Optimized Rollback and Re-computation

Hasnain Lakhani   Rashid Tahir   Azeem Aqil   Fareed Zaffar
Lahore University of Management Sciences

Dawood Tariq   Ashish Gehani
SRI

***Abstract*—Large data processing tasks can be effected using workflow management systems. When either the input data or the programs in the pipeline are modified, the workflow must be re-executed to ensure that the final output data is updated to reflect the changes. Since such re-computation can consume substantial resources, optimizing the system to avoid redundant computation is desirable. In the case of a workflow, the dependency relationships between files are specified at the outset and can be leveraged to track which programs need to be re-executed when particular files change. Current distributed systems cannot provide such functionality when no predefined workflows exist. In this paper, we present an architecture that provides functionality to produce both correct output as well as fast re-execution by leveraging the provenance of data to propagate changes along an implicit dependency graph. We explore the tradeoff between storage and availability by presenting a performance analysis of our rollback and re-execution scheme.**

I. Introduction

Scientific data is often derived by automating and sequencing several inputs and processes through a set of predefined rules and operations [12]. Such systems are typically modeled as directed acyclic graphs (DAGs) or dataflow networks [24], where participants and applications are represented as vertices with data and control dependencies. Workflow management systems allow users to create and manage instances of such pipelined operations.

Workflow management systems are typically used to execute long-running computational and data-intensive pipelines of operations, often across local and distributed resources [19]. Such systems are typically implemented as services where users submit their jobs and retrieve their results after the computations have been finalized. Given the computation- and data-intensive nature of typical scientific workflows, once scheduled for execution, individual jobs can potentially run for days. This model is appropriate for several scenarios in a diverse set of application areas, such as bioinformatics, seismic activity research, high-energy physics, astronomy, and climate monitoring, that require high-performance and high throughput computations. If the job completes successfully, the results of the computation can be retrieved by the clients. Using such systems, data can be constantly tracked as it is modified, consumed, or produced by monitoring and controlling the interaction of workflow participants and applications [24]. Scientific data produced as a result is therefore a product of significant time and computational resources. For example, researchers at the Southern California Earthquake Center use Coral with the Pegasus workflow management system [5] to perform large-scale Grid-based seismic hazard research. Each workflow contains approximately 840,000 tasks that require 17,000 CPU hours, generating 66 GB of data [3]. Similarly, each piece of information from Fermilab's Collider Detector is the output of a month of processing dozens of terabytes of raw data, and the cost to analyze a single protein stored in the Protein Data Bank is $70,000 [2]. Consequently, there is substantial adverse economic impact if large amounts of such computation and expensive data must be discarded because of either faults in the workflow processing or simply a need to run experiments with different sets of inputs or values. Research in this area has also focused on introducing fault tolerance for the workflow management systems.

Several scenarios necessitate the need for fault tolerance and flexibility in such computation- and data-intensive pipelines of operations: (i) A fault in the workflow specifications or one of the processes or applications used in the workflow can invalidate the results. A modified or newer version of a process or inputs can similarly invalidate the computation. (ii) Dataflow pipelines in both local or distributed environments are potentially prone to termination or corruption due to the problems of data inconsistency, job failures, network variations, network latency, and server outages. (iii) Empirical scientific research is often an iterative process where, based on results from experiments, a user can vary or re-calibrate inputs to some or all programs or processes.

An apparently feasible solution to the problem is to stop the workflow and restart the system by submitting a new job. This is not optimal for several reasons: (i) A lot of useful work may have already occurred that could potentially be unrelated to the changed inputs or modified applications and programs. (ii) In typical batch processing scenarios, jobs are often submitted to queues. Re-submitting the job to the queue again can mean longer delays and thus higher costs. As far as the client is concerned, the performance, utility, cost, and usefulness of such systems can potentially improve a lot if a reasonable fraction of useful work already done can be preserved across various different invocations of the workflow.

Several different approaches have been proposed in the research literature to address these issues and to reduce the overhead of recovering from such faults or changes to data inputs: (i) the ability to freeze workflow state so that it is preserved across several runs, (ii) re-executing only parts of the workflow so that redundant computations are not performed unless necessary, and (iii) the ability to checkpoint data and the ability to cache intermediate results [14] [17] [11]. Minimizing redundant work and preserving useful work across various different runs requires the ability to hone in on exactly what processes or information has been tainted in the case of faults and exactly what minimally needs to be re-computed in case of rollbacks and re-executions. This requires each user of the system to be familiar with the workflow design and process-level implementation details.

