Adaptive Task-Oriented Resource Allocation for Large Dynamic Workflows on Opportunistic Resources

Thanh Son Phung
University of Notre Dame
tphung@nd.edu

Douglas Thain
University of Notre Dame
dthain@nd.edu

***Abstract*—Dynamic workflow management systems offer a solution to the problem of distributing a local application by packaging individual computations and their dependencies on-the-fly into tasks executable on remote workers. Such independent task execution allows workers to be launched in an opportunistic manner to maximize the current pool of resources at any given time, either through opportunistic systems (e.g., HTCondor, AWS Spot Instances), or conventional systems (e.g., SLURM, SGE) with backfilling enabled, as opposed to monolithic or message-passing applications requiring a fixed block of non-preemptible workers. However, the dynamic nature of task generation presents a significant challenge in terms of resource management as tasks must be allocated with some *unknown* amount of resources pre-execution but are only observable at runtime. This in turn results in potentially huge resource waste per task as (1) users lack direct knowledge about the relationship between tasks and resources, and thus cannot correctly specify the amount of resources a task needs in advance, and (2) workflows and tasks may exhibit stochastic behaviors at runtime, which complicates the process of resource management.**

In this paper, we (1) argue for the need of an adaptive resource allocator capable of allocating tasks at runtime and adjusting to random fluctuations and abrupt changes in a dynamic workflow without requiring any prior knowledge, and (2) introduce Greedy Bucketing and Exhaustive Bucketing: two *robust, online, general-purpose, and prior-free* allocation algorithms capable of producing quality estimates of a task's resource consumption as the workflow runs. Our results show that a resource allocator equipped with either algorithm consistently outperforms 5 alternative allocation algorithms on 7 diverse workflows and incurs at most 1.6 ms overhead per allocation in the steady state.

I. Introduction

Dynamic workflow management systems are becoming increasingly prevalent in supporting and executing large-scale scientific and data analytic computations [1]–[3]. This is due to several reasons, most notably (1) tasks' definitions and dependencies are generated and inferred at runtime, thus removing the need for declaring a static DAG in advance, (2) local computations are automatically translated and packaged into tasks to be executed on remote workers, thus removing unnecessary efforts and frustrations, and 3) workers can be deployed opportunistically to maximize available resources.

Since each task is executed independently from each other, workers can be deployed opportunistically to maximize the available pool of resources at any given time during a workflow execution, as opposed to monolithic or message-passing applications requiring a deployment of a fixed block of non-preemptible workers and increasing the user's wait time in the batch queue. From the perspective of an administrator, such opportunistic worker deployment also increases the resource utilization of the local HPC facility, as workers can be deployed by submitting many small pilot jobs to take advantage of the backfilling strategy commonly seen in large batch systems (e.g., HTCondor [4], SLURM [5], SGE [6]) and utilize unused resources as they become available over time. Furthermore, large cloud vendors have been offering opportunistic resources in their data centers at an extremely low cost (up to 91% discount) [7]–[10], thus greatly reducing the monetary barrier and opening the door to practical utilization of a huge amount of opportunistic computational resources with little cost and changes to the worker deployment configuration code.

Specifying resources for each task (e.g., cores, memory, disk) is crucial to the efficiency and performance of a workflow as it limits the waste of resources during a task run and helps the underlying execution system make better scheduling decisions [11]. However, the inherent dynamicity of this class of workflow system poses a significant *dilemma* to the process of resource allocation: **tasks must first be specified with some unknown amount of resources in order to be scheduled for deployment and execution, but the optimal amount of resources is only visible to the workflow manager upon task completion.** This problem is further exacerbated as workflows may change over each run, reflecting the evolution of application logic, modifications to input data, updates to underlying software libraries, or random external factors affecting the state of a workflow system at any given time. Individual tasks may also differ in resource consumption between runs if they are inherently stochastic. The combination of dynamicity and stochasticity of workflows and tasks thus makes the problem of resource allocation challenging.

In this paper, we argue that an allocation algorithm X can only be considered a complete solution to the above challenge by addressing all stated problems, and thus following these 4 design goals:

- **General-purpose**: X should be able to run generically with any dynamic workflow without relying on any workflow- or task-specific feature.
- **Prior-free**: X should not rely on past information of a

workflow (e.g., previous traces/logs) to allocate resources for the current run. Many works [12]–[14] apply machine learning techniques to customize solutions to specific applications. These techniques however are costly to train, prone to overfit, and vulnerable to substantial stochastic changes in workflows.

- **Online**: since dynamic workflows generate tasks at runtime instead of having a DAG in advance, *X* must operate in an online manner, i.e., be able to collect information and predict resource allocations as the workflow runs.
- **Robust**: *X* must be able to perform well under a variety of distributions and unexpected changes to workflows when compared to related works [11], [15].

We hence introduce two resource allocation algorithms, **Greedy Bucketing** and **Exhaustive Bucketing**, that attempt to minimize the expected resource waste of all tasks in a workflow. Each algorithm (1) models the expected resource waste of tasks in a workflow, (2) collects a list of resource records of completed tasks, and (3) carefully extracts a potential resource specification from this list to allocate subsequent tasks in the workflow. This design thus makes these algorithms *general-purpose* (no task-specific feature is used), *prior-free* (only information about completed tasks in the current workflow run is collected), *online* (resource prediction is derived on demand), and *robust* (resource prediction changes as the workflow changes its behavior during its run.)

We use two production workflows, ColmenaXTB [16] and TopEFT [17], and further generate a diverse set of 5 synthetic workflows following 5 different resource distributions (*Normal*, *Exponential*, *Uniform*, *Bimodal*, *Phasing Trimodal*) to evaluate the robustness and performance of these allocation algorithms. Our results obtained from running 7 workflows with 20-50 workers deployed opportunistically on a local HTCondor cluster show that the bucketing algorithms consistently outperform 5 alternative algorithms and can reach as high as 96% resource efficiency and incur at most 1.6ms overhead per allocation prediction in the steady state, thereby substantially reducing the resource waste and increasing the resource efficiency on a variety of workflow's behaviors.

II. Problem Framework

A. Background

Figure 1 shows the general architecture of a dynamic workflow system. The entire software stack including the application runs on the manager node, whereas computations (e.g., functions) packaged as tasks are executed at remote workers. At runtime, an application calls possibly thousands of expensive functions, each of which is labeled as remote/asynchronous and prepared for remote execution. Functions are then packaged as tasks by marshalling its arguments, detecting and packaging library dependencies, wrapping error handling code, etc., and sent to the workflow manager, which resolves tasks' dependencies by constructing a dependency graph between tasks and passes ready tasks to the task scheduler. The task scheduler upon receiving a ready task will provision some

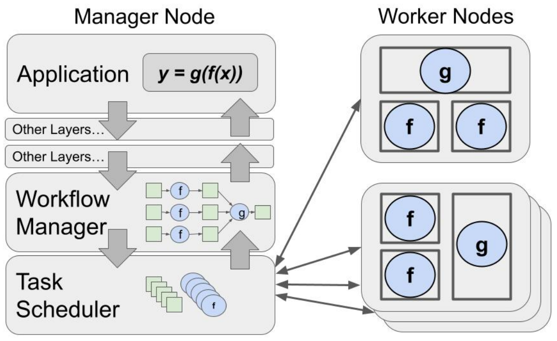


Fig. 1: Dynamic Workflow System Architecture
Distributing a local application using a dynamic workflow system involves detecting, resolving, and packaging tasks and their dependencies at runtime. Ready tasks are then sent to workers to execute and results are sent back to the application.

amount of resources to the task and send it to an available worker to be executed. A worker then allocates the specified portion of its resources to the task, executes it, records its resource consumption, and takes proper actions if the task exceeds its allocation. Finally, the result of the execution is returned transparently through the software stack to the application. **Note that this paper addresses the problem of resource allocation to tasks which happens at dispatch time - after task dependencies are resolved and before tasks are scheduled to workers for execution.**

B. Definitions and Assumptions

Our problem space then requires the definitions of only two entities: *Task* and *Allocation*. A *task T* (c, m, d, t) is an isolated executable program that consumes at most c cores, m MBs of memory, and d MBs of disk in t seconds when executed. Note that the 4-tuple (c, m, d, t) is *not known* prior to execution. An *allocation A* (c_a, m_a, d_a, t_a) is a declaration of the resource requirement for a task before execution, which tells the workflow execution engine to allocate to the task c_a cores, m_a MBs of memory, and d_a MBs of disk over t_a seconds. In many batch and workflow execution systems [4], [5], [18], [19], a task is monitored for its resource consumption using standard OS metrics and killed the moment the execution system detects it over-consuming its allocation. We shall follow this reasonable behavior and accordingly introduce the following set of assumptions:

1) **The optimal amount of resources to allocate to each task is unknown prior to its execution.**
2) **A task can only execute if its allocation is specified.**
3) **A task can only consume up to its allocated amount of resources during its execution.**
4) **If a task over-consumes its allocation at any given time, its execution is terminated and the task must be retried with a bigger allocation.**

Therefore, a task executes successfully only if $c \leq c_a$, $m \leq m_a$, $d \leq d_a$, $t \leq t_a$. The main problem then stems directly from assumption (1): **resource allocation is *uncertain*, as a**

large allocation risks a large resource waste, and a small allocation risks task failure and retry.

