumption can be accurately implemented on real, finite resources.

Note that (1) is a strong assumption, as for example, it assumes that wall-time is not affected by the amount of memory available, which is not true for jobs that would need to swap to disk. Our developments to date do not consider this interplay of resources.

3.4 Computing First Allocations

Now we come to the crux of the problem: how should we select the resources for the *first* execution of a job? We present solutions for two different objectives: Section 3.4.1 minimizes the amount of resources wasted, while Section 3.4.2 maximizes the throughput relative to resources consumed. In practice, both of these methods give similar values, so Section 3.4.3 explains when and why the two methods converge. We use the following notation throughout this section:

r	resource peak value, per job
a_i	the i^{th} allocation tried.
a_m	the largest allocation tried, usually the maximum resource peak across jobs, or the largest node available
τ	execution time, per job
$\bar{\tau}$	average execution time of all jobs
$\bar{\tau}_\mathrm{r}$	average execution time of jobs with resource peak r
$p(r,\tau) = p(r\|\tau)p(\tau)$	joint probability density of all resource peaks and execution times
$P(r)$	cumulative probability of the resource peaks

3.4.1 Minimizing Waste

Using static allocations and the slow-peaks model, we need two variables to describe the usage of a single resource per job: the *resource peak* r, which is the maximum usage value (e.g., peak resident memory, or peak disk storage used), and the *execution time* τ, which is the time it takes for the job to successfully complete (i.e. wall-time). We define resource usage as how many resource units are required over time: $\mathrm{usage}(\mathrm{r},\tau) = \mathrm{r}\tau$ (e.g., MB·s). Similarly, consider

an allocation sequence $a_1, a_2, \ldots, a_m$, with $a_i < a_{i+1}$, in which allocations are used in order until the job completes without exhausting resources. The resource waste for such allocation sequence is defined as:

$$\text{waste}(\mathrm{r}, \tau, a_i) = \begin{cases} (a_i - \mathrm{r})\tau, & \text{if } a_i \geq \mathrm{r} \\ a_i\tau + \text{waste}(\mathrm{r}, \tau, a_{i+1}) & \text{otherwise,} \end{cases}$$

with the assumption that all resource peaks are no larger than a_m. There are several ways in which waste can be minimized, but the common thread is to choose an appropriate total waste expression, and a sequence of allocations that minimize it. We describe how to minimize the *expected* value of total waste using a *two-step* policy. In the two step policy all jobs are first tried with some allocation a_1, and if that allocation is exhausted, then they are tried with the maximum allocation a_m available (say, the one corresponding to the largest computational node currently in the system). The two-step policy is simple to implement, but multi-step allocations might be also used. We conjecture that the larger the distribution tail, the better a multi-step allocation performs, but this remains an open problem that we have not evaluated.

Given allocations a_1, and a_m, we treat r and τ as random variables with joint probability density $p(\mathrm{r}, \tau)$, and find the expected waste to be:

$$\begin{aligned} \mathrm{E}[\text{waste}(\mathrm{r}, \tau, a_1)] = \int_0^\infty \Bigg(& \underbrace{\int_0^{a_1} (a_1 - \mathrm{r})\tau p(\mathrm{r}, \tau)\mathrm{d}\mathrm{r}}_{\text{first-allocation succeds}} \\ & + \underbrace{\int_{a_1}^{a_m} ((a_m + a_1 - \mathrm{r})\tau p(\mathrm{r}, \tau)\mathrm{d}\mathrm{r}}_{\text{final allocation succeds}} \Bigg) \mathrm{d}\tau \\ = & a_1 \underbrace{\int_{a_1}^{a_m} \int_0^\infty \tau p(\mathrm{r}, \tau)\mathrm{d}\tau\mathrm{d}\mathrm{r}}_{\text{mean wall-time for all jobs}} \\ & + a_m \int_{a_1}^{a_m} \underbrace{\int_0^\infty \tau p(\tau|\mathrm{r})\mathrm{d}\tau}_{\text{mean wall-time taks w. peak r}} p(\mathrm{r})\mathrm{d}\mathrm{r} \\ & - \underbrace{\int_0^\infty \int_0^\infty \mathrm{r}\tau p(\mathrm{r}, \tau)\mathrm{d}\tau\mathrm{d}\mathrm{r}}_{\text{resources effectively used}}, \end{aligned}$$

Using $\bar{\tau}$ for the mean wall-time across all jobs, $\bar{\tau}_r$ for the mean wall-time for jobs with peak r, converting integrals to summations, and probability densities to probability distributions, the allocation a_1 that minimizes waste is found with:

$$\text{argmin}_{a_1} \left\{ a_1\bar{\tau} + a_m \sum_{\mathrm{r}>a_1}^{a_m} \bar{\tau}_\mathrm{r} p(\mathrm{r}) \right\}. \quad (1)$$

Note that $\bar{\tau}$, $\bar{\tau}_\mathrm{r}$ and $p(\mathrm{r})$ can be easily computed by keeping running averages as historical data becomes available. In practice, we found it convenient to group neighboring peaks to reduce the number of averages kept (say every minute for time, or every 100MB for memory or disk). Further, note that the allocation is given only in terms of the resource, but not in terms of time. Assumption 3 of the slow-peaks model simply states that, in a worst-case scenario, if resource exhaustion occurs, it does so at the end of execution. This assumption is used in the minimization, but no time deadlines are enforced per job unless wall and cpu-times are considered as resources (see Section 3.4.4).

