Efficient Provenance Alignment in Reproduced Executions

Tanu Malik, Yuta Nakamura
School of Computing,
DePaul University
Chicago, IL, 60604, USA
tanu,nakamura1@cdm.depaul.edu

Ashish Gehani
Computer Science Laboratory
SRI, International
Menlo Park, CA 94025, USA
gehani@csl.sri.com

Abstract

Reproducing experiments entails repeating experiments with changes. Changes, such as a change in input arguments, change due to non-determinism in the runtime, or a change in the invoking environment may alter results. If the resulting outcome alters significantly, perusing results is not sufficient—users must determine if the experiment computed the same steps and causally analyze the impact of a change. This can be both challenging and time-consuming. In this paper, we present *provenance alignment* as a method for comparing a reproduced execution with recorded provenance of the original execution. Our alignment is based on comparing counters and hashes that take into account the specific location in the program, the control flow by which execution arrived there, and data inputs. Experiments show that the method has a low overhead to compute a match and realigns with a small look-ahead buffer.

1 Introduction

It is often necessary to reproduce an experiment and determine if reproduced results are consistent with results obtained from the original experiment. For computational experiments, reproducing an experiment often entails a change, such as a change in input arguments, change due to non-determinism in the runtime, or a change in the invoking environment. If a change alters the result of a reproduced experiment significantly—making it inconsistent—users often want to causally analyze how the presence or absence of a change impacts the result.

Provenance from the original experiment, if available, aids in such causal analysis. Consider reproducing a computational experiment in a changed environment. If the reproduced experiment fails to run in the new environment due to a missing dependency, the user must determine the experiment's configuration and dependencies. This can be arduous given complex experiments and numerous dependencies [2]. Previous work has shown that [9], [1], albeit changes to environments, if reproduced experiment reuses dependency provenance collected from the original experiment, it will repeat successfully.

Provenance reuse of this form produces a consistent result by minimizing the impact of the change (in this case, the environment) [7]. When there are changes due to input arguments or changes due to non-determinism, reusing provenance, however, is equivalent to replay of the original experiment, which will produce the original consistent result. For a reproduced execution, changes due to input arguments or due to non-determinism may cause execution path differences in the execution of the reproduced experiment, and lead to inconsistent results [5]. To reuse provenance in the context of these changes, we need to analyze differences between the original and reproduced executions. this requires determining where the two executions align and diverge, in terms of their data and control flows.

In this paper we present a provenance alignment technique that meaningfully compares a reproduced execution with the provenance of its original execution to identify inconsistencies in results that may arise due to change in input arguments or due to runtime non-determinism. The alignment relies on collected operating system provenance of the original execution. System call-based provenance does not capture intra-process program control flow. Consequently, the two traces need to be aligned to identify where divergences arise when the program input is modified.

To align provenance, the reproduced execution compares, on-the-fly its current state with the provenance of the original execution. If there is divergence from the provenance of the original execution, the reproduced execution continues, and skips unmatched execution instructions. Further alignment happens based on comparing the data flow and control flow paths of both the original and reproduced execution. Our alignment is based on comparing counters and hashes that take into account the specific location in the program, the control flow by which execution arrived there, and data inputs. Differences due to non-deterministic system calls can either be ignored by the user or redirected to a database lookup to allow for provenance replay.

The rest of the paper is organized as follows: Section 2 describes an example; Section 3 presents a formal framework; Section 4 presents alignment overheads; Section 5 discusses related work and we conclude in Section 6. Appendix includes supplementary material.