C. Goals and Metrics

The goal of any predictive allocation algorithm is to attain the performance of the oracle: zero resource waste and 100% resource efficiency. We will now give precise definitions of these metrics, assuming that for a given resource *R*, a task *T* is allocated with a units of resources and consumes at most c units during its execution of *t* seconds.

Resource Waste: There are two types of waste that a task can incur during its execution: *Internal Fragmentation* and *Failed Allocation*.

1) *Internal Fragmentation*: This is defined to be $t \cdot (a - c)$, i.e., the difference over time between a task's allocation and its actual consumption, given that $c \leq a$. This type of waste is minimized to 0 when the predicted resource specification is equal to the peak resource consumption of *T*, i.e., $a = c$.
2) *Failed Allocation*: When task *T* over-consumes its allocation, i.e., $c > a$, then by assumption 4 in Section II-B, T must be retried with a bigger allocation specification. This implies that the previous allocation didn't accomplish any work, and thus incurred an unavoidable waste. Therefore, *Failed Allocation* is defined to be $\Sigma_{i=1}^{k}(a_i \cdot t_i)$, where k is the number of failed allocation attempts and each pair of (a_i, t_i) is the amount of allocated resource and execution time at the ith attempt.

Combining *Internal Fragmentation* with *Failed Allocation*, we define the resource waste for task *T* to be:

$$ResourceWaste(T) = t \cdot (a - c) + \Sigma_{i=1}^{k}(a_i \cdot t_i)$$

Thus, the resource waste for task T is optimal (equal to 0) iff $a = c$ and $k = 0$. Intuitively, a task incurs no resource waste when it is allocated once and that allocation is equal to its peak resource consumption.

Resource Efficiency: For a given type of resource *R*, we introduce *Absolute Workflow Efficiency (AWE)*, a metric that tracks the absolute efficiency in resource usage of a workflow. To avoid a clutter of symbols, we first define $C(T_i)$ and $A(T_i)$ to be the resource consumption and total resource allocation of task i, respectively, where

$$C(T_i) = c_i \cdot t_i$$

$$A(T_i) = a_i \cdot t_i + \Sigma_{j=1}^{k_i}(a_{ij} \cdot t_{ij})$$

AWE is then defined to be:

$$AWE(\{T_i\}_1^n) = \frac{\Sigma_{i=1}^{n} C(T_i)}{\Sigma_{i=1}^{n} A(T_i)},$$

where $\{T_i\}_1^n$ is a sequence of tasks from T_1 to T_n in workflow W. Intuitively, the numerator tracks the total consumption of all tasks in W, and the denominator tracks the total allocation of all tasks in W. Thus, this metric considers the entire workflow as a whole and measures the ratio of total useful resource consumption over the total resource allocation. Furthermore, this metric is *independent* of the number of available workers and thus fits well in the common situation where dynamic workflows are run on opportunistic resources with workers joining and leaving the worker pool over time, and will be used as the main metric for evaluation in Section V. **Note that W is allocated optimally iff its *AWE* is equal to 1, i.e., W uses *all* of the allocated resources.**

D. Additional Problems and Solution Design

In addition to the stated problem of uncertainty in Section II-B, we outline further issues that a complete solution must address by breaking them down into two categories: *Internal Stochasticity* and *External Stochasticity*.

1) Internal Stochasticity: This refers to random and/or uncontrollable elements that are *contained* within a workflow's run, such as:

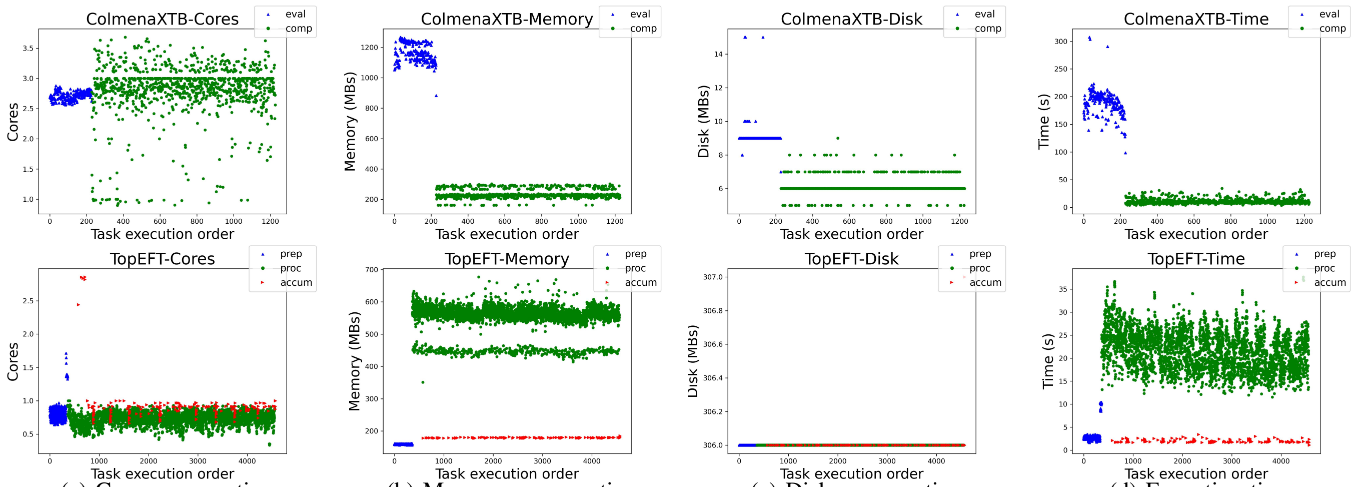
1) *Arbitrary ordering of task execution*: Tasks are submitted sequentially but potentially executed in an arbitrary order due to a variety of factors, including: data dependencies between tasks, priority of tasks, data locality on workers, available capacity of workers, etc.
2) *Specialization of tasks*: Tasks can be core-, memory-, or I/O-intensive to reflect different purposes in a workflow.
3) *Arbitrary structure of workflows*: Workflows are often organized into phases of tasks, each of which serves a different purpose, and thus possibly consume a different amount of resources depending on the current phase.
4) *Arbitrary moving resource distribution*: This issue arises from the combination of (1) and (2), causing a workflow to possibly exhibit an arbitrary and moving resource usage distribution depending on which specialized tasks are being executed.

2) External Stochasticity: This refers to the random and/or uncontrollable elements between executions of the same workflow, such as:

1) *Current system state*: On shared premises, the compute cluster and storage servers are subject to arbitrary usage from other users, which in turn affects the workflow's execution differently between runs and contributes to (1) and (4) in Section II-D1 [20].
2) *Evolution of workflows*: Workflows may behave differently to reflect updates to the underlying software dependencies of tasks, the arrival of a new input distribution, or the changes in tasks or phases of a workflow.
3) *Inherent stochasticity of tasks*: Tasks may be inherently stochastic (e.g., Monte-Carlo-based simulation or SGD-based ML training tasks) and thus may consume resources differently between runs.

