Provenance-based Workflow Diagnostics Using Program Specification

Yuta Nakamura, Tanu Malik, Iyad Kanj
School of Computing, DePaul University
Chicago, IL USA
ynakamu1, tanu.malik@depaul.edu, ikanj@cdm.depaul.edu

Ashish Gehani
Computer Science Laboratory, SRI
Menlo Park, CA, USA
gehani@csl.sri.com

***Abstract*—Workflow management systems (WMS) help automate and coordinate scientific modules and monitor their execution. WMSes are also used to repeat a workflow application with different inputs to test sensitivity and reproducibility of runs. However, when differences arise in outputs across runs, current WMSes do not audit sufficient provenance metadata to determine where the execution first differed. This increases diagnostic time and leads to poor quality diagnostic results. In this paper, we use program specification to precisely determine locations where workflow execution differs. We use existing provenance audited to isolate modules where execution differs. We show that using program specification comes at some increased storage overhead due to mapping of provenance data flows onto program specification, but leads to better quality diagnostics in terms of the number of differences found and their location relative to comparing provenance metadata audited within current WMSes.**

I. Introduction

Workflow management systems (WMS) (1; 2; 3) stitch together different scientific modules into a workflow application and help to automate and monitor their execution. Workflow applications, however, often lack a formal testing framework. Thus, if failures happen during execution (4) or the results are not as expected (e.g., obey unlikely distributions), it is often challenging for users to be able to quickly localize the error *i.e.,* narrow the cause down to a module and, more precisely, a function or line number in the module.

One option is to debug workflow applications. However, debugging requires manual intervention, which is often not available in the case of long-running, batch workflow application. In the absence of debugging, provenance metadata, which represents information about data flows and collected during workflow execution is useful to conduct error localization.

Provenance metadata represents lineage information about the outputs of each workflow module. However, audited provenance data in most WMSes (1; 2; 3) consists of only partial information about workflow execution. Even the most low-level provenance systems (5; 6; 7) audit data flows and not program control flows. Consequently, using provenance metadata to determine fault and error locations within a specific module or a function within a module often becomes insufficient.

Consider the workflow application in Figure 1, which consists of five processing modules with input and output data flows. In this application, when the input `parameter` file

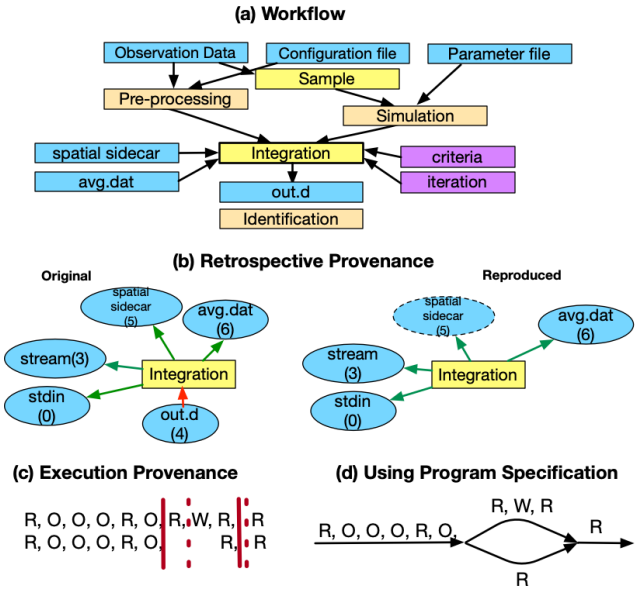

Fig. 1: (a) A workflow application's (b) execution provenance is different across original and reproduced execution. (c) Localizing the cause with execution provenance is ambiguous. (d) Program specification precisely localizes the difference.

is changed, the (true) location at which workflow execution changes is a *control flow* statement within the `Integration` module. Consequently, the final output is produced in one execution and not the other. We show that provenance data audited at different levels by current WMSes (1; 2; 3) cannot determine this location.

- Prospective provenance (8) describes the workflow, all the modules and the inputs/outputs (Figure 1(a)). Being descriptive, this form of provenance only records changes to file across any two executions, which in this case is the `parameter` file, i.e., it does not narrow the location of difference to any specific module.
- Retrospective provenance (5; 9) represents execution at the granularity of process and files, including intermediate files, consumed from previous modules. In Figure 1(b), this form of provenance detects a changed intermediate file produced from the Simulation module, but the Integration process is a black-box—it does not locate any differences within the

module.

- Execution provenance (10; 11) audits all input/output system call events in a process using tools such as *ptrace*. As Figure 1(c) shows at this level, we do observe that the execution behavior of the `Integration` module has changed as number of system calls differ. However, we still do not know the location of the difference as the system calls do not encode any function-level information, and aligning them without function stack information is ambiguous. Figure 1 (c) shows two out of three possible alignments, each of which semantically map to different locations where execution differed (dashed and solid lines).

As the example shows provenance helps to narrow to a specific module but does not report where in the module execution differed. In this paper, we use *program specification* to precisely locate semantic constructs—functions, and within a function specific statements—where execution provenance started differing. A program specification represents expected behavior of the program and is used for static analysis. Represented as a directed graph, it includes all functions specified in a program and, for a given function, all control flow properties such as branches, iteration, and recursive calls. Since execution provenance consists of system calls, which are functions, once mapped onto a specification, location of functions and control flows are precisely known.

The primary contributions of the paper are:

Obtaining a concise representation of program specification. Program specification obtained via source code transformation is too verbose consisting of all instructions. Since execution provenance consists of only data flow system calls, we must prune away *irrelevant* nodes in the directed graph of a program specification, *i.e.,* nodes to non-system call instructions. Node removal, however, changes the structure of the resulting graph. We provide a node removal algorithm that keeps system call data flow paths invariant.

Using the functional context of a system call on the program specification. In Figure 1, the mapping of execution provenance and program specification assumes same functional context. However, in general, the functional context must be determined. We present two contributions here: First, we show that mapping execution provenance at the granularity of system calls to program specification requires exponential state. Second, we use function-level traces of system calls and represent them at *per* function level and not as one large sequence to map onto the program specification.