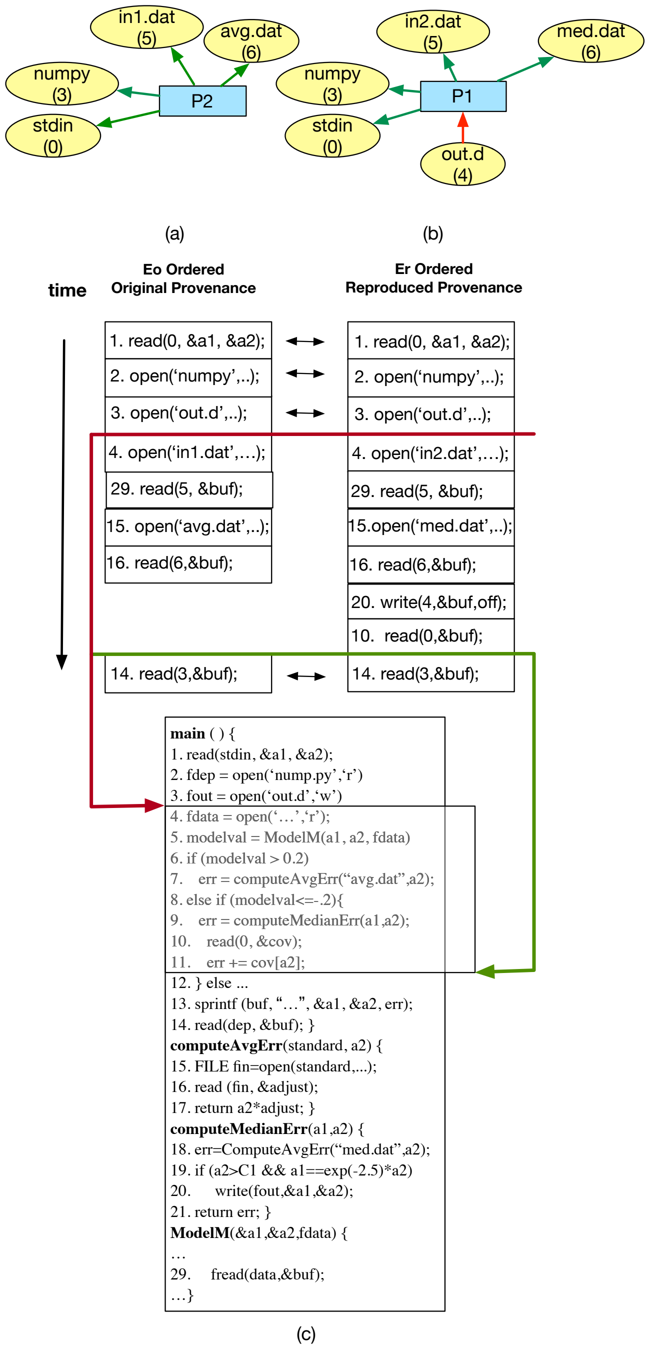
2 Provenance Alignment: Example

We illustrate through an example how the provenance alignment method detects differences in execution. We assume Alice shares with Bob a container (to rule out environment differences) consisting of application source code, binaries, and its system and data dependencies. We also assume that Alice has compiled her program with debug information and executed scripts with tracing information to include program locations in binaries. Alice also provides required documentation to run the application program with input arguments (parameters and datasets). Bob repeats the program by changing an input dataset (`in1.d` $\rightarrow$ `in2.d`) and observes an inconsistent result.

The container stores operating system provenance from Alice's original execution, E_o, and Bob's reproduced execution, E_r (Figure 1(a) and (b)). For efficiency' sake, we assume system provenance is recorded at the granularity of data and process-related system calls, namely `open, read, write, close, fork, execve`, etc. The number of nodes and edges do not match (and so the inconsistency), but in the course of alignment we are interested in determining when nodes and edges match, versus when they do not.

To perform alignment, each system call in E_r is matched with a system call listed in E_o. For example, Lines 1-3 in E_r match in terms of system call specification, arguments to the system call, and the order of execution, and so E_r, the reproduced execution, is determined to be aligned with E_o here in terms of provenance. Bob's execution observes a divergence in the arguments of the system call at Line 4: E_o and E_r open different files. The reproduced execution, E_r becomes unaligned from the provenance of the original execution, E_o. E_r continues to remain unaligned for the next few system calls. The alignment tracking stops progressing on E_o and remains stalled at Line 3. As E_r proceeds from Line 4, the provenance alignment method determines if any subsequent system call of E_r aligns with the next k system calls of E_o. For $k = 6$, this happens at Line 14 of E_r, at which point the reproduced execution aligns again with the provenance of the original execution.