If we further assume that resource peaks r and wall-time τ are independent variables, we find the expected waste to be:

$$\begin{aligned} \mathrm{E}[\text{waste}(\mathrm{r}, \tau, a_1)] = a_1\bar{\tau} + a_m\bar{\tau} & \underbrace{\int_{a_1}^{a_m} p(\mathrm{r})\mathrm{d}\mathrm{r}}_{\text{approx. from histograms}} \\ & - \underbrace{\int_0^\infty \int_0^\infty \mathrm{r}\tau p(\mathrm{r}, \tau)\mathrm{d}\tau\mathrm{d}\mathrm{r}}_{\text{resources effectively used}}, \end{aligned}$$

and the minimum waste can be found by:

$$\text{argmin}_{a_1} \left\{ a_1 + a_m \sum_{\mathrm{r}>a_1}^{a_m} p(\mathrm{r}) \right\}. \quad (2)$$

It is interesting that wall-time does not appear in this expression. Even though we should not expect independence of r and τ to hold for every workflow, Equation 2 is simpler to compute as only one histogram needs to be kept, and performs extremely well in practice.

3.4.2 Maximizing Throughput

HTC facilities typically accumulate the quantity of resources consumed by a user and either charge it against some total quota, or charge the user money for total resources consumed. Therefore, we define *user-perceived throughput* as the amount of work completed relative to the resources consumed. This allows the quality of a job sizing algorithm to be evaluated independently of the size of the cluster, the batch system's scheduling algorithm, or other factors outside of the user's control. To this end, it is advantageous to think of workflows as consisting of an infinite number of jobs, running on an infinite, continuous pool of resources. This captures the steady state of a finite executing workflow, running on finite resources.

Given allocation a_1, throughput (#jobs/seconds) is given by:

$$\lim_{\#\text{jobs}\to\infty} \frac{\#\text{jobs completed with } a_1 + \#\text{jobs completed with } a_m}{\text{total time run at } a_1 + \text{total time run at } a_m}$$

Consider the throughput with a resource maximum a_m and first-allocation a_1. For jobs with peak $\mathrm{r} \leq a_1$, per every a_m units of the pool of resources, a_m/r jobs can run simultaneously in τ time. Conversely, if $\mathrm{r} > a_1$, then only a single job can be completed per every a_m, and this takes 2τ (one τ for failure under the slow-peaks assumption plus one τ for the job to successfully complete). If n_r is the number of jobs with peak r, and $\bar{\tau}_\mathrm{r}$ their mean wall-time, our throughput expression becomes:

$$\lim_{\#\text{jobs}\to\infty} \frac{(a_m/a_1)\sum_{\mathrm{r}\leq a_1} n_\mathrm{r} + \sum_{\mathrm{r}>a_1} n_\mathrm{r}}{\sum_{\mathrm{r}\leq a_1} \bar{\tau}_\mathrm{r} n_\mathrm{r} + 2\sum_{\mathrm{r}>a_1} \bar{\tau}_\mathrm{r} n_\mathrm{r}}.$$

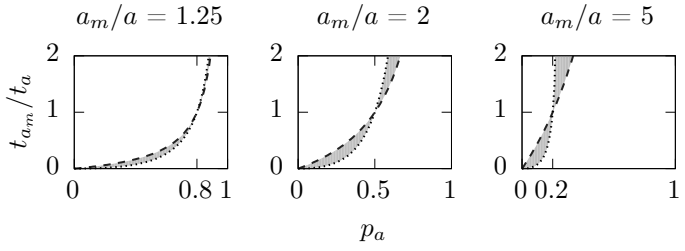


Fig. 7: Comparison of Min-Waste vs Max-Throughput
For a binomial distribution of resource sizes a and a_m with probabilities p_a and $1-p_a$, and wall-times t_a and t_{a_m}, the shaded regions show the value of the ratio t_{a_m}/t_a for which optimizing by minimum waste or maximum throughput yields different allocations. The dashed lines showed the minimum ratio for t_{a_m}/t_a for which it is advantageous to use first allocation a according to the minimum waste optimization. The dotted lines show the same information, but considering maximum throughput.

After arranging terms, and dividing each of the terms by the total number of jobs, the first-allocation that maximizes the expected throughput is:

$$\text{argmax}_{a_1} \left\{ \frac{(a_m/a_1)P(\text{r} \leq a_1) + P(\text{r} > a_1)}{\bar{\tau} + \sum_{\text{r}>a_1} \bar{\tau}_\text{r} p(\text{r})} \right\}, \qquad (3)$$

with cumulative probabilities $P(\text{r} \leq a) = \sum_{\text{r}\leq a} p(\text{r})$, and $P(\text{r} > a) = 1 - P(\text{r} \leq a)$. As in the case of minimum waste, all terms are easily computed from histograms that can be incrementally updated.

3.4.3 Comparing Min Waste and Max Throughput

As we will see in Section 4, the allocations computed using minimum waste or maximum throughput often yield the same first allocations in practice. The similarity is a function of how close the wall-times (i.e., the different $\bar{\tau}$) are across the different resources sizes (i.e., the different r). To explore this idea further, consider a simple workflow, which consists of two types of jobs: a small job, that uses a units of a resource, runs precisely for t_a seconds, and occurs p_a of the time; the other, a big job, uses $a_m > a$ units of a resource, runs precisely for t_{a_m}, and occurs $1 - p_a$ of the time. If we fix the ratio $a_m/a > 1$, can we compute the minimum ratio t_{a_m}/t_a for which using first allocation a becomes worth it?

In Fig. 7, the curves show the minimum value of t_{a_m}/t_a for which first trying allocation a is advantageous. The dashed line shows the values for minimum waste, while the dotted line for maximum throughput. The shaded areas show the sole regions where the two minimization methods give different results. Note that around $t_{a_m}/t_a = 1$, the result for both optimization methods is the most similar.

3.4.4 Practical Considerations

One aspect we have not directly addressed is any "penalty" for retries. Resubmitting a job still involves some use of resources, even when the job is not executing. This means that bytes written, and received by computational node when a job is dispatched should be accounted to the job, together with the adding the time to dispatch to the overall job wall-time.