3) Design Goals: The combination of uncertainty and internal and external stochasticity substantially complicates the problem of resource allocation. We thus believe that the following 4 design goals are necessary conditions for an allocation algorithm *X* to be considered a complete solution:

1) **General-purpose**: Since the goal of minimizing resource waste only concerns a task's resource consump-



(a) Core consumption (b) Memory consumption (c) Disk consumption (d) Execution time

Fig. 2: Resource consumption of tasks in ColmenaXTB (top) and TopEFT (bottom).
Left to right: cores, memory, disk, execution time. Each point in a plot is a task's peak resource consumption.

tion and allocation, X should be accordingly generic and not rely on any workflow- or task-specific feature.

2) **Prior-free**: Elements in *External Stochasticity* discourage the use of prior information about a workflow, as such information is application-specific and vulnerable to substantial changes to a workflow.
3) **Online**: Element (4) in Section II-D1 directly requires X to be an online algorithm and adapt to the ever-changing resource distribution of tasks.
4) **Robust**: Elements (3) and (4) in Section II-D1 demand X to be sufficiently robust to work with arbitrary resource distribution and phase changes in a workflow run.

III. Case Study: Colmena-XTB and TopEFT

To better understand the stochastic nature of a workflow execution, we examine the resource logs of two large-scale production workflows, ColmenaXTB and TopEFT, and show how several elements of stochasticity as discussed in Section II-D appear in and impact these workflows.

A. Overview

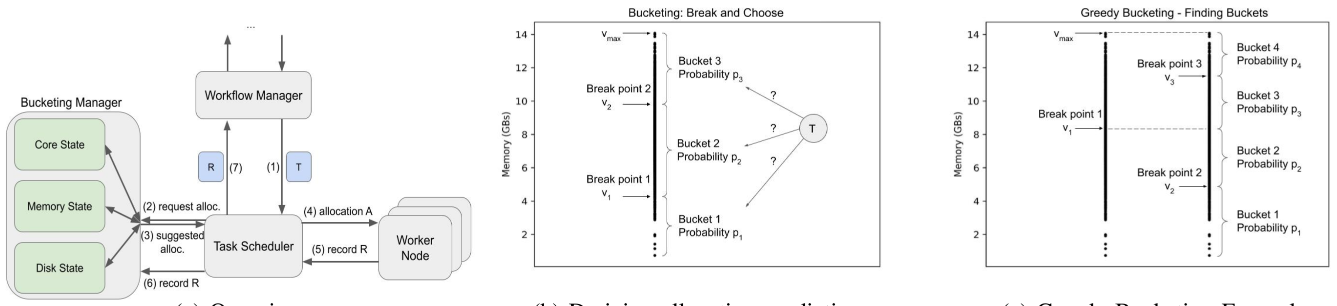
Both ColmenaXTB and TopEFT workflows follow the architecture in Figure 1. ColmenaXTB is an application combining neural network inferences with molecular dynamics analysis to drive large campaigns of molecular search and design. It defines two functions: (1) *evaluate_mpnn*, which takes in a list of candidate molecules and outputs a ranking of those molecules, and (2) *compute_atomization_energy*, which computes the energy values from top-ranked molecules. ColmenaXTB runs on top of a suite of distributed execution frameworks, including: (1) Colmena [16], a Python library driving the molecular search campaign by carefully submitting tasks and examining returned results (2) Parsl [1], a Python-native workflow manager that constructs a task dependency graph on the fly and send ready tasks to (3) Work Queue [18], a manager-worker distributed programming library that handles the allocation, deployment, and execution of tasks, along with result collection and worker management on various underlying distributed systems.

TopEFT structurally operates in the same manner. Its goal is to apply the effective field theory (EFT) to detect new physics by processing a large quantity of events produced by the Large Hadron Collider (LHC). TopEFT defines three functions: (1) *preprocessing*, which scans through a list of metadata files to find relevant event datasets, (2) *processing*, which analyzes a given quantity of events, and (3) *accumulating*, which merges processed results into a complex multi-level histogram. It passes these functions down to Coffea [21], a high-performance data processing library. Coffea first submits all preprocessing tasks to fetch metadata files and identify relevant event datasets, then logically divides events between processing tasks or partially accumulated results between accumulating tasks. All tasks generated by Coffea are sent to Work Queue for remote execution, as described above.

B. Workflows' Resource Consumption

Figure 2 shows the resource consumption of tasks in ColmenaXTB (top row) and TopEFT (bottom row). Within each row, from left to right, we vary the resource types from cores, memory, disk, to execution time. Within each plot, the x-axis tracks the order of task submission (each submitted task gets an incremental ID from 0) and the y-axis displays the magnitude of a given resource type. Each point in a plot is then a given task's peak resource consumption. ColmenaXTB has 228 *evaluate_mpnn* tasks and 1000 *compute_atomization_energy* tasks, and TopEFT has 363 *preprocessing* tasks, 3994 *processing* tasks, and 212 *accumulating* tasks.

Specialization of tasks: For both workflows we can clearly see major differences in resource consumption between tasks of different categories, especially in memory consumption. In ColmenaXTB, while *evaluate_mpnn* tasks use from 1 GB to 1.2 GBs of memory, *compute_atomization_energy* tasks only



(a) Overview (b) Deriving allocation predictions (c) Greedy Bucketing Example

Fig. 3: Bucketing algorithms

(a) Interactions between the bucketing manager with relevant components. (b) Resource predictions are probabilistically derived from a set of computed buckets. (c) Example: Greedy Bucketing greedily and recursively computes a set of buckets.

hover around 200 MBs, suggesting that different categories of tasks need different amount of resource allocations. However, *preprocessing* and *accumulating* tasks in TopEFT consume an almost equivalent amount of memory (around 180 MBs), suggesting that different categories of tasks should instead be allocated *independently* from each other, as different categories don't necessarily show a correlation in resource consumption.

Arbitrary structure of workflows: We also see the phasing behavior arising in both workflows due to internal application logic, which is most obviously seen in the memory consumption of ColmenaXTB tasks. ColmenaXTB first submits only *evaluate_mpnn* tasks to rank all molecules, and then submits only *compute_atomization_energy* tasks to process top-ranked molecules. This phase change suggests that phases of a workflow must be detected and tasks must be allocated accordingly at different points in time.

Inherent stochasticity of tasks: Tasks' core consumption in ColmenaXTB shows another interesting behavior where tasks of the same category don't consume a similar amount of resources. *compute_atomization_energy* tasks in ColmenaXTB are not consistent in their core consumption at all, ranging from 0.9 to 3.6 cores. The memory consumption of *processing* tasks in TopEFT shares the same behavior but in a puzzling way, where tasks can seemingly be separated into two clusters of tasks in terms of memory consumption (one around 580 MBs and the other around 450 MBs). The core consumption of TopEFT tasks shows another aspect of task stochasticity: *outliers*. While most tasks, irrespective of their categories, use one core or less during execution, some tasks go as high as three cores, risking execution failure due to resource exhaustion if a static allocation was made.

This case study on ColmenaXTB and TopEFT demonstrates several elements of stochasticity in both workflows and shows that the resource allocation problem for dynamic workflows is indeed practical and challenging. As the structure of these workflows is quite common, we further believe that the above stochastic elements aren't just restricted to ColmenaXTB and TopEFT but are generalizable to other large-scale and possibly more complex production workflows. This case study thus validates both the design goals of a complete solution as stated in Section II-D3 and a practical need for a resource allocation algorithm conforming to such goals.

IV. Bucketing Algorithms

In this section, we introduce two novel allocation algorithms forming the basis of an adaptive resource allocator, *Greedy Bucketing* and *Exhaustive Bucketing*, that rely on the principle of the bucketing approach. We first give the general idea of the bucketing approach, describe how *Greedy Bucketing* and *Exhaustive Bucketing* operate within this framework, and then detail a few heuristics to incorporate these algorithms into a resource allocator. Most importantly, we will show that these algorithms directly match the design goals stated at the end of Section II-D3.