Once the execution aligns, Bob is informed of locations in the code where the reproduced execution differs. Thus Bob does not have to peruse the entire code but the highlighted portion, shown in Figure 1(c). Looking at the code, Bob learns that inconsistent results happen due to a data-dependent control flow in Alice's program. In particular, the specific input argument, which Bob changed, also changed the output of the computational model `ModelM`, leading the reproduced execution to take a different control path: E_o calls `ComputeAvgErr` and E_r calls `ComputeMedianErr`. The above example shows unaligned execution traces based on a simple program. In general, programs consist of a large number of files and may align/unalign at multiple locations.



```
main ( ) {
1. read(stdin, &a1, &a2);
2. fdep = open('nump.py','r')
3. fout = open('out.d','w')
4. fdata = open('…','r');
5. modelval = ModelM(a1, a2, fdata)
6. if (modelval > 0.2)
7.    err = computeAvgErr("avg.dat",a2);
8. else if (modelval<=-.2){
9.    err = computeMedianErr(a1,a2);
10.    read(0, &cov);
11.    err += cov[a2];
12. } else ...
13. sprintf (buf, "…", &a1, &a2, err);
14. read(dep, &buf); }
computeAvgErr(standard, a2) {
15. FILE fin=open(standard,...);
16. read (fin, &adjust);
17. return a2*adjust; }
computeMedianErr(a1,a2) {
18. err=ComputeAvgErr("med.dat",a2);
19. if (a2>C1 && a1==exp(-2.5)*a2)
20.     write(fout,&a1,&a2);
21. return err; }
ModelM(&a1,&a2,fdata) {
…
29.    fread(data,&buf);
…}
```

Figure 1: The provenance of (a) original, and (b) reproduced execution. Ellipses are entities and rectangles are activities. File descriptors from system calls are in brackets, e.g., (`in.dat(5)`). Provenance is compared and aligned. The example program in (c).

3 Provenance Alignment

We first describe the metadata, consisting of source code ordering, system call functions, and function arguments, which determines if executions are aligned or not. We then define control-flow alignment, and data-flow alignment in terms of this metadata, and finally describe how to compute provenance alignment, *i.e.,* establish equivalence between two ordered provenance graphs.

3.1 Metadata for Alignment

The provenance associated with the execution of a program P is maintained as a 2-tuple $E_P : (R, <_p)$, where R is a set of relationship operations, and $<_p$ (program order) is irreflexive partial orders on R.

Operations in R correspond to system calls such as $read, write, open, close, send, recv, fork, execve$, etc. Each operation of R stores four types of metadata: $(pid, name, content\_val, uid)$, in which pid identifies the executing program (activity) under which this operation runs. *name* is the an array of name(s) of the entity(ies) or activity that the operation is aiming to act upon [1] $content\_val$ identifies the content of the first name as a hash. If name refers to a file (entity), it is the hash of the file content. If name refers to a process it is the hash of the process input arguments and the binary that gave rise to the process. *uid* is an arbitrary unique tuple identifier; it serves to coaelsce operations that have same metadata but differ in $content\_val$.

To begin with we limit execution to sequential programs i.e., single-threaded programs. Thus, programs may fork, giving rise to new processes, but there is no shared memory between parent and child, nor are there any synchronization operations. Thus, program order, $<_p$, is a union of total orders of different processes. Specifically, given two distinct operations $o_1, o_2 \in R$ in which *pid*'s are not equal then there is no known ordering between o_1 and o_2, *i.e.*, $(o_1 \nless_p o_2 \wedge o_2 \nless_p o_1)$ We discuss extension of our formalism to multi-threaded programs in Appendix C.

To compare the original and the reproduced execution, each maintains a counter-based tuple in *uid*. Algorithm 1 shows the how the counter is uniquely computed. The method `CounterProg()` is called whenever the program's execution reaches system calls. Assume the system call in program P occurs at program location L in a function N defined within a program file F. These are passed as arguments to the method, which maintains a global counter, $globalUid$, and returns a tuple of $(F, N, L, globalUid)$.

`CounterProg()` computes counters using postorder depth first search, *i.e.,* a callee's counter is incremented before subsequent counters of the calling function are computed. Note, the reverse of DFS postorder is a valid topological ordering and preserves the program order [8]. The incremented counter is used to compute the total number of system calls along a path from the beginning of the program to the current execution point.