Differencing disambiguated execution provenance. Given function-level traces of system calls, we must determine function calls and specific control flow statements where the first difference in execution arose. Offline edit-distance based methods require a large state-space and an appropriate cost-model, which is often missing. We present a greedy algorithm that compares paths obtained from mapping two function-level traces of system call onto the program specification to precisely identify locations where the traces diverge, and

```
int main(int argc, char **argv) {
  int arg = atoi(argv[0]);
  F(arg);
  return 0;
}

void F(int arg) {
  char buf1[10];
  char buf2[10];
  int fd;

  fd = open("a.txt", "rw");
  if(arg == 1) {
    read(fd, buf1, 10);
    write(fd, "hello", 5);
  } else {
    read(fd, buf1, 10);
    read(fd, buf2, 10);
  }
  close(fd);
}
```

T_1	T_2
O	O
R	R
W	R
C	C

Fig. 2: Program P with two traces T_1 and T_2.

locations where the traces converge. Our greedy approach precisely determines if two traces differ due to control flow or difference in number of loops.

Experimental prototype. We have developed an experimental prototype, `ProvScope`, based on LLVM IR (12) as program specification and Intel Pin (13) to obtain system call functional context. We use scientific workflows obtained from WorkflowHub.eu (14), and, for specific modules, generate multiple runs by changing input parameters and datasets. For these modules, we use SPADE (5) to generate execution provenance. We then use `ProvScope` to identify locations in the module where the difference arose. We show that those locations are more precise than what is reported by provenance auditing or workflow management systems, with a 20% storage cost over typical provenance auditing for higher precision.

The rest of the paper is organized as follows. Section II presents the overview of our approach. Section III-A describes how to obtain concise representation of program specifications. Section III-B describes how to obtain functional context, and Section III-C outlines the greedy algorithm. We present implementation and experiments in Section V-B, related work in Section VI and conclude in Section VIII.

II. ProvScope Overview

Given the source code of a deterministic program P and the transformation of the code into a specification $P_{spec}(P)$, the objective of `ProvScope` is to precisely determine locations in the specification where execution provenance T_1 of P, obtained by executing P with input arguments $\{I_1, \ldots, I_n\}$, differs from execution provenance T_2 of P, obtained by executing P with input arguments $\{I'_1, \ldots, I'_n\}$.

We represent execution provenance of T of $P(I_1, \ldots, I_n)$ as a trace $T = (t_1, \ldots, t_n)$ where in each t_i represents a system call label. The label consists of only the specific type of system call such as *read (R), write (W), open (O), close (C)*. Arguments of system calls are ignored as part of labels since we are only interested in location where an execution control flow changes and not a change to the argument. System call labels are present in the specification of program P.

Example 1: Figure 2 shows a program P with one input argument. When executed with `arg == 1` it produces trace $T_1 = (O, R, W, C)$ and when executed with `arg == 2` it produces trace $T_2 = (O, R, R, C)$.

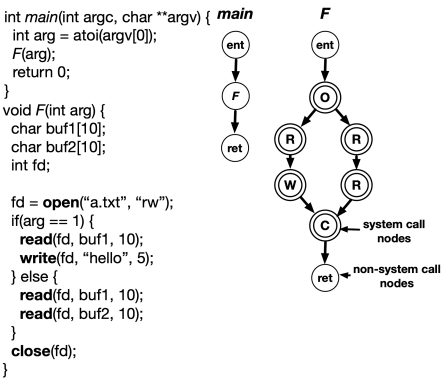

Fig. 3: The program P in Figure 2 and its specification consisting of two directed graphs. Node with double stroke represents system calls.

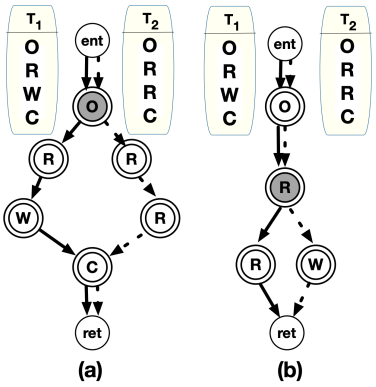

Fig. 4: Paths corresponding to traces and the location of the divergence point (shaded gray). For the same traces, the location of the divergence point changes based on the specification.

Given three types of program elements: (i) function calls, (ii) system calls, and (iii) special instructions: entry and return, the specification, $P_{spec}(P)$ is a translation of P to the program elements and the control flow between the elements. We assume the specification is represented as a set of directed graphs $P_{spec}(P) = \{G_1, \ldots, G_n\}$, in which (i) G_1 corresponds to entry function "main" of P, and (ii) each $G_i(V_i, E_i)$ is a control flow graph of a distinct function element that appears in P. Vertices V_i in $G(V_i, E_i)$ also correspond to program elements, *i.e.,* function elements (including this function element for a recursive call), system calls, and special instructions, and edges E_i are control flow edges between those elements. Thus, each G_i 'belongs' to a function and its nodes 'call' other functions, system calls, and special instructions including itself. The representation of P_{spec} as a set of graphs is necessary per the abstract stack model (15) so that spurious edges are not introduced.

Example 2: Figure 3 shows $P_{spec}(P)$, which consists of two directed control flow graphs, one for function **main** and one for function **F**. As shown in the Figure, the specification does not consist of other program elements, such as global variable assignment and other instructions, but only the three program elements. The edges of the specification map to the control flow of the program: in main it is sequential flow and in **F** it is a branched flow.

To determine where execution trace T_1 differs from execution trace T_2, we map the trace T_i onto a qualifying path S in the specification, and obtain the divergence points. We define the qualifying path as follows:

Definition 1. *Given a trace $T = (t_1, \ldots, t_s)$, let $H = (v_1, \ldots, v_t)$ be a path in $P_{spec}(P) = G$, iff*

i) v_1 is an entry point of function "main", and v_t a return call of function "main",

ii) for every $v_i \in H$, there exists a node in $P_{spec}(P)$,

iii) for every edge $(v_i, v_{i+1}) \in H$, one of the below must be true:

1) if v_i is a function node in G_i, then v_{i+1} is the special instruction entry node of the graph G_j called by the function of v_i,

2) if $v_i \in G_j$ is a return node, then v_{i+1} is one of the destination nodes of the node in G_i that called G_j function, or

3) edge (v_i, v_{i+1}) corresponds to an edge in some G_i.

iv) T and H are non-contradicting, i.e., the nodes in $P_{spec}(P)$ that correspond to T occur in the same sequence as the system calls t_i of T.

Example 3: Figure 4 shows paths on the specification that correspond to the traces T_1 and T_2 of Figure 2. The Figure 4 shows how the same traces map to paths in different program specifications.

We define the divergence point as the point where execution traces differ.

