Combining Static and Dynamic Storage Management for Data Intensive Scientific Workflows

Nicholas Hazekamp, Nathaniel Kremer-Herman, Benjamin Tovar, Haiyan Meng,
Olivia Choudhury, Scott Emrich, and Douglas Thain
Department of Computer Science and Engineering, University of Notre Dame
{nhazekam,nkremerh,btovar,hmeng,ochoudhu,semrich,dthain}@nd.edu

***Abstract*—Workflow management systems are widely used to express and execute highly parallel applications. For data-intensive workflows, storage can be the constraining resource: the number of tasks running at once must be artificially limited to not overflow the space available in the filesystem. It is all too easy for a user to dispatch a workflow which consumes all available storage and disrupts all system users. To address these issues, we present a three-tiered approach to workflow storage management: (1) A static analysis algorithm which analyzes the storage needs of a workflow before execution, giving a realistic prediction of success or failure. (2) An online storage management algorithm which accounts for the storage needed by future tasks to avoid deadlock at runtime. (3) A task containment system which limits storage consumption of individual tasks, enabling the strong guarantees of the static analysis and dynamic management algorithms. We demonstrate the application of these techniques on three complex workflows.**

I. Introduction

Workflow management systems are widely used to express and execute highly parallel applications in bioinformatics, high energy physics, earth science, data mining, and other fields [1], [2], [3], [4], [5], [6], [7], [8], [9]. A workflow is a set of independent programs which communicate with each other by producing and consuming files or other data streams. Workflows are often organized into a graphical representation. A view of the entire workflow allows the workflow manager to perform resource management by scheduling tasks within limited global resources.

Resource limitations can cause workflow deadlock when the user attempts to utilize more resources than are available. For a system administrator, the limiting resource for throughput is often cores or memory, as their goal is achieving maximum throughput of the system. The limiting resource for a user is storage, which is often quota-based. This establishes storage as the finite resource at risk of causing deadlocks, since a user would have to actively free space after use. This contrasts with cores and memory, which may limit the throughput, but are released at job completion. Regardless of the throughput speed, sufficiently large workflows will fill the limited storage, causing the workflow to deadlock. As storage reaches its limit, the number of tasks that can run simultaneously is limited by the available storage space. A large proportion of the storage consumed may lie in intermediate files between tasks in a workflow. As a result, the order in which tasks execute has a significant effect on the storage consumption of the workflow as a whole. An intermediate file must exist long enough to be consumed by the next task, and then may be deleted. Previous work [10] focused on basic reduction of storage usage, using clean-up jobs to delete files which are no longer needed. From this we can conclude that depth-first execution strategies, that focus of freeing files quickly, will consume less storage than a breadth-first execution strategy, which can result in deadlock if insufficient space is left to process the files already created.

The problem is exacerbated by storage being the most casually managed resource in high performance computing centers. Most batch systems schedule based on job resource needs for cores and memory. Although data centers are often over-provisioned for storage, the process by which a user attains more storage is often tied to paperwork or payment. Neither of these methods prevent or mitigate the chance of deadlock. Users typically have permission to write into a global filesystem with a quota, as well as local temporary storage without physical limits. This can cause tasks to exhibit unexpected behavior and may crash outright when the user's storage space is completely exhausted. Storage exhaustion may also occur if the shard of the filesystem within which the user is located is full even though the user's workflow is within their quota. Though procuring more physical storage hardware could resolve this issue, it depends upon the system's partition policy. Additionally, unlimited use of local storage, such as scratch space, can also result in deadlock when in heavy use. If many users greedily consume local storage, one or more users' work could deadlock due to local storage exhaustion. When this happens all other tasks fail and work halts. The user is left with no recourse except to attempt an execution while deleting files and canceling jobs manually as they approach the storage constraints. Little information is available in advance to understand exactly how much storage a workflow will require.

We present three coordinated techniques that allow users to manage storage at runtime in user-level workflow-structured applications. First, we present how the graph structure of workflow applications allows us to observe the minimum and maximum storage needs of a workflow *before execution*. This enables a scheduler to judge whether a workflow will be able to *run to completion* at all in the available storage, or whether more resources must be obtained. Second, we demonstrate

an online accounting method by which a workflow manager tracks not only the actual storage use, but the storage needed to complete the current graph structure. This method is necessary to *avoid deadlock* that would be caused by a naive approach. Third, we demonstrate several techniques by which individual tasks may be monitored and contained so they do not overflow their expected storage consumption. A key challenge here is to *identify storage exhaustion* so the workflow manager can stop the task and re-plan. As a case study, we demonstrate how these techniques have been applied to a selection of workflows, including a binary tree, the Montage image analysis workflow, and a genomic analysis workflow. For each, we demonstrate that the measurement technique accurately captures the minimum and maximum storage footprint, that arbitrary runtime storage limits are respected, and that individual tasks are contained with modest overheads compared to uncontrolled execution.

II. Background

We assume a workflow is a directed acyclic graph (DAG) of *programs* and *files* produced and consumed by each program. Nodes in the DAG are logically ready for execution when all input files exist at the execution site. The workflow structure is recorded in a file authored by the end user, while the individual programs and files are assumed to be present in the filesystem at the execution site or accessed via URLs in the workflow itself. Files may be divided into three categories: *input files* which exist before the workflow begins, *output files* which are produced by the workflow and must be retained, and *intermediate files* which are created within the workflow but may be deleted once they are no longer needed. The size of input files is known in advance, and an estimate of the size of intermediate and output files is stated in the workflow.

Figure 1 shows the relationship between a workflow, running tasks, and the available storage. The workflow itself is fed into a *workflow manager* which dispatches individual tasks to a batch system that selects an execution node for each task. Each task has a *task sandbox* in which it executes, which must be large enough to contain the inputs, outputs, and auxiliary files needed for that single task. Likewise, the workflow as a whole has a *workflow sandbox* which must be large enough to contain the inputs, outputs, and intermediates files of the workflow.

Typically, the workflow sandbox is stored in a large parallel filesystem used for user data and home directories, such as Panasas [11], Lustre [12], or Ceph [13]. We assume that this storage is large and fast, but of finite capacity. There are two common configurations for the task sandbox. Some systems have storage local to each node, so the task's storage consumption is different from the workflow. Some systems do not have storage local to each node, and so the task's sandbox exists in the global filesystem and must be accounted against the workflow's storage. In either case, each task's consumption must be allocated and measured.

In the context of workflow and task storage management, we make three contributions:

1 - Static analysis of the storage footprint. We present an algorithm which statically determines the storage footprint

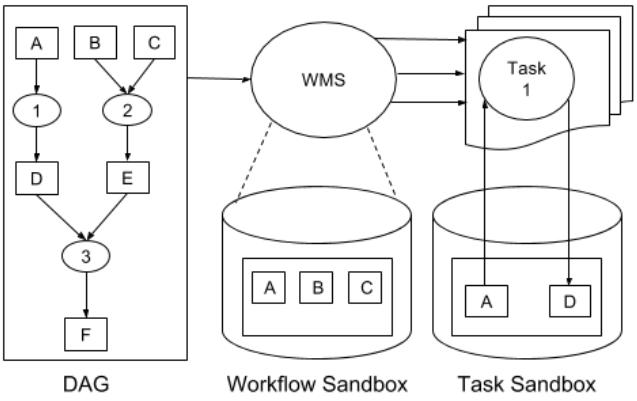


Fig. 1: System Architecture. In the DAG, files are represented as squares and programs as circles. When a node is submitted, the input files are sent to the task sandbox, where upon completion the output files are retrieved. These files are held in the workflow sandbox, which is managed by the Workflow Management System.

of a DAG before execution. The algorithm uses a single-pass, bottom-up approach to determine the storage needs for the execution of each task. As the algorithm traverses up the tree it accumulates values that correspond to the storage needs of the task's descendants in a global context. Once the traversal is complete, the set of head nodes is used to determine the overall needs of the DAG as well as inform concurrent branches on the loose ordering to maintain limits. The algorithm makes a single pass through the graph, limiting the cost of analysis. Utilizing this analysis to prevent deadlock, as opposed to a scheduling algorithm, limits the runtime costs while still providing a precise minimum and maximum data footprint.

