Fast, Transparent, and High-Fidelity Memoization Cache-Keys for Computational Workflows

Vassilis Vassiliadis
IBM Research Europe - Dublin
vassilis.vassiliadis@ibm.com

Michael A. Johnston
IBM Research Europe - Dublin
michaelj@ie.ibm.com

James L. McDonagh
IBM Research Europe - Daresbury
james.mcdonagh@uk.ibm.com

***Abstract*—Computational workflows are important methods for automating complex data-generation and analysis pipelines. Workflows are composed of sub-graphs that perform specific tasks. Certain sub-graphs may appear in multiple workflows. This implies that the same task, with the same input, may execute multiple times thereby wasting computational resources. Additionally, hybrid cloud environments spanning across on-premises and public cloud deployments are increasingly popular. Identical tasks may not only execute multiple times but also on multiple heterogeneous environments. Memoization, a technique to reduce the number of redundant task executions, can tackle this problem. There is a significant body of work for the memoization of workflows. However, prior studies have only been able to satisfy at most two of the three desired features of workflow memoization: transparency (leveraged automatically), performance, and fidelity (identify desired redundant tasks). This paper first introduces a taxonomy for understanding memoization. Next, we describe a novel algorithm that uses a walk in the workflow graph to transparently generate fast and high-fidelity memoization cache-keys, overcoming limitations in prior approaches. We evaluate our scheme using three industry motivated quantum chemistry workflows and two geo-distributed OpenShift clusters. Our experimental evaluation results show the implemented scheme is fast (O(ms)) and can remove nontrivial sources of false positives and negatives that other methods may generate. Finally, we illustrate the proposed workflow memoization scheme can reduce the duration of our benchmarks by up to 10.55x nearly matching the theoretical maximum speedup of memoization for these workloads (10.69x).**

***Index Terms*—computational workflows, memoization, hybrid cloud, workflow execution**

I. Introduction

Computational workflows are popular research accelerators for a broad range of scientific domains [1], [2]. Such computational graphs contain nodes to represent tasks, and edges to define data dependencies. Tasks range from lightweight applications to resource-intensive simulations with long execution times. Workflows provide a way to encode and automatically deploy complex pipelines of tools and services. These pipelines often comprise initialization, computation and analysis steps that can process large volumes of data [3]–[7].

Workflow developers tend to reuse the definitions of frequent execution steps and combine them to produce new workflows. This has led to the emergence of workflow graphs for common tasks within a scientific field [8]–[11]. Using common blocks in workflows assists in building and maintaining workflows. However, it may increase the computational footprint of workflows. Intuitively, executing a workflow implies the execution of all its tasks. In turn, tasks belonging to frequently used sub-graphs may end up processing the same inputs and generating the same outputs but in multiple workflows. Additionally the use of both on-premises and cloud-based clusters within organizations for computational workflow execution is becoming more prevalent due to the ease of their creation and popular Kubernetes [12] frameworks like Argo [13] and Kubeflow [14]. However, identical tasks may execute multiple times on multiple, heterogeneous environments.

The dynamic programming technique of memoization [15] uses a cache of task executions to eliminate redundant executions of equivalent tasks. We investigate workflow memoization and identify three desirable characteristics of such systems: a) *high-fidelity*: memoization must identify as many redundant tasks (true positives) as possible; without incorrectly reusing cached results (false positives) or missing optimization opportunities (false negatives), b) *fast*: memoizing a workflow node must be faster than executing the workflow node, and c) *transparent*: memoization should be leveraged automatically, not requiring workflow developers/users to modify workflows, or adopt specific languages or frameworks. Prior works have demonstrated varying levels of success in meeting these criteria. Some approaches sacrifice transparency to be fast with high fidelity [13], [16]–[22]. Others are fast and transparent, but have fidelity issues with workflow nodes that consume file directories or digital artifacts (e.g. S3 buckets) [14], [23] resulting in false negatives or worse false positives.

The research contributions of this paper are:

1) A taxonomy that enables the comparative evaluation of different computational workflow memoization methods. The taxonomy introduces the concepts of black/white-box scope, memoization intent, task equivalence criteria, surrogates and vulnerabilities (e.g. false-negative/positive).
2) A novel cache-key generation method for computational workflow memoization based on traversing the workflow graph. It is fully transparent and we show it provides higher fidelity than current comparable state of the art methods [14], [23], particularly for nodes that consume collection of files, while meeting performance goals.
3) A performance evaluation of our method, including workflow traces [24], on three industry motivated scientific workflows that share a common sub-graph. Our walk-in-the-graph can generate cache-keys in $\mathcal{O}(ms)$ regardless of the amount of data that the workflow tasks consume.

Section II introduces our memoization taxonomy for computational workflows and Section III discusses prior work. In Section IV we introduce our methodology and compare it with prior work using our taxonomy. Section V discusses the implementation of our methodology. Section VI presents our benchmarks and input dataset. Next, we evaluate the performance of our methodology in Section VII. Finally, we present our conclusions in Section VIII.

II. Memoization taxonomy for computational workflows

Computational workflow memoization is nuanced and here we introduce a simple taxonomy to aid comparative evaluation of workflow memoization methods. In general, memoization aims to eliminate redundant executions of a function f on some inputs x. The core of any memoization method is how it determines equivalence between a to-be-executed $f'(x')$ (the memoization candidate) and a set of already executed $f(x)$. In the case of computational workflows, f is the application of the task associated with a workflow node to its inputs.

Three characteristics of $f(x)$ are suitable for equivalence: its interface (signature and description of the computation), input/output values, and internals (implementation & execution details). Memoization methods do not execute the memoized task so they must base equivalence on inputs, interface and implementation. These reduce to two possible scopes: *black-box*, where only the interface and inputs are used; or *white-box*, which consider information on the internals. Thus 'scope' is the highest rank in our taxonomy and consists of two taxa (groups): black-box and white-box (Figure 1).

Adopting a black-box scope can be desirable and not just the result of trading-off exactness for non-functional characteristics like performance. It enables a separation-of-concerns where a different quality-of-service method could ensure correctness across implementations. It can also allow wider matching of tasks across multiple differing implementations which ultimately are expected to provide the same result.

A memoization method, in addition to its *scope*, also has an *intent*, how it defines equivalence within this *scope*, and *criteria*, how it determines this equivalence. Hence intent and criteria are the second and third ranks in our taxonomy. A method's intent is either implicitly defined by its criteria or explicitly stated. However, in practice examples exist were the intent must be inferred based on the other characteristics, starting from the scope, as criteria are too lax. An example intent is to want to ensure bit-wise identity between the outputs of the tasks, which we term here an *exact* intent.

Evaluating a memoization method involves considering the pros and cons of its *intent* along with, and separate to, its *fidelity*. This is the ability of its criteria to identify redundant tasks w.r.t its intent. An incorrect identification can either be a false-positive, a function the criteria returns as equivalent is not as intended, or false-negative, the criteria find no equivalent function where one that matches by intent is present. The potential false-positives/negatives are caused by vulnerabilities

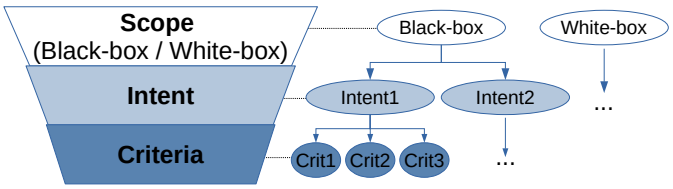


Fig. 1. We use the taxonomic ranks *scope*, *intent*, and *criteria* to classify workflow memoization methods. The highest rank, scope, has two taxa (groups), black-box and white-box. The lower ranks can have many taxa. Comparisons between methods should take place in the context of their lowest common ancestor: all methods can be compared on scope, methods with same scope can be compared on intent, methods with same intent can be compared on criteria. Care must be taken with comparisons between taxa that do not fit this constraint e.g. comparing criteria with different intents.

in the criteria. Each vulnerability can be associated with risk-factors that indicate how likely the vulnerability is to be hit.