Further, so far we have ignored internal fragmentation of resources. Equations 2, and 3 give the allocations for the minimum waste and maximum throughputs possible under the slow-peaks model. This is done by ignoring internal resource fragmentation, or in other words, by running on machines that perfectly fit the allocations computed. For a practical implementation we have two options: try to mimic the infinite pool of resources on real practical machines, or modify the expressions so that they deal directly with internal fragmentation. Mimicking the pool of infinite resources gets us closer to the theoretical extrema, but is not always practical. On the other hand, modifying the optimization expressions can always be done, but comes at the expense of increasing waste and decreasing throughput.

To mimic the infinite pool of resources, we need to take a closer look at a_m. There are not many restrictions on a_m, other than it has to be large enough to fit the largest resource peak of the workflow. Also, note that a single computational node may host several a_m allocations (e.g., a node with 16GB of memory, with a maximum allocation per job of 4GB). If a_1 exactly divides a_m, then all the computational nodes effectively act as a continuous pool of resources, with scheduling of jobs reducing, or eliminating internal fragmentation [25]. This can be achieved even when a_m occupies an entire node. The farther a_m is from a multiple of a_1, the farther we are from modeling the continuous pool of resources, and the more internal fragmentation is created.

For the second option, we can take internal fragmentation into account by not allowing fractions of jobs when computing throughput:

$$\text{argmax}_{a_1} \left\{ \frac{\lfloor a_m/a_1 \rfloor P(\text{r} \leq a_1) + P(\text{r} > a_1)}{\bar{\tau} + \sum_{\text{r}>a_1} \bar{\tau}_\text{r} p(\text{r})} \right\}.$$

For minimum waste, Equations 1 and 2 remain the same, but the (open) range of allocations $(a_m/2, a_m)$ is removed from the set of possible solutions.

Wall-time and cpu-time as resources deserve special considerations regarding first-allocations. Our model does not consider job checkpointing or migration, as it is common in some HPC systems. If we again think of allocations as boxes, an allocation for a resource such as memory defines a box for a job to run along two dimensions, memory and time. Enforcing first-allocations means that we can stack more boxes along the memory axis, while the time axis comes into play for minimizing waste or maximizing throughput. For wall-time and cpu-time, as resources, we only have the time axis, thus in general, enforcing such allocations is detrimental for both wasted resources and throughput. There are cases, however, where a scheduler may take advantage of information provided by the first-allocation computation. For example, without checkpointing or migration, wall-time first-allocations are better interpreted as guarantees of resource availability rather than limits to enforce on a job. Consider a system with queues for short and long running jobs[6]. The wall-time first-allocation can be used to decide whether submitting a new job to the short-running jobs' queue is appropriate, and to compute the throughput expected of running all jobs first allowing for retries in the

6. In an opportunistic/dedicated system, the limit for the short-running jobs' queue can be taken as the average time to eviction, while the dedicated resources act as the queue for long-running jobs.

	naive			brute-force		min. waste	max. through
resource	max. peak	$P(0.95 > \mathrm{r})$	$P(0.5 > \mathrm{r})$	min. waste	max. throug.	Equation 2	Equation 3
first allocation							
cores(cores)	5	3	2	2	2	2	2
cores_avg(cores)	2.9	1.5	0.8	1	1	1	1
memory(MB)	3830	2416	914	1350	1350	1350	1350
disk(MB)	2657	1338	1241	1300	1300	1300	1300
proportion of wasted resources per task							
cores	58%	34%	13%	13%	13%	13%	13%
cores_avg	70%	48%	64%	23%	23%	23%	23%
memory	72%	57%	62%	32%	32%	32%	32%
disk	55%	16%	53%	15%	15%	15%	15%
throughput normalized							
cores	1.00	1.58	2.18	2.18	2.18	2.18	2.18
cores_avg	1.00	1.74	1.41	2.69	2.69	2.69	2.69
memory	1.00	1.51	1.75	2.54	2.54	2.54	2.54
disk	1.00	1.88	1.07	1.91	1.91	1.91	1.91
percentage of tasks retried							
cores	0%	5%	9%	9%	9%	9%	9%
cores_avg	0%	5%	50%	7%	7%	7%	7%
memory	0%	5%	49%	8%	8%	8%	8%
disk	0%	5%	48%	6%	6%	6%	6%
overhead							
overhead (s)	—	—	—	5.05	5.37	0.40	0.40
538078 tasks read in 28.68 seconds							

Fig. 8: First-allocations for the CMS analysis workflow as a whole.
The fixed allocations are particular value of the distribution of jobs seen: max-peak is the maximum value seen, $P(0.95 < \mathrm{r})$ chooses the 95^{th} percentile, and $P(0.50 < \mathrm{r})$ chooses the median (50^{th} percentile). The rest of the allocations optimize for minimum waste or maximum throughput using the slow-peaks model. We show both brute-force computations considering every resource summary, and computations using Equations 2, and 3. The value of a_m, the maximum allocation used, was max-peak. Per allocation, we also show the average waste per job, the throughput as compared to the max-peak allocation, the percentage of jobs that were retried, and the time it took to compute the allocation. For histograms, we use partitions of 10-seconds for time, and one MB for disk and memory. Allocations were computed on an off-the-shelf 4-core workstation.

long-running jobs' queue. Thus, the first-allocation computation to be helpful as an analysis tool for characterizing and detecting inefficient workflow dispatches.

4 Evaluation

To evaluate our method, we apply it to data collected from production workflows runs on the HTCondor batch system described above. For each job in a workflow, we use the resource monitoring tool to capture the cores, memory, disk, etc. actually consumed by each job. Using this data, we use offline analysis to demonstrate: (1) there exist real applications that suffer from the job sizing problem; (2) our methods for computing minimum waste and maximum throughput are confirmed by producing equivalent results from a brute-force search of the solution space; and (3) these methods produce better results than simple statistical measures. Finally, we demonstrate the entire system operating online and show that it converges to accurate job sizes.