2 - Online management of the storage footprint. The dynamic management of the DAG is performed in two ways: storage allocation and task dispatch. The storage allocation works as a transaction hierarchy. When a task is committed, space is reserved for descendants as well. When a node is selected to run, the data footprint from static analysis provides a limit and the length of that commitment. When a task completes, the appropriate files are deleted, and the committed storage is updated. The task dispatch component works in conjunction with the storage allocation, by using the commitment hierarchy and the eligible node's footprint to determine if there is enough available space. If so, the node's footprint is committed and the node is submitted for execution.

3 - Runtime containment of individual tasks. The assumptions made about file size in the static and dynamic algorithms must eventually be enforced at the execution site. The necessity for strict runtime storage containment for a task is two-fold: first it gives a guarantee to the workflow management system that the task in question will only use as much space as it has been allocated. Secondly, it honors the resource request between the task and the execution node which states the task will not consume excess resources, which could alter the availability of the node for future tasks. We provide these guarantees by mounting a loop device filesystem,

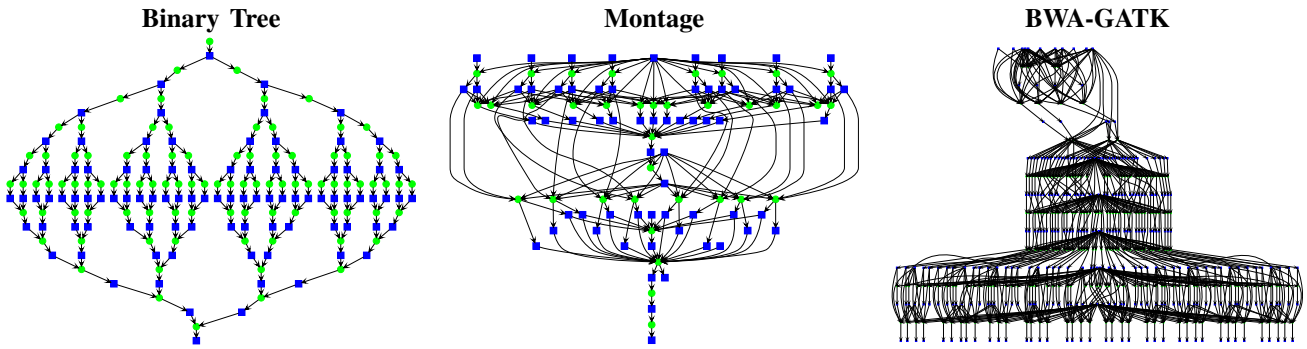


Fig. 2: Example Workflows

Three example workflows used to evaluate storage management strategies. **Binary Tree** *is a synthetic workflow which generates 1GB files in a tree structure, resulting in a large amount of intermediate data.* **Montage** *is a widely used workflow benchmark which produces image mosaics from raw astronomical images.* **BWA-GATK** *is a bioinformatics workflow that performs alignment and genotyping of sequences related to the oak tree. Each graph shown is reduced in cardinality from the actual workflow in order to more clearly show the general structure.*

a file backed pseudo-disk, in the place of the task sandbox. From the task's perspective, it appears that it has an entire device to itself with the right amount of space needed to fully execute and report back to the workflow management system. The loop device also guarantees that the task will consume no more space than it has been allocated. If the loop device is exhausted by accident, a library interpositioning tool detects the exact moment of overage and communicates the details to the workflow manager.

III. The Storage Footprint

We begin static analysis by defining the storage footprint of a workflow and computing the footprint manually for several simple examples. This will serve as a basis for the static algorithm in the next section.

For any given workflow, the **absolute maximum** storage it can consume occurs when all files (input, intermediate, and output) exist simultaneously. If the target storage system has enough space for the sum of all files mentioned in the workflow, then it is not necessary to delete any files during execution, and there is no storage management problem.

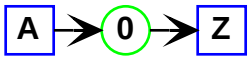
If storage space is limited, then we may delete intermediate files incrementally, whenever they have been consumed and are no longer needed by any node in the workflow. We define the **storage footprint** of the workflow to be the maximum amount of storage consumed during execution with the policy that all files are deleted at the first possible opportunity.

Footprint is affected by the concurrency used to execute the workflow. We define two extreme values of the footprint:

- **Maximum storage footprint** is the largest possible footprint achieved when tasks run with maximum possible concurrency, subject to the workflow ordering constraints.
- **Minimum storage footprint** is the smallest possible footprint, achieved when only one workflow branch is executed at a time, and possibly concurrent tasks are executed in the order that minimizes the footprint.

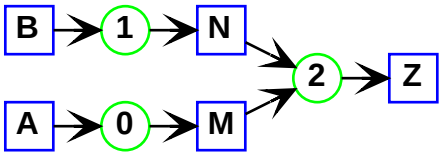
By computing these values *before* executing the workflow, we can give the user a realistic assessment of the likelihood of success. If the available storage is less than the minimum footprint, the workflow cannot run to completion *at all*, so the end user is advised to look for another system or acquire more storage. If the available storage is between the minimum and maximum footprint, the workflow can complete but concurrency must be limited dynamically. If the available storage is at or above the maximum footprint, then the workflow can run at maximum concurrency if files are deleted at the first opportunity.

Here are the footprints of a few simple workflows:



Example 1: A single task 0 reads an input file A and produces an output file Z. The size of a file is defined as $|X|$, where X is the file. At some point during the execution (however briefly) both A and Z must exist simultaneously, so the footprint of the workflow is the sum of the size of the files $|A| + |Z|$ which we abbreviate as $|AZ|$. After the task completes, A may be deleted, but Z remains, so the residual file of the workflow is Z.

Example 2: Two tasks execute in sequence. Intermediate file M is then created by executing task 0 with input A, after which file A is no longer needed and can be deleted. Next, output file Z is created by executing task 1 with input M, at which point file M can also be removed. This results in only output file Z remaining in the end. A and M must exist simultaneously, and M and Z must exist simultaneously, so the footprint is the $\max(|AM|, |MZ|)$, and the residual is the sole output Z.



Example 3: Two tasks combine outputs in a third. This workflow can execute three different ways:

- Case 1: If tasks 0 and 1 can execute simultaneously, then files A, B, M, and N must co-exist. Files A and B can be deleted, at which point M, N, and Z must co-exist. The footprint is $\max(|ABMN|, |MNZ|)$.
- Case 2: If task 0 executes first, then files A, B, and M co-exist, after which file A can be deleted. Then, task 1 executes, so files B, M, and N co-exist. File B can now be deleted. Finally, task 2 executes, so M, N, and Z co-exist. The footprint is the largest of the three steps: $\max(|ABM|, |BMN|, |MNZ|)$
- Case 3: If task 1 executes first, the combinations are similar to Case 2: $\max(|ABN|, |AMN|, |MNZ|)$

As can be seen, the maximum storage footprint occurs in case 1, while the minimum storage footprint is the minimum between cases 2 and 3. This presents the runtime manager of a workflow with a trade off – increased concurrency can result in increased storage consumption. If this is not carefully quantified, storage may be accidentally exhausted.

IV. Example Workflows

Figure 2 shows three workflows that we use as running examples. Each of these workflows can be generated at various scales; the figure shows a low-degree example to make the macro-structure clear. Each presents a somewhat different storage management challenge; we can give a qualitative sense of the footprint by examining the workflow structure.

The **Binary Tree** workflow is a synthetic benchmark consisting of processes that each consume and produce a single file of 1MB. Each file is consumed by two children until the desired depth d, and then the data is reduced to a single file in a similar manner. If all branches are executed concurrently, the maximum footprint is $2^d + 2^{d-1}$, when two levels co-exist at once. The minimum footprint is $d + 2$, when a single branch plus one task must exist simultaneously. Many values in between may occur if the workflow proceeds unevenly. The footprint tends to peak in the middle of execution.