We use the term *high-fidelity* to identify methods with limited and low-risk vulnerabilities. This is a somewhat qualitative measure as it is dependent on the intent and the risk-factors which are difficult to measure. However as a first approximation, given two memoization methods within same scope and similar intent an examination of the vulnerabilities of each can be used to determine a relative fidelity.

Vulnerabilities arise from using surrogates for input and internal characteristics of a function when determining equivalence. Examples of input surrogates are hash of a file/directory, or UID of a data artifact (e.g. S3 bucket), rather than the contents of inputs themselves. The surrogate of the interface is often the function signature because a description of the expected computation is either not included or not provided in a way amenable to comparison. Although surrogates cause vulnerabilities, they do not all cause the same number, or severity of vulnerabilities. Surrogates may be desirable as a way to realize certain intents. For example to reuse a cached-task's output even if the inputs of the memoization-candidate differ slightly from those of the cached-task.

Within a scope a wide variety of criteria can be used to achieve a desired intent. This leads to a range of methods that can be qualitatively thought of as going from *direct*, which use no surrogates, through a spectrum where the criteria become progressively less strict and/or use more surrogates [25], [26]. Near the *direct* end we place *strong* equivalence criteria that use hashes of inputs as these have very low chance of error [16]–[19], [22], [27].

Up to now we have considered the functional aspects of memoization. The non-functional aspects of performance and transparency are also important and often influence the criteria. Performance refers to the time it takes to search for a match for a memoization candidate. Memoization is useful only if it is faster than executing the candidate. Transparency refers to ease-of-adoption of a memoization method. A completely transparent method requires no interaction from user or developer, thus eliminating all barriers to leveraging it.

In summary, a memoization method is classified by its scope, intent and criteria (Figure 1). Understanding this clas-

sification is key to understanding the applicability (when it is intended to be used) of the method and its fidelity (how well it performs). For example, there is rich literature on examining and ensuring correctness between differing implementations of a function [15], [25], [26], [28]–[30] and white-box memoization methods [16]–[19], [21], [22] may use these techniques. However, the objective of a black-box method is to identify equivalence at the interface level. Therefore the potential of finding a match with a cached function that has different output, while important to consider, is intended; it is not a vulnerability.

III. Related Work

Improving workflow time-to-solution and reducing the cost of execution are two research areas in the domain of workflow scheduling. For example, [31] introduces scheduling policies that optimize multiple objects such as workflow duration, cost of executing workflows, energy consumption, and reliability. [32] optimizes the execution of workflows by improving the data-locality of workflow nodes. However, memoization is complimentary to such techniques. It does not seek to find the optimal way to run a task but rather to skip its execution.

Some methods assume developers use certain libraries and runtime systems in their programs. [18] augments the workflow runtime [33] in which tasks communicate over a message-based interface with memoization functionality. [17] describes the Parsl python library and its memoization system for python functions. Flyte [16] also supports memoization for python methods. Contrary to Parsl, Flyte can also perform memoization across the boundaries of a single cluster. In [19], [20], on-premises servers generate data and a cost model decides whether future tasks can benefit by reusing the generated data. In that case, the system caches the data in the cloud. In [19], [21], [22] tasks communicate via message-based schemas. The orchestrator uses these input/output messages to identify tasks that have been executed before and their associated cached output messages. The orchestrator memoizes a task by replaying cached output messages instead of executing it.

Other prior work does not impose restrictions to the programming languages and software libraries that workflow developers use in their tasks. Argo [13] is a Kubernetes-native workflow orchestrator with memoization functionality. However, the onus is on the workflow developer/user (non-trasparent), to configure workflow memoization by annotating jobs with memoization metadata. This is an error-prone and tedious process that can result in sub-optimal results. Also, [13] is limited to reusing cached jobs within the confines of a single Kubernetes namespace on a single cluster. The Seven Bridges platform [23] is a commercial workflow orchestrator with support for memoization at the granularity of processes. It can perform memoization, without the need for the workflow developer to manually annotate workflows. However, [23] does not memoize jobs that consume directories or reuse remotely cached node outputs. Kubeflow [14] is a popular framework to deploy pipelines on Kubernetes. Currently, it supports using Argo [13] as its execution backend, among other choices. It also contains its own, and more advanced, memoization system. This system builds cache-keys of nodes out of the base image, command-line, and code a node including any additional customizations (e.g. resource request). Guixwl is a workflow management language extension for the GNU Guix [27] package manager whose primary focus is reproducibility. It encodes the hashes of the task logic (e.g. script, container, etc) as well as resources, and all referenced inputs in the cache-key of the task. Moreover, Guixwl stores memoization metadata under a regular directory where paths under it are the keys of cached tasks. This lack of abstracted storage limits the scaling of its memoization system.

IV. A fast and high-fidelity black-box memoization method

Our memoization system targets Directed Acyclic Graph (DAG) workflows where each task (node in DAG) corresponds to the execution of a program with some command-line options on some inputs. Inputs can be files, directories or any serializable object. The requirements we seek to satisfy are:

1) **Black-box scope**: we wish to match tasks based on their inputs and interfaces only. We do not consider program source code, execution details, and platform characteristics of the task. This enables memoization across heterogeneous computing platforms and between different resource requests.
2) **Transparency**: Workflow developers and users should be able to automatically leverage workflow memoization simply by switching on memoization. This also removes potential sources of errors.
3) **Performance**: negligible overhead for memoization. Intuitively, the higher the overhead the worse the performance benefits of memoization.
4) **Fidelity**: We wish to limit vulnerabilities with respect to state-of-the-art methods with similar requirements as 1-3.

The *intent* of our scheme is for a program to be memoized if it is executed with the same program options, not including any related to resource-requests, and the same inputs as a previous execution. We differentiate between program and executable (a compiled instance of the program). We expect that the same version of a program should match on different platforms. We use the signature of a task as a surrogate for its interface. We construct the signature from its *inputs* (edges), command-line, and container image. For the *inputs* of a node the criteria are:

1) Any inputs which are not outputs of other nodes are bitwise identical.
2) For inputs coming from producer tasks, those tasks have identical interfaces (as per the *interface* criteria).

Criterion 1 uses hashes for the inputs of root nodes for and hence is *strong*. This gives an "anchor" for the application of Criterion 2. Criterion 2 is our walk-in-the-graph method. It combines the label given to an input by its producer with the interface of the producer. Criterion 2 is a recursive surrogate and criterion 1 terminates the recursion. These criteria enforce that the chains of tasks leading to the cached and memoization candidate nodes have identical interfaces. Notably, our

criteria are suitable even for non-deterministic nodes or nodes that consume outputs of non-deterministic nodes. Concretely, neither the contents, nor the hashes, of node outputs are part of any node's signature.

A. Limitations

Our equivalence criteria for input surrogates are stricter when compared to other methods using input surrogates (see next section). This strictness introduces a vulnerability that could lead to false negatives. Tasks that are identical in terms of interface and inputs will not be equivalent if a different chain of tasks produced the inputs. We have not observed this scenario in our experiments. However we identify two scenarios where it could be possible and believe it is an interesting case to consider for future refinement. The scenarios are: (A) Where the chain of nodes in one workflow is a sub-graph of the chain of nodes in the other i.e. one workflow has nodes that feed into the root of the other; (B) Where the chain of nodes are different but when each set is viewed as single block with a black-box scope, they evaluate as identical.