We evaluate these results on three different applications:

The application we show in the greatest detail is the **CMS-analysis** workflow already described above in Section 3.2 with complete histograms displayed in Fig. 5. It consists of 538,078 jobs in five categories (DIGI, LHEGS, mAOD, and RECO) labelled by the user.

CMS-simulation is another production high energy physics workflow used to generate simulated data for the CMS experiment. This was also executed using the Lobster workload manager on top of the HTCondor batch system at the University of Notre Dame. It consists of 30,630 jobs divided into three categories (ttW, ttZ, and MERGE) identified by the user. These tasks are largely computation-bound and have a lower degree of heterogeneity than CMS-analysis.

BWA-makeflow is a bioinformatics workflow build on the Burrows Wheeler Alignment (BWA) tool [26]. It consists of 826 jobs, divided in three steps: split (2 jobs), analysis (822 jobs), and join (2 jobs). The split jobs subsample a 30GB input file into 822 parts, each of which is fed into an analysis job that uses BWA to query a reference database of about 30 MB. The results of each analysis job (approximately 23 MB each) are then fed into the join jobs which merge and reconcile the results.

4.1 Offline Analysis

The collection of resource measurements was used to perform an offline evaluation of various methods of job sizing for CMS-analysis which are shown in Fig. 8. (This table includes all tasks together without distinguishing between categories.) Each column indicates a different method of job sizing: in order, always using the maximum peak value, always using the 95^{th} percentile, always using the 50^{th} percentile, brute-force searches for the minimum waste and maximum throughput, and finally our solution using Equations 2 and 3 for minimum waste and maximum throughput. The first group of rows indicates the values of cores, memory, and disk selected by each method. Following groups indicate the percent of wasted resources of each type,

the throughput relative to the max-peak method, and the percentage of tasks retried.

We can highlight a number of points from Fig. 8:

First, the max-peak method wastes 70%, 72%, and 55% of cores, memory, and disk (respectively) compared to 23%, 32%, and 15% of the maximum throughput method. The other naive methods of picking job sizes at the 50^{th}, or 95^{th} percentiles offer higher throughput than max-peak, but still have considerable waste. Some amount of this waste is due to the fact that all tasks are evaluated together without distinguishing between categories; we will show later the effect of analyzing categories separately.

Gains in throughput are normalized to the maximum per resource. The more uniform the resource value across jobs, the less advantageous this strategy becomes. For example, compare the memory and disk resources, with an original 72% and 55% of waste, respectively. Using the maximum throughput first-allocations, waste is reduced to 32% and 15%, with a throughput increase of 2.54 and 1.91 respectively. However, note that we can only say that a first-allocation strategy will have at least a 1.91X increase in throughput, but this does not mean that the disk is the computational bottleneck. Which resource is the bottleneck depends on the computational site.

The brute force method confirms the minimum-waste and maximum-throughput equations, although it is likely an expensive approach to use in practice. Using counts in the histograms, first-allocations can be computed using fewer data points, and in linear time. In contrast, brute-force uses every resource summary to find the minimum waste and maximum throughput in a quadratic time computation. We used histograms with divisions at 30s for time, and 50MB for disk and memory. These small divisions were chosen to show that the more efficient method generates the same allocation as brute-force calculations. In production, differences in the order of seconds, or megabytes are most likely irrelevant, and larger divisions can be used to filter some of the noise from the measured data.

Finally, note that minimum waste and maximum throughput yield the same results for this particular workflow, but as noted in Section 3.4.3, they are not guaranteed to be the same.

4.2 Category Analysis

We applied the same techniques to all three workflows, but in the interest of space we show a smaller set of results in Fig.s 9, 10, and 11 for the CMS-analysis, CMS-simulation, and BWA-makeflow workloads, respectively. Each table shows the number of tasks in each category and the first allocation for memory selected by the max-peak, min-waste, and max-throughput methods for each separate category. These tables show the effect of exploiting category information provided by the user, rather than simply treating the entire workflow as a bag of equivalent tasks.

For example, Fig. 9 shows the effect of categorization in CMS-analysis. We observe an immediate decrease in waste and a very noticeable increase in throughput when using categories, even for fixed policies. Note that the percentage of retries becomes much more manageable, even with some slight increases for some columns. Similarly, not shown in the figure, the number of retries for wall-time decreased

category	count	max. peak	min. waste	max. th.
digi	22911	3830	3600	3600
merge	1041	2289	1700	1700
mAOD	2544	2311	2250	2250
lhegs	500000	2192	1350	1350
reco	11582	2910	2850	2850
(all)	538078	3830	1350	1350
proportion wasted resources				
with cats.	—	51.9%	27.2%	27.2%
w/o cats.	—	71.8%	31.5%	31.5%
throughput				
with cats.	—	1.70	2.65	2.65
w/o cats.	—	1.00	2.54	2.54
retries				
with cats.	—	0%	<1%	<1%
w/o cats.	—	0%	7.8%	7.8%

Fig. 9: Memory Allocations (MB) for CMS-analysis

category	count	max. peak	min. waste	max. th.
ttW_mAODv2	10161	2000	1900	2000
merge	1587	900	850	850
ttZ_mAODv2	18882	2000	1900	2000
(all)	30630	2000	1900	2000
proportion wasted resources				
with cats.	—	14.1%	14.5%	14.0%
w/o cats.	—	15.9%	16.1%	15.9%
throughput				
with cats.	—	1.09	1.09	1.10
w/o cats.	—	1.00	1.00	1.00
retries				
with cats.	—	0%	3.9%	<1%
w/o cats.	—	0%	3.8%	0%