Definition 2. *Given two paths $H_1 = (u_1, ..., u_\lambda), H_2 = (v_1, ..., v_\mu)$, u_i $(1 \leq i < \lambda)$, is a divergence point iff.*

$$(1 \leq j < \mu) \wedge (u_i = v_j) \wedge (u_{i+1} \neq v_{j+1})$$

There might be multiple divergence points in two traces, and our algorithm in Section III-C returns all points, often the user is interested in the first point of divergence. The divergence point is shown as shaded nodes on both the specification.

III. Locating Divergence Points

To obtain a precise mapping between a trace element and a node in the specification, we need a specification $P_{spec}(P)$ that consists of only the valid elements of P, i.e. program elements necessary to search for trace elements, namely function call labels, system call labels, and special instructions. The following three sub-sections describe (i) how to obtain a *reduced* specification for any program P (Section III-A); (ii) obtain a trace that maps to a path in the reduced specification (Section III-B) and (iii) given two paths from the reduced specification, locate divergence points using a greedy procedure (Section III-C).

A. Obtaining Program Specifications

It is possible for a program P to have a control flow in which no system call is involved. Nodes that are part

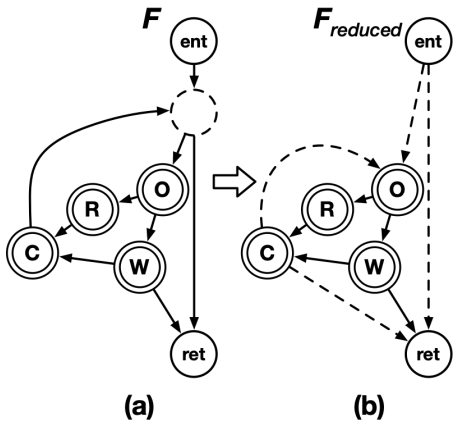

Fig. 5: (a) Redundant node (dashed) in a function graph. (b) Removal of redundant node retains all control flows via system calls in the function graph.

of such control flows can be pruned away as they increase the size of the specification graph but do not increase the preciseness of locating the divergence point. However, pruning must be performed in such a way that any control flow path corresponding to system calls is retained. To retain all control flow paths involving system calls we define a redundant node in a function graph G_i as follows:

Definition 3. *A node in G_i is* redundant *if it does not contain any of the program elements: (i) a function label, (ii) a system call label or (iii) special instructions: entry and return.*

A program's specification (obtained via LLVM-IR-based tools (12)) consists heavily of such redundant nodes and removing the *redundant nodes*, as our experiments show, significantly reduces the size of the program specification (by about 90-95%), allowing the number of the paths in the specification to be linear in terms of the trace consisting of system call labels. Algorithm 1 removes redundant nodes by deleting any self loop edge from a redundant node and connecting every incoming neighbor of the redundant node to all outgoing neighbors of the redundant node. The operation **remove()** is equivalent to creating a complete bipartite graph (16) between the incoming neighbors and the outgoing neighbors of the redundant node. We show that despite applying Algorithm 1 on a every G_i of $P_{spec}(P)$, a path that could be previously enumerated using the system call trace can still be enumerated without any loss of information. This path invariant is important as it does not reduce the accuracy with which we identify divergence points, since no new trace element is added or deleted. We formalize this as a theorem.

Let s, t be two non-redundant nodes in a function graph G_i. A sequence $S = \langle \ell_1, \ldots, \ell_r \rangle$ of labels, where $r \in \mathbb{N}$, is said to be an s-t-labels-sequence if there is an s-t path H in G (we omit the subscript i for simplicity) such that sequence of labels is obtained by concatenating the labels of the vertices on H is identical to S. Note that redundant nodes do not have labels, and other significant labels are from program elements viz. function and system calls, and special instructions.

Algorithm 1 Procedure for removing redundant nodes in a program specification

```
1: function REMOVE(G)
2: n_e ← FINDREDUNDANTNODES(G)        ▷ A list of redundant nodes in G.
3: While n_e ≠ ∅
4:     u ← n_e.get()
5:     if u has a self-loop then RemoveEdge(u, u)
6:     dsts ← getOutNodes(u)
7:     srcs ← getInNodes(u)
8:     For dst ∈ dsts Do
9:         REMOVEEDGE(u, dst)
10:    For src ∈ srcs Do
11:        REMOVEEDGE(src, u)
12:        For dst ∈ dsts Do
13:            ADDEDGE(src, dst)
14: return G
```

Theorem 1. *Let $G = (V, E)$ be a directed graph. The following statements hold:*

(i) Let $\{v_1, \ldots, v_k\}$ be a set of redundant nodes in G. For any two permutations π and π' of $\{v_1, \ldots, v_k\}$, the graph obtained from G after applying the removal operation following the order prescribed by π is the same graph obtained from G after applying the removal operation following the order prescribed by π'.

(i) Let v be a redundant node in G and let G' be the graph obtained from G after applying the operation **remove()**. *Then, for any two non-redundant nodes s, t in G, there is a bijection between the set of s-t-labels sequences in G and the set of s-t-labels sequences in G'.*

(iii) Let $G' = (V', E')$ be the graph obtained from G after the iterative application of the **remove()** *operation to the redundant nodes in G and the graphs resulting from G during this process. Then G' is computable in time $\mathcal{O}(n^2)$, where n is the number of vertices in G.*

Proof. (i) It is easy to see that the **remove()** operation is commutative, and the statement follows.

(ii) Let s and t be two non-redundant nodes in G. First, note that since both s and t are non-redundant nodes, both are present in G'. The map f is the identity function ι, that is, it maps each s-t-labels sequence in G to itself in G'. The function ι is well-defined. This can be seen as follows. Let $S = \langle \ell_1, \ldots, \ell_r \rangle$, where $r \in \mathbb{N}$, be an s-t-labels sequence in G, and let H be a path that induces S. If H does not contain redundant nodes, then H remains in G', and hence, $S = \langle \ell_1, \ldots, \ell_r \rangle$ is an s-t-labels sequence in G'. Otherwise, the redundant nodes in H will be removed resulting in an s-t path H' in G'. Since redundant nodes are unlabeled, it follows that the sequence obtained by concatenating the labels of the vertices on P' is S, and hence S has an image under ι (which is obviously unique).