In our scheme if a node is memoized then its producers are also memoized back to the root-nodes which use strong equivalence criteria for inputs. Under a black-box scope, a node whose producers are all memoized, and that has equivalent interface, will be correctly memoized. The potential vulnerability for false-positives then lies in how interface equivalence is determined. This scenario requires a node with the same inputs (number, type, and names), command-line, and container image, as a node in the cache but intending to perform a different task. We consider this situation extremely unlikely to occur. Note here *different* means the aim is to compute a different output/result, as opposed to seeking to compute the same result but using e.g. a different random seed. This later case we do not consider to be a different task.

Every black-box & transparent method is sub-optimal for nodes with implicit inputs. This includes those using direct, strong, and our equivalence criteria. Such nodes do not expose their inputs in their interface (e.g. nodes may download hard-coded URLs). This edge case is technically a vulnerability. Importantly, it is only possible for definitions of workflows that are inherently incompatible with any black-box scope.

With our black-box scope we assume that changes to the resource request of a node (e.g. CPU architecture, resource request, etc) do not fundamentally alter its output, or that a separate sub-system prevents this possibility. However, we recognise that the first assumption may be invalid or that there is no such sub-system. To this end, we provide users the option to include resource information in the memoization cache-key (thus switching our scope to white-box).

B. Comparison with related work

We note that very few implementations explicitly describe the intent of their memoization method. Where this is relevant for comparing their intent to ours (see above) we infer their intent from their scope and criteria. In particular where a method's criteria use surrogates for inputs we assume the intent is to find identical matches. The vulnerabilities which these surrogate open cannot be intentional. They are incompatible with the intent of finding identical matches.

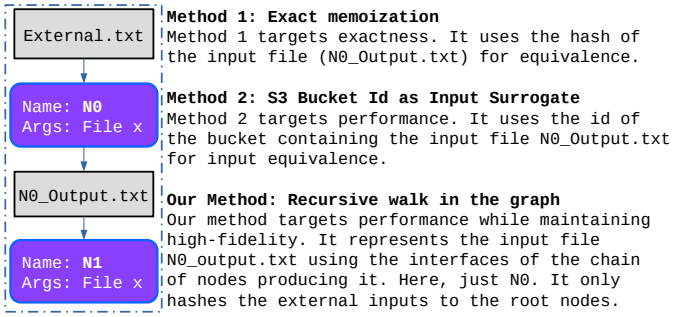


Fig. 2. Diagram showing 3 different memoization methods to evaluate task equivalence in a simple two-node workflow. The memoization candidate is *N1*. It has a single input, the file that *N0* generates (N0_Output.txt).

Some methods use strong equivalence criteria for the inputs of workflow nodes [16]–[19], [21], [22], [27]. This introduces overheads that can lead to performance or scalability issues as inputs grow. Our method avoids these issues by its use of a walk-in-the-graph as a surrogate (Section VII-A). We note that these methods may have an intent e.g. exact, that prefers to use strong equivalence criteria rather than optimize non-functional properties like performance.

Other memoization systems [25], [26], often referred to as "approximate memoization", use lax surrogates in their criteria. This allows for example, their equivalence criteria to match two nodes which have different interfaces. However this does not satisfy our requirements and intent.

Some implementations [16]–[19], [21], [22] are non-transparent and hence introduce extra requirements. For example, they can require access to the source code of the programs that workflow nodes execute, or metadata about the purpose of a program's execution that a developer, or a user, needs to provide. This lack of transparency reduces their ease of adoption by workflow developers and users alike. In comparison our method strives for maximum transparency.

Of the methods discussed in Section III there are two, implemented in Kubeflow [14] and Seven-Bridges Platform [23], which are transparent and provide high-performance via use of surrogates on inputs, matching our requirements. These methods include function execution details, specifically resource requests, in their equivalence criteria and hence have a white-box scope. However if we narrow our focus to situations where resource-requests do not change, we can consider these methods under a black-box scope. Although they do not explicitly specify their intent we believe they closely match ours given the requirements they seek to satisfy and the criteria they use. Thus we consider their criteria as existing alternatives to our proposed criteria.

The memoization system in Kubeflow [14] uses "artifacts" (e.g. S3 Bucket ID) as surrogates of inputs/outputs of tasks, thus providing high-performance even when inputs are very large. This introduces a variety of vulnerabilities as it assumes

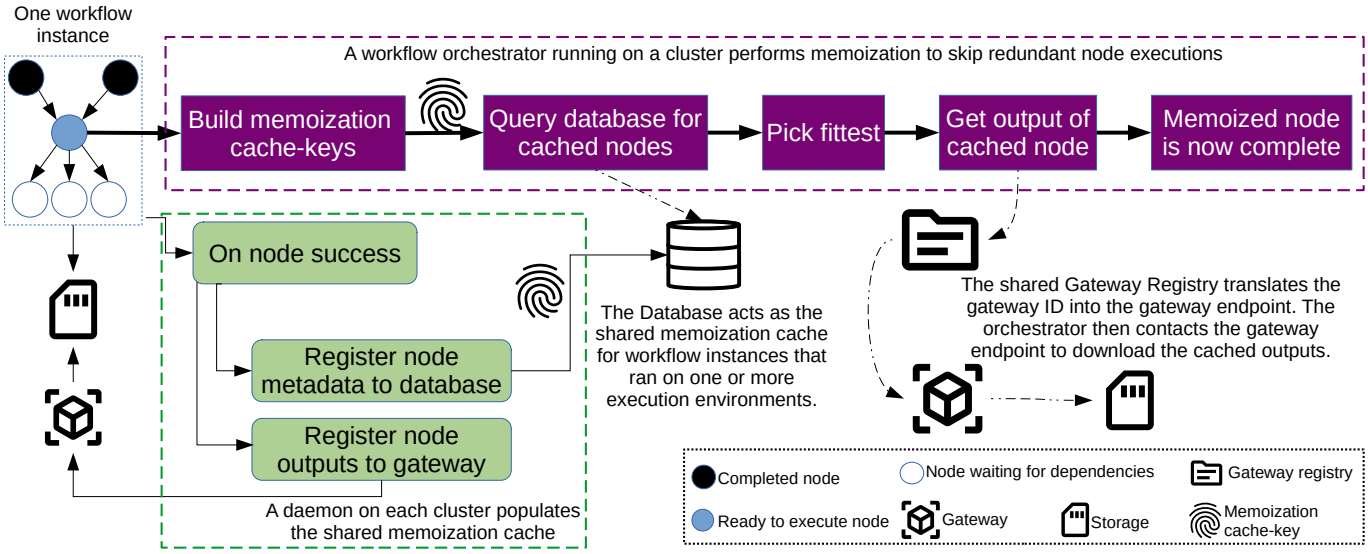


Fig. 3. Diagram of a memoization system that spans multiple and potentially heterogeneous clusters. One daemon per cluster, listens for scheduling events. It then populates the shared memoization cache each time it receives a *"workflow node success"* event. A workflow orchestrator uses the memoization cache to skip redundant node executions and memoize them instead. There can be multiple workflow orchestrators and daemons active on the same at the same time.

that artifacts are immutable between executions of workflows, and that two identical nodes in two workflow instances store their outputs under two artifacts with identical ids. The first assumption causes false positives, and the second one false negatives. The memoization system in Seven-Bridges Platform [23] does not memoize jobs that consume directories or support remote memoization caches. Both limitations cause false negatives.