Fig. 10: Memory Allocations (MB) for CMS-Simulation

category	count	max. peak	min. waste	max. th.
Join	18	4	4	4
Split	18	1304	50	50
Analysis	7398	321	300	300
(all)	7434	1304	300	300
proportion wasted resources				
with cats.	—	21.0%	16.2%	16.2%
w/o cats.	—	78.9%	20.6%	20.6%
throughput				
with cats.	—	4.38	4.56	4.56
w/o cats.	—	1.00	4.15	4.15
retries				
with cats.	—	0%	1.4%	1.4%
w/o cats.	—	0%	1.4%	1.4%

Fig. 11: Memory Allocations (MB) for BWA-makeflow

from 88% to 13% when using categories. This results highlight the importance of three concepts for an efficient use of resources: historical data to create resource allocations, monitoring to enforce such allocations, and coarse categorical knowledge about the workflow from the user.

In Fig. 10, we show the first-allocations for CMS-simulation. This figure highlights the advantages of having some coarse information regarding the workflow: without categories, the effect of first-allocation is negligible. However, when labeling and processing jobs based on category, an increase in throughput of about 7% is seen. Further, not shown in the figure, the 95^{th} percentile allocation has a throughput of 0.97, and wasted more resources (18%)

		fixed max.	online no cats.	online + cats.	oracle prediction
CMS ana.	cores	1.00	2.18	2.24	2.51
	cores_avg	1.00	2.69	2.76	3.60
	memory	1.00	2.54	2.62	4.00
	disk	1.00	1.91	1.97	4.13
CMS simul.	cores	1.00	1.94	1.95	1.99
	cores_avg	1.00	1.10	1.11	1.19
	memory	1.00	1.00	1.12	1.42
	disk	1.00	8.89	9.09	32.33
BWA	cores	1.00	1.97	2.03	2.05
	cores_avg	1.00	1.00	1.03	1.11
	memory	1.00	4.15	6.15	7.32
	disk	1.00	51.30	55.88	60.96

Fig. 12: Summary of Results

first-allocation	waste	throughput	retries	overhead
fixed				
max-peak	highest	lowest	none	none
$P(r < 0.95)$	moderate	low	least	none
$P(r < 0.50)$	high	low	highest	none
brute-force				
min waste	lowest	moderate	moderate	high
max th.	moderate	highest	most	high
slow-peak online histograms				
min waste Eq. 2	lowest	moderate	moderate	negligible
max th. Eq. 3	moderate	highest	most	negligible

Fig. 13: Rules of Thumb

as compared to the maximum peak allocation. Similarly, there is a reduction of throughput (0.98) and increase in waste (18%) for the allocation computed using Equation 2 (not shown in figure), in which the dependency on time is eliminated. This indicates that a fixed first allocation that simply tries to handle resource peak outliers may be detrimental to the execution of the workflow.

Fig. 11 shows the results for memory first-allocations per category and with the workflow as a whole. Observe that by simply dividing by categories, we get an increase in throughput of at least 4.38 times, or 4.56 times by allowing 1.4% of the jobs to be retried. Fig. 11 also highlights the coarse information regarding the workflow, as the split category shows a significant difference between the maximum peak and first-allocations computed.

In Fig. 12 summarizes the effect of these techniques on each of the three workloads, compared to a perfect "oracle" prediction which knows in advance whether the job will succeed or fail within a given job size. Note that even though online computations give a noticeable gain compared to the fixed maximum peak allocation, in some cases they are somewhat far from the value of oracle predictions. This leaves the door open for improvements, as we discuss in Section 6.

Finally, in Fig. 13 shows a general comparison for the different first-allocation policies. Note that these are general guidelines, as the actual values for waste and throughput are a function of the workflow's resource distributions.

4.3 Online Execution

The results so far have been computed offline, taking advantage of a complete recorded history of resource consumptions. This allows us to use a maximum observed value of a resource (i.e., a_m in Equations 2 and 3), and to precompute first-allocations before any jobs of a workflow is actually executed. In a more general setting, such historical information is not available, and first-allocations need to be computed during execution as partial information from the workflow becomes available.

To implement these techniques online, we proceed as follows:

1) Set the maximum allowable value a_o for the resource. For example, this could be the size of the largest computational node at the site.
2) Run jobs allocating a_o units of the resource until some statistics converge. (For example, wait for some convergence of the resource usage variance when at least n jobs are completed.)
3) Record the maximum resource value observed, and set a_m to it.
4) Compute the first-allocation with the historical data now available.
5) New jobs are deployed using the computed first-allocation. On resource exhaustion, jobs are retried using a_m.
6) If a jobs exhaust resources using a_m, then go to step 2.

In Fig. 14 we show the first-allocation computations for the first 12 hours of the CMS-analysis workflow. The maximum computational node considered was 6 cores, with 6 GB of RAM and 12 GB of disk. Observe the spike back to the maximum allowed size at about 6 hours, which correspond to a new job resource maximum usage observed. In Fig. 15 we show the time-series for the total of cores, memory and disk, both committed and used for the 11 days the workflow ran. The gaps in time correspond to manual restarts activated by the user. The dynamic range of the committed resources responds in great part to the availability of resources in our opportunistic setting using HTCondor. At the bottom of Fig. 15.d we include a normalization to memory committed resources. Note that the expected usage stabilizes at about 68% (or 32% wasted resources), that follows closely the one obtained previously when historic data was available (Fig. 8). This shows the scalability of our intuition regarding allocations: do not compute how many allocations, but rather, how big they should be.