Since ι is the identity function, it follows that it is injective. To show surjectivity, let $S' = \langle \ell'_1, \ldots, \ell'_r \rangle$ be

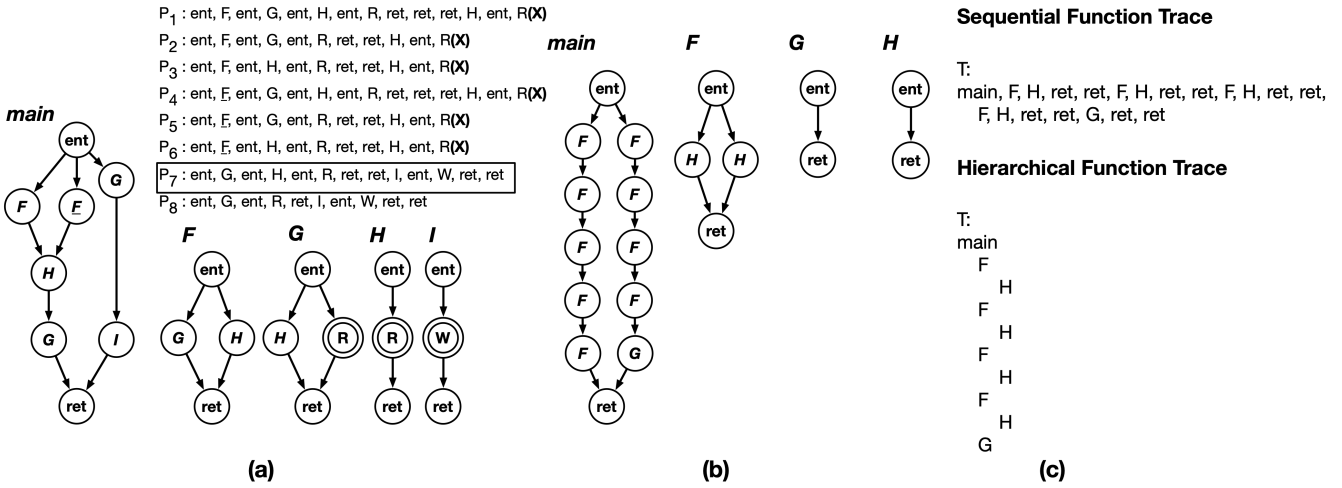

Fig. 6: Function level traces prune search paths and thus reduce search complexity. In (a) system call traces will require 8 paths and function call trace will require 2 paths. The F function call is underscored to keep the node unique (but with the same function call label). R**(X)** represents the point where the unmatched system call was found. (b) shows that function calls must also be dis-ambiguated, and (c) shows how hierarchical function trace is useful for reducing search complexity. The indentations represent the caller-callee relationships.

an s-t-labels sequence in G' corresponding to a path H'. Then either H' is in G, and hence S' is also an s-t-labels sequence in G, which must be the preimage of S' under ι, or H' resulted from the removal of redundant nodes from an s-t path H in G. Since redundant nodes are unlabeled, it follows that the sequence S of labels of the vertices of P is identical to S', and hence, S is the preimage of S' under ι, thus proving the surjectivity of ι.

(iii) We can compute G' from G by applying the procedure given in Algorithm 1, while avoiding adding edges to redundant nodes during the process (since such edges will eventually have to be removed). Since the number of vertices in G' is at most that of G, it follows that the total number of edges (and vertices) that were removed from G and/or added to G' during the process is $\mathcal{O}(n^2)$, and hence the above procedure can be implemented in time $\mathcal{O}(n^2)$.

□

B. Mapping the Trace to Path(s) in the Specification

We assume the specification is reduced, unless stated otherwise. We also assume each trace corresponds to one or more paths in the set of graphs corresponding to a specification.

Consider the program specification consisting of four functions and a trace $T_S = (R, W)$. In order to search for this sequence of system calls, the naive approach would need to go through 8 paths as written in the Figure 6 (a) (the "X" represents the point it stops traversing, for the path contradicts with the system call trace). The paths are counted using depth-first search (DFS), though, in general, one will have to traverse the same node multiple times as long as the trace does not contradict with the path unlike normal DFS.

On the other hand, if a corresponding function trace for system calls is available then we reduce the number of paths to search for significantly. In Figure 6 (a) consider the function call trace, $T_F = main, G, H, R, ret, ret, I, W, ret, ret$. For this trace, we only need to traverse 1 path as the G function call eliminates the candidate P_1 through P_6 immediately in Figure 6 (a). The observation made in this example has also been confirmed by our experimental evaluation, which demonstrates that using T_F (as opposed to T_S) greatly reduces the time complexity, thus making the identification of root causes more scalable.

However, changing the audit granularity still creates several paths if the same function is repeated several times as shown in Figure 6 (b). As shown in the Figure, one would have to enumerate $2^4 = 16$ paths before going into the correct branch of the first divergence at the entry point. To ameliorate this problem, we represent function call traces hierarchically, as illustrated in Figure 6 (c). With this transformation paths only need to be searched on a per function level. For example, in Figure 6 (b), only 2 paths in $main$ need to be enumerated, and given a valid path, just 1 path in each F, for a total of 2 paths.

C. Path Alignment of Two Traces

Let T_1 and T_2 be two function traces, let P_1, P_2 be the two paths as computed in the previous subsection, that correspond to T_1 and T_2, respectively. Assume that $P_1 = (u_1, \ldots, u_r)$ and $P_2 = (v_1, \ldots, v_s)$. To align P_1 and P_2, we use a greedy method. Let u_i be the vertex in P_1 with the smallest index i such that u_i appears in P_2; let v_j be the vertex on P_2 with the smallest index j such that $v_j = u_i$. Then, the greedy method aligns u_i with v_j and recurses to align the postfixes $P_1' = (u_{i+1}, \ldots, u_r)$ and $P_2' = (v_{j+1}, \ldots, v_s)$ of P_1 and P_2,

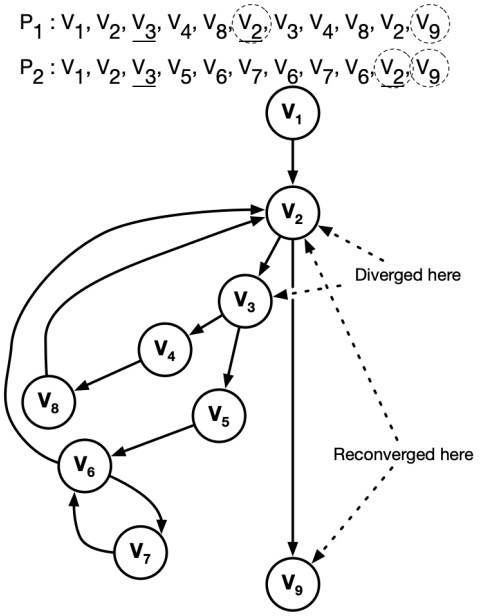

Fig. 7: An example of the greedy approach. In the paths on the top-right, underlined nodes are the divergence points and nodes with dotted circles are where the paths converged. Each v_i represents function call, syscall, or entry/return instructions.

respectively. The method stops when the return/last node of P_1 is aligned with the return/last node of P_2 or the paths terminate. This greedy method is applied at the top-level of their hierarchical function-call traces T_1 and T_2, and then recursively compute an alignment between every two aligned nodes on P_1 and P_2 that correspond to a function call.