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

The *uid* tuple is uniquely computed if the program does not have any loops. In case of loops, the provenance trace will consist of *uid* tuples which have same values on (N, F, L) but differs on $globalUid$. Using a self-join on the trace, we group operations that have same values on (N, F, L), but in which $globalUid$'s increment by a fixed number. For all grouped operations which repeat periodically, we assume their system calls belong to a loop. The tuple is revised to include a count of the number of times N, F, L repeat periodically. An example using loop counts is described in Appendix A.

Algorithm 1: Basic Counter Computation

```
1  $globalUid = 0$
2  $mainProg = true$
3  $currentFunc = uidTuple = Empty$
4  CounterProg(F,N,L,globalUid):
       Input  : syscall in function N in program file F at
                line number L
       Output : uidTuple
5      if $mainProg$ then
6          $currentFunc = N$
7          $mainProg = false$
8      if $(N == currentFunc)$ then
9          $uidTuple = (F, N, L, globalUid + 1)$
10     else
11         $currentFunc = N$
12         CounterProg(F,N,L,globalUid)
```

Control-flow alignment. In this the reproduced and original execution are aligned if and only if the *uid* tuples match. A *uid* tuple in control flow alignment matches if the (N, F, L) values are same and the $globalUid$ of the reproduced execution is larger or equal than the $globalUid$ of original execution. Content need not be preserved.

Definition. *Executions* $E_1 : (OP_1, <_{p_1})$ *and* $E_2 : (OP_2, <_{p_2})$ *are said to be equivalent if there exists a one-one mapping (bijection) between* OP_1 *and* OP_2 *that preserves* $<_p$ *and uidTuple in every operation.* In Section 3.2, we apply this definition to Example 1.

Data-flow alignment. To define data flow alignment we need to align operations not only on *uid* but also the $content\_val$. The (hash) value of the content cannot be simply compared. $content\_val$ of past operations that are causal to this content must also be same. To make a comparison on $content\_val$, we define $ext(E_i)$ as the external inputs of an execution, and *occurs_before_order*, $<_{ob}$, as an irreflexive transitive closure on the program order, $<_p$. We assume a given data flow graph[2], G on $<_{ob}$, that has an edge from a read operation to all the write operations that the read operation affects. Given a data flow graph, the hash value of the *content_val* is computed similar to counter computation by performing a postorder DFS and computing the hash values based on the hash values of its children (or lineage ancestors in data flow graph). Appendix B describes the algorithm for computing hash values.

Definition. *Two executions* $E_1 : (OP_1, <_{p_1})$ *and* $E_2 : (OP_2, <_{p_2})$ *are said to be data-flow aligned if and only if* $ext(E_1) = ext(E_2)$ *and there is a one-one mapping between* OP_1 *and* OP_2 *that preserves G on* $<_{ob}$*, the data flow.*

3.2 Provenance Alignment

Assuming collected provenance of a reproduced and original execution as E_o and E_r, the provenance alignment method jointly steps over the operations of E_o and E_r one at a time, determining if they match (based on control or data flow alignment). If they unalign, then the lockstep processing stops. The alignment maintains a buffer of size k, which stores the next k system calls from the original execution (Line 7 and 8 in Algorithm 2). The procedure only steps through the reproduced execution further and determines if any system call of the reproduced execution matches with a system call in the buffer (Line 11). If it does match, the alignment method determines the offset between reproduced and original execution using *globalUid* and resumes joint iteration from that system call for reproduced and original execution, using the offset to subtract *globalUiD* and align calls further. Otherwise, the executions remain unaligned.

Algorithm 2: Provenance Alignment

```
1  offset = 0;Alignment($E_r, E_o, k, type, offset$):
2      $align = true$;
3      while ($align$ && ($i \neq E_{rn}$ || $j \neq E_{om}$)) do
4          if (Match($E_{ri}$, $E_{oj}$,type,offset) ) then
5              Increment i and j
6          else
7              $align = false$; foreach $t$ in $j \ldots j{+}k$ do
8                  lookahead[cnt] = $E_{ot}$
9              foreach $t$ in $i \ldots i{+}k$ do
10                 if ($E_{rt}$ in lookahead) then
11                     align = true
12                     $offset = globalUid_R - globalUid_O$
13                     break;
14             i = t; j = index of matched lookahead;
```

[2]Such a data flow graph is obtained by dynamic tainting of syscalls using systems such as LIBDFT [3].