The **Montage** [14] workflow computes mosaics of astronomical images, and is widely used as a benchmark for evaluating workflows. It can be generated at a variety of scales by varying the angle of sky (and thus number of images) to be processed. Although Montage has been previously used to explore storage consumption [15], it is an unusual workflow in that most intermediate files it creates are used by multiple later stages, such that most files cannot be deleted until the final chain of individual jobs. Thus, the footprint tends to increase slowly until reaching a peak near the end of the workflow.

The **BWA-GATK** [16] workflow combines two common bioinformatics tools into a large scale parallel application. A

Algorithm 1 Algorithm to Measure Storage Footprint.

Term	Definition
n	Current node under examination
n.descendants	Nodes that utilize a file produced by n
n.children	Nodes related to n where no other descendant of n is parent, subset of descendants
n.residual_nodes	List of nodes where all children are used
n.residual_files	Files held until nearest residual node
n.diff	Difference in size between n.min_footprint and n.residual_files
n.run_footprint	Files used/created during n's execution
n.diff_order_footprint	Files forming largest footprint during diff order traversal
n.wgt_order_footprint	Files forming largest footprint during wgt order traversal
n.min_desc_footprint	Files forming minimal space needed to execute to next residual
n.min_footprint	Files forming minimal space needed to execute self and children
n.max_desc_footprint	Files forming maximal space needed to execute to next residual
n.max_footprint	Files forming maximal space needed to execute self and children

procedure MeasureStorageFootprint(n)

for all c **in** n.children **do**
  MeasureFootprint(c)
end for
n.residual_nodes $\leftarrow$ n + ($\bigcap_{c}^{\text{n.children}}$ c.residual_nodes)
if |n.children| < 2 **then**
  n.residual_files $\leftarrow$ n.outputs
else
  n.residual_files $\leftarrow \max($n.outputs, $\bigcup_{c}^{\text{n.children}}$ c.residual_files$)$
end if

n.run_footprint $\leftarrow$ n.inputs + n.outputs

n.diff_order_footprint
tmp footprint $\leftarrow$ n.outputs
for all c in n.children sorted by c.diff **do**
  if |footprint| + |c.min_footprint| $>$ n.diff_order_footprint **then**
    n.diff_order_footprint $\leftarrow$ footprint + c.min_footprint
  end if
  tmp footprint $\leftarrow$ footprint + c.residual_files
end for

n.wgt_order_footprint
  $\leftarrow \max_{c}^{\text{n.children}}$(c.min_footprint) + $\sum_{r}^{\text{n.children - c}}$ (r.residual)
n.min_desc_footprint
  $\leftarrow \min($n.diff_order_footprint, n.wgt_order_footprint$)$
n.min_footprint
  $\leftarrow \max($n.parent_footprint, n.min_desc_footprint$)$
n.max_desc_footprint $\leftarrow \bigcup_{c}^{\text{n.children}}$ (c.max_footprint)
n.max_footprint
  $\leftarrow \max($n.parent_footprint, n.max_desc_footprint$)$
n.diff $\leftarrow$ |n.min_footprint| $-$ |n.residual_files|
end procedure

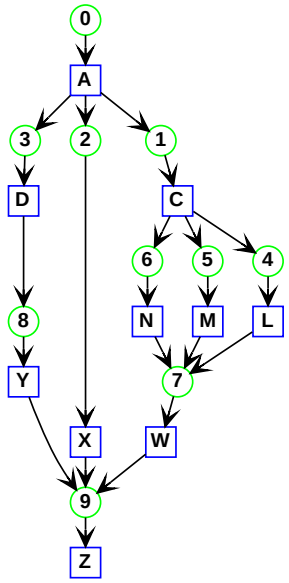


Fig. 3: Worked Example DAG

Node (Res Nodes)	Min Footprint	Max Files	Residual Footprint	Run Footprint
9 {9}	4 {WXYZ}	4 {WXYZ}	1 {Z}	4 {WXYZ}
8 {9,8}	2 {DY}	2 {DY}	1 {Y}	2 {DY}
7 {9,7}	4 {LMNW}	4 {LMNW}	1 {W}	4 {LMNW}
6 {9,7,6}	2 {CN}	2 {CN}	1 {N}	2 {CN}
5 {9,7,5}	2 {CM}	2 {CM}	1 {M}	2 {CM}
4 {9,7,4}	2 {CL}	2 {CL}	1 {L}	2 {CL}
3 {9,8,3}	2 {AD}	2 {AD}	1 {D}	2 {AD}
2 {9,2}	2 {AX}	2 {AX}	1 {X}	2 {AX}
1 {9,7,1}	4 {CLMN}	4 {CLMN}	3 {LMN}	2 {AC}
0 {9,0}	4 {AWXY}	7 {ACDLMNX}	3 {WXY}	1 {A}

Fig. 4: Worked Example Variables

Worked example of DAG, in Figure 3, with the relevant variables shown in Figure 4.

large genomic query file is split into task-sized pieces, and the BWA alignment tool aligns the queries to a reference dataset. Then, the GATK tool uses a Bayesian algorithm to compute the quality of successful alignments. The workflow size is increased by adding more data from multiple organisms. This workflow has an irregular footprint over time: each stage of the workflow produces files which are used only once and then may be deleted. In some cases, single files are consumed by single tasks in parallel, so the footprint changes in a fine-grained manner. In other cases, multiple files must be consumed by all tasks in a stage, creating a storage barrier in which nothing is deleted until all tasks complete.

These three workflows are not intended to present a complete profile of workflow behavior, rather show common behaviors which may result in deadlock given certain conditions. The static and dynamic algorithms we present are suitable for running these workflows in most distributed computing environments. In this paper we utilized a centralized batch system due to its wide-spread use, but the work we present does not rely on a batch system environment. Further, we do not make any assumptions about typical file sizes, workflow behavior, or workflow needs. We do assume the user can accurately estimate characteristics of their workflow to account for any special needs or storage behavior it may experience during execution.

V. Static Analysis Algorithm

The algorithm used to analyze the abstract DAG utilizes a single pass bottom-up approach for determining the estimated storage utilization. The algorithm is presented in Algorithm 1. The goal of the algorithm is to determine the storage using information gathered at each node and passed upward.

A residual node is defined as any node that is the lone child of another node or nodes. This is useful as it provides a limit on the look-ahead needed to accurately determine storage needs of the parent node. If a node's children culminate in a single node the storage impact of these nodes is limited between the node and the residual node where they culminate. The storage for the node's children can be calculated and saved at the node for future use. The value at the node now represents the space needs to be committed for guaranteed execution.

Traversal begins from leaf nodes, where the residual nodes and files include itself and its outputs. The only relevant footprint at this location is the run footprint, which is also the minimum and maximum footprints. As we traverse up the DAG, there are two states in which a node resides. A node in the first case has only a single child node, at which point the residual nodes, files, and run footprint are equivalent to the child's residual nodes, files, and run footprint respectively. As there is only one possible ordering for child execution, the minimum and maximum footprints are defined as the minimum and maximum respectively, between the node and its child's footprint.