Workflow management systems were designed to abstract away such details from typical users and this reduces the utility of such system for all but the most savvy users of the system. Another problem with the situation is that in the case of a workflow, the dependency relationships between files and processes are specified at the outset and can be leveraged to track which programs or components of the workflow need to be re-executed when particular files or inputs change. Current distributed systems cannot provide such functionality when no predefined workflows exist. Even in the case of a workflow definition, extant schemes [11] often only allow the ability to rollback at the granularity of workflow processes.

Our scheme relies on a fine-grained provenance collection framework to transparently collect and store soft-state in the system, which can also act as a detailed history of all the process and data artifacts in the system. We then provide a novel scheme that leverages this information to preserve all the useful work that has been performed by the system while ensuring fast re-execution and correct output. Our system allows us to provide this functionality even without having any explicit workflow specifications in place. We also explore the inherent tradeoff between storage space and data availability in such an environment. We present a performance analysis of our scheme. We note that while a key assumption in the current implementation is that all the data and processes reside on a single system, the same concepts can be generalized to a distributed environment as long as a detailed provenance record is kept on all the nodes involved.

The rest of the paper is organized as follows. Section II describes the background and introduces the provenance infrastructure that forms the core of our re-execution scheme. Section III gives more details about our approach and the general algorithms and architecture of our scheme. We describe the implementation and performance analysis of our scheme in Section IV. Finally, in Sections V and VI, we conclude with a discussion of related work and future directions.

II. Data Provenance

Data Provenance in our context refers to the history of a process or a piece of information. A detailed provenance record on a system essentially allows us to record the whole enactment of a workflow as a series of operations by recording the creation of data and processes, and all the data and control dependencies between them. Such a record is typically kept as a DAG that shows the causal relationship between objects. There are several models that have been proposed to standardize the sharing of provenance information across various system entities. The Open Provenance Model (OPM) model [18] is one such digital representation of causal dependencies that provides a layer of compatibility when sharing provenance information. OPM categorizes the various different entities in a system as artifacts, processes, and agents. A sample provenance graph is shown in Figure 1.

SPADE [8] is a novel user-space framework that allows for transparent storage and propagation of provenance at all levels of a system. Built using the OPM model, the SPADE framework includes provenance producers, storage, visualization tools, and consumers. At the heart of the system is the SPADE kernel that fuses all of this information together, stores it in persistent storage, and allows users to access this information through a uniform interface. SPADE also provides analysis tools that can be used to navigate the provenance information generated without the need for any external software.

SPADE combines various data streams being generated in a system to create a coherent view of all the interacting artifacts in a system. The underlying storage data model in SPADE is a directed graph structure $G = (V, E)$ based on OPM. The types of vertices $v \in V$ are Processes, Artifacts, and Agents. The types of edges $e \in E$ are `used`, `wasGeneratedBy`, `wasTriggeredBy`, `wasDerivedFrom`, and `wasControlledBy`. As described earlier, causal dependencies and relationships between entities are depicted by edges that define how processes are generated, and how an artifact was created, derived, or transformed from another object. Each of the vertices also has a set of attributes describing it in detail. During the course of execution on a system, SPADE can transparently collect such provenance information through system call interposition. SPADE responds to all process- and I/O-related system calls to record these relationships in persistent storage. Since multiple processes can read and write to the same set of files, this can potentially lead to cycles in the provenance graphs where a process reads writes to a file and then reads from the same file at a later stage. To maintain the DAG semantics, SPADE deals with this problem by versioning the files, creating a newer version at each write. This ensures that at each step, we know exactly what version of the file a process read or wrote to explicitly.

The overall result is a detailed visual and quantitative history of each input, process, and data output in the system. It also acts as a detailed record of all the processing whether or not there is a workflow specification in place. SPADE provides novel filters to make sure the system is not deluged by the volume of audit data generated while maintaining enough context so that we can use the data for advanced analysis [7]. By relying on SPADE as a provenance generation, collection, and management subsystem to generate an implicit dependency graph, we are able to collect a fine-grained history of processing that can then be used to affect fast re-execution and rollbacks while preserving useful work and CPU cycles.