A. The Bucketing Approach

Figure 3a shows a quick overview of the bucketing approach. (1) The workflow manager sends a ready task T to the task scheduler to be deployed and executed. (2) The task scheduler asks the bucketing manager for the amount of resources to allocate T. (3) The bucketing manager maintains a separate state for each resource type, queries each state for a value, and sends back a suggested resource allocation A for T. (4) The task scheduler sends T with allocation A to a worker for execution. (5) The worker returns the completed task with result and resource record R and (6)(7) sends R to both the bucketing manager and the workflow manager to update their respective states. Note how the bucketing approach revolves around a bucketing manager that solely interacts with the task scheduler to perform two operations: respond to an allocation request upon a ready task and update its internal states upon a completed task. As *Greedy Bucketing* and *Exhaustive Bucketing* only diverge on how to update the internal bucketing states and share the resource prediction approach, we will now explain the prediction approach and delay state updates to Sections IV-B and IV-C.

As each resource type is managed independently, we will focus on one resource for simplicity. Figure 3b shows an example of how the bucketing approach predicts a new task's allocation based on a given bucketing state. Assume that we have a synthetic workflow containing 2,000 tasks of the same

Algorithm 1 Greedy Bucketing

```
procedure GREEDYBUCKETING(lo, hi, L)
    if lo == hi then return [lo]
    min_cost, break_idx = $\infty$, None
    for $i$ = lo to hi do
        cost = compute_greedy_cost(lo, $i$, hi, L)
        if cost < min_cost then
            min_cost, break_idx = cost, $i$
    if break_idx == hi then return [hi]
    lo_indices = GreedyBucketing(lo, break_idx, L)
    hi_indices = GreedyBucketing(break_idx+1, hi, L)
    return lo_indices.concat(hi_indices)
end procedure
```

category, each of which's memory consumption in GBs is sampled from the normal distribution $\mathcal{N}(8, 2)$. Further assume that the application decides to submit another task of the same category immediately after all 2000 task executions in this workflow, which requires the resource allocator to specify some amount of memory. As the first 2000 tasks execute successfully and return to the application, the resource allocator has access to these tasks' resource consumption records. It then sorts these records by resource value and tries to find if there are potential clusters among these values, each representing a group of tasks consuming a similar amount of resources. For now assume that these clusters are found to be three intervals $(0, v_1]$, $(v_1, v_2]$, and $(v_2, v_{max}]$. The allocator then breaks the sorted list of records based on v_1 and v_2 into three buckets accordingly, where each bucket contains all records in the respective interval. Each bucket is then reduced to two elements: the representative value and the probability value. Let **B** be the set of all buckets, and B_i be the i^{th} bucket, then the representative value of B_i is the maximum value of all records in a bucket:

$$B_i.rep = \max_{r \in B_i.records}(r.value),$$

and the probability value of B_i is the ratio of the number of records contained in B_i:

$$B_i.prob = \frac{\#records \in B_i}{\Sigma_{B_j \in \mathbf{B}} \#records \in B_j}$$

Upon a request to allocate a new task, the allocator randomly chooses a bucket among the list of buckets based on the probability values defined above and returns the chosen bucket's representative value. This value doesn't guarantee that the next task will not exceed its resource allocation however, so when that task returns with a resource exhaustion status, the allocator only considers buckets that have the representative values greater than that of the previously chosen bucket. If there are no such buckets, implying that the previous bucket's representative value is the greatest one seen so far, then the allocator doubles the task's previous peak resource consumption until the task succeeds.

It is straightforward to see why the bucketing approach so far follow the design goals listed in Section II-D3. The bucketing approach (1) operates using only the resource records of tasks, making it *general-purpose*, (2) uses only resource records of tasks completed in the current workflow run, making it *prior-free*, and (3) derives allocation predictions on demand from the task scheduler, making it *online*. To partially address the *robust* design goal, we observe that when workflows make abrupt changes and exhibit the phasing behavior, more recent task records serve as a better guidance to allocate subsequent tasks and should contribute more to the probability values of buckets than older ones. To build this observation into the bucketing approach, we add into each record of a task a *significance* value, and the higher the value the more recent or significant the task record is (we will briefly discuss how to set this value in Section V.) Thus, the probability value of each bucket B_i is now updated to:

$$B_i.prob = \frac{\Sigma_{r \in B_i.records} r.sig}{\Sigma_{B_j \in \mathbf{B}} \Sigma_{r \in B_j.records} r.sig}$$

where $r.sig$ is the significance value of record r.

It's quite challenging to address the arbitrary moving resource distribution aspect of the *robust* design goal, and this is where *Greedy Bucketing* and *Exhaustive Bucketing* diverge on the method to capture and model the problem's complexity. While *Greedy Bucketing* attempts to find a bucketing state that minimizes the expected resource waste in a greedy and recursive manner, *Exhaustive Bucketing* computes the expected resource waste of all possible combinations of buckets and chooses the one with the lowest waste.

B. Greedy Bucketing

The question *Greedy Bucketing* tries to answer is straightforward: given a list of records, should it break that list into exactly two sublists or not, and if yes, where exactly is the break point? Assume the answer is yes and the break point is v_1. *Greedy Bucketing* first breaks the list from one interval $(0, v_{max}]$ into two: $(0, v_1]$ and $(v_1, v_{max}]$, forming two buckets B_{v_1} and $B_{v_{max}}$. Given this configuration, and assume that the next task T follows the resource consumption behaviors of completed tasks, then T has a probability of $B_{v_1}.prob$ to consume an amount of resources in the interval of $(0, B_{v_1}.rep]$, and a probability of $B_{v_{max}}.prob$ to consume an amount of resources in the interval of $(B_{v_1}.rep, B_{v_{max}}.rep]$. Since the bucketing approach probabilistically chooses a bucket to allocate the next task, it will choose bucket B_{v_1} with a probability of $B_{v_1}.prob$ and $B_{v_{max}}$ with a probability of $B_{v_{max}}.prob$. Assume the resource consumption of T is v_{lo} if it falls within B_{v_1} and v_{hi} if it falls within $B_{v_{max}}$, then four cases can occur:

1) T falls within B_{v_1} and *Greedy Bucketing* chooses B_{v_1}: this happens with a probability of $B_{v_1}.prob^2$ and incurs an expected resource waste of $W_{lo,lo} = B_{v_1}.prob^2(B_{v_1}.rep - v_{lo})$.
2) T falls within B_{v_1} and *Greedy Bucketing* chooses $B_{v_{max}}$: this happens with a probability of $B_{v_1}.prob \cdot B_{v_{max}}.prob$ and incurs an expected resource waste of $W_{lo,hi} = B_{v_1}.prob \cdot B_{v_{max}}.prob(B_{v_{max}}.rep - v_{lo})$.
3) T falls within $B_{v_{max}}$ and *Greedy Bucketing* chooses B_{v_1}: this happens with a probability of $B_{v_{max}}.prob \cdot$

Algorithm 2 Exhaustive Bucketing

```
procedure EXHAUSTIVEBUCKETING(L)
    min_cost, break_indices = $\infty$, None
    for k = 0 to L.length-1 do
        for P in combinations(k, L) do
            cost = compute_exhaust_cost(P, L)
            if cost < min_cost then
                min_cost, break_indices = cost, P
    return break_indices
end procedure
```

$B_{v_1}.prob$ and incurs an expected resource waste of $W_{hi,lo} = B_{v_{max}}.prob \cdot B_{v_1}.prob(B_{v_1}.rep + B_{v_{max}}.rep - v_{hi})$, as T exhausts $B_{v_1}.rep$ amount of resources and thus retries with $B_{v_{max}}.rep$ amount of resources.

4) T falls within $B_{v_{max}}$ and *Greedy Bucketing* chooses $B_{v_{max}}$: this happens with a probability of $B_{v_{max}}.prob^2$ and incurs an expected resource waste of $W_{hi,hi} = B_{v_{max}}.prob^2 \cdot (B_{v_{max}}.rep - v_{hi})$.

Thus, the expected resource waste of the next task under *Greedy Bucketing* is $W = W_{lo,lo} + W_{lo,hi} + W_{hi,lo} + W_{hi,hi}$.