In the case where there are multiple nodes, the ordering can affect the overall footprint. To evaluate the interplay between children we find the common subset of residual nodes shared by each child. This becomes the residual node set for the current node. Of the remaining uncommon residual nodes, the residual files and the largest footprint are found for each child. The node's residual file set is the union of all children's residual files. We evaluated two methods of traversal, each with its own goal. The straightforward approach is to find the largest footprint among the children and add the remaining children's residual files. The other approach is to order the children by the difference between their minimum footprint and residual files. Traversing the nodes in this order favors nodes that consume and release space instead of nodes that hold their consumed space. To pick between which is better we select the minimum between the two as they both account for executing all of the nodes. The minimum footprint

Workflow Size	Min (GB)	Max (GB)	Abs (GB)	Tasks	Analysis Time(s)
Binary Tree					
3	5	12	22	22	0.0015
5	**7**	**48**	**94**	**94**	**0.0065**
10	12	1536	3070	3070	0.2776
15	17	49152	98302	98302	10.631
Montage					
0.01	0.07	0.12	0.13	35	0.0041
1	1.87	2.78	2.98	998	1.9558
1.99	**4.01**	**6.49**	**9.33**	**2984**	**2.3500**
BWA-GATK					
1	2.69	2.69	3.614	53	0.0117
5	2.75	13.46	17.99	265	0.0439
10	**2.82**	**26.93**	**35.94**	**530**	**0.0749**
25	3.06	67.31	89.83	1325	0.1659
50	3.43	134.63	179.64	2650	0.3082
100	4.19	269.25	359.24	5300	0.4145
500	10.25	1346.26	1796.11	26500	2.8163

TABLE I: Static Analysis Results

Each section of this table refers to one of the three workflows used in our analysis. The size column refers to different attributes for each workflow. Size refers to the number of split levels in Binary Tree, the degree of the sky being analyzed in Montage, and the number of individual samples included in BWA-GATK. Min shows the estimated minimum footprint, Max the estimated maximum footprint, and Abs the sum of all files in the workflow. Tasks are the number of tasks created in Makeflow. Analysis Time shows the additional time needed to determine the footprint for each case.

is determined by utilizing the largest of the current nodes footprint and the footprints reported by its children. The maximum is the sum of all the children nodes' maximum footprints.

Figures 3 to 4 shows the algorithm applied to a simple workflow. Assuming each file is of size 1, the minimum footprint (ACLMN) occurs if the bottom-most branch of the workflow is executed first. Files LMN exist simultaneously, before being reduced to W, allowing the rest of the workflow to proceed. The maximum footprint (ACDLMNX) occurs when all three branches execute concurrently, so that DLMNX all must exist at once, along with the input files (AC) to the tasks that created them.

Table I shows the results of applying this algorithm to the three example workflows previously described, as implemented in Makeflow [9]. For various sizes of each workflow, we compute the minimum footprint, maximum footprint, and absolute maximum. The size of the three workflows are given as the tree depth for Binary Tree, the degrees of resolution for Montage, and the number of organisms for BWA-GATK. The bold lines indicate the configurations actually run below. As discussed above, the maximum footprint of Binary Tree grows exponentially, the maximum footprint of Montage is close to the minimum footprint, and the maximum footprint of BWA-GATK is roughly linear with the width of the workflow. With the exception of the very large binary tree, the single-pass algorithm executes in a mater of seconds.

A. Limitations

This static analysis perform well in an organized consistent environment, but there are several factors that affect execution. The first is untracked files. Often in execution programs create log files or auxiliary status files. These files have limited scientific worth outside of performance and error logs, but occupy no space. If they are specified in the task then we account for them and remove them when no longer needed. When they are not specified they clutter space and are never cleaned causing more contention. Second when files vary significantly from expected size, such as log files, the static algorithm does not recompute to account for this. Ideally, we recompute and reallocate using the dynamic management, but if the limit is already close it may be beyond the point where a change will help.

To combat these issues, we have additional mechanisms to help determine if the environment is prohibiting forward progress outside of the bounds of our management. This comes up by actively cleaning old files and watching the working directory. If the filesystem report almost full utilization of storage error messages are printed to bring the users attention to the issue, though the static analysis does little to help prevent this.

VI. Dynamic Storage Management

In the previous section, the static allocation defines the storage bounds of the workflow execution. The dynamic algorithm utilizes the static analysis results to enforce the space reservations needed for future execution.

The dynamic storage allocator uses a data structure that tracks the current space utilized by files and reservations of current and future nodes. The data structure is initialized with a base size, called the base allocation, which is specified by the end user with the help of the static analysis results. The base allocation defines the upper limit of available space for this execution of the workflow. Within the base allocation, reservations are created to hold space for a node and its ancestors. This creates a hierarchy in the data structure of node reservations above their descendant reservations. When a file is created it is accounted for in the current reservation and tracked until deletion.

A. Dynamic Storage Algorithm

The dynamic algorithm consists of three stages for each node's execution: verification, allocation, and release.

When a node is logically ready for execution, the dynamic storage allocator checks if there is sufficient space to run the selected node. The allocator utilizes the residual node set to compare against the existing data structure. The allocator starts with the lowest residual node (nodes later in the workflow) and compares the data structure with the required space for the residual node. If the residual node reservation does not exist in the data structure, the allocator checks for sufficient space in the base allocation. If the reservation exists, but there is not sufficient space, the allocator checks if the existing hierarchy can be enlarged to reserve the additional size. If there is sufficient space in the reservation hierarchy, the allocator continues up the residual list using the available space to check nodes. If there is not sufficient space, the node is postponed for submission.

If the verification step is successful, the node is allocated and submitted. To allocate a node, its residual nodes are passed to the allocator. The allocator creates a reservation for each residual node that does not exist in the data structure and grows smaller allocations to accommodate. This proceeds for all residual nodes until a hierarchy exists above the base allocation. In this hierarchy, any lower node is at least as large as the nodes above it. If multiple nodes share a common residual node, the common node is at least as large as the sum of the higher nodes. In this way, space is reserved for the widest part of the ancestors to guarantee space.

After execution, the allocator must release the completed reservations. During this release stage, files that are no longer needed are deleted and their space is marked as available in the data structure. The reservation for the completed node is released, and files within the reservation are transferred to the containing reservation. These files continue to exist until they are no longer needed. If output files are larger than the existing reservation, the reservation attempts to grow and accommodate the increased size. This can cause deadlock or failure from resource exhaustion. Due to these changes, the static analysis is outdated and can no longer guarantee execution.

B. Worked Example

Figure 5 demonstrates the dynamic algorithm using the earlier worked example (Figures 3 to 4).

Step 1: With Node 0 available to run, the algorithm checks the stack to ensure there is enough space. Traversing the residual nodes of Node 0, Node 9 is added to the stack with the size needed for the nodes between 0 and 9. Node 0 is then reserved on top of Node 9. Once executed, File A moves from the space reserved by Node 0 to Node 9, and node 0's reservation is removed.

Step 2: With File A existing, Nodes 1, 2, and 3 are available to run. The algorithm will check Node 1. Traversing Node 1's residual nodes, Node 9 is reserved. With the remaining space in Node 9, the algorithm can fit Node 7 and Node 1. A reservation is then added to 7 above 9, and a reservation for Node 1 is created on Node 7. After completing Node 1, File C is created and Node 1's reservation is removed.

Step 3: The available nodes to run are Nodes 2, 3, 4, 5, and 6. There is not enough space for Node 2 and 3. However, in the reservation for Node 7, there is space for Nodes 4, 5, and 6. Space for Nodes 4, 5, and 6 is reserved. Upon completion, files L, M, and N are created and the node reservations are released. File C is no longer needed and is removed.

Step 4: With Nodes 2, 3, and 7 available, again we check which nodes will fit. The reservation for node 7 already exists, so Node 7 is executed. After outputting file W, files L, M, and N are removed along with node 7's reservation.

Step 5: Space now exists for Node 2 and 3, so reservations are created for Nodes 2, 3, and 8. After running, file A is removed, and reservations 2 and 3 are released.

Step 6: Reservation for Node 8 exists. File Y is created. File D is removed, and Node 8's reservation is released.

Step 7: Reservation for Node 9 exists. File Z is created. File W, X, and Y are removed. Node 9's reservation is released.

Step 8: Final output File Z exists. The workflow is complete.