Finally, most methods discussed use execution specific information, including execution platform architecture, e.g. via hashing the container images, or resource request in their criteria, and hence are classified under a white-box scope. As a result, unlike our method, most cannot perform memoization across heterogeneous platforms without changing their criteria to fall under a black-box scope.

C. Scalability of our memoization system

For simplicity, we evaluate our approach using MongoDB as the database backend and a simple Flask application to serve cached node outputs. However, our memoization cache-keys are compatible with any key-value storage. Indeed, deployments containing industry-standard key-value storage solutions (such as Redis [34]) scale well because we rely on simple key queries. Intuitively, key-value storage solutions are optimized for such operations. Moreover, our approach is inherently more scalable than memoization methods using strong equivalence criteria [16]–[19], [21], [22], [27]. We only hash the external inputs to workflows which are typically smaller than the node outputs. Our method strains the filesystem less than those using strong equivalence criteria and incurs much smaller overhead (Section VII-A). Moreover, our approach enables memoization between heterogeneous computing platforms thereby further improving the scalability of our memoization system (Section VII-D).

Finally, our implementation (Section V-A) includes a step to select a matching cached node that has not been evicted from the cache. This step enables plugging in cache eviction policies [35] to our system in the future. These could asynchronously evict cached node outputs and memoization metadata without risk of adversely affecting the memoization mechanism.

V. Design and Implementation Details

A. High level architecture

Figure 3 shows multiple workflow instances, running on multiple clusters using a shared memoization cache and a shared gateway registry. The memoization cache is a distributed database that holds memoization metadata for workflow nodes that executed in the past. The gateway registry maintains a record of known gateways and can translate a gateway id to a gateway endpoint. Each cluster contains a memoization daemon and a gateway. The daemon populates the memoization cache and the gateway serves the outputs of cached workflow nodes that are stored on the cluster.

To populate the memoization cache, each memoization daemon consumes scheduling events that workflow orchestrators on the same cluster emit. When a workflow node completes, the daemon generates the cache-key of the node. Then, it registers the node's cache-key and locations of its outputs to the database and the local gateway.

In the same figure, we can see the steps that the workflow orchestrator takes to memoize a ready-to-execute node during workflow execution. In particular, the workflow orchestrator builds the memoization cache-key of a ready-to-execute node.

Algorithm 1 Walking the graph to generate the memoization cache-key of a workflow node.

```
1:  function NodeCacheKey(n: WorkflowNode) ▷ This is the Entry point; it returns the cache-key of a node. Walk may visit the producers of the node in HashReference()
2:      exec ← ResolveExecutable(n.command)
3:      ref(r) ← HashReference(r) for r ∈ n.references
4:      args ← ReplaceReferences(n.command, ref)
5:      args ← ResolveVariables(args)
6:      if global.useTagsForRequests == false then
7:          conf ← ResolveResourceRequests(n.fields)
8:      else
9:          conf ← n.fields
10:     end if
11:     obj ← (exec, args, ref, n.backend, n.envVars, conf)
12:     if n.useCustomHashKeyFunction then
13:         return n.HashKey(obj)
14:     end if
15:     return SerializeThenHash(obj, "md5")
16: end function
17: function HashReference(dr: DataReference) ▷ Returns hash of DataReference, may walk the graph via NodeCacheKey()
18:     if dr.producedByNode then
19:         dr.hash ← NodeCacheKey(dr.producer)
20:     else
21:         dr.hash ← HashFiles(dr.referencedFiles)
22:     end if
23:     return dr.hash+":"+dr.optionalPath+":"+dr.refType
24: end function
```

Then, it queries the database for nodes with matching cache-keys and picks the fittest one i.e. most recent node that completed successfully and whose cached outputs have not been evicted. Next, it retrieves the cached outputs, and marks the memoized node as complete. At this point the memoization phase is done, the workflow orchestrator then tags nodes with no pending dependencies as ready-to-execute. This process memoizes a single node and the orchestrator can greedily repeat it to memoize as many nodes as possible.

B. Generating the memoization cache-key of a node

Our execution model resembles Linux® processes. Nodes in the DAG have a command-line and store their outputs under their working directory. Nodes consume paths, streams, and contents of files that other nodes produce, or are external inputs. The edges in the DAG indicate task dependencies. We use the term DataReference to refer to an edge.

Algorithm 1 implements the walk on the workflow graph (graph traversal). The walk begins at the node for which we compute the cache-key (Line 1). It follows edges to producers to generate the hash of the associated DataReference (method $HashReference(r)$ in Line 3). Upon visiting an edge it hashes files to terminate the walk if the referenced files are external inputs. Otherwise, it follows the edge to the producer node to generate the producer's cache-key (Lines 18-22). $NodeCacheKey(node)$ then uses these edge hashes (Lines 3-4) to rewrite the command-line of $node$. Additionally, it optionally does not expand references to resource requests in the arguments e.g. CPU cores, memory capacity, etc (Lines 6-10). This enables reusing cached node executions that were executed with different resources or even on different CPU architectures.

```
executable:
  path: /gamess/rungms
arguments: >
{{<cache-key-of-producer>\<name>:<type>}}
01 {{self.resourceRequest.cores}}
inputEdges:
- {{<cache-key-of-producer>\<name>:<type>}}
backend:
  image: gamess:1.0.0
environmentContext:
  envVar: value  # ...omitted...
```

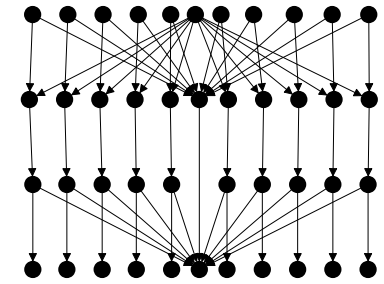
Fig. 4. Memoization metadata for executing GAMESS on a molecule using a number of CPU cores on Kubernetes. A producer node generates the input molecule file. The node cache-key is the hash of the dictionary.

The last step in $NodeCacheKey(node)$ packs the node's transformed arguments, edge hashes, backend metadata and executable metadata into a dictionary (Line 11). The memoization cache-key is the hash of this dictionary (Lines 12-15). If a node uses a multi-architecture manifest image we can generate one cache-key containing the hash of each entry in the manifest. In our experiments, we use the hash of the multi-architecture manifest. This enables our method to find matching cached nodes even if they executed on a different CPU architecture. Figure 4 presents the resulting memoization dictionary for an execution of GAMESS [36] on Kubernetes.

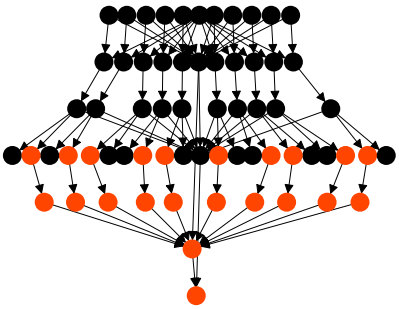
Finally, we note that it is easy to modify our algorithm for different memoization intents and criteria. Take for example a memoization intent that wants to ensure bit-wise exactness for file inputs of tasks, on the condition that the files are cheap to hash. The memoization criteria of a such a system can be:

1) Any inputs which are not outputs of other nodes are bit-wise identical.
2) Any inputs which are outputs of other nodes and are cheap to hash are also bit-wise identical.
3) Remaining inputs are coming from producer nodes tasks and are costly to hash. Those tasks have identical interfaces (as per the *interface* criteria).