Algorithm 2 shows the pseudo-code for the combined procedure of finding paths given traces, and using the obtained paths to find divergence points using the greedy method. Note that the algorithm finds *all* points of divergence. To find all points of divergence it must find a point of reconvergence as well. Figure 7 illustrates how the greedy approach finds both the points of divergence/re-convergence, given two paths. The algorithm finds V_3 as the first point of divergence, but re-converges at V_2 since nodes V_4 and V_8 are not found in P_2. V_2 again becomes a divergence point as the post-fixes of P_1 and P_2 do not align, finally reconverging at V_9.

Algorithm 2 Procedure to find the enumerate the path and find divergence/convergence point

function PROVENANCEALIGNMENT(T_{F_1}, T_{F_2}, PS, DP)
  if T_{F_1} and T_{F_2} are equal at every function level **then**
    return DP;
  if T_{F_1} and T_{F_2} are equal at a current function level **then**
    for ($i = 0$; $i < T_{F_1}$.length i++)
      ProvenanceAlignment(T_{F_1}[i], T_{F_2}[i], PS, DP);
    return DP;
  $ps = PS[T_{F_1} \rightarrow$ funcName];
  $\langle int, int \rangle$ alignedPairs $\leftarrow$ Null;
  $i = j = 0$;
  $P_1 =$ getPossiblePath(T_{F_1}, ps);
  $P_2 =$ getPossiblePath(T_{F_2}, ps);
  $\langle node, List\langle int \rangle \rangle$ $match \leftarrow$ Null
  while ($i < P_1.length$ and $j < P_2.length$) **do**
    if ($P_1[i] == P_2[j]$) **then**
      alignedPairs.append((i, j));
    else ▷ Diverged
      DP.append(P_{i-1});
      if $match$ has not been filled **then**
        for ($k = j$; $k < P_2.size()$; $k = k + 1$)
          match[$P_2[k]$].append(k);
      jStart = j;
      while ($i < P_1.size()$)
        if ($P_1[i]$ is found in match) **then**
          $j = \min_{j \in match[P_1[i]]} \{j: j \geq jStart\}$; ▷ Converged
          alignedPairs.append((i, j));
          $i = i + 1$, $j = j + 1$;
          break
        else
          $i = i + 1$;
  for ($t = 1$; $t <$ alignedPairs.length - 1; $t = t + 1$)
    ProvenanceAlignment(T_{F_1}[alignedPairs[t][0]], T_{F_2}[alignedPairs[t][1]], PS, DP);
  return DP;

IV. PROVSCOPE ARCHITECTURE AND IMPLEMENTATION

Figure 8 shows the architecture of `ProvScope` in a typical distributed workflow management system consisting of a central job-submission node and several worker nodes. We assume scientific modules run locally within a node, though the workflow application can be entirely distributed executing different modules at different nodes. The setup consists of using SPADE (5), which provides provenance auditing for distributed environments for generating system call traces, and Intel PIN (17) for generating function call traces. Both these auditing tools collect the traces from scientific modules running at each compute node locally at a node, which is then collected centrally. `ProvScope` is used for causal diagnostics at the central node, based on program specifications of workflow applications, generated using LLVM toolchain (12). We particularly note that SPADE and PIN are generic tools that work at the operating system interface and therefore can be coupled with any workflow management system. In our case we have integrated them with Galaxy (18) and Nextflow (19) WMSes, and they can be easily used with other WMSes.

LLVM compiler toolchain generate specifications at a per-function level for program source code and any system dependent libraries. Since most workflow applications depend on scientific computing modules, obtaining source code of programs or depedencies is often not difficult. In LLVM, per-function program specification is a well-specified representation (12) comprising of a collection of "basic blocks" of non-branching sequences of instructions, in which the instructions of relevance are function and system calls. The last instruction in a block can direct the control flow to another block. To translate library dependencies, and most importantly, `glibc` into program specification, we instead use *musl libc*,

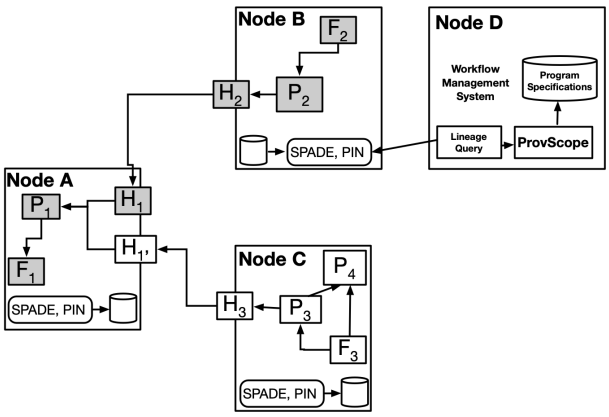

Fig. 8: ProvScope architecture. P_i, H_i, and F_i represent process, host, and file *i.e.,* nodes of retrospective provenance.

which is built into an LLVM bitcode library (20). Further, sometimes, programs invoke system calls indirectly through glibc, i.e., the call may be accessed through a function pointer or a struct field. To facilitate mapping of such calls, we divirtualize references to system calls using pointer analysis (21), which is a form of static analysis for simplifying latent system calls in functions.

To generate provenance traces, we use SPADE. SPADE auditing requires no instrumentation of the application. SPADE uses Linux auditing to intercept each system call (Viz. *read*, *write*, *open*, *close*, *send*, *recv*, *fork*, *execve*, etc.) to determine the running process' state and the arguments to the system call[1]. For some file-related system calls, SPADE hashes the content, while for some others such as *open*, *clone*. SPADE examines input arguments and the process control block, and extracts the file path parameter.