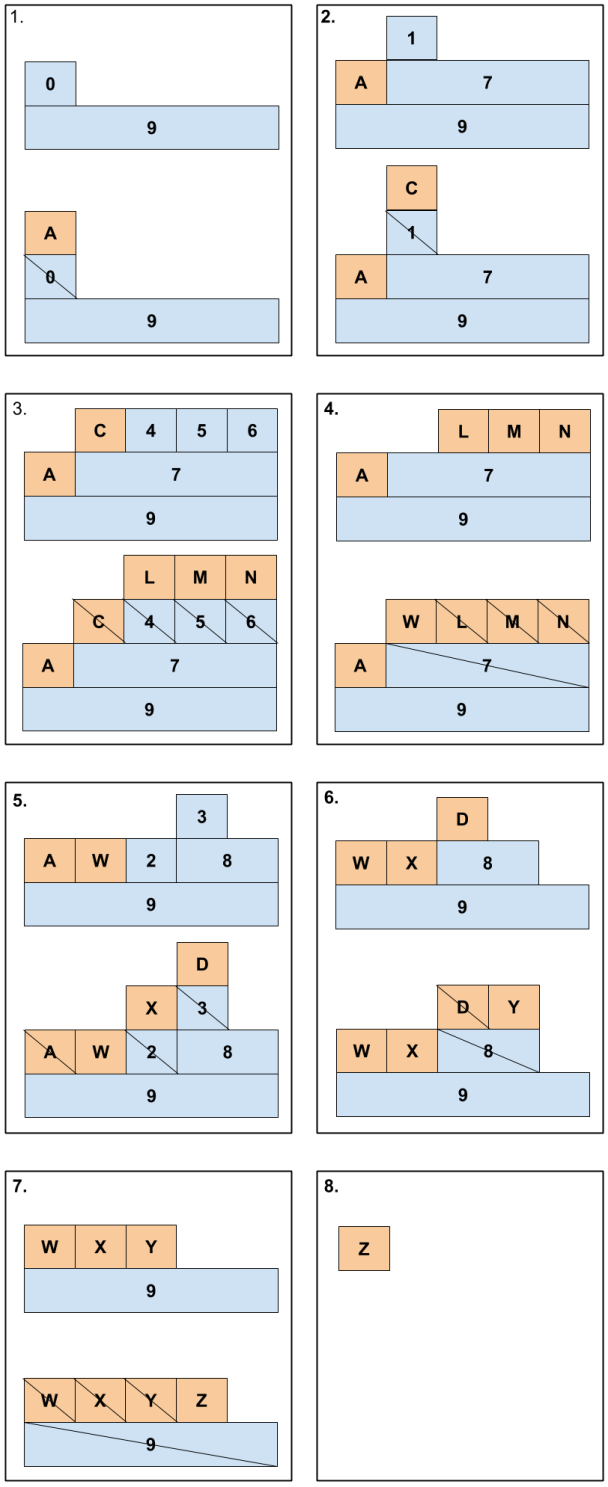


Fig. 5: Dynamic Allocation Data Structure. Shown is the execution of the worked example from Figure 3. Numbered boxes represent storage allocations for nodes. Lettered Boxes represent allocations for specific files. Boxes with slashes through them are allocations which are being removed because either the node is complete or the file is no longer needed. Allocations are removed when no longer relevant.

C. Impact of Local Storage

Local storage can affect the dynamic storage management in several ways. First, if the local storage that is utilized for execution is accounted for within the same quota, then the dynamic management needs to adjust. In this case, we need to account for both the active space utilized during execution and the space consumed in caches are used for data movement. In the prior case, when a node is allocated the requisite space needed for execution is added to the allocation, but removed after execution. Caches are more difficult as they are essentially copies of the existing data and may persist between execution increasing the complexity. One method for handling this is to set a limit on individual size of a cache such that old files are removed and the space is statically accounted for.

Second, in cases where temporary space is unaccounted for, such as in scratch spaces or local temporary directories, we do not need to account for space. These resources can still become contentious. In these cases we rely on the remote execution to report on limited space to maintain limits. Unfortunately, system wide storage contention on scratch storage is not in the purview of this paper.

Currently, we do not consider either as our remote execution happens on local temporary space outside of the quota, though inclusion of these factors may become necessary in some execution environments.

VII. Task Constraints

The methods we have presented so far have focused on the storage footprint at the workflow level, but we must also be concerned with constraining resources at the task level. To prevent a compute node from exhausting storage, we sandbox each task into its own directory to prevent interference with other host processes. However, a directory is not strict enough to prevent storage exhaustion. Enforcement of storage constraints is not well supported in POSIX, but Linux loop devices provide passive storage constraint of a sandbox. In a loop device sandbox, a task cannot progress once it has used the storage space available to the device. This prevents the problem of storage exhaustion from impacting other tasks since the exhausting task is contained within a loop device. A loop device is considered a pseudo-device. Within the system it is viewed as a physical device, while in reality it is a file created to a defined size then mounted in the sandbox namespace.

A. Greedy Space Allocation

There is a need to be cautious when allocating space for each task at the workflow site. If not enough space is allocated to a task, and the resulting outputs are larger than the allocation, we will exceed the task's allocated space. However, if too much space is allocated to each task we risk wasting time waiting on unused space, or worse, deadlocking due to lack of space left for progress. We assume the space required by a task includes the space needed to execute the task. We also assume that user labeling of tasks is accurate enough to prevent deadlock due to requesting exorbitant storage space. Users can label their tasks with a slight overhead to account for any intermediate data produced during task execution. This overhead is in addition to the allocation size of input and output data. While this approach is somewhat greedy, it prevents the dynamic algorithm from committing more tasks to a resource-limited execution space than the space could handle. The extra overhead is added to a loop device's guaranteed storage space.

B. Mechanisms

Even with a hard limit to a task's disk space, another problem remains. The workflow management system must receive a report when the task uses up all of its allocated storage. Accurate reporting of this failure is non-trivial. Storage exhaustion failures must be reported immediately. There are multiple methods to report this kind of failure.

One way to report failures is through a sentinel process. This kind of process periodically collects information on a task, and it will report back to the workflow management system when it notices that the task has surpassed its requested storage. The sentinel process is not an ideal solution for resource-constrained systems. It cannot guarantee that it will kill a task and report its storage exhaustion as soon as it happens. This is not the case with a dynamic library reporting mechanism which we have implemented to use with loop devices.

We chose loop devices because they provide a generic interface. However, this is not a perfect solution either as it requires superuser access. Unlike the abstracted view of sentinel processes, a loop device is a kernel-provided measure to ensure storage constraint. Upon storage exhaustion, a task will fail and throw an `ENOSPC` error. This is the key to letting the workflow management system know that the loop device is full. At this point, the workflow management system may decide to re-dispatch the task with a larger allocation.

To properly capture task storage exhaustion, we have implemented a dynamic library which overloads the `write(2)` and `open(2)` standard functions via LD_PRELOAD. This is to catch any `ENOSPC` errors due to a loop device filling its capacity as soon as it occurs. By intercepting errors relating to the loop device's capacity in this method, we avoid any complications from a task having similar error handling. If a task does have error handling which catches `ENOSPC` errors, it may report back with a different, more ambiguous error code. By our handling of the `write(2)` and `open(2)` calls with a library interpositioning tool, we can know that a task exhausted its storage space regardless of the task's internal error handling.

C. Performance Overhead

We measured the performance overhead of loop device allocations with three micro-benchmarks: read, write, and file metadata throughput. Our test platform has the following specifications: 3GB RAM, 2 x Intel Core 2 Duo processors, 250GB SATA drive (16 heads, 7,200 RPM, 16,383 cylinders, 63 sectors/track, queue depth 32, maximum I/O data transfer rate 300 MB/s, maximum sustained data transfer rate 125 MB/s, internal data transfer rate 1,695 Mb/s). Read and write throughput tested a loop device's speed at sequentially reading

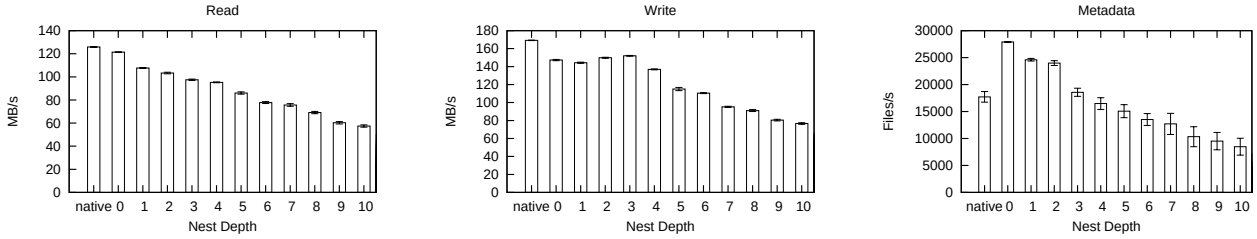


Fig. 6: Loop Device Performance of ext4 Filesystem

a file 95% of its allocation size and sequentially writing a file 95% of its allocation size respectively. Metadata throughput was measured by timing a loop device's creation of 10,000 blank files, retrieving file status on each, then deleting each file. We used the commonly available ext4 filesystem for these tests.