As *Greedy Bucketing* doesn't know the actual value of v_{lo} or v_{hi} of T, it estimates these values using a weighted average of values of records that fall in the same bucket, as follows:

$$v_{lo} = \frac{\Sigma_{r \in B_{v_1}.records} r.value * r.sig}{\Sigma_{r \in B_{v_1}.records} r.sig}$$

$$v_{hi} = \frac{\Sigma_{r \in B_{v_{max}}.records} r.value * r.sig}{\Sigma_{r \in B_{v_{max}}.records} r.sig}$$

This is the gist of *Greedy Bucketing* and represented by the procedure *compute_greedy_cost* in Algorithm 1. As we don't know where v_1 is at, *Greedy Bucketing* scans through the list of records and computes each record's expected resource waste as if it is the break point. It then chooses the record incurring the minimum amount of resource waste and outputs that record as a break point. However, if v_{max} is chosen, then *Greedy Bucketing* stops its computation and returns no break point as having only one bucket containing all records yields the minimum amount of waste.

The final technique of Greedy Bucketing is shown in the last 4 lines of Algorithm 1, where it recursively calls itself on two portions of the records and finds possibly more break points. Figure 3c shows an example of this behavior as *Greedy Bucketing* first finds a break point v_1, then recursively calls itself on smaller portions of the list of records to find v_2 and v_3, yielding a list of break points $[v_1, v_2, v_3]$ and the final configuration of 4 buckets, guaranteeing that each call to *Greedy Bucketing* finds the local optimum that minimizes the expected local resource waste.

C. Exhaustive Bucketing

Algorithm 2 shows how *Exhaustive Bucketing*, instead of finding buckets greedily like *Greedy Bucketing*, considers all possible configurations of buckets in a list of records, computes the expected resource waste produced by each configuration, and chooses the best one. Let L be a list of records of tasks. Since each record can form its own bucket, there can be at least 1 bucket and at most $L.length$ buckets. Thus, the outer for loop runs from 0 to $L.length - 1$ as there can be at least 0 break point and at most $L.length - 1$ break points that separate these buckets. The inner for loop then considers all combinations of break points of length k that can be drawn from list L and returns the one that yields the lowest expected resource waste. We now focus on the gist of *Exhaustive Bucketing*: the procedure *compute_exhaust_cost*.

Assume the next task's resource consumption v_i falls within the bucket B_i in a list of buckets **B** of length N. As *Exhaustive Bucketing* doesn't know v_i, it estimates this value in a similar way to how *Greedy Bucketing* estimates v_{lo} and v_{hi}:

$$v_i = \frac{\Sigma_{r \in B_i.records} r.value * r.sig}{\Sigma_{r \in B_i.records} r.sig}$$

Note that the bucketing approach randomly chooses a bucket according to its probability value. Since a task can be anywhere in N buckets and the allocator can choose any of the N buckets, we have N^2 cases to handle. Let $T[i,j]$ be the expected resource waste when the next task falls within bucket B_i and *Exhaustive Bucketing* chooses bucket B_j. If $i \leq j$, then

$$T[i,j] = B_j.rep - v_i,$$

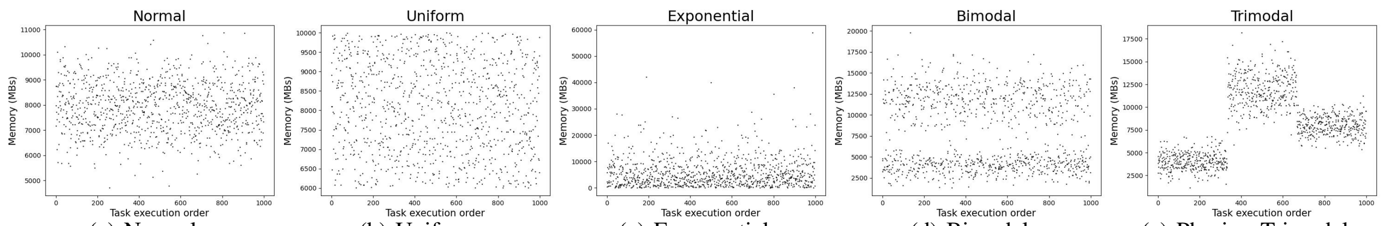
as the allocation from B_j is sufficient for the next task's execution. Otherwise,

$$T[i,j] = B_j.rep + \Sigma_{k=j+1}^{N} \frac{B_k.prob}{\Sigma_{m=j+1}^{N} B_m.prob} * T[i][k]$$

To see this, note that $B_j.rep$ is the resource waste from *Failed Allocation* as bucket j's representative value is less than the amount of consumed resource of the next task. As described above, *Exhaustive Bucketing* now only considers higher buckets from $j+1$ to N, and probabilistically chooses one of these buckets to re-allocate the task. As the pool of buckets is reduced to $[j+1, N]$, $\frac{B_k.prob}{\Sigma_{m=j+1}^{N} B_m.prob}$ simply renormalizes the probability value of bucket B_k. $T[i][k]$ is defined as above, and thus $T[i][j]$ is intuitively equal to the sum of the current resource waste of the next task plus the expected resource waste of that task's next allocation. If $j < i$ and $j < k$, then $T[i][j]$ depends on $T[i][k]$. Therefore, the table T should be filled from the last column to the first column per row. After computing T, the expected resource waste of a list of buckets **B** is then:

$$W_{\mathbf{B}} = \Sigma_{i=1}^{N} \Sigma_{j=1}^{N} B_i.prob * B_j.prob * T[i][j],$$

as there's a probability of $B_i.prob * B_j.prob$ that the next task falls within B_i and *Exhaustive Bucketing* chooses B_j. As mentioned above, the algorithm's job now is to apply *compute_exhaust_cost* to every configuration of buckets and choose the best one.



(a) Normal $N(8,1)$ (b) Uniform $U(6,10)$ (c) Exponential $E(6)$ (d) Bimodal $N_1(4,1)$, $N_2(12,2)$ (e) Phasing Trimodal $N_1(4,1)$, $N_2(12,2)$, $N_3(8,1)$

Fig. 4: Memory consumption of tasks in five synthetic workflows.
x-axis: Task execution order, numbered from 1 to 1000. y-axis: amount of memory used in MBs.

D. *Integrating Bucketing Algorithms*

Additional refinements are needed to integrate these algorithms into a resource allocator. Both *Greedy Bucketing* and *Exhaustive Bucketing* assume the existence of a list of records. To get these records in the first place, an allocator runs in the exploratory mode for a while and allocates each task some predefined amount of resources until the it collects an enough number of records (Details are delayed to Section V.)

As discussed in Section III-B, an allocator treats each category of tasks independently and uses a separate instance of a bucketing manager per category. Within each category, the bucketing manager maintains a separate instance of a resource state that follows either the *Greedy Bucketing* or the *Exhaustive Bucketing* algorithm. Thus, an adaptive resource allocator would more or less adhere to the following pseudocode:

```
class Allocator:
    def __init__(self, workflow_metadata):
        #initialize the list of bucketing instances,
        #one per category

    def add(self, task_record):
        #add task record to the appropriate
        #bucketing instance

    def predict(self, task_category):
        #call the appropriate bucketing instance
        #to compute the list of buckets and allocate
        #the next task accordingly
```

To conclude this section, we now address the call to *combinations* in Algorithm 2. There are $\frac{N!}{k!(N-k)!}$ ways to choose k records out of N records, which grows exponentially as the allocator continues to accumulate tasks and increases N over time. To avoid this computational problem and spread the number of considered records evenly, a call to *combinations(k, L)* instead operates as follows:

1) form a list of $k-1$ candidate break points L to break the space of records evenly, so $L[i] = \frac{v_{max}*i}{k}$ for $i = 1$ to $k-1$, where v_{max} is the maximum value in all records.
2) for each candidate break point, map its value to the closest record that has a lower value than it and remove all duplicate or empty mappings.
3) return newly found records as a list of break points to be considered.

V. Evaluation

We begin this section by describing the settings that *Greedy Bucketing* and *Exhaustive Bucketing* use in all experiments. To understand and evaluate the performance and robustness of these algorithms, we additionally generate five synthetic workflows along with ColmenaXTB and TopEFT. We conclude this section by presenting our analysis on these workflows' performance under 7 allocation algorithms, and thus show that the allocation predictions made by the bucketing algorithms are performant and reliable[1].