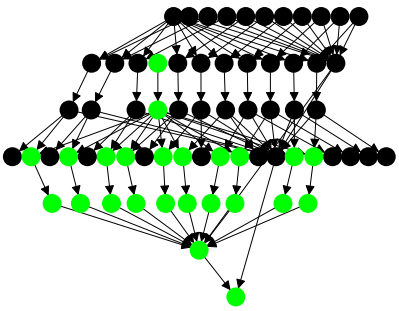
This memoization system need only modify $HashReference(dr)$ to hash files generated by nodes if their size is smaller than some threshold. Criterion 3 would then become the recursive walk in the graph, and criteria 1 and 2 would behave as "anchors" for the application of criterion 3. See Section IV for more details on defining the *interface* for criteria.



(a) W_{base} workflow with 43 BN_{base} nodes in black.



(b) W^{α} workflow with 43 BN_{base} in black and 22 BN^{α} nodes in lighter color.



(c) W^{β} workflow with 43 BN_{base} in black and 24 BN^{β} nodes in lighter color

Fig. 5. Workflows processing 10 molecules with color coded blocks of nodes. There are only a handful of batch-size invariant nodes: those which aggregate multiple upstream nodes.

VI. Experimental setup

A. Benchmark workflows and input dataset

In this paper, we evaluate our memoization system with three industry motivated quantum chemistry Workflows: W_{base}, W^{α}, and W^{β}. We used these workflows in the context of Project Photoresist [37] to accelerate the discovery and synthesis of more sustainable Photo-Acid Generators (PAGs) for chemically amplified photoresists [38].

Our workflows consist of three functional Blocks of Nodes (BN): BN_{base}, BN^{α}, and BN^{β} shown in Figure 5. W_{base} contains only one computationally intensive step, BN_{base}. In both W^{α} and W^{β} we reuse BN_{base} from W_{base} and introduce another computationally intensive block of nodes: BN^{α} for W^{α} and BN^{β} for W^{β}. The fact that complex workflows consist of common functional blocks [8]–[11] increases the benefits of memoization. In our experiments, instances of W^{α} and W^{β} can reuse cached nodes (BN_{base}) of W_{base} calculations for the same molecule.

W_{base} estimates the energetic gap between the Highest Occupied Molecular Orbital and Lowest Unoccupied Molecular Orbital (HOMO-LUMO gap). This energetic gap is related to conductivity in semiconductors, optical absorption and other properties. This workflow consists of just the BN_{base} block of nodes which: a) parse a standard string representation of a molecule, known as the Simplified Molecular Input Line Entry System (SMILES) [39], to generate 3D atomic coordinates using RDKit [40]; b) produce GAMESS US [36] inputs; c) perform geometry optimization via GAMESS US; and finally, d) extract and aggregate energies. This workflow represents a common task in the domain. Other workflows use it as a base layer. It involves the iterative diagonalization of large multidimensional matrices for quantum chemical geometry optimizations. In this work, we use semi-empirical methods. These methods are considerably cheaper compared to alternative *ab initio* or Density Functional Theory (DFT) methods. In the domain, such processes consume the majority of the computational expense. Finally, we note that BN_{base} forms the basis of all three workflows in the experiments campaign.

W^{α} builds from the W_{base} workflow's step *(c)*. Specifically, it extends the calculation and analysis to estimate the minimum energy required to remove one electron from the molecule. The workflow consists of the BN_{base} and BN^{α} nodes. The BN^{α} nodes add a single molecular energy evaluation step. This step iteratively solves a set of linear equations describing the electronic structure of the system in a particular configuration. Overall, BN^{α} is computationally cheaper than BN_{base}.

W^{β} builds from steps *(c)* and *(d)* of workflow W_{base}. Specifically, W^{β} introduces a prediction of the wavelength at which a molecule has its maximum photonic absorbance (lambda max). This prediction is made by Time Dependent Density Functional Theory (TDDFT). As such, W^{β} consists of BN_{base} and BN^{β} blocks of nodes. BN^{β} nodes evaluate the energy of electronically excited states. BN^{β} is more computationally expensive than BN_{base}.

Finally, we use a dataset of 80 molecules [41] to perform experiments involving the three workflows described above. The molecules represent a selection of sulfonium PAGs retrieved from academic and publicly available patent literature [42]. The range of characteristics of the molecules, in terms of size (number of atoms and number of electrons), chemistry, and their distribution are typical of sulfonium PAGs. Thus we expect the trends we observed for this data-set to generalize to larger PAG data sets.

B. Execution platforms

We use 2 OpenShift deployments, one with IBM POWER8™ and another with Intel® x86-64™ CPUs. The POWER cluster is hosted in a multi-tenant, on-premise, data

TABLE I

W^{α} AND W^{β} INSTANCES REUSE LOCALLY CACHED BN_{base} NODES FROM CACHED W_{base} INSTANCES (AVERAGE OF 8 RUNS). MEMOIZATION, ON AVERAGE, SKIPS 85.56% AND 7.5% OF THE TOTAL EXECUTION TIME FOR W^{α} AND W^{β} RESPECTIVELY. AS SUCH, IT REDUCES THE WORKFLOW DURATION OF W^{α} BY A FACTOR OF 10.55X AND THAT OF W^{β} BY A FACTOR OF 1.04X.

	W^{α}	W^{β}
Workflow nodes	65	67
Memoized nodes	43	43
Total walltime (sec)	2583.1	31559.3
Total re-execution walltime (sec)	373.0	29349.20
Total memoization overhead (sec)	6.21	6.03
Workflow duration with re-execution (sec)	525.19	5930.01
Workflow duration with memoization (sec)	49.76	5684.17
Speedup to workflow duration	10.55x	1.04x
Execution walltime skipped (%)	85.56%	7.5%
Mean memoization ovhd to re-execute (sec)	0.017	0.015
Mean memoization ovhd to memoize (sec)	0.145	0.142

TABLE II

W^{α} AND W^{β} INSTANCES REUSE REMOTELY CACHED BN_{base} NODES FROM CACHED W_{base} INSTANCES (AVERAGE OF 8 RUNS). MEMOIZATION, ON AVERAGE, SKIPS 85.42% AND 6.8% OF THE TOTAL EXECUTION TIME FOR W^{α} AND W^{β} RESPECTIVELY. AS SUCH, IT REDUCES THE WORKFLOW DURATION OF W^{α} BY A FACTOR OF 8.38X AND THAT OF W^{β} BY A FACTOR OF 1.003X.

	W^{α}	W^{β}
Workflow nodes	65	67
Memoized nodes	43	43
Total walltime (sec)	2587.35	32509.55
Total re-execution walltime (sec)	377.25	30299.45
Total memoization overhead (sec)	109.76	123.69
Workflow duration with re-execution (sec)	525.19	5930.01
Workflow duration with memoization (sec)	62.68	5914.57
Speedup to workflow duration	8.38x	1.003x
Execution walltime skipped (%)	85.42%	6.8%
Mean memoization ovhd to re-execute (sec)	0.54	0.55
Mean memoization ovhd to memoize (sec)	2.81	3.12

center in Europe. This subset of the data center contains 7 bare-metal nodes. Each node features 1 TB of DDR3 RAM and two POWER8™ 8335-GCA @2.92GHz processors for a total of 20 physical cores. The cluster sits behind a VPN but has access to the public internet, including the x86-64 cluster. The x86-64 cluster is a multi-tenant OpenShift deployment hosted on a public cloud data center in the USA. It contains 8 nodes, each of which is a Virtual Machine that is equipped with 32 GB of DDR4 RAM and 8 cores, out of the 14 physical cores, of an Intel® Xeon™ CPU E5-2683 v3. The x86-64 cluster does not have access to the VPN that the POWER cluster uses but supports inbound/outbound internet traffic.