Data assurance

As discussed earlier, scientific data is typically an output of long chains of computation. The utility of any such piece of data depends on the assurances that can be made about it. By tracking each data object and process, we create a provenance graph and assurance attributes that can help

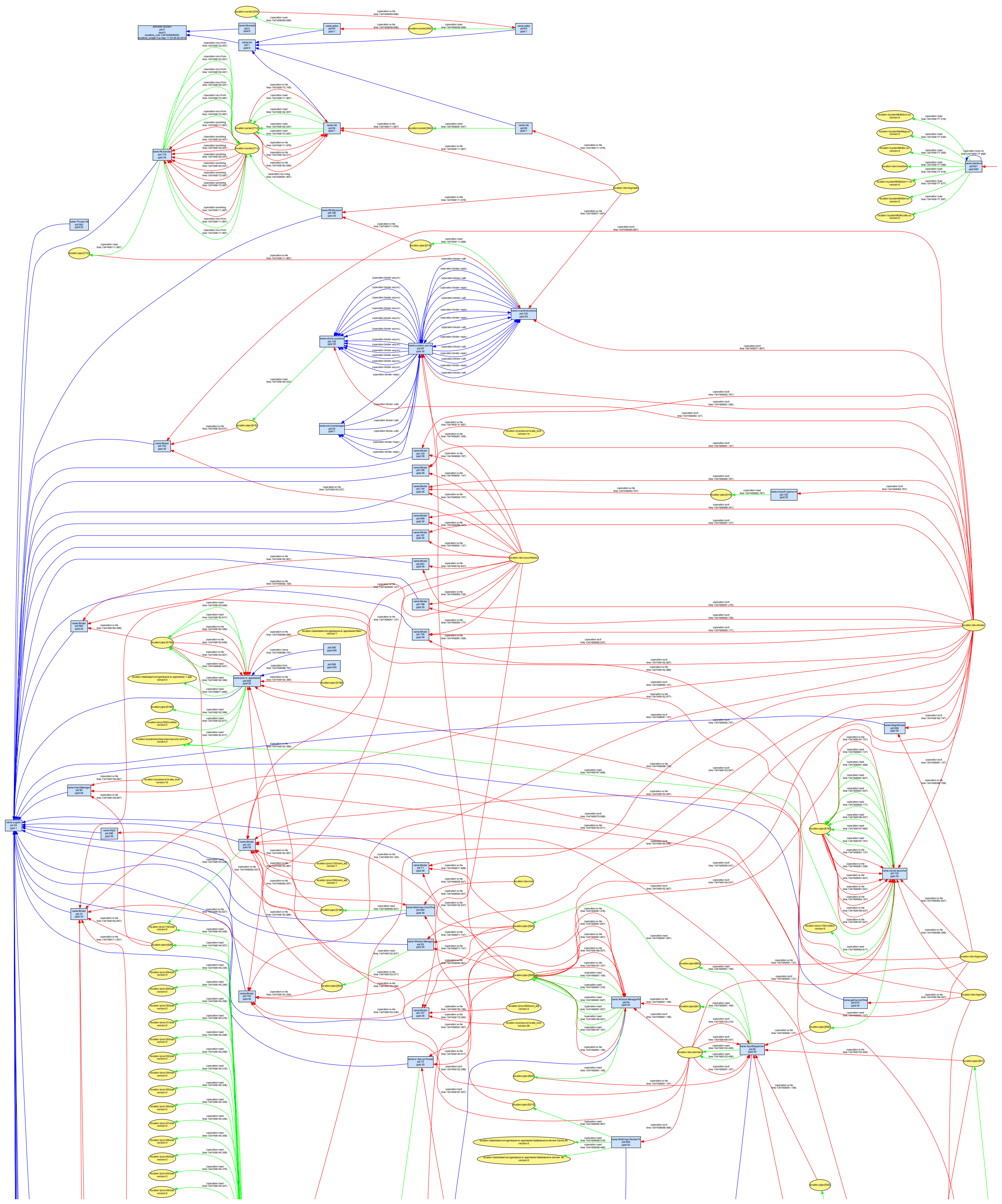

Fig. 1. Part of a provenance graph emitted by SPADE. The data and control dependencies in a typical execution become sufficiently complicated that determining the minimal re-computation needed when inputs or programs change is challenging.

us verify data trustworthiness dynamically, facilitating real-time awareness of each object's security, reproducibility, and correct computation. While there is extra storage overhead for keeping the metadata, the performance overhead imposed is minimal, particularly considering the increased assurance for the resulting data.

Detailed provenance information also helps ensure survivability by letting us hone in on the exact causes if something is found not to work correctly. Our scheme provides us the ability to restart, roll back, or re-execute the computation pipeline from an arbitrary point while preserving the order, correctness, and all the useful work from previous computation. To see why this holds, consider an information flow graph where vertices are data objects and edges are created when data from one object is read and then written into another object. If a particular data object is compromised, all the objects connected to it in this graph are affected. Forward and backward flow analysis through the provenance graph can help us locate the tainted components precisely. We can also perform what-if analysis on the various inputs to our computation through keeping such records.