A. *Settings*

We implement the bucketing algorithms in the core scheduler of Work Queue [18] to minimize changes to all workflow applications. For each experiment, we run the corresponding workflow on opportunistic workers with 16 cores, 64 GBs of memory, and 64 GBs of disk. The number of workers varies from 20 to 50 depending on the availability of the local HTCondor cluster. Algorithm-wise, there are many ways to set the significance value of a task record. In all experiments we simply set it to the task ID, so the task's record with ID 1 has a significance value of 1, and so on. For *Exhaustive Bucketing*, it is from our experience running both algorithms that the number of buckets rarely exceeds 10 at any given time, so we restrict k to at most 10 in the outer for loop of Algorithm 2. In the exploratory mode of both algorithms, we allocate each task 1 core, 1 GB of memory, and 1 GB of disk until 10 records are retrieved. If a task exhausts any type of resources during this phase, it is simply retried by doubling the amount of respective resources.

To evaluate the performance of the bucketing algorithms, we use 2 naive algorithms, *Whole Machine* and *Max Seen*, and 3 alternative algorithms that align to our design goals, *Min Waste*, *Max Throughput*, and *Quantized Bucketing*. Naive algorithms are straightforward: *Whole Machine* simply allocates each task a whole worker and thus serves as our baseline, and *Max Seen* allocates each task the maximum resource value seen so far in the current workflow run. On the other hand, *Min Waste* and *Max Throughput* follow the description of respective algorithms in [15], and *Quantized Bucketing* follows

[1]All logs are available at https://github.com/tphung3/ipdps2024-resource-paper.

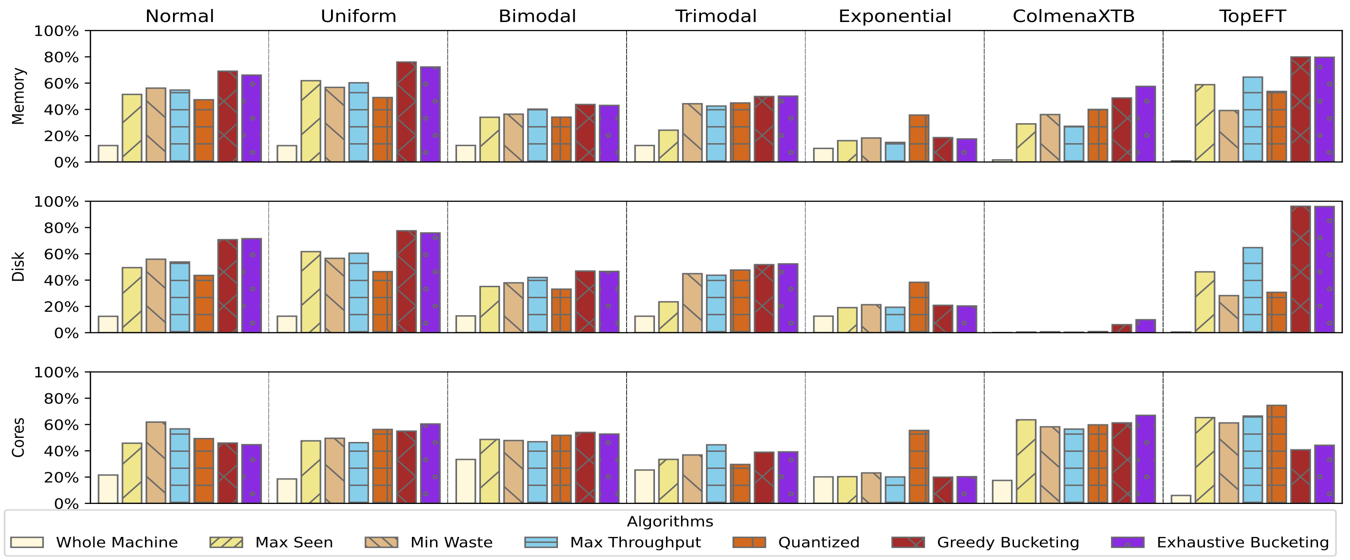


Fig. 5: Absolute Workflow Efficiency in cores, memory, and disk of 7 workflows across 7 allocation algorithms.

	10	200	1000	2000	5000
GB	11.2	586.4	14588.2	62207.2	441050.7
EB	14.4	76.5	323.5	567.8	1632.0

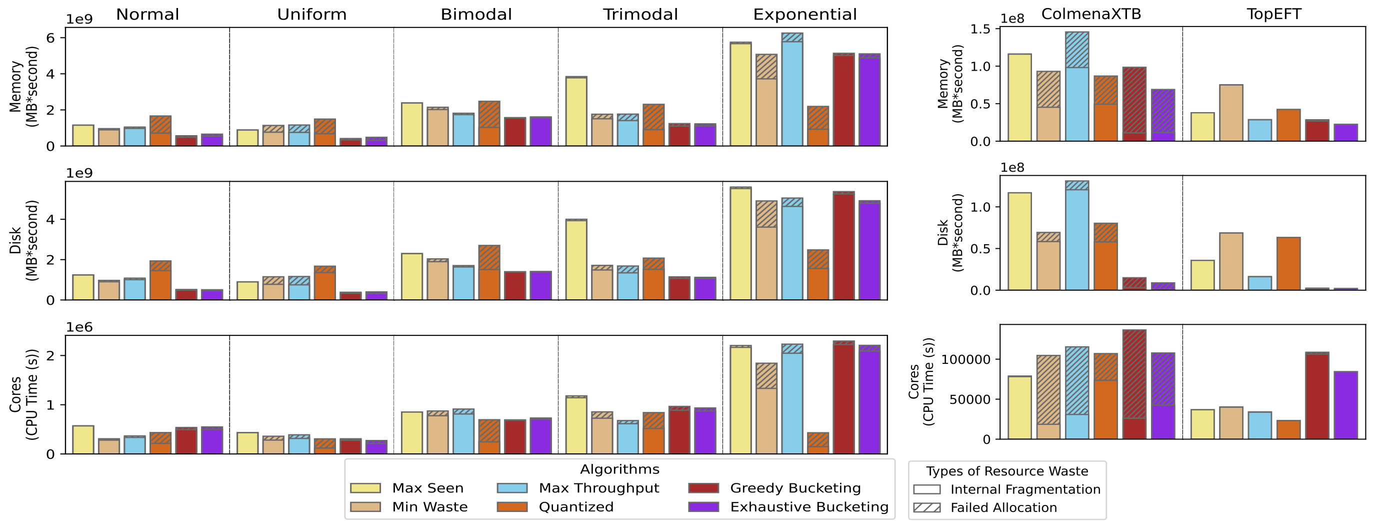
TABLE I: Average time (in μs) to compute a new bucketing state and derive a new allocation.
(Columns) The total number of records in the record list
(Rows) GB: Greedy Bucketing, EB: Exhaustive Bucketing

the description in [11]. Our main metric here is Absolute Workflow Efficiency (*AWE*) as it measures the efficiency of the workflow as a whole and is *independent* from the variable number of workers arising from the use of opportunistic resources as discussed in Section II-C. **Note that an optimal algorithm that attains 100% resource efficiency and zero resource waste is not realistic as the resource allocation problem is online: tasks are generated and required a resource allocation at runtime but the optimal allocation value is only observable at the end of their executions.**

B. Synthetic Workflows

We generate five synthetic workflows (*Normal*, *Uniform*, *Exponential*, *Bimodal*, and *Phasing Trimodal*) to evaluate the bucketing algorithms' performance under a variety of workflow's behaviors. Each workflow contains 1000 tasks of the same category to account for the worst-case scenario where there's a large discrepancy between tasks' resource consumption within a category. Figure 4 shows the memory consumption of tasks in these workflows, where each task's consumption is sampled from the respective distribution (we skip cores and disk to save space as disk shares the same distribution with memory and cores have a slightly different distribution.) Each synthetic workflow is designed to capture a certain possible stochastic behavior of a dynamic workflow: *Normal* and *Uniform* for common randomness, *Exponential* for outliers, *Bimodal* for specialization of tasks, and *Phasing Trimodal* for moving resource distribution from phases. Figure 5 shows the *Absolute Workflow Efficiency* in different resource types of these workflows under 7 allocation algorithms. A quick look shows that different distributions reflect different levels of resource efficiency as the amount of resource saved depends on the "hardness" of respective distributions (e.g., it is easier to allocate the *Uniform* workflow than the *Exponential* workflow as there are more extreme outliers in the latter.) We can additionally see that *Whole Machine* performs very poorly, as expected. *Max Seen* performs fairly well across different resource types and synthetic workflows, but is usually 5-25% less efficient than the best algorithm depending on the configuration. *Min Waste* and *Max Throughput* perform comparably well against *Max Seen* but the performance advantage doesn't appear frequently across configurations. *Quantized Bucketing* trails behind these two algorithms, but significantly excels at the *Exponential* workflow. This is because it separates the buckets at the 50th quantile, which reduces the number of retries on average and thus reduces the effect of outliers on the resource waste.