VII. PERFORMANCE EVALUATION

Section IV-B presented a comparative evaluation of our memoization method against prior work, including fidelity and vulnerability aspects. Next, we focus on the performance evaluation. In Section VII-A we show our method can generate cache-keys at $\mathcal{O}(ms)$, outperforming memoization methods with strong input equivalence criteria [16]–[19], [21], [22], [27]. Section VII-C evaluates the fidelity of our method in an example scenario with workflows that slightly differ from each other. Finally, in Section VII-D, we demonstrate that our method, for our benchmark-set, may result in speedup of up to $10.55x$ nearly matching the theoretical speedup that memoization has to offer for local memoization ($10.69x$). For cross-cluster & heterogeneous environments) scenarios it delivers speedup up to $8.38x$. The speedup reduction is mainly due to the network latency and network bandwidth between the on-premises cluster in Europe and the public cloud datacenter in the USA.

A. Memoization overhead

First, we define the term "skipped execution time". This is the would-be execution time of redundant tasks that memoization identifies as *skipped execution time*, or in short $Skipped$. We compute $Skipped$ as:

$$Skipped = 1 - \frac{totalReExecution + totalMemoOverhead}{totalExecutionWallTime} \quad (1)$$

Next, we examine the overhead of our memoization algorithm. Table I presents the profiling data for memoizing W^{α} and W^{β} with locally cached instances of W_{base}. Table II shows the profiling data for memoizing W^{α} and W^{β} with remotely cached instances of W_{base}. The overhead of reusing remotely cached nodes (remote memoization) is larger than that of local (Table II). Specifically, we find that in remote memoization the overhead to identify nodes that cannot be memoized increases by just 0.4 seconds (from 0.017 seconds for local). Whereas, the overhead to fully memoize a node increases to 3 seconds (from 0.145 seconds for local). The increase is due to the network latency and bandwidth between the POWER cluster in Europe and the x86-64 cloud instance in the USA public data-center. REST-API calls from the EU cluster to the USA data-center take about 0.4 seconds longer to complete and the network bandwidth is on average a modest 82.7 Mb/s. In both memoization experiments, our method takes on average 0.004 seconds to generate a cache-key and never more than 0.1 seconds. Cache-keys which take the longest to generate, are all for nodes that consume external inputs. Moreover, the mean skipped execution time for nodes is orders of magnitude larger ($2210.1/43 = 51.4$ seconds) than the task re-execution time. As a result, we consider that our approach, even in the case of remote memoization, incurs negligible overhead.

Table III presents the memoization overhead breakdown for workflows that process all 80 molecules (*mega* workflows). Although the total input size of nodes is ∼17 GB our method hashes just ∼600 KB worth of files. Contrary to methods using strong equivalence, our method requires $\mathcal{O}(ms)$ to generate cache-keys for nodes that only consume outputs of other nodes. Often, only root nodes consume external inputs.

We can estimate the minimum overhead of any method using strong equivalence criteria as the time it would take to

TABLE III
BREAKDOWN OF THE MEMOIZATION OVERHEAD FOR THE MEGA WORKFLOWS IN TABLE IV. OUR METHODOLOGY ENDS UP HASHING JUST 562KB OUT OF THE 17.34GB THAT WORKFLOW NODES REFERENCE.

	mega-W^α	mega-W^β
Total walltime (sec)	20935.4	260054.1
Total skipped execution walltime (sec)	20870.1	259985.2
Total memoization overhead (sec)	63.36	50.87
generate cache-keys (sec)	2.03	2.06
query-cache (sec)	2.63	2.64
pick-fittest cached node (sec)	0.23	0.22
get cached node outputs (sec)	58.47	45.95
Mean memoization ovhd to re-execute (sec)	0.019	0.018
Mean memoization ovhd to memoize (sec)	0.162	0.131
total filesize of inputs to nodes	16.57 GB	17.34 GB
total filesize that memoization hashes	547.6 KB	562.29 KB

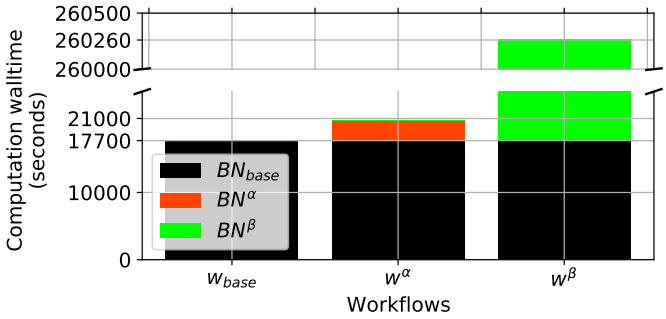


Fig. 6. Computation walltime, in seconds, for the POWER cluster to process the entire dataset. The computation walltime is reported at the granularity of blocks of nodes.

hash the input set of nodes. Such a method would spend 147 seconds just to hash the 17.34 GB worth of inputs to nodes. Our system calculates the cache-keys of the same nodes 71.46x faster. Our implementation took 50.87 seconds to complete all memoization steps for all the workflow nodes (Table III), 90% of that time (45.95 seconds) is spent retrieving cached outputs. A system with strong equivalence criteria, beyond spending 147 seconds to hash files, would also need to hash the definition of nodes, search the cache, and retrieve outputs to completely memoize the workflow nodes in mega-W^β.

B. Baseline experiments: re-executing tasks

We compare our memoization approach against re-execution and release traces of 3 workflows [24] and our input molecule dataset [41]. For our baseline measurements we process all 80 input molecules, in batches of 10, for each workflow on each cluster. All nodes execute their tasks and do not get memoized.

Figure 6 shows the total execution walltime for our 3 workflows and the time each workflow spent executing nodes of the 3 blocks (BN_{base}, BN^α, and BN^β) on the POWER cluster[1]. The BN_{base} nodes are common in all 3 workflows. These are the only ones that can be memoized between 2 instance of different workflows. They also happen to execute about the same time in each of the 3 workflow instances.

[1]The x86-64 cluster features a similar distribution of execution times.

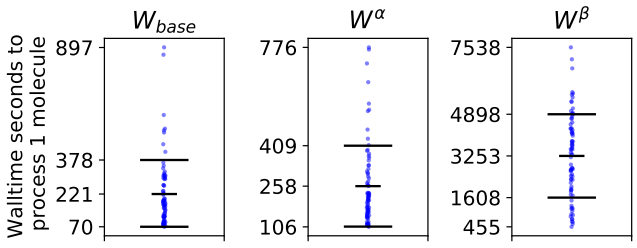


Fig. 7. Walltime seconds required by the workflows to process each of the 80 molecules on the POWER cluster. Horizontal lines indicate $mean + std$, $mean$, and $mean - std$ points. Recall that W^α and W^β contain BN_{base} nodes, which are also present in W_{base} (Figure 6).