III. Approach

The essential idea behind our scheme is similar to the caching and restart approach [11] that checkpoints and saves the state of processes or entire systems periodically. This saved state can subsequently be used to restore the execution state of a program or the system at some later point. The scheme works at the granularity of a workflow, and re-execution cannot take advantage of useful work done inside a workflow module. The use of automatically generated fine-grained provenance information allows our scheme to work at a much finer granularity. A key distinguishing factor in our scheme is that we are able to provide fast re-execution and correct output even when no workflow definition or management system is in place and does not force the users to think in terms of workflow primitives. Our design also obviates the need for explicit checkpointing and bookkeeping since the provenance subsystem automatically generates a succinct record of the processing, which can be leveraged at a later stage. A number of assumptions are made in our design, including: (i) the provenance metadata is complete and does not omit provenance events, (ii) it does not create false events, (iii) it does not alter the chronology of events, and (iv) it is stored in a persistent tamper-proof environment. Several hash-chain-based and checksum-based approaches have been suggested to guarantee the trustworthiness of provenance metadata. We do not address these issues.

As described earlier in the context of SPADE and the OPM model, any set of processing tasks involving data and control dependencies, whether specified through a workflow or not can be represented by a directed acyclic graph where vertices represent the processes and edges represent the data flows. Figure 1 shows one such provenance graph that signifies data and control dependencies. Note that the OPM convention of using octagons for Agent vertices, rectangles for Process vertices, and ellipses for Artifact vertices is followed. (SPADE also encodes the OPM semantics of vertices and edges in the color in which they are rendered when output is generated in Graphviz format [8].)

Algorithm 1 Identifying the forward subgraph for re-execution

```
Require: Provenance graph G
         Set S $\Leftarrow$ Inputs that changed
Ensure: Set C (descendants to be re- computed)
  for i $\in$ S do
    Queue Q $\Leftarrow$ $null$
    $Enqueue$(Q) $\Leftarrow$ i
    while Q $\neq$ $empty$ do
      x $\Leftarrow$ $Dequeue$(Q)
      if x $\notin$ C then
        $Put$(C) $\Leftarrow$ x
        $Enqueue$(Q) $\Leftarrow$ $Successors$(x) – C
      end if
    end while
  end for
```

With this metadata in hand, the problem reduces to identifying the minimal subgraph that is essential to ensure the fastest re-execution and the correctness of the output. In the worst case scenario, the entire computation would need to be performed. Depending on the type of function performed, our system allows us to hone in on the smallest number of operations and steps that need to be performed. This essentially provides a script, parts of which can be replayed to recreate data artifacts or to re-execute systems and processes. Once the exact subgraph has been identified from the original provenance graph, the vertices in the subgraph can be re-executed in a topological order so that tasks are submitted and re-executed in the order they happened originally, preserving the correctness of the output data. Scheduling heuristics can be used to further improve the performance of the scheme.

Our scheme allows users the ability to perform a variety of tasks, including:

1) Re-executing computation done between time t_1 and t_2.
2) Re-executing everything with a new input i_1.
3) Rolling the system back to arbitrary time t_3.

The user of the system can pose these tasks as command line queries to our system. The system automatically rewrites these commands as SPADE queries that search the provenance store to identify the minimal set of processes and files that would be needed to perform the above-mentioned tasks. The SPADE analysis tools mentioned earlier provide us the ability to extract detailed file and process dependency information, while giving us the ability to query their provenance metadata along several different dimensions such as process name, file name, time of process invocation, file modification time, and file size.

We first describe the working of our system in the context of Task 1. We query the provenance subsystem to identify all the vertices modified between time t_1 and t_2. This set of vertices S is used as an input to Algorithm 1 and is essentially the set