The two novel bucketing algorithms perform well and consistently better than alternative algorithms, except for the (Cores-Normal) and (Trimodal-Cores) configurations. For the *Normal* and *Uniform* workflows, bucketing algorithms perform well, reaching 60-80% efficiency. Additional built-in variations from the *Bimodal* and *Trimodal* workflows reduce the efficiency of bucketing algorithms to 40-50%, as tasks can come from any mode with its own variance. Both bucketing algorithms struggle against the Exponential workflow, where only around 20% efficiency is achieved and is only slightly better than the *Whole Machine* baseline. This workflow poses the most resource waste as it is easy to incur a large internal fragmentation waste to small tasks and a large failed allocation waste to the occasionally large tasks. However, our analysis on these workflows' performance reinforces that the bucketing algorithms excel in most variations of workflows' behaviors and don't produce catastrophic waste in corner cases.



(a) Synthetic Workflows (b) Production Workflows

Fig. 6: Resource Waste in cores, memory, and disk of 7 workflows across 6 allocation algorithms.

C. Production Workflows

The *Absolute Workflow Efficiency* of ColmenaXTB and TopEFT is shown in Figure 5 (last two columns). We can see in Figure 5 that while *Min Waste*, *Max Throughput*, and *Quantized Bucketing* are more efficient by 20-30% than *Greedy Bucketing* and *Exhaustive Bucketing* in allocating cores to tasks in TopEFT (we believe the first few outliers cause this issue, which can be mitigated by running *Quantized Bucketing* initially then switching over), the bucketing algorithms are consistently more efficient in other resource types, reaching close to 100% disk efficiency and 80% memory efficiency for TopEFT. While TopEFT tasks always consume 306 MBs of disk, other predictive algorithms fail to achieve an efficiency close to 100% due to several reasons: (1) contrary to the conservative exploratory strategy of the bucketing algorithms in subsection V-A, other algorithms allocate a whole machine instead (16 cores, 64 GBs of memory and disk), trading an expensive exploratory cost with a guarantee of successful task execution, and (2) *MaxSeen* allocates resources to task using a histogram with the bucket size of 250, resulting in a rounded-up 500-MB disk allocation for a 306-MB disk consumption in the steady state. Due to the low disk consumption of tasks in ColmenaXTB (around 10 MBs as shown in Figure 2) and the policy of allocating 1 GB of disk to tasks in the exploratory mode, the *Absolute Workflow Efficiency* of ColmenaXTB in disk goes extremely low to single-digit efficiency for all allocation algorithms.

Table I shows the average time for both bucketing algorithms to compute a new bucketing state and a new allocation with varying sizes of the list of records. *Greedy Bucketing* is quite expensive later on (almost half a second to compute an allocation at 5000 records) due to the recursions at the end of each iteration. On the other hand, *Exhaustive Bucketing*, despite its name, runs quite fast and uses around a millisecond to compute a new allocation at 5000 records. This is due to the optimization as discussed at the end of Section IV, as *Exhaustive Bucketing* doesn't have to search for buckets but instead applies a compute function to every given configuration of buckets. Note that Table I assumes the worst-case scenario that each task requires a re-computation of the bucketing state, which is not necessarily the case. As shown in Figure 3a, a sequence of ready tasks can share the same bucketing state if there's no completed tasks in-between (no resource record to update), and a sequence of completed tasks can be batched into a large update if there's no ready tasks in-between (no need to update until there's a resource request).

D. Resource Waste Analysis

Figure 6 shows the resource waste of all 3 resource types for 7 workflows with 6 allocation algorithms broken down into *Internal Fragmentation* and *Failed Allocation* (we remove the baseline Whole Machine algorithm for better visualization).

For the synthetic workflows, we see that the *Max Seen* algorithm performs according to its strategy and mostly over-estimates resource allocations to tasks. The same trend applies to other algorithms except *Quantized Bucketing* with varying degrees, showing that the predictive algorithms favor over-estimation over under-estimation. This is a behavior consistent with the goal of minimizing waste, as an under-allocation not only incurs resource waste from a failed allocation, but also carries the same risk of an over-estimation as the failed task has to be allocated again. *Greedy Bucketing* and *Exhaustive Bucketing* penalize the under-allocation closely to *Max Seen*, whereas *Min Waste* and *Max Throughput* allow a 20-30% of resource waste to come from failed allocations.

For the ColmenaXTB workflow, waste from failed allocations dominates the proportion of the total waste in most algorithms, except *Max Seen*. This can be explained by the diverse resource distribution in each type of resource in the workflow. Such diverse resource distributions in multiple resource dimensions can trickle a loop of failed allocations, as the algorithms have to predict exactly how much resource a task needs and an under-prediction in any resource at any attempt will cause the task to be under-allocated and thus must be retried with a bigger allocation. In contrast, the TopEFT workflow shows a somewhat less diverse resource distributions in all 3 resource dimensions, especially in disk and cores. This helps the algorithms to narrow down that only the memory dimension needs to be closely tracked and thus makes it easier for the algorithms to predict the allocations, resulting in the fact that most allocations from the predictive algorithms are over-allocations.

E. Summary

The ability of the novel bucketing algorithms to consistently make quality predictions in a diverse set of combinations of workflow's behaviors and resource types therefore demonstrates that the bucketing approach accurately models the problems outlined in Sections II-B, II-C, II-D and follows the design goals in Section II-D3 and that the core strategies in *Greedy Bucketing* and *Exhaustive Bucketing* are effective at finding useful buckets in a given list of task records. Since the *Exhaustive Bucketing* algorithm delivers higher resource consumption efficiency than alternative approaches in most cases (comparable to the *Greedy Bucketing* algorithm) and much faster time to compute a new allocation (computation time grows linearly with the amount of completed tasks as demonstrated in Table I), it is the recommended algorithm to allocate unknown workflows and tasks.

VI. Related Works

A large quantity of algorithms, strategies, and heuristics on improving workload's resource consumption efficiency have been produced, tracked, and compiled over time by a number of research groups. Witt et al. [13] provide an extensive survey on the approach of modeling tasks' resource consumption using black-box machine learning models. We argue that while machine learning is a promising approach, it requires a large quantity of labeled data and takes a long time to train and infer, and thus needs careful design to work as an online algorithm. Pupykina et al. [22] focus on the management of only memory consumption of tasks in HPC systems, and thus doesn't account for cores and disk.

Other papers focus on different objectives to improve a workflow's execution. Zhang et al. [12] leverage reinforcement learning techniques and present a scheduling inspector module that optimizes the average task wait time. Thekkepurayil et. al. [23] schedule workflows in cloud systems according to a variety of objectives, including optimal resource consumption, quality of services, load balancing, etc. Li et al. [24] instead focus on scheduling decisions that make sure workflows are fault-tolerant and data are placed efficiently across geo-distributed data centers. Several groups focus on the problem of minimizing the energy consumption of workflows in clouds with a budget constraint, as presented by Choudhary et al. [25] and Taghinezhad-Niar et al. [26]. In this paper, we model the problem of resource consumption in a general way with only two entities: *Task* and *Allocation*, and thus believe that our solution can be applied to both cloud and HPC systems and incorporated into these work independently.