To audit function calls, we use PIN (13; 17), a dynamic binary instrumentation framework for x86 instruction-set architectures to create function traces. In general the output of PIN can be very large (in GBs) as Pin sees every instruction in the user process that is executed; we have modified PIN to only output functions that lead to system calls, which, as experiments show, drastically reduces the size of the function trace. PIN produces hierarchical function trace, and we preprocess the trace to produce, T_F, the function trace for each called function. Mapping of function calls to system calls is done via process identifiers.

ProvScope works for multi-threaded applications. For multi-threaded application, ProvScope separates the traces based on the association time intervals and *pid* to infer causal dependence and partial orders. Thus during alignment it only compares system and function calls related to a given process as determined by the partial order.

V. Experimental Evaluation

A. Experiment Setup

Our experiments are conducted on a high-end machines running the Linux kernel 5.4.0-84-generic on Ubuntu 20.04 having Intel(R) Xeon(R) Silver 4215R CPU@ 3.20GHz with 3 TB of SSD and 20 GB of RAM. SPADE and Intel PIN were installed. ProvScope and SPADE query module was only installed on the central machine from which jobs were submitted. All output data including the differentials are currently reported in JSON format, which includes a link to the specific function in the program specification and the vertex identifier with the function where the differential started or ended.

1) Real-world Workflows: We have downloaded workflows from WorkflowHub.eu (14), and conducted experiments on modules that are used frequently (for instance, rm, uniq, sort) and used as main program (minimap2, bwa, and mcf).

2) Module Specification and Provenance Traces: Table I provides details about the specific modules. The complexity of each module is quantified in terms of the program specification, which is an abstract control flow graph represented in terms of the number of instructions and edges in the graph, the number of back edges which informs about the number of loops that exist, the number of unique functions calls, and which modules make recursive function calls. Further, we create two runs of the program: an original and a reproduced run. These two runs correspond to the execution of the module in which input parameters or datasets are varied. We state the number of traced function and system calls.

We would like to highlight that the abstract control flow graph does not directly correspond to the LLVM-IR obtained by directly compiling the program/module with the clang compiler. Data in Table I represents the LLVM-IR after some pre-processing. In particular, LLVM-IR graph vertices that have no associated system calls or functional calls are removed. This reduction is necessary to improve the scale and complexity of the program specification and enumerating provenance traces on them. In our experiments, we show the total amount of reduction in graph size with pre-processing.

B. Experiments

We have designed our experiments to answer three high-level questions: (i) How accurately does ProvScope identify the differences within a module in comparison to baselines?; (ii) What is the overhead of ProvScope in comparison to baselines?; and (iv) What are the storage requirements of ProvScope?.

Baselines. We compare ProvScope with system and function call traces obtained from SPADE and PIN. To compare SPADE system calls, we used simple *diff* command in Linux, which is based on an efficient implementation of the dynamic program that solves the longest common subsequence problem (27). We term ProvScope as **PS** and SPADE as **SP**.

1) How accurately is ProvScope in identifying the location of a differential?: Since the number of function calls far exceeds the number of system calls in executions except for very few exceptions, we would expect the finer granularity for the place of divergence when SPADE and ProvScope are compared. This can be measured by the sheer number of divergences in both programs, which is shown in Figure 9. As we

[1]We include system calls that operate on more than one entity such as link(), which operates only on metadata and not content.

TABLE I: Workflow Modules. (Inst: Instructions; Func.: Function; Recur: Recursion; Orig: Original; Repr: Reproduced)

		Program Specification					# of System Calls		# of Function Calls	
Id	**Module**	**# Inst.**	**# Edges**	**# BackEdges**	**# Func.**	**# Recur.**	**Orig.**	**Repr.**	**Orig.**	**Repr.**
1	rm (22)	37328	5959	194	304	10	13	22	1083	1242
2	uniq (22)	26843	4694	170	230	10	21	18	1569	1321
3	b2sum (22)	39583	5209	181	242	6	15	12	4585	3482
4	sort (22)	64104	9740	392	499	11	57	57	4848	4889
5	cat (22)	27720	3931	161	187	3	13	22	1339	1428
6	bzip2 (22)	50437	6343	365	190	5	38	148	1338	1183522
7	mcf (23)	21492	3580	192	112	6	1203	2359	179438513	535273359
8	minimap2 (24)	145341	15360	856	397	9	187	69	203037	101416
9	bwa (25)	155056	17831	1061	556	8	136	136	73665	73407
10	sed (26)	113866	16053	736	427	26	22	22	7627	7909
11	grep (26)	125336	17951	880	501	36	36	39	7740	26265

Report by SP	Modules
Correct Reconvergence	rm, uniq, sort, cat, bzip2, bwa, sed
Incorrect Reconvergence	b2sum, mcf, minmap2, grep

TABLE II: Modules with correct/incorrect reconvergence observed by **SP**

can see, the number of differences are bigger for `ProvScope` in all the modules because of the finer granularity of the trace except for b2sum. b2sum falls within the exceptional case, which is when SPADE reconverges at the system call which the corresponding stack trace is different. In other words, let's say that at *write* system calls in one path reconverges with the *write* system calls in another path. Nonetheless, since SPADE lacks the function call stack information, the write system call in one path could come from *printf*, and the other from *fwrite*. In the Table II, 1 represents the program such wrong reconvergences are found, and 0 not found. If the paths reconverges with such places, it could diverge and reconverge again with different call stack information, hence could lead to the bigger number of divergences. Through Table II and Figure 9, we have shown the relationships, if "the location of reconvergence is correct with function call stack information", then "the number of divergences for `ProvScope` is larger than that of SPADE" empirically. The bars for minimap do not exist because the numbers of those modules were significantly larger (SP: 2821, PS: 2840).

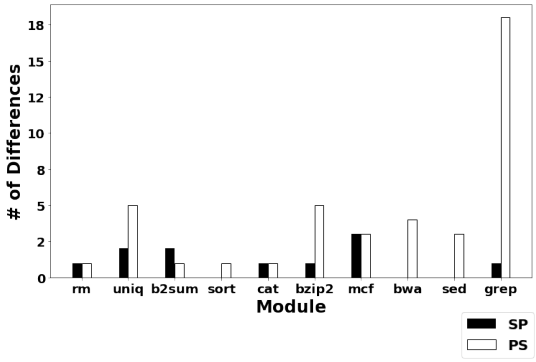

Fig. 9: # of differential locations

2) What is the overhead of `ProvScope`?: We consider this overhead both in terms of generating execution provenance traces at the level of system and function calls, as well as the overhead for determining the location of differences. In general, function call tracing is significantly more than system call tracing. Figure 10 shows for specific modules, the overhead of the tracing for both system call trace and function call trace. Function call trace overhead costs exponentially higher than system call trace. Thus our strategy in `ProvScope` is to use SPADE for system calls, and apply PIN only to specific modules that SPADE informs are different.