We already demonstrated data flow alignment in Section 2.
Example: Control flow alignment. E_r and E_o iterate in lockstep for the first seven system calls (Lines 1-4, 29, 15-16) in example 1. The reproduced execution unaligns at program location 20 with the original execution and remains unaligned for the next system call. The original execution stores in lookahead buffer program location 14. When the reproduced execution reaches program location 14, the *globalUid* of reproduced execution is 10 and of original execution is 8. Since F, N, L match and the executions are unaligned, an offset of 2 is computed and used for subsequent alignment. Since the system calls belong to two different functions, the user is informed of the two causal functions, `main` and `ComputeMedianErr` that cause the executions to be misaligned.

4 Experiments

We evaluate the runtime performance of the method, and its ability to correctly inform misalignment based on an input. Experiments are on a machine with Intel 3.4GHz CPU (2 cores), 16GB RAM, and 32-bit Ubuntu 18.08.

Benchmark Datasets We have used four programs for evaluation, three benchmark programs and a scientific program implementing a computational hydrology model. The benchmark programs help to precisely determine alignment in case of for-loops, control-flow, and data-flow alignment. The scientific program consists of several modules. Each module implements several different methods and analyzes a scientific variable, such as temperature, evapotranspiration, etc. A configuration file describes user's choice of modules and respective methods to compute the output from the model. Provenance alignment informs the user if a change in a module specified in the configuration file affects a choice of method in another module.

Program	LOC	# Syscalls	Changed input
401.bzip2	5739	10	Input data file
429.mcf	1579	11	Input data file
473.astar	4285	18	Configuration file
PySumma	54K	349	Configuration file

Table 1: Program details

Experiment Details Each program is compiled with debug information so line information is part of the executable. Scripts were run with tracing enabled. We have implemented our provenance alignment method within the context of a container system, Sciunit [7], to avoid environment changes. Each original execution of the program is intercepted at specific system calls and the files are recorded within a Sciunit container directory. Each reproduced execution is run from the set of files recorded within this directory.

Correctness of Alignment For the reproduced and the original execution to perfectly align the lookahead buffer must be sufficiently large. Too small a buffer and the executions may never align, and too large a buffer may result in unnecessary comparisons between system calls. We assess the impact of different values of the lookahead buffer in benchmark programs. In particular, we first run the benchmark programs for increasing values of k but with the same input. Since the input is same, system calls must align. For each program, we determine the lowest maximum k at which all system calls align. For this value of k, we change the input as described in Table 1, and determine the total number of times control or data flow re-alignment happens. We then double the value of k once to determine if the number of re-alignment changes determining how sensitive is alignment to size of k. Figure 2 shows the result. The benchmark programs merely increase the number of realignments by one, but PySumma has a significant change.

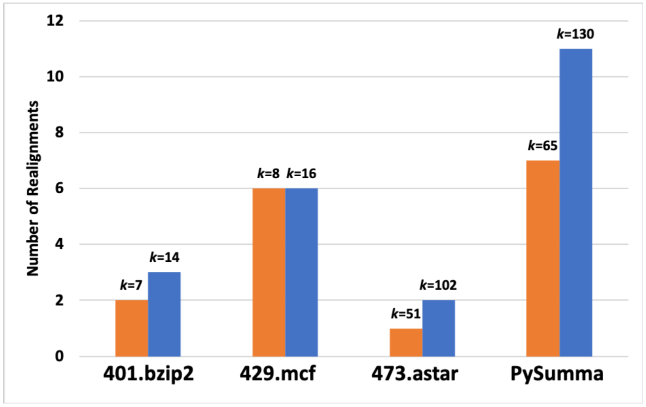


Figure 2: Changes in number of realignments for different k values

Performance We study the additional time taken to perform control and data flow alignment. We run the reproduced program twice. In the first run, we do not change the input so the reproduced execution perfectly aligns with the original execution. The overhead is thus for recording of system calls and for hashing the input data. In the second run, the reproduced execution is run with a different input. Since the change leads to programs taking different paths and having different system calls, the overhead includes that for matching, creating buffers, and realignment. The overheads for both control and data flow alignment are between 5-15% for each of the benchmark programs. for pySumma, the control flow alignment overhead is between 5-8% but data flow alignment is between 35-50%. As is observed the overhead for both control and data flow alignment is very low for benchmark programs. The overhead for the real program remains low for the control flow alignment as the traces simply need to compare the *globalUID* values, but increases in data flow alignment depending on the change in the type of input. If the input causes a change in the configuration of several modules, then the execution is misaligned most of the time.