Other research groups, while also attempt to reduce the resource waste of workflows, extract workflow- or task-specific information. Rodrigo et al. [27] optimize the turnaround time of a workflow by analyzing the task dependency graph. Tanash et al. [28] use metadata of tasks to train several machine learning models and predict a task's memory consumption. Witt et al. [29] instead use the input size of each task as a parameter to infer tasks' resource consumption. Rodrigues et al. [30] use a machine learning model to process and infer a task's memory consumption based on its LSF job specification.

Many papers also extract information from a chronological list of resource records accumulated during a workflow's execution. Tovar et al. [15] present two strategies that aim to minimize waste and maximize throughput, each relying on the policy of at-most-once retry. Our bucketing algorithms instead relax the policy of at-most-once retry by using a bounded list of buckets. Phung et al. [11] study the heterogeneity of tasks and allocate tasks based on their categories using the k-means and quantile clustering methods. Fan et al. [14] train a reinforcement learner to schedule tasks based on the availability of the local cluster instead of predicting tasks' resource consumption.

VII. Conclusion and Future Works

The nature of executing dynamic workflows on opportunistic resources complicates the process of resource management, as tasks are generated dynamically at run time and consume an unknown amount of resources but must be specified a resource allocation in advance. To avoid potentially huge resource waste due to over- or under-allocation, this paper presents two allocation algorithms, *Greedy Bucketing* and *Exhaustive Bucketing*, that are designed to be general-purpose, online, prior-free, and robust to allocate tasks on-the-fly. Results show that the bucketing algorithms perform well on a diverse set of workflows and outperform alternative algorithms, showing that the algorithmic design of the bucketing algorithms is correct in addressing potential stochasticity of workflow's behaviors and useful in predicting tasks' resource consumption.

In the future, we target to evaluate our algorithms on even larger workflows ($>$ 10,000 tasks). We hypothesize that the bucketing algorithms should perform even better on larger workflows since they are shown to perform well and quickly converge to a steady state on workflows of around 4,500 tasks.

Other future works include: potential optimizations of the bucketing algorithms, an extension to additional resource types, and exploring other approaches and deriving alternative solutions to the problem of resource allocation.

Acknowledgement

This work was supported by National Science Foundation Grant OAC-1931348. We thank Ben Tovar, Logan Ward, Kevin Lannon, and Kelci Morman for providing us with the code and data for the runs. TaskVine is available at http://ccl.cse.nd.edu/software/taskvine.

References

[1] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster, M. Wilde, and K. Chard, "Parsl: Pervasive parallel programming in python," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 25–36. [Online]. Available: https://doi.org/10.1145/3307681.3325400

[2] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th Python in Science Conference*, K. Huff and J. Bergstra, Eds., 2015, pp. 130 – 136.

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

[4] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the condor experience." *Concurrency - Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[5] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*, 2003.

[6] "Oracle Grid Engine: An Overview," Oracle Corporation. Accessed: 2023-09-27, Tech. Rep., 2010 [Online].

[7] A. Inc. (2023) Amazon spot instances. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances.html.Accessed:2023-09-28

[8] G. LLC. (2023) Google spot vms. [Online]. Available: https://cloud.google.com/spot-vms.Accessed:2023-09-28

[9] M. Corporation. (2023) Azure spot virtual machines. [Online]. Available: https://learn.microsoft.com/en-us/azure/virtual-machines/spot-vms.Accessed:2023-09-28

[10] A. Group. (2023) Alibaba preemptible instances. [Online]. Available: https://www.alibabacloud.com/help/en/ecs/user-guide/overview-4. Accessed:2023-09-28

[11] T. S. Phung, L. Ward, K. Chard, and D. Thain, "Not all tasks are created equal: Adaptive resource allocation for heterogeneous tasks in dynamic workflows," in *2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS)*. IEEE, 2021, pp. 17–24.

[12] D. Zhang, D. Dai, and B. Xie, "Schedinspector: A batch job scheduling inspector using reinforcement learning," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 97–109. [Online]. Available: https://doi.org/10.1145/3502181.3531470

[13] C. Witt, M. Bux, W. Gusew, and U. Leser, "Predictive performance modeling for distributed batch processing using black box monitoring and machine learning," *Information Systems*, vol. 82, p. 33–52, May 2019. [Online]. Available: http://dx.doi.org/10.1016/j.is.2019.01.006

[14] Y. Fan, Z. Lan, T. Childers, P. Rich, W. Allcock, and M. E. Papka, "Deep reinforcement agent for scheduling in hpc," 2021.

[15] B. Tovar, R. F. da Silva, G. Juve, E. Deelman, W. Allcock, D. Thain, and M. Livny, "A Job Sizing Strategy for High-Throughput Scientific Workflows," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 2, pp. 240–253, 2018.

[16] "Colmena," 2021 [Online], exaLearn and Parsl Teams. Available: https://colmena.readthedocs.io/en/latest/index.html.

[17] A. Basnet *et al.*, "Topeft/topcoffea: Topcoffea 0.1 (v0.1)," (2021), zenodo. Available: https://doi.org/10.5281/zenodo.5258003.

[18] 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.

[19] B. Sly-Delgado, T. S. Phung, C. Thomas, D. Simonetti, A. Hennessee, B. Tovar, and D. Thain, "Taskvine: Managing in-cluster storage for high-throughput data intensive workflows," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1978–1988. [Online]. Available: https://doi.org/10.1145/3624062.3624277

[20] T. S. Phung, B. Clifford, K. Chard, and D. Thain, "Maximizing data utility for hpc python workflow execution," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 637–640. [Online]. Available: https://doi.org/10.1145/3624062.3624136

[21] "Coffea," 2021 [Online], fermi National Accelerator Laboratory. Available: https://github.com/CoffeaTeam/coffea.

[22] A. Pupykina and G. Agosta, "Survey of memory management techniques for hpc and cloud computing," *IEEE Access*, vol. 7, pp. 167 351–167 373, 2019.

[23] J. Kakkottakath Valappil Thekkepurayil, D. P. Suseelan, and P. M. Keerikkattil, "Multi-objective scheduling policy for workflow applications in cloud using hybrid particle search and rescue algorithm," *Serv. Oriented Comput. Appl.*, vol. 16, no. 1, p. 45–65, mar 2022. [Online]. Available: https://doi.org/10.1007/s11761-021-00330-4

[24] C. Li, J. Liu, M. Wang, and Y. Luo, "Fault-tolerant scheduling and data placement for scientific workflow processing in geo-distributed clouds," *J. Syst. Softw.*, vol. 187, no. C, may 2022. [Online]. Available: https://doi.org/10.1016/j.jss.2022.111227

[25] A. Choudhary, M. C. Govil, G. Singh, L. K. Awasthi, and E. S. Pilli, "Energy-aware scientific workflow scheduling in cloud environment," *Cluster Computing*, vol. 25, no. 6, p. 3845–3874, dec 2022. [Online]. Available: https://doi.org/10.1007/s10586-022-03613-3

[26] A. Taghinezhad-Niar, S. Pashazadeh, and J. Taheri, "Energy-efficient workflow scheduling with budget-deadline constraints for cloud," *Computing*, vol. 104, no. 3, p. 601–625, mar 2022. [Online]. Available: https://doi.org/10.1007/s00607-021-01030-9

[27] G. P. Rodrigo, E. Elmroth, P.-O. Östberg, and L. Ramakrishnan, "Enabling workflow-aware scheduling on hpc systems," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 3–14. [Online]. Available: https://doi.org/10.1145/3078597.3078604

[28] M. Tanash, B. Dunn, D. Andresen, W. Hsu, H. Yang, and A. Okanlawon, "Improving hpc system performance by predicting job resources via supervised machine learning," in *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*, ser. PEARC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3332186.3333041

[29] C. Witt, J. van Santen, and U. Leser, "Learning low-wastage memory allocations for scientific workflows at icecube," in *2019 International Conference on High Performance Computing Simulation (HPCS)*, 2019, pp. 233–240.

[30] E. R. Rodrigues, R. L. F. Cunha, M. A. S. Netto, and M. Spriggs, "Helping hpc users specify job memory requirements via machine learning," in *2016 Third International Workshop on HPC User Support Tools (HUST)*, 2016, pp. 6–13.