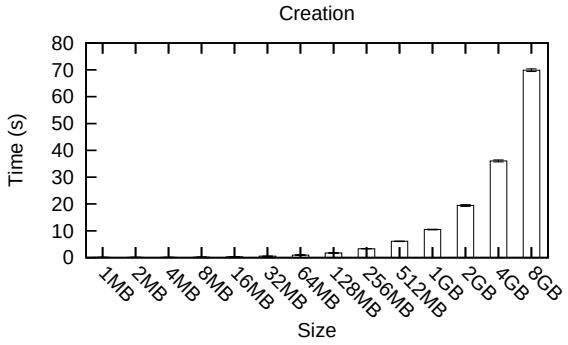


Fig. 7: Loop Device Creation Performance of ext4 Filesystem

Each micro-benchmark test ran for twenty iterations then created a nested loop device within the current device. We provide results from the first ten nest depths since the trend in performance can be seen that early. As shown in Figure 6, this nesting shows an overall trend toward reduction in device performance the deeper the device is nested. We note that there were some minor increases in performance between adjacent nest depths such as between nest depths 1, 2, and 3 in the write test. These were caused by outliers, but the overall trend shows a reduction in performance the further down we go in nest depth. Though nesting loop devices is feasible, it is not the expected behavior of a workflow. It does demonstrate, however, that an entire workflow can run within a storage container and still reasonably perform. The aim of these micro-benchmarks was to demonstrate that a loop device does not sacrifice much in the way of performance in order to provide a resource guarantee and extra safety for the native filesystem.

We tracked the creation and deletion times for a loop device of varying sizes. Like in the performance overhead testing, we ran each test for twenty iterations. As demonstrated in Figure 7, we began our tests with a size of 1MB and proceeded exponentially to a size of 8GB. The creation time for loop devices under a gigabyte took trivial time on our test platform. Larger loop devices took much more time to instantiate, yet they still took about a minute to set up. All deletion times took less than half a second; faster than a user would notice. Since the removal of a loop device is akin to the removal of a file, the majority of the work was in removing file references used for the device.

We also measured overhead while running the BWA-GATK workflow once with no loop device workers and the second time with only loop device enabled workers. Ten workers were used for both tests. Each worker ran on identical hardware, used a single core, and was allocated 10GB of memory and 32GB of disk space to use as a sandbox.

Figure 8 shows the effect using loop devices has on runtime and throughput of the whole workflow. The overhead is negligible primarily because the allocation and deletion times are very short compared to the length of most tasks. We can also see that the read and write performance is still quite comparable to the non-loop device run. The normal execution of BWA-GATK ran for around 4174 seconds (approx. 1.16 hours) while the loop device enabled test ran for about 4273 seconds (approx. 1.19 hours). The total performance overhead resulted in a 99 second longer execution which is negligible compared to the total runtime. Like in the runtime overhead, the throughput does not see a significant overhead incurred. The overall throughput (tasks per second) of both runs of the workflow are generally the same throughout the workflow's lifetime.

D. Loop Device Evaluation

The results presented in Figure 6 display the median value across each nest depth along with its respective median absolute deviation. This format was chosen to present the most accurate representation of loop device performance since a few instances of outliers caused the use of mean and standard deviation to be unfairly skewed. As is demonstrated by Figure 6, the performance of a loop device mounted with the ext4 filesystem performs comparably in many cases with the native filesystem, which in our testing is also ext4. By using non-trivial sizes for the test allocations, our test suite simulates sustained I/O as would be expected of a batch job task completing a realistic work load as opposed to burst I/O from a task smaller than 1GB. Considering the vast majority of workflows execute within a nest depth of zero, the most pertinent results are located within the first few nest depths compared against the native filesystem's performance. The rare case of a nested loop device task would occur if a batch job task were to execute a workflow (separate from the workflow

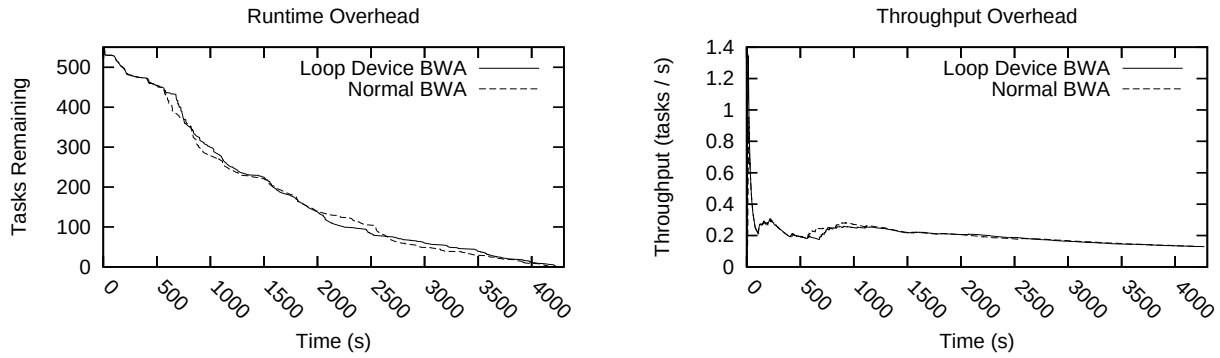


Fig. 8: Loop Device Overhead running BWA-GATK

which spawned this task) locally. For added perspective, we have provided the performance results down to a nest level of ten. This would be indicative of the performance of a batch job task recursively executing ten separate workflows locally. In both read and write tests, we examine the throughput (in MB/s) of the given nest depth (from the native filesystem to a loop device nest depth of ten).

Considering read performance first, we note an equivalent performance to the native filesystem with a nest depth of zero (a single loop device). However, as nest depth is increased (a loop device is mounted within a parent loop device) there is a linear decrease in read speed. We demonstrate that the added constraint provided by loop devices does not come at the cost of performance for the vast majority of cases. Should a user execute a workflow which would make use of nested loop devices, they must consider the nest depth at which they see diminishing returns for their increased resource security. Reasonable, though not comparable, performance can be expected within the first four depth levels of loop devices. After this point, read performance begins to become more divergent from the speed of the native filesystem.

Write performance degrades about as gracefully as read performance though it has a lower initial throughput. The initial few loop device nest depths have reasonable performance though not as comparable as in the case of the read tests. Again, the user would have to consider the added cost slightly decreased write speed for resource guarantees. Depending on the user's preferences, they may find it in their best interests to execute without loop device constraint.

Metadata performance is notably faster for the first four depth levels. This is due to the asynchronous operation of loop devices which is enabled by default. Because the metadata performance is measured upon how quickly the device can instantiate 10,000 empty files, we note this asynchronous batching method is quicker for metadata operations. It is worth noting that the lead a loop device will have over the native filesystem drops off after a nest depth of five, at which point the metadata performance is equivalent to the native filesystem. Eventually, this performance degrades which would necessitate the user to determine if the added security of loop devices at that depth outweighs its slower performance.

VIII. Overall Evaluation

Figure 9 shows the behavior of the dynamic algorithm implemented in Makeflow [9]. The workflows are evaluated by submitting each task to Work Queue [17], with up to 100 tasks running simultaneously on remote nodes. For each, we chose a storage footprint limit (dotted line) that is larger than the minimum footprint (solid line) but less than the maximum footprint. The dark shaded area on each graph shows the storage actually consumed, while the light shaded area shows the "committed" storage value.

A. Configuration

Each workflow was created and run using the emboldened configuration entry in Table I for each corresponding tool. All three workflows executed using a shared filesystem accessed by batch job workers. Each worker was allocated a single core and at least 8GB of memory. The amount of storage allocated to the worker was based on the the sizes from Table I. The machines used to run the batch job workers were part of a campus-scale cluster and thus varied in their hardware configurations however the method used to acquire batch job workers ensured the worker had access to the requested amount of resources in order to begin work.