TABLE IV
MEGA WORKFLOWS WITH BATCH SIZE 80 REUSE LOCALLY CACHED EXECUTIONS OF SAME WORKFLOWS BUT WITH BATCH SIZE 10. WORKFLOWS ONLY RE-EXECUTE TASKS WHICH ARE BATCH SIZE DEPENDENT, MEMOIZATION SKIPS MORE THAN 99.69% OF THE TOTAL PROCESS WALLTIME OF NODES.

	mega-W^α	mega-W^β
Workflow nodes	485	487
Memoized nodes	411	413
Total walltime (sec)	20935.4	260054.1
Total re-execution walltime (sec)	65.3	68.9
Total skipped execution walltime (sec)	20870.1	259985.2
Total memoization overhead (sec)	63.36	50.87
Workflow duration with re-execution (sec)	928.25	7556.15
Workflow duration with memoization (sec)	28.7	24.76
Speedup to workflow duration	32.34x	305.18x
Execution walltime skipped (%)	99.69%	99.97%

Figure 7 shows the total execution walltime of workflows to process one molecule when they execute each task without checking for memoization opportunities. Each point in W^α and W^β is one point from W_{base} increased by the execution walltime of BN^α and BN^β nodes respectively. The nodes that W^α introduces are less computationally intensive than the BN_{base} ones. However, the nodes that W^β introduces are an order of magnitude more computationally intensive than the nodes in BN_{base} (Section VI-A).

C. Memoization and subtle differences in task chains

We evaluate memoization when the batch-size of an under-execution workflow differs from that of the cached workflows. We show that our cache-key algorithm is robust even in the presence of subtle differences between the nodes under execution and those in the cache. To this end, we populate the memoization cache with workflows that processed 10 molecules each (batch-size = 10) and then execute mega-workflows that process the entire dataset (batch-size = 80).

Table IV shows the profiling metadata for the mega instances of W^α and W^β. We report that both mega instances behave similarly. For example, mega-W^β executes just 74 out of 487 nodes and memoizes the rest. These 74 nodes are batch-size dependent and involve aggregating multiple molecule pipelines. The re-executed nodes execute for just 68.9 seconds and memoization skips the remaining execution walltime of the experiment which is is $259,985.2$ seconds. In other words,

memoization incurs an overhead of just 50.87 seconds to skip 99.97% of the total execution walltime and thus brings the workflow duration to 24.76 seconds from $7,556.15$.

Our memoization system identifies all memoization opportunities and delivers a 305.18x speedup. Moreover, it avoids false positives and negatives even when there are differences in batch sizes between otherwise identical workflows. The resulting system introduces negligible overhead for workloads that are extremely suitable for memoization. This enables users to run expensive to execute workflows once, potentially using multiple heterogeneous platforms. Then, they may run multiple large scale analyses on the outputs and intermediate results of the cached workflows with a negligible overhead.

D. Memoization performance

We can use Formula 1 to calculate the theoretical maximum skipped execution time of memoization for W^{α} and W^{β} workflows that reuse cached W_{base} workflow instances. We just set the memoization overhead to 0 to compute that memoization may skip up to 85.7% of the execution time of W^{α} and 6.9% of W^{β} (Figure 6). After we factor in the dependencies of W^{α} and W^{β} we can also compute the maximum theoretical speedup as well. It is $525.19/49.125 = 10.69$x for W^{α} and $5930.01/5683.58 = 1.04$x for W^{β}.

Both W^{α} and W^{β} add a few nodes on top of W_{base}. In the case of W^{α} these new node (BN^{α}) are computationally cheaper than those of W_{base} (BN_{base}). On the contrary, W^{β} introduces nodes (BN^{β}) with execution time an order of magnitude bigger than the execution time of W_{base} nodes (BN_{base}). In general, memoization can significantly boost the performance of workflows such as W^{α} whose execution time is mostly due to nodes that get memoized. Memoization offers 10.55x speedup for W^{α} in local mode (Table I) and 8.38x in remote mode (Table II). However, in scenarios where re-executed workflow nodes end up executing for longer than those that get memoized, the speedup is not as impressive. Still, it provides a time-to-solution advantage (e.g. W^{β}).

Although remote memoization offers significant speedup it tends to be less performant than local memoization because of the increased latency, and lower bandwidth between the cluster that executes the workflow and the memoization cache. Interestingly, we find that even a mediocre network bandwidth (82.7 Mb/s) is good enough for remote memoization to result in substantial speedup (8.38x). We also highlight, that our memoization architecture (Figure 3) enables multiple, even heterogeneous platforms, to use the same memoization cache. In turn, this enables a workflow to memoize its nodes using a combination of locally and remotely cached nodes. During the "pick fittest" step, the memoization system could prioritize locally cached nodes over remotely cached ones.

Finally, beyond task re-execution time, queuing time and other orchestrator overheads may also play a significant role in the duration of a workflow. Memoized nodes effectively reduce the resource pressure to clusters and thus enable other jobs to execute sooner. Interestingly, workflow memoization has the potential to improve the queuing time of any workload on the same cluster even if it is not part of the under-memoization workflow. This non obvious benefit is especially important in the context of multi-tenant environments.

VIII. Conclusions

In the pursuit of faster time-to-solution the quickest calculation is the one you do not have to run. This work realizes this maxim by demonstrating a memoization system for computational workflows that avoids re-execution of tasks. Prior work involved a trade off between high-fidelity with transparency (programming language restrictions or requirement of memoization hints) and performance. In contrast, we demonstrate that our method avoids the need for this trade off by automatically extracting memoization hints based on the dependencies between nodes.

Our fast cache-key generation algorithm, which takes $\mathcal{O}(ms)$, means memoization can be worthwhile even for tasks that take $\mathcal{O}(seconds)$ to complete, depending on size of output to be retrieved. In the future we plan to model the overhead so that the workflow orchestrator can intelligently choose between memoizing or re-executing a task. We also evaluated memoization between heterogeneous clusters in different continents. This is especially relevant in hybrid or multi-cloud scenarios. We find that even with a limited network bandwidth of 82.7 Mb/s a significant number of tasks benefit from memoization. We note that all the figures discussed in this paper pertain to the worst-case scenario where queuing time is 0. In reality, execution environments are often shared and managed by a batch scheduler. Consequently, the potential impact of memoization can be equally as dramatic to not only workflows, but also any other workload on the same cluster.

We adopt a black-box scope (Section IV) which uses a task's interface (signature and description of the computation) and inputs for determining task equivalence. However, we see cases where fuzzier criteria for the interface could be useful and indeed for a memoization method that acts at a sub-graph level (set of nodes) instead of node level. In the future we plan to investigate cases where more lax memoization could be desirable. For example, to memoize a BN_{base} sub-graph with another quantum chemistry code, which has improved performance, and produces the same numerical analysis. This requires intelligent criteria (Section II) as well as quality control sub-system(s).

Acknowledgment

This work was supported by the STFC Hartree Centre's *Innovation: Return on Research programme*, funded by the Department for Business, Energy & Industrial Strategy.

References

[1] I. J. Taylor, E. Deelman, D. B. Gannon, M. Shields *et al.*, *Workflows for e-Science: scientific workflows for grids*. Springer, 2007, vol. 1.

[2] R. F. da Silva, R. Filgueira, I. Pietri, M. Jiang, R. Sakellariou, and E. Deelman, "A characterization of workflow management systems for extreme-scale applications," *Future Generation Computer Systems*, vol. 75, pp. 228–238, 2017.

[3] S. Kelling, W. M. Hochachka, D. Fink, M. Riedewald, R. Caruana, G. Ballard, and G. Hooker, "Data-intensive science: a new paradigm for biodiversity studies," *BioScience*, vol. 59, no. 7, pp. 613–620, 2009.

[4] C. Draxl and M. Scheffler, "Nomad: The fair concept for big data-driven materials science," *Mrs Bulletin*, vol. 43, no. 9, pp. 676–682, 2018.