5 Related Work

Managing systems with limited intervention of system administrators is the goal of autonomic computing, which has been used to address various problems related to self-healing, self-configuration, self-optimization, and self-protection in distributed systems. For instance, the provisioning of virtual machines is studied in [27] and an approach to tackle service overload, queue starvation, the "black hole" effect and job failures is sketched in [28]. An autonomic feedback-loop process to quantify incident degrees of workflow executions from metrics measuring performance-related issues is proposed in [29]–[31]. Although the process constantly monitors workflow executions to perform corrective actions, it does not tackle the resource provisioning problem.

Monitoring systems, such as Ganglia [32], Nagios [33], and Munin [34], offer a system administrators a wealth of information for troubleshooting at a site-level, but they are

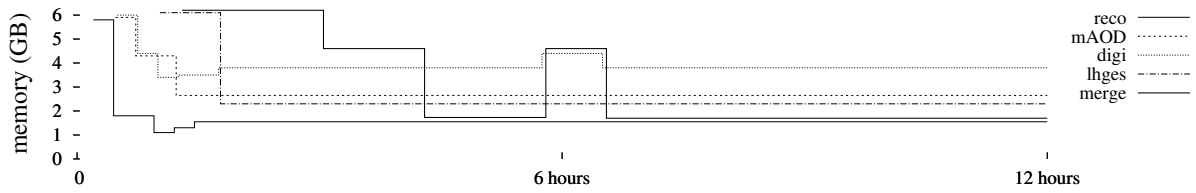


Fig. 14: Computing first-allocations with partial information for the HEP CMS analysis workflow.
The first-allocations for memory per category for the first 12 hours of a run of 11 days are shown. The spike around 6 hours corresponds to a new job resource usage maximum being observed.

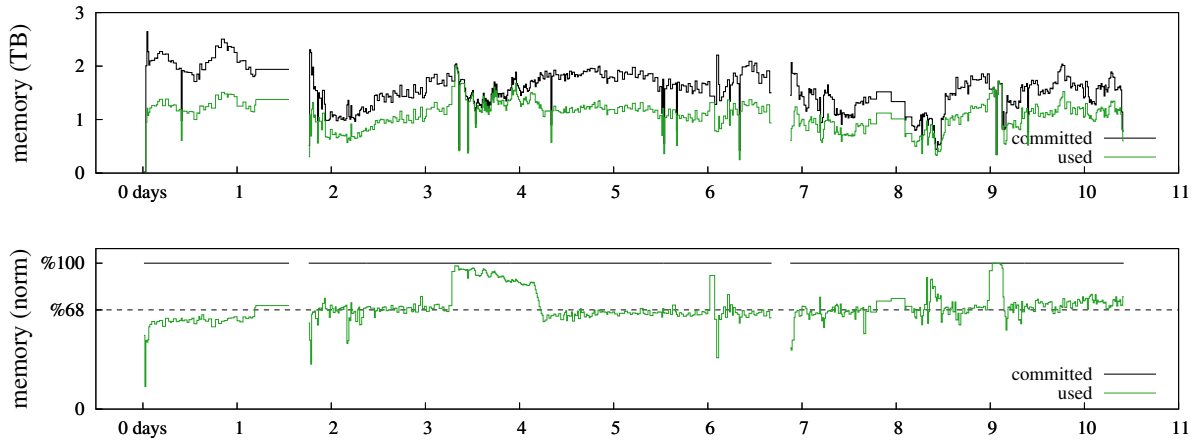


Fig. 15: Time-series for committed and used memory for the HEP CMS analysis workflow.
The great variation in committed resources corresponds to the variation of the availability of opportunistic resources in our campus HTCondor pool. Note, however, that the percentage of used resources remains stable, at about 68%.

not available to unprivileged users, and provide only coarse job resource usage information. Site specific systems, such as TACC_Stats [35] and NCAR [36], collect resource usage for HPC workloads as time series allowing for an analysis per job. `ParaTrac` [37] focuses on monitoring data-intensive workflows by interposing all I/O operations via an overlay file system implemented using FUSE [38]. Full resource workflow profiles are also generated by polling information from `procfs, taskstats` [39].

Our developments focused on workflows that consist of bag of jobs. In a study by [8], several characteristics of general workflows in the cloud are observed that are amenable to our approach: workflows usually consist of bag of jobs, with several resources to be allocated and enforced, heterogeneity is managed by grouping jobs in bag (e.g. categories), the distributions tend to be heave tailed, and the number of available resources is highly dynamic. Labeling of categories is sometimes done implicitly, such as in MapReduce applications, in which it is understood that mappers and reducers have different resource requirements. For workflows running on grids [40], users are often able to label the stages of a multi-step job, such as retrieving data from an instrument or database, running an analysis, and extracting statistics.

In this paper we did not touch on scheduling jobs. Scheduling algorithms usually assume that resource information is available [14]–[17]. For example, back-filling algorithms [41], follow a first-come-first-serve policy, that uses a priori job sizes and execution times to schedule in partially available resources. Our first-allocation computations can be used as input to such algorithms, to take advantage of particular site architectures, or respect given site policies, such as [42], [43]. Along these lines, [44] uses machine learning techniques to predict the wall-time of a job. Unlike our approach, where a user minimizes waste or maximizes throughput for jobs in a workflow, the objective of [44] was to improve job slowdowns of a batch system using a back-filling algorithm. One particular issue that we do not handle is the resource waste given the external fragmentation in a computational node. Such fragmentation may be reduced using schedulers such as [45].

6 Conclusions and Future Work

This paper introduced a feedback-loop for efficient use of computational resources for scientific workflows. We dealt with computational resources descriptions in three stages: an historical archive of past job executions, monitoring/enforcement of current job executions, and allocations for provisioning future executions. We presented strategies to minimize the expected waste, or maximize the expected throughput and applied them to real production scientific workflows running on $O(25K)$ opportunistic cores.