Each workflow is run in four different configurations:

The **No Limit** row shows each workflow run with maximum concurrency and no explicit storage limits. The ordering of tasks is solely due to logical constraints, and the exact concurrency achieved depends upon the performance of the tasks in the batch system. As can be seen, each workflow meets or exceeds the minimum footprint, and in the absence of control, exceeds the desired limit or deadlocks.

The **Naive** row shows the workflow system attempting to enforce the desired storage limit by simply examining each file as it is submitted. Storage is committed for the immediate output files of a task when it is submitted. Tasks are only submitted if the committed value can be kept below the limit. This can result in deadlock (as shown) if the workflow manager commits too much space and is unable to execute tasks to consume created files.

We note that both the binary workflow and BWA-GATK experience deadlock using the naive strategy given the storage

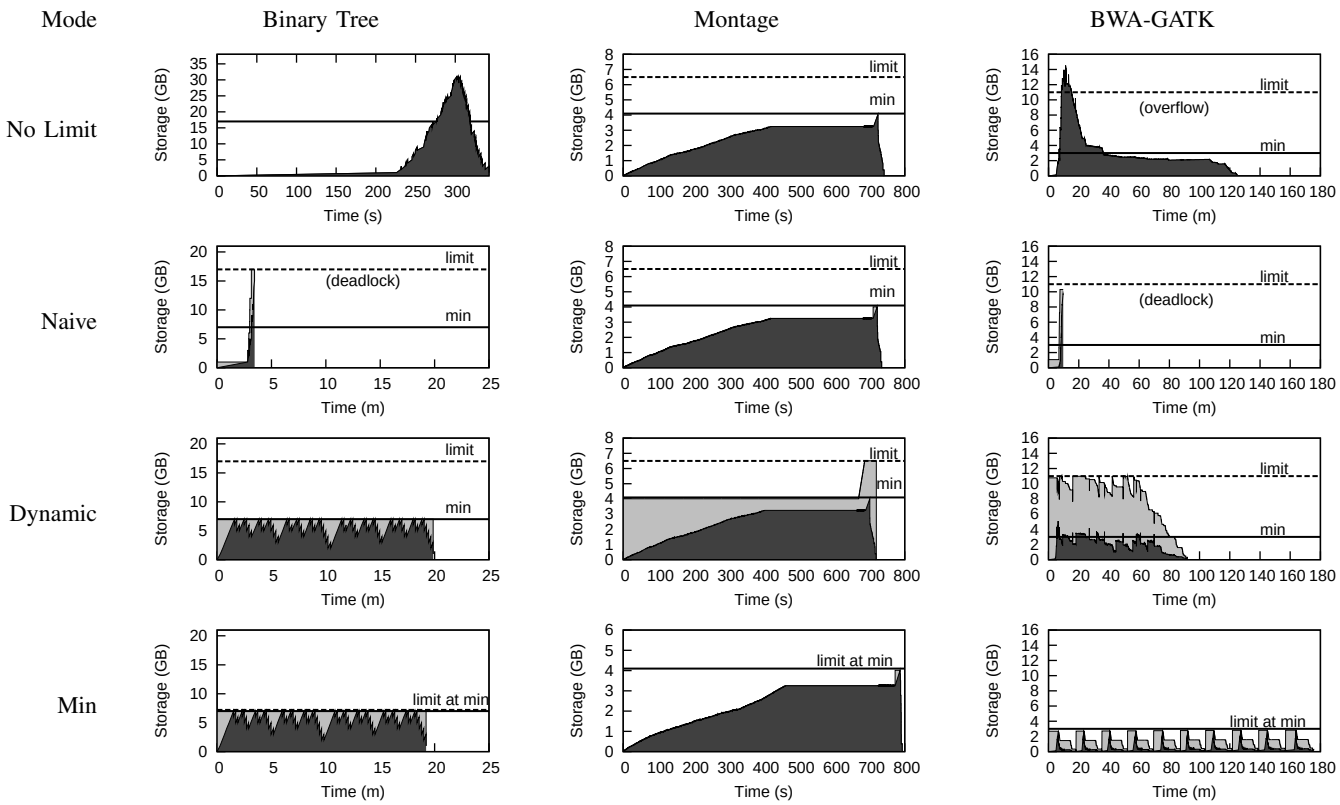


Fig. 9: Storage Limits Applied to Example Workflows

Timeline of storage consumption for each of the three example workflows, in different configurations. The top row shows uncontrolled executions which exceed the desired limit. The second row shows naive storage limits which can result in deadlock. The third row shows a limit applied using the dynamic algorithm. The fourth row shows the minimum storage footprint enforced. In each graph, the dark area shows storage actually consumed, while the light area indicates storage committed to future use.

limit. The binary workflow's rapid expansion behavior caused the naive approach to very quickly exhaust its storage. In BWA-GATK, deadlock occurred when a group of the most data-intensive tasks in the workflow were dispatched at the same time. Some of the data-intensive tasks remained but could not be scheduled because there was not enough storage available.

The **Dynamic** row shows the workflow system using the dynamic algorithm described in this section to limit submission of new tasks. We note that the committed storage increases more rapidly than in the naive case because the footprint of each task is committed before submission. The storage actually consumed does not always rise to the level of the committed storage, which is an upper bound on all possible executions following each node.

The **Min** row shows the workflow system using the dynamic algorithm, but the limit is set to the minimum possible value. As previously mentioned, the Montage Min graph utilizes the file list approach to allocation management.

Overall, we observe that the dynamic online algorithm is able to enforce the desired constraints without falling into deadlock. As noted above, the committed value is an upper bound on actual consumption. Noting this, there still exists room for improvement in the upper bound. The difficulty in optimally utilizing an upper bound is tied closely with the runtime behavior of the workflow. Improvements could be made if the execution time of tasks is combined with the expected growth of a branch in order to overlap branch executions more smoothly. The footprint is currently defined as a means of preventing over-commitment. Optimal storage use focuses more closely upon scheduling as a solution and relies on consistent execution behavior while static footprint analysis relies only on file sizes.

During real execution, we expect the static analysis and dynamic storage algorithms to have broad appeal. The static algorithm can help a user understand the needs and behavior of their workflow better while the dynamic algorithm will help keep the workflow constrained when necessary. The runtime task constraint using loop devices is another way of guaranteeing the user's storage requirements are not exceeded. We believe all three tools have general appeal to scientific workflows regardless of the workflow's behavior or size. The limiting factor of our tools' appeal comes from the user's storage needs.

IX. Related Work

A wide variety of work [18], [19], [20], [21], [22] has proposed different approaches for solving the classic DAG scheduling problem. These approaches typically focus on minimizing the overall execution time, while considering utilization of various resources as a means of determining good candidates. However, in a storage constrained environment

these algorithms may over commit the system with limited calculations on the persistent storage needed to hold files produced. As we have shown, naive accounting of storage without consideration for dependencies can lead to deadlock.

Previous work by the Pegasus project demonstrated the importance of *minimizing* the data footprint of a workflow. The basic approach is to insert cleanup jobs [10] to remove unneeded files after their tasks have been completed, in a manner akin to garbage collection. By itself, this step achieves the maximum footprint described above without computing it in advance. In addition, the DAG itself can be restructured by adding constraints so that concurrent branches are not run simultaneously. This was first done manually [23] and later automatically [24], in both cases by transforming the DAG statically before execution. Our work builds upon these previous approaches by first providing a *static analysis* that quickly computes storage bounds in advance of execution, without committing to a specific order of execution. In addition, the *dynamic algorithm* is able to maintain a given storage limit while still adapting to the concurrency that is encountered at runtime.

It is important to note that we assume a worker can be limited to a specified size. This can be done in several ways. One technique is to employ an online resource monitor [15] which actively watches the task sandbox, making sure it stays within its specified resource constraints. If it exceeds these constraints, the process is killed and reported to the master. Another technique is staging data ahead of the job to increase performance and accuracy of accounting [25], [15], [26]. In this work, we demonstrate the use of loop device filesystems to allocate storage resources independently of the files that consume those resources.