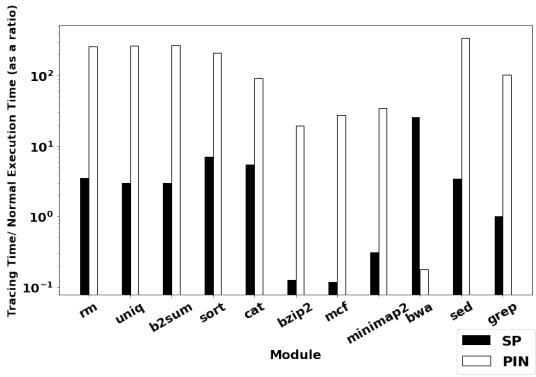

Fig. 10: Tracing Overhead over Normal Execution

We compare the overhead of more accurate reporting of `ProvScope` over the baselines. The baselines, in effect, require computation of a dynamic program, where as in `ProvScope` computing differentials consists of searching for a path on the program specification and the alignment method. We report how expensive is the computing the differentials or the dynamic program with respect to merely re-executing the module again. This is reported as overhead ratio on the y-axis. Figure 11 shows that (i) simple `diff` comparison for most program is often times more expensive. But, the overhead of simple `diff` is more significant for all the executions except for `mcf` and `bwa`. In both cases, we have couple of functions in which path finding is quite ambiguous and `ProvScope` takes more time, but as our previous result shows is far more accurate in reporting the number of differentials, and the type of differential.

3) Storage overhead: We consider this question from the overhead of program specification graphs, the state mainte-

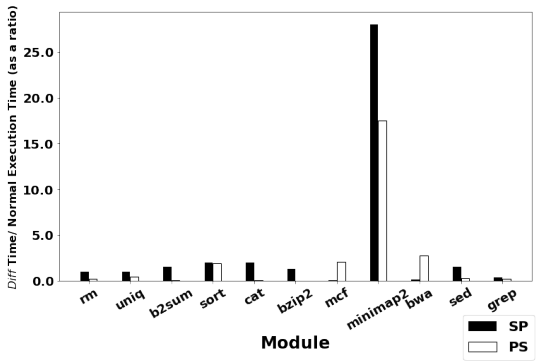

Fig. 11: Comparison Overhead over Normal Execution

nance in the algorithm, and the overhead of the function call traces. Figure 12 shows the reduction in the size of the graph after preprocessing the program specification as obtained from LLVM toolchain. As Figure 12 shows there is significant reduction in all structural units of the graph: the instructions, edges, backedges (implies the number of loops), and functions that are not relevant to system calls, with instruction reduction rate of at least 95%.

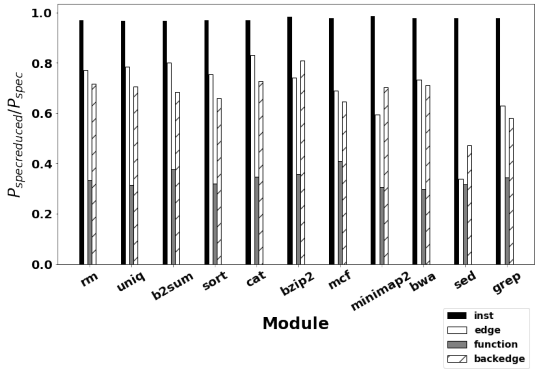

Fig. 12: Reduction in Program Specification

The overhead of the algorithm in terms of stack maintenance is negligible (few Kbs) due to significantly reduced size of the function trace (see next experiment) and the program specification. Here we would like to point out that when we used only system call traces, and not function call traces, the path enumeration did not terminate for any of our programs, and the state grew exponentially (28; 29). Since our leased machines had high memory capacity this lead to the program running for hours and extended into days.

Finally, we report the storage overhead of the function traces for the original and the reproduced execution. We also preprocessed PIN traces to remove function calls that do not lead to system calls. We compare function traces obtained by Intel PIN versus the size of the trace actually used by `ProvScope`. As we see the reduction is significant in the number of function calls bringing the function call traces to about 5-10% of the original function trace.

VI. Related Work

As large number of scientific workflows are created and shared, the emphasis has increasingly shifted from development to analysis. Most scientific workflow management systems (30) allow users to compare the specifications of workflows. However, since the specification is different from a run, in case of failure or unexpected output, the causes are often not known (31). Comparing workflow runs may help analysis users better understand why an expected output has not been produced (32).

TABLE III: Reduction in PIN Function Traces for Original (O) and Reproduced (R). 2 right rows represents the reduced Original and Reproduced Traces.

Module	O.	R.	red. O.	red. R.
rm	1083	1242	115	144
uniq	1569	1321	287	269
b2sum	4585	3482	195	107
sort	4848	4889	670	675
cat	1339	1428	51	89
bzip2	1338	1183522	220	55022
mcf	179438513	535273359	375866	740932
minimap2	203037	101416	118783	33269
bwa	73665	73407	2110	2114
sed	7627	7909	2484	2692
grep	7740	26265	2081	2915

Collecting provenance at the operating system level provides a broad view of activity across the computer and does not require programs to be modified. Workflows, are often composed of executables from different binaries built from different programming languages, and runtime support. Collecting workflow-level provenance creates portability issues as well as makes it difficult to compare across runs. Several systems (5; 7; 33) collect provenance by monitoring executions at the operating system. Collected execution provenance allows for querying multiple aspects of information regarding an entity: what is the origin of the entity; how the entity is derived; and when it originated. These systems, currently, do not provide tools and utilities for comparing provenance needed for comparative analysis.

Some edit-based methods (9; 34) to compare provenance traces have recently been developed. These methods either do not apply to system call traces (34) or do not identify differences within a process or across processes of a distributed system (9). The work closest in spirit to our work is determining root causes in climate model. While this approach advocates the use of source code for determining differences between an ensemble and an experiment run to identify a root-cause for failure, it analyses source code at the variable level. Consequently, differences that arise due to control flows and iterations are not accounted for, which the authors mention as a limitation. In this paper, we have effectively shown how to use program specification to identify differentials between two runs and also explain them.