There are several ways in which the job sizing problem could be further explored:

Our results are based on a conservative accounting of resources called the *slow-peaks model*. The model is based on a worst-case scenario, in which a job resource exhaustion occurs at the time the job would have completed successfully.

This assumes that the resource exhaustion can be detected, which is not a trivial problem in itself [18] and that the job does not modify its behavior according to the resources available. A more sophisticated analysis could address jobs where one resource selection affects another.

The manner in which we account for the time it takes to transfer files to a computational node deserves further exploration. In our current implementation, such times can be easily accounted as part of the total running time. However, this does not accurately describe scenarios where transfer of one job can be overlapped with the computation of another.

Finally, we have evaluated a two-step process of making a first allocation, then jumping to a maximum allocation on failure. We conjecture that a multi-step process might have further advantages on distributions with very long tails. This requires evaluating both the number of steps as well as the strategy for increase between steps.

ACKNOWLEDGEMENTS

This work was funded by DOE under the contract number ER26110, "dV/dt - Accelerating the Rate of Progress Towards Extreme Scale Collaborative Science".

REPRODUCIBILITY

The data used in this paper, the waste minimization and throughput maximization computations (with C and python interfaces), are available at:

`https://github.com/cooperative-computing-lab/efficient-resource-allocations`

The resource monitoring tool, the resource feedback loop, and the min-waste and max-throughput techniques are implemented in the Makeflow workflow management software, available here:

`https://ccl.cse.nd.edu/software/makeflow`

REFERENCES

[1] J. Montagnat *et al.*, "Workflow-based comparison of two distributed computing infrastructures," in *2010 5th Workshop on Workflows in Support of Large-Scale Science (WORKS)*. IEEE, 2009, pp. 1–10.

[2] O. A. Ben-Yehuda *et al.*, "Expert: Pareto-efficient task replication on grids and a cloud," in *2012 IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2007, pp. 167–178.

[3] H. Arabnejad *et al.*, "Fairness resource sharing for dynamic workflow scheduling on heterogeneous systems," in *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA)*. IEEE, 2012, pp. 633–639.

[4] D. Poola *et al.*, "Enhancing reliability of workflow execution using task replication and spot instances," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 10, no. 4, p. 30, 2015.

[5] 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.

[6] I. Casas *et al.*, "A balanced scheduler with data reuse and replication for scientific workflows in cloud computing systems," *Future Generation Computer Systems*, 2016.

[7] N. Zakay and D. G. Feitelson, "On identifying user session boundaries in parallel workload logs," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2012, pp. 216–234.

[8] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12. New York, NY, USA: ACM, 2012, pp. 7:1–7:13.

[9] R. Ferreira da Silva *et al.*, "Toward fine-grained online task characteristics estimation in scientific workflows," in *8th Workshop on Workflows in Support of Large-Scale Science*, 2013.

[10] I. Sfiligoi, "Estimating job runtime for cms analysis jobs," in *Journal of Physics: Conference Series*, vol. 513, no. 3. IOP Publishing, 2014, p. 032087.

[11] R. Ferreira da Silva *et al.*, "Online task resource consumption prediction for scientific workflows," *Parallel Processing Letters*, vol. 25, no. 3, 2015.

[12] ——, "Characterizing a high throughput computing workload: The compact muon solenoid (CMS) experiment at LHC," *Procedia Computer Science*, vol. 51, pp. 39–48, 2015.

[13] D. Thain, T. Tannenbaum, and M. Livny, "Distributed Computing in Practice: The Condor Experience," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[14] J. Blythe *et al.*, "Task scheduling strategies for workflow-based applications in grids," in *5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, may 2005.

[15] T. D. Braun *et al.*, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 61, no. 6, pp. 810–837, jun 2001.

[16] H. Casanova *et al.*, "Heuristics for scheduling parameter sweep applications in grid environments," in *9th Heterogeneous Computing Workshop*, 2000.

[17] A. Mandal *et al.*, "Scheduling strategies for mapping application workflows onto the grid," in *14th IEEE International Symposium on High Performance Distributed Computing*, 2005.

[18] G. Juve, B. Tovar, R. F. da Silva, D. Krol, D. Thain, E. Deelman, W. Allcock, and M. Livny, "Practical Resource Monitoring for Robust High Throughput Computing," in *Workshop on Monitoring and Analysis for High Performance Computing Systems Plus Applications at IEEE Cluster Computing*, 2015.

[19] "ltrace," http://ltrace.org.

[20] J. Keniston *et al.*, "Ptrace, utrace, uprobes: Lightweight, dynamic tracing of user apps," in *Ottowa Linux Symposium*, 2007.

[21] 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.

[22] "CCTools," http://www3.nd.edu/ ccl/software/download.

[23] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, "Condor – a distributed job scheduler," in *Beowulf Cluster Computing with Linux*, T. Sterling, Ed. MIT Press, October 2001.

[24] 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.

[25] C. Bays, "A comparison of next-fit, first-fit, and best-fit," *Commun. ACM*, vol. 20, no. 3, pp. 191–192, March 1977.

[26] D. Peters, X. Luo, K. Qiu, and P. Liang, "Speeding up large-scale next generation sequencing data analysis with pbwa," *J. Biocomput.*, vol. 1, no. 1, 2012.

[27] N. Van *et al.*, "Autonomic virtual resource management for service hosting platforms," in *ICSE Workshop on Software Engineering Challenges of Cloud Computing*, 2009, pp. 1–8.

[28] P. Collet *et al.*, "Issues and scenarios for self-managing grid middleware," in *2nd workshop on Grids meets autonomic computing*. ACM, 2010, pp. 1–10.

[29] R. Ferreira da Silva *et al.*, "Self-healing of workflow activity incidents on distributed computing infrastructures," *Future Generation Computer Systems*, vol. 29, no. 8, pp. 2284–2294, 2013.