Makeflow, upon which this work is built, has many other similar solutions for parallel execution and organizations. Uintah[27] is a parallel design environment, where workflows are constructed of components that create and consume data to other components or into data warehouses. Similar to how files are used in in Makeflow space could be monitored by watching the streams. Other task based systems such as Charm++[28], Legion[29], Swift[6], and Work Queue[17] are task based programming language extensions that provide built-in methods for data movement, task execution, and remote resources. However these approaches are often fluid and with out the static structure of Makeflow, the static analysis this is built on is not possible.

Storage limitations are also addressed in work that focuses on utilizing RAM and temporary storage. Spark[30], [31] and its concept of Resilient Distributed Datasets address both a performance issues as well as the storage issue by holding the data in memory. This process works well for moderately sized intermediates that are created, but requires that enough memory is available to maintain performance. The limitation of this approach only come from limited memory on machines and the rapid growth of data size to be processed. The experiments utilized in this paper could easily be executed within the memory of even modest machines, but as this data scales the experiments will need to rely on persistent storage for check pointing and execution. As the amount of memory grows, this will continue to be a great approach for many workflows.

X. Conclusion and Future Work

We have shown three techniques for managing storage space in workflows: a static algorithm for offline analysis of storage consumption, a dynamic algorithm which enforces runtime limits while avoiding deadlock, and a containment mechanism for individual tasks. As discussed, the dynamic algorithm is an upper bound on consumption and can overestimate in cases where the same file is used in multiple places in the workflow.

This work is a preliminary step to more fully understanding the interplay of storage constraints in a DAG workflow. Future work will seek to make a tighter upper bound on runtime consumption. There is also work to be done in understanding how estimated or inaccurate file sizes affect performance and accuracy of algorithm results. This includes looking at the scale of re-computation on sufficiently different file sizes.

References

[1] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, B. Berriman, J. Good, A. Laity, J. Jacob, and D. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming Journal*, vol. 13, no. 3, 2005.

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

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

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

[5] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. Nieva de la Hidalga, M. P. Balcazar Vargas, S. Sufi, and C. Goble, "The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud," *Nucleic Acids Research*, vol. 41, no. W1, pp. W557–W561, 2013. [Online]. Available: http://nar.oxfordjournals.org/content/41/W1/W557.abstract

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

[7] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[8] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 59–72, Mar. 2007. [Online]. Available: http://doi.acm.org/10.1145/1272998.1273005

[9] 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.

[10] Arun Ramakrishnan, Gurmeet Singh, Henan Zhao, Ewa Deelman, Rizos Sakellariou, Karan Vahi, Kent Blackburn, David Meyers and Michael Samidi, "Scheduling data-intensive workflows onto storage-constrained distributed resources," *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, 2007.

[11] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the panasas parallel file system," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, ser. FAST'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 2:1–2:17. [Online]. Available: http://dl.acm.org/citation.cfm?id=1364813.1364815

[12] "Lustre: A scalable, high-performance file system," Cluster File Systems, Inc., Tech. Rep., Novemeber 2002. [Online]. Available: http://www.cse.buffalo.edu/faculty/tkosar/cse710/papers/lustre-whitepaper.pdf

[13] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 307–320. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298485

[14] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. C. Laity, E. Deelman, C. Kesselman, G. Singh, M. Su, T. A. Prince, and R. Williams, "Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking," *Int. J. Comput. Sci. Eng.*, vol. 4, no. 2, pp. 73–87, Jul. 2009. [Online]. Available: http://dx.doi.org/10.1504/IJCSE.2009.026999

[15] Gideon Juve, Benjamin Tovar, Rafael Ferreira da Silva, Dariusz Krol, Douglas Thain, Ewa Deelman, William Allcock and Miron Livny, "Practical resource monitoring for robust high throughput computing," *Workshop on Monitoring and Analysis for High Performance Computing Systems Plus Applications*, 2015.

[16] N. Hazekamp, J. Sarro, O. Choudhury, S. Gesing, S. Emrich, and D. Thain, "Scaling Up Bioinformatics Workflows with Dynamic Job Expansion," in *IEEE International Conference on e-Science*, 2015.

[17] 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.

[18] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, no. 4, pp. 406–471, Dec. 1999. [Online]. Available: http://doi.acm.org/10.1145/344588.344618

[19] R. Sakellariou and H. Zhao, "A hybrid heuristic for dag scheduling on heterogeneous systems," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, April 2004, pp. 111– .

[20] H. Topcuouglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002. [Online]. Available: http://dx.doi.org/10.1109/71.993206

[21] M. M. Eshaghian and Y.-C. Wu, "Mapping heterogeneous task graphs onto heterogeneous system graphs," in *Heterogeneous Computing Workshop, 1997.(HCW'97) Proceedings., Sixth*. IEEE, 1997, pp. 147–160.

[22] L. He, S. A. Jarvis, D. P. Spooner, and G. R. Nudd, "Dynamic, capability-driven scheduling of dag-based real-time jobs in heterogeneous clusters," 2004.

[23] Gurmeet Singh and Karan Vahi and Arun Ramakrishnan and Gaurang Mehta and Ewa Deelman and Henan Zhao and Rizos Sakellariou and Kent Blackburn and Duncan Brown and Stephen Fairhurst and David Meyers and G. Bruce Berriman, "Optimizing workflow data footprint," 2007.

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

[25] Ann Chervenak, Ewa Deelman, Miron Livny, Mei-Hui Su, Rob Schuler, Shishir Bharathi, Gaurang Mehta and Karan Vahi, "Data placement for scientific applications in distributed environments," *Proceedings of Grid Conference 2007*, 2007.

[26] G. Juve and E. Deelman, "Resource provisioning options for large-scale scientific workflows," *Third International Workshop on Scientific Workflows and Business Workflow Standards in e-Science (SWBES) in conjunction with Fourth IEEE International Conference on e-Science (e-Science 2008)*, 2008.

[27] J. D. de St. Germain, J. McCorquodale, S. G. Parker, and C. R. Johnson, "Uintah: a massively parallel problem solving environment," in *Proceedings the Ninth International Symposium on High-Performance Distributed Computing*, 2000, pp. 33–41.

[28] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[29] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 66:1–66:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389086

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

[31] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

XI. Author Biography

Nicholas Hazekamp received the B.S. in Computer Science and Chemistry from Hope College and is pursuing a Ph.D. in Computing Science at the University of Notre Dame. His research interests include parallel and distributed computing. He enjoys board games and going to the park with his dog.

Nathaniel Kremer-Herman received the B.A. in Computer Science at Hanover College in Indiana. He is pursuing a Ph.D. in Computer Science & Engineering at the University of Notre Dame. His research interests include scientific applications of distributed computing systems, philosophy of science, and computer-based education.

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 Post-doctoral 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.

Haiyan Meng received her Ph.D. in Computer Science from the University of Notre Dame in May 2017 and is currently a Site Reliability Engineer at Google Inc.. Her Ph.D. research focuses on improving the reproducibility of scientific applications with execution environment specifications, and proposes two preservation approaches and prototypes for the purposes of both result verification and research extension.

Olivia Choudhury received the B.Tech degree in Computer Science and Engineering from West Bengal University of Technology, India and Ph.D. in Computer Science and Engineering from the University of Notre Dame, IN, USA. She is currently a postdoctoral researcher at IBM T.J. Watson Research Center. Her research interests include genomics, healthcare effectiveness, high performance computing, and predictive modeling.

Scott Emrich received the B.S. in Biology and Computer Science from Loyola College in Maryland and the Ph.D. in Bioinformatics and Computational Biology from Iowa State University. His research interests include computational biology, bioinformatics and parallel computing, including arthropod genome analysis with applications to global health and ecology.

Douglas Thain received the B.S. in Physics from the University of Minnesota and the Ph.D. in Computer Sciences from the University of Wisconsin - Madison. He is currently an Associate Professor of Computer Science and Engineering at the University of Notre Dame, where his research focuses on scientific applications of distributed computing systems.