5 Related Work

Collecting provenance at the operating system level provides a broad view of activity across the computer and does not require applications to be modified. Currently these systems however do not reuse provenance. Lakhani *et al.* [4] showed how to reuse OS provenance for replay, but did not consider an independent reproduced execution with changed inputs.

Comparing provenance traces has been extensively studied in workflows, which model provenance at the descriptive level. [6] provides an excellent survey of DAG comparison methods and tools. Here we focus on methods that compare retrospective or execution provenance of workflows, especially for the purpose of explaining inconsistencies in results. VisTrails encapsulates a workflow experiment as a set of files, and detects changes to files contents by hashes, allowing users to later check that provided data does indeed match the original experiment. Comparing files based on hash-value or code is not enough; it is essential to understand how the files are linked and executed.

Why-diff [6] is a step-forward and considers the retrospective or execution provenance of the workflow. The execution is not monitored using OS events. Consequently execution provenance is derived from programmatic constructs of blocks of activities and inputs and outputs (entities) to these blocks; such program-based execution provenance can easily miss differences due to data-dependent control flow or due to different time orders which can only be sufficiently monitored at the OS level. Even if programmatic and OS provenance is same, Why-diff only considers graph differencing algorithms based on node and edge matching. Implementing a match function is application-specific. We describe specifics of the match function in terms of contents, specification, and line numbers, and use that to order the directed graphs in the order of execution and make comparisons. Our matching procedure can thus explain differences at the code level, intuitively understood by the user.

6 Conclusion

Causality tracking helps to contrast independent experiment runs and explain how changes to an experiment affects outputs. In this paper, we have presented alignment as a method for contrasting the original and the reproduced provenance executions. Our method uses operating system provenance which has high fidelity to the actual execution of the program. We show how our method can detect data-dependent control flows in program and inform the misalignment in a usable way (at the source code level). Experiments show the method has insignificant overhead in terms of time and space.

Acknowledgments

This work is supported in part by National Science Foundation Grants CNS-1846418, ICER-1639759, ICER-1661918, and a US-Department of Energy Fellowship.

References

[1] Fernando Chirigati, Rémi Rampin, Dennis Shasha, and Juliana Freire. ReproZip: Computational reproducibility with ease. In *SIGMOD'16*, pages 2085–2088, 2016.

[2] Philip J. Guo and Dawson Engler. CDE: Using system call interposition to automatically create portable software packages. In *USENIX*, 2011.

[3] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, pages 121–132, 2012.

[4] H. Lakhani, R. Tahir, A. Aqil, F. Zaffar, D. Tariq, and A. Gehani. Optimized rollback and re-computation. In *Hawaii International Conference on System Sciences*, 2013.

[5] Paolo Missier, Simon Woodman, Hugo Hiden, and Paul Watson. Provenance and data differencing for workflow reproducibility analysis. *Concurrency Computation: Practice and Experience*, 28(4):995–1015, March 2016.

[6] P. Thavasimani, J. Cala, and P. Missier. Why-diff: Exploiting provenance to understand outcome differences from non-identical reproduced workflows. *IEEE Access*, 7, 2019.

[7] Dai Hai Ton That, Gabriel Fils, Zhihao Yuan, and Tanu Malik. Sciunits: Reusable research objects. In *IEEE eScience*, Auckland, New Zealand, 2017.

[8] Wikipedia. Depth first search. https://en.wikipedia.org/wiki/Depth-first_search, 2020. [Online; accessed 20-Mar-2020].

[9] Zhihao Yuan, Dai Hai Ton That, Siddhant Kothari, Gabriel Fils, and Tanu Malik. Utilizing provenance in reusable research objects. *Informatics*, 5(1), 2018.