[5] D. G. Smith, D. Altarawy, L. A. Burns, M. Welborn, L. N. Naden, L. Ward, S. Ellis, B. P. Pritchard, and T. D. Crawford, "The molssi qcarchive project: An open-source platform to compute, organize, and share quantum chemistry data," *Wiley Interdisciplinary Reviews: Computational Molecular Science*, vol. 11, p. e1491, 2020.

[6] T. A. Barnes, E. Marin-Rimoldi, S. Ellis, and T. D. Crawford, "The molssi driver interface project: A framework for standardized, on-the-fly interoperability between computational molecular sciences codes," *Computer Physics Communications*, vol. 261, p. 107688, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465520303386

[7] J. L. McDonagh, W. C. Swope, R. L. Anderson, M. A. Johnston, and D. J. Bray, "What can digitisation do for formulated product innovation and development?" *Polymer International*, vol. 70, no. 3, pp. 248–255, 2021.

[8] D. Garijo, P. Alper, K. Belhajjame, O. Corcho, Y. Gil, and C. Goble, "Common motifs in scientific workflows: An empirical analysis," *Future Generation Computer Systems*, vol. 36, pp. 338–351, 2014.

[9] D. De Roure, C. Goble, and R. Stevens, "The design and realisation of the experimentmy virtual research environment for social sharing of workflows," *Future Generation Computer Systems*, vol. 25, no. 5, pp. 561–567, 2009.

[10] P. Mates, E. Santos, J. Freire, and C. T. Silva, "Crowdlabs: Social analysis and visualization for the sciences," in *International conference on scientific and statistical database management*. Springer, 2011, pp. 555–564.

[11] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher *et al.*, "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.

[12] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade," *Queue*, vol. 14, no. 1, pp. 70–93, 2016.

[13] C. N. C. Foundation, "Argo workflows: Container-native workflow engine," 2021, accessed 12 March 2021. [Online]. Available: https://argoproj.github.io/projects/argo/

[14] "Kubeflow: The machine learning tookit for kubernetes," 2021, accessed 12 March 2021. [Online]. Available: https://www.kubeflow.org/

[15] D. Michie, ""memo" functions and machine learning," *Nature*, vol. 218, no. 5136, pp. 19–22, 1968.

[16] flyte, "Flyte: The workflow automation platform for complex, mission-critical data and ml processes at scale," 2021, accessed 12 March 2021. [Online]. Available: https://flyte.org/

[17] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster *et al.*, "Parsl: Pervasive parallel programming in python," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, 2019, pp. 25–36.

[18] R. M. Richard, C. Bertoni, J. S. Boschen, K. Keipert, B. Pritchard, E. F. Valeev, R. J. Harrison, W. A. De Jong, and T. L. Windus, "Developing a computational chemistry framework for the exascale era," *Computing in Science & Engineering*, vol. 21, no. 2, pp. 48–58, 2018.

[19] G. Heidsieck, D. De Oliveira, E. Pacitti, C. Pradal, F. Tardieu, and P. Valduriez, "Adaptive caching for data-intensive scientific workflows in the cloud," in *International Conference on Database and Expert Systems Applications*. Springer, 2019, pp. 452–466.

[20] C. Pradal, S. Dufour-Kowalski, F. Boudon, C. Fournier, and C. Godin, "Openalea: a visual programming and component-based software platform for plant modelling," *Functional plant biology*, vol. 35, no. 10, pp. 751–760, 2008.

[21] L. Bavoil, S. P. Callahan, P. J. Crossno, J. Freire, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "Vistrails: Enabling interactive multiple-view visualizations," in *VIS 05. IEEE Visualization, 2005.* IEEE, 2005, pp. 135–142.

[22] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, "Kepler: an extensible system for design and execution of scientific workflows," in *Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004.* IEEE, 2004, pp. 423–424.

[23] S. Bridges, "Seven bridges platform: Actionable informatics for biomedical research," 2021, accessed 12 March 2021. [Online]. Available: https://www.sevenbridges.com/

[24] V. Vassiliadis, A. M. Johnston, and L. J. McDonagh, "Traces of scientific workflow instances with and without memoization," Jul. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6619609

[25] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 922–927, 2005.

[26] G. Tziantzioulis, N. Hardavellas, and S. Campanoni, "Temporal approximate function memoization," *IEEE Micro*, vol. 38, no. 4, pp. 60–70, 2018.

[27] L. Courtès, "Functional package management with guix," *CoRR*, vol. abs/1305.4584, 2013. [Online]. Available: http://arxiv.org/abs/1305.4584

[28] U. A. Acar, G. E. Blelloch, and R. Harper, "Selective memoization," *ACM SIGPLAN Notices*, vol. 38, no. 1, pp. 14–25, 2003.

[29] J. Tuck, W. Ahn, L. Ceze, and J. Torrellas, "Softsig: software-exposed hardware signatures for code analysis and optimization," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2, pp. 145–156, 2008.

[30] K. Parasyris, G. Georgakoudis, H. Menon, J. Diffenderfer, I. Laguna, D. Osei-Kuffuor, and M. Schordan, "Hpac: evaluating approximate computing techniques on hpc openmp applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[31] M. R. Hoseinyfarahabady, H. R. Samani, L. M. Leslie, Y. C. Lee, and A. Y. Zomaya, "Handling uncertainty: Pareto-efficient bot scheduling on hybrid clouds," in *2013 42nd International Conference on Parallel Processing*. IEEE, 2013, pp. 419–428.

[32] I. Casas, J. Taheri, R. Ranjan, L. Wang, and A. Y. Zomaya, "A balanced scheduler with data reuse and replication for scientific workflows in cloud computing systems," *Future Generation Computer Systems*, vol. 74, pp. 168–178, 2017.

[33] H. van Dam, E. Apra, R. Bair, J. Boschen, E. Bylaska, W. De Jong, T. Dunning, N. Govind, R. Harrison, K. Keipert *et al.*, "Nwchemex–computational chemistry for the exascale era," *Bulletin of the American Physical Society*, vol. 65, 2020.

[34] Redis, "Redis: Remote dictionary server," 2009, accessed 21 December 2021. [Online]. Available: https://redis.com/

[35] M. Bilal and S.-G. Kang, "A cache management scheme for efficient content eviction and replication in cache networks," *IEEE Access*, vol. 5, pp. 1692–1701, 2017.

[36] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su *et al.*, "General atomic and molecular electronic structure system," *Journal of computational chemistry*, vol. 14, no. 11, pp. 1347–1363, 1993.

[37] I. Research, "Project photoresist: Finding and synthesizing a new molecule in less than a year," 2022, accessed 27 May 2022. [Online]. Available: https://research.ibm.com/interactive/photoresist/

[38] E. O. Pyzer-Knapp, J. W. Pitera, P. W. Staar, S. Takeda, T. Laino, D. P. Sanders, J. Sexton, J. R. Smith, and A. Curioni, "Accelerating materials discovery using artificial intelligence, high performance computing and robotics," *npj Computational Materials*, vol. 8, no. 1, pp. 1–9, 2022.

[39] D. Weininger, "Smiles, a chemical language and information system. 1. introduction to methodology and encoding rules," *Journal of chemical information and computer sciences*, vol. 28, no. 1, pp. 31–36, 1988.

[40] G. Landrum, "Rdkit: Open-source cheminformatics," 2006, accessed 12 March 2021. [Online]. Available: http://www.rdkit.org

[41] V. Vassiliadis, A. M. Johnston, and L. J. McDonagh, "Molecule dataset used in workflow memoization experiments," Jul. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6620058

[42] I. B. Machines, "Ibm circa," https://circa.res.ibm.com/, 2021.