VII. Discussion and Future Work

Debugging is a complex task involving program behavior and human understanding of the program. Our approach aims to simplify this process by automating fine-grained execution provenance and mapping it to the program specification, which concisely represents program behavior, especially in WMSes

where manual debugging is challenging. While program specification hinges on source code being available, debugging also needs source codes.

Workflow management systems do not provide a comprehensive failure repository, as with software artifacts (26). Consequently, to examine failures, we must explore individual modules and their software repositories. Extending our prototype to scale to more realistic distributed workflows is part of our future work. In addition, our prototype currently does not extend to message passing programs, which requires us to address the concurrency and non-determinism issues in provenance capture, an open issue in current provenance auditing systems. Finally, real workflow systems audit provenance at different granularities, thus making it challenging to perform experiments within it. Thus, we choose to audit provenance at the operating system level.

VIII. Conclusion

Using workflow management systems (WMS) for automating the execution of scientific modules has become common in scientific computing. However, when failures and errors arise, current WMSes are not equipped with differencing and debugging tools to perform diagnostics. Auditing provenance metadata as part of WMSes is becoming essential. As our work shows, low-level provenance metadata is, however, insufficient to narrow down the cause within a specific workflow module. We have presented `ProvScope` that combines program specification with execution provenance to determine precise locations where execution runs may diverge. The collection of function call traces makes alignment efficient, as we show. We show types and locations of differences that `ProvScope` reports in comparison to other edit-distance based measures.

IX. Acknowledgements

Tanu Malik and Yuta Nakamura are supported by the National Science Foundation under grants CNS-1846418, NSF ICER-1639759, ICER-1661918. Tanu Malik and Ashish Gehani are supported by National Aeronautics Space Agency under grant AIST-21-0095-80NSSC22K1485.

References

[1] E. Deelman, K. Vahi *et al.*, "Pegasus, a workflow management system for science automation," *FGCS*, 2015.

[2] D. Hull, K. Wolstencroft, and et. al., "Taverna: a tool for building and running workflows of services," *Nucleic acids research*, vol. 34, no. suppl_2, pp. W729–W732, 2006.

[3] I. Altintas, C. Berkley, and et. al., "Kepler: an extensible system for design and execution of scientific workflows," in *SSDBM*. ACM, 2004, pp. 423–424.

[4] N. Yigitbasi, M. Gallet, and et. al., "Analysis and modeling of time-correlated failures in large-scale distributed systems," in *2010 11th IEEE/ACM International Conference on Grid Computing*, 2010, pp. 65–72.

[5] A. Gehani and D. Tariq, "SPADE: Support for Provenance Auditing in Distributed Environments," in *Middleware*, 2012.

[6] T. Pasquier, X. Han, T. Moyer, A. Bates, O. Hermant, D. Eyers, J. Bacon, and M. Seltzer, "Runtime analysis of whole-system provenance," in *CCS*, 2018.

[7] D. J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, "Hi-fi: collecting high-fidelity whole-system provenance," in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012, pp. 259–268.

[8] C. Lim, S. Lu, and et. al., "Prospective and retrospective provenance collection in scientific workflow environments," in *2010 IEEE International Conference on Services Computing*. IEEE, 2010, pp. 449–456.

[9] P. Thavasimani, J. Cała, and P. Missier, "Why-diff: Exploiting provenance to understand outcome differences from non-identical reproduced workflows," *IEEE Access*, 2019.

[10] D. H. Ton That, G. Fils, Z. Yuan, and T. Malik, "Sciunits: Reusable research objects," in *IEEE eScience*, 2017.

[11] Q. Pham, T. Malik, and et. al., "LDV: Light-weight database virtualization," in *ICDE'15*, 2015, pp. 1179–1190.

[12] C. Lattner, "The architecture of open source applications: Llvm," *The architecture of open source applications*, 2014.

[13] V. J. Reddi, A. Settle, and et. al., "Pin: a binary instrumentation tool for computer architecture research and education," in *Workshop on Computer architecture education*, 2004.

[14] "Workflow.eu," https://workflowhub.eu/, 2018.

[15] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *POPL*, 1995.

[16] R. Diestel, *Graph Theory, 4th Edition*. Springer, 2012.

[17] Intel, "PIN: A Dynamic Binary Instrumentation Tool," https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html, 2012.

[18] E. Afgan, A. Lonie, J. Taylor, and N. Goonasekera, "CloudLaunch: Discover and deploy cloud applications," in *Future Generation Computer Systems*, 2018.

[19] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame, "Nextflow enables reproducible computational workflows," vol. 35, pp. 316–319, 2017.

[20] "Musllvm: The musl libc llvm bitcode library," 2019. [Online]. Available: https://github.com/SRI-CSL/musllvm

[21] SRI, "Clam: Crab for Llvm Abstraction Manager," https://github.com/seahorn/clam, 2019.

[22] "Coreutils - gnu core utilities," http://www.gnu.org/software/coreutils/, 2016.

[23] "Spec cpu2006 benchmark," https://www.spec.org/cpu2006, 2006.

[24] "minimap2," https://lh3.github.io/minimap2/, 2018.

[25] "Burrow-wheeler aligner for short-read alignment," https://github.com/lh3/bwa, 2009.

[26] "Software-artifact infrastructure repository," https://sir.csc.ncsu.edu, 2006.

[27] J. W. Hunt and M. D. MacIlroy, *An algorithm for differential file comparison*. Bell Laboratories Murray Hill, 1976.

[28] P. Boonstoppel, C. Cadar, and R. D. Engler, "Rwset: Attacking path explosion in constraint-based test generation," 2008.

[29] S. Krishnamoorthy, S. M. Hsiao, and et. al., "Tackling the path explosion problem in symbolic execution-driven test generation for programs." IEEE, 2010.

[30] J. Freire, C. T. Silva, S. P. Callahan, E. Santos, C. E. Scheidegger, and H. T. Vo, "Managing rapidly-evolving scientific workflows," in *IPAW*, 2006.

[31] P. Missier, S. Woodman, H. Hiden, and P. Watson, "Provenance and data differencing for workflow reproducibility analysis," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 4, pp. 995–1015, 2016.

[32] S. Cohen-Boulakia and U. Leser, "Search, adapt, and reuse: the future of scientific workflows," *ACM SIGMOD Record*, vol. 40, no. 2, pp. 6–16, 2011.

[33] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, "Practical whole-system provenance capture," in *SoCC*, 2017.

[34] Z. Bao, S. Cohen-Boulakia, S. B. Davidson, A. Eyal, and S. Khanna, "Differencing provenance in scientific workflows," in *ICDE*, 2009.