A Provenance Alignment using Loops

We consider a small program with loops in which the input data changes leads to the loop being executed different number of times in the original and the reproduced execution. Figure 3(a) shows the program and Figure 3(b) and (c) shows the sequence of system calls in the original and reproduced executions. The original execution runs the loop with `n=1`, `m=2` causing two *read* system calls at program location 6 with $globalUid$ 3 and 4 (shown at the top of each system call box), and one *write* system call at program location 7 with $globalUid$ of 5. The reproduced execution runs the loop with `n=2, m =1` causing two sets of consecutive read and write system calls at program location 6 and 7 and $globalUid$ of 3 and 4 respectively, and then 5 and 6, respectively.

The executions remain in lockstep for the first three system calls for both control and data flow alignment. Assuming a large $k = 5$, in control flow alignment, the reproduced execution does not find a system call in which *uid* is greater than equal to *uid* of system calls of original execution in lookahead buffer. This constraint is not true till the reproduced execution reaches program location 8 when the executions align again. Note if the role of reproduced and original execution are reversed a false alignment may happen at *write* system call with $globalUid$ of 5. In this example, data flow alignment does not happen due to differing content of *send* system call which is based on loop content.

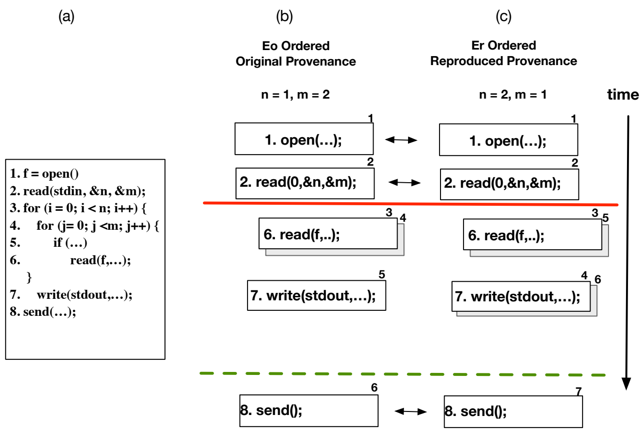

```
1. f = open()
2. read(stdin, &n, &m);
3. for (i = 0; i < n; i++) {
4.     for (j= 0; j <m; j++) {
5.         if (...)
6.             read(f,...);
   }
7.     write(stdout,...);
8. send(...);
```

Figure 3: The loop program (a) with (b) original, and (c) reproduced execution. The dashed line shows where control flow re-alignment happens. Data flow alignment, depending on content, may not happen.

B Computing Hash Values

To compute the hash value of $content\_val$ in an operation, a data flow graph, $G(V,E)$, is assumed in which V are the operations and $E \subseteq V \times V$ are edges between operations which have a data flow. Given such a G, the hash value is computed by performing a postorder DFS on G, *i.e.,* a hash value of a node is computed after computing the hash values of the successors (ancestors in a data flow lineage graph). The hash value of a node is computed based on the content of the node and the hash values of its successors. By adding the hash values of successors to the content of the node, we can simply compare the hash value of $content\_val$ in each operation without performing expensive traversals on G at each system call. Algorithm 3 presents the pseudo code for computing this hash values.

C Alignment for Multi-threaded Programs

To extend our provenance alignment method to multithreaded programs, the set of operations OP in R expands to include

Algorithm 3: Hash Values using Postorder DFS

```
1  DFSHash(G):
       Input  : G
       Output : hash value of vertex in G
2      foreach vertex in G do
3          if vertex is not visited then
4              hashVal = DFS(vertex)

5  DFS(vertex):
6      mark vertex visited
7      foreach successor v' of v do
8          if v' not yet visited then
9              hashVal += DFS(v')
10     return hash(data content of vertex) + hashVal
```

synchronization operations such as thread fork and join, lock acquire and release, monitor entry and exit, etc. The operation metatadata is collected at a thread level instead of process level. The execution is a three tuple consisting of $(OP, <_p, <_s)$, where OP is a set of operations, and $<_p$ (program order) and $<_s$ (synchronization order) are irreflexive partial orders on OP. The counter computation does not change and is computed per thread level. Control and data flow alignment are described in which the synchronization order is also preserved, and alignment matches records which belong to the same thread. We have not tested alignment on multi-threaded programs since currently we do not intercept lock acquiring system calls and do not have a robust implementation of data flow taint analysis in multithreaded programs. This is part of our future work.