[30] ——, "Controlling fairness and task granularity in distributed, online, non-clairvoyant workflow executions," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 14, pp. 2347–2366, 2014.

[31] ——, "Characterizing a high throughput computing workload: The compact muon solenoid (CMS) experiment at LHC," *Procedia Computer Science*, vol. 51, pp. 39–48, 2015.

[32] M. L. Massie *et al.*, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, jul 2004.

[33] "Nagios," http://nagios.org.

[34] "Munin," http://munin-monitoring.org.

[35] C.-D. Lu *et al.*, "Comprehensive job level resource usage measurement and analysis for xsede hpc systems," in *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery (XSEDE)*, 2013.

[36] D. D. Vento *et al.*, "System-level monitoring of floating-point performance to improve effective system utilization," in *Supercomputing*, 2011.
[37] N. Dun *et al.*, "Paratrac: A fine-grained profiler for data-intensive workflows," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010.
[38] "Fuse: Filesystem in userspace," http://fuse.sourceforge.net/.
[39] "taskstats," http://www.kernel.org/doc/Documentation/accounting/taskstats.txt.
[40] I. Taylor *et al.*, *Workflows for e-Science: Scientific Workflows for Grids*. Springer, 2007.
[41] D. A. Lifka, "The ANL/IBM SP scheduling system," in *Job Scheduling Strategies for Parallel Processing, IPPS'95 Workshop, Santa Barbara, CA, USA, April 25, 1995, Proceedings*, 1995, pp. 295–303.
[42] S. Bardhan and D. A. Menascé, "Predicting the effect of memory contention in multi-core computers using analytic performance models," *IEEE Trans. Computers*, vol. 64, no. 8, pp. 2279–2292, 2015.
[43] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *SIGOPS European Conference on Computer Systems (EuroSys)*, Prague, Czech Republic, 2013, pp. 351–364.
[44] E. Gaussier, D. Glesser, V. Reis, and D. T. Denis, "Improving backfilling by using machine learning to predict running times," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15, 2015, pp. 64:1–64:10.
[45] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 455–466, Aug. 2014.

Benjamin Tovar is a research software engineer at the University of Notre Dame. In his current role, he is the lead maintainer of CCTools, a suite of tools to quickly enable scientist the use distributed, high-throughput computing. Prior to his position at Notre Dame, he was a Postdoctoral fellow in the area of control engineering in robotics at Northwestern University, and he received a Ph.D. in Computer Science from the University of Illinois Urbana-Champaign, where he studied algorithmic modeling for robotics.

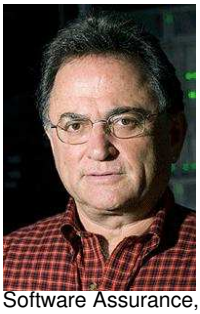
Rafael Ferreira da Silva is a Research Assistant Professor at the USC Computer Science Department, and a Computer Scientist at the USC Information Sciences Institute (ISI). His research focuses on the efficient execution of scientific workflows on heterogeneous distributed systems (e.g., clouds, grids, and supercomputers), computational reproducibility, and Data Science?workflow performance analysis, user behavior in HPC/HTC, and citation analysis (for publications). Dr. Ferreira da Silva received his PhD in Computer Science from INSA-Lyon, France, in 2013.

Gideon Juve worked as a Computer Scientist in the Science Automation Technologies group at the USC Information Sciences Institute. He received his BS, MS and PhD degrees in Computer Science from USC in 2004, 2008, and 2012. His research focused on enabling and optimizing large-scale, data-intensive scientific workflows on clusters, grids and clouds. He now works on the automation team at SpaceX.

Ewa Deelman is a Research Professor at the USC Computer Science Department and a Research Director, at the USC Information Sciences Institute (ISI). Dr. Deelman's research interests include the design and exploration of collaborative, distributed scientific environments, with emphasis on automation of scientific workflows, management of computing resources, and management of scientific data. Her work involves close collaboration with researchers from a wide spectrum of disciplines. At ISI she leads the Science Automation Technologies group, responsible for the development of the Pegasus Workflow Management software. She founded the annual Workshop on Workflows in Support of Large-Scale Science, held in conjunction with the Super Computing conference. In 1997, Dr. Deelman received her PhD in Computer Science from the Rensselaer Polytechnic Institute.

William E. Allcock is since 2006 a Senior Storage Engineer in High Performance Computing at the Argonne Leadership Computing Facility (ALCF), where he holds overall responsibility for all I/O and storage-related activities, as well as integration of the systems software stack for the ALCF. He joined the Distributed Systems Laboratory at Argonne in 2000 where he developed technology required for computational grids and served as a senior member. He was a driving force behind the Globus project. Allcock served as liaison on many international grid projects, including the prestigious Earth System Grid.

Douglas Thain is an Associate Professor in the Department of Computer Science and Engineering at the University of Notre Dame. He received the B.S. in Physics from the University of Minnesota - Twin Cities and the M.S. and Ph.D. in Computer Sciences from the University of Wisconsin - Madison, where he contributed to the Condor distributed computing system. At Notre Dame, he works closely with researchers in multiple fields of science and engineering to attack scientific problems using large scale computing. His research team creates and publishes open source software that is used around the world to harness large scale computing systems such as clusters, clouds, and grids.



Miron Livny is a Professor of Computer Science at the University of Wisconsin-Madison, Principal Scientist at Core Computational Technology of the Wisconsin Institutes for Discovery, Chief Technology Officer of the Wisconsin Institutes for Discovery, Director of the UW Center for High Throughput Computing (CHTC), Director of the Software Assurance Marketplace and the Technical Director of the Open Science Grid (OSG). His research interests include: Distributed Processing Systems, High Throughput Computing, Software Assurance, Cyberinfrastructure