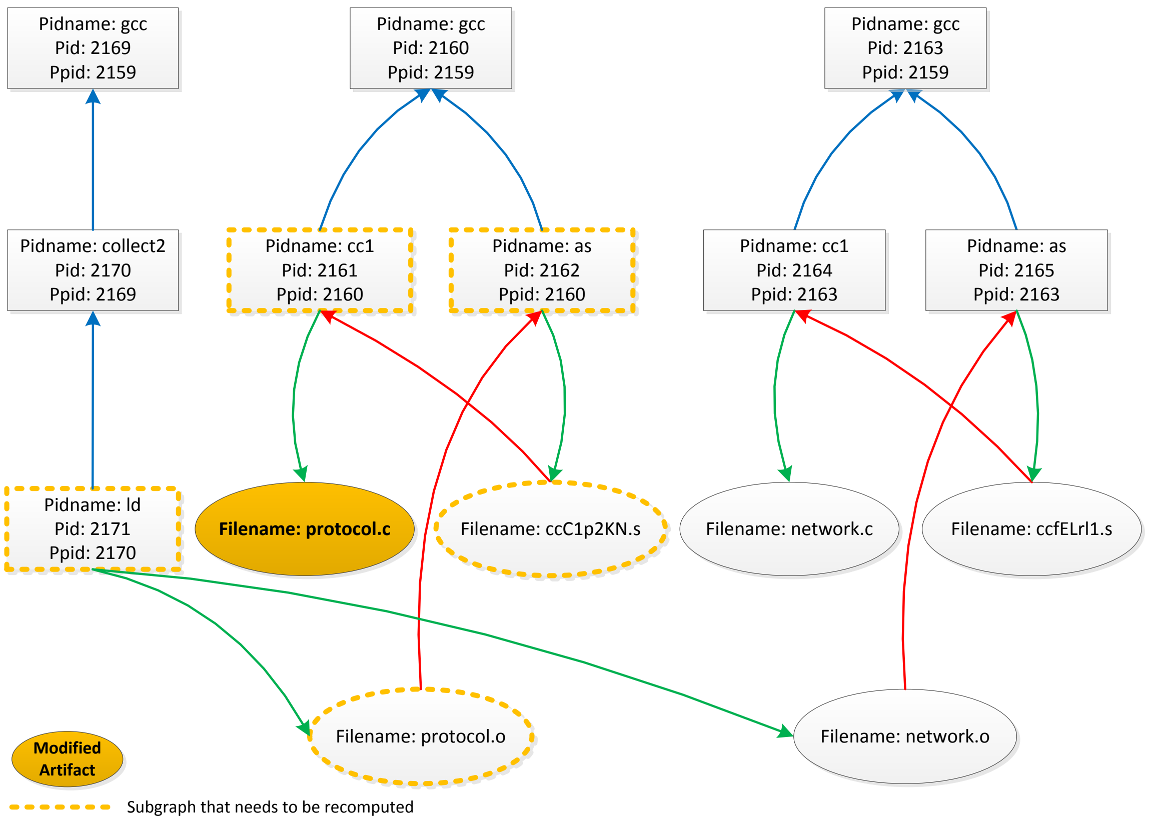

Fig. 2. When inputs change, provenance metadata is used to determine the minimal re-computation needed.

of data artifacts that were changed between time t_1 and t_2, necessitating the need for re-executing their dependents. The situation is similar for Task 2 where the set S is essentially the input i_1. Once such vertices are identified, forward flow data analysis, a graph traversal and coloring algorithm, helps us mark all the descendant processes that would need to be re-executed and all the artifacts that would need to be created again.

Algorithm 2 introduces the liveness property for provenance metadata. Liveness refers to whether the provenance metadata about an input to a process is synchronized with the current state of that data in the filesystem. A file is considered live if its last modified time in the provenance metadata is the same as its last modified time in the filesystem. Since provenance is a record of the history of processing, some of the inputs to processes that were recorded in the past may have been overwritten, modified, or deleted during subsequent processing. The longer such processing goes on, the more the state of the system and the provenance metadata are likely to become desynchronized.

The correctness property for re-execution and rollback requires that those inputs should be identified and regenerated by backtracking through the provenance graph and re-executing their parent vertices. The algorithm iteratively pushes the re-execution frontier back as it keeps adding more vertices to the colored set C. The provenance metadata of files is compared to the filesystem metadata to determine if an input has been modified since it was referenced in the provenance graph. If the data is not synchronized with the provenance, the parent vertices for such inputs are added to the colored set and the algorithm continues.

The algorithm eventually comes to a halt when either the whole provenance graph has been colored or a minimal set of vertices has been identified that need to be re-executed to perform the task. The algorithm guarantees that all the inputs to these tasks are present in their original order in the filesystem, thus ensuring the correctness of the output. Once all such vertices have been identified, a topologically sorted list of all the process vertices is used as a script to effect fast re-execution.

Rollback

While useful processing is preserved in the case of Tasks 1 and 2 involving re-executions, the system maintains no such state in the case of rollbacks where there is no way of undoing the changes made by processes and files in a system. As an addition to our scheme, we explored the use of a versioning filesystem with continuous snapshot ability to provide an efficient rollback capability. The resulting scheme is similar to the original one described. However, since the versioning filesystem allows us the ability to access older versions of files, we do not need to regenerate inputs that are

Algorithm 2 Backward flow

```
Require: Provenance graph G
         Set C ⇐ Algorithm 1
Ensure:
  Set C (of all vertices that need to be re-computed)
  for i ∈ C do
    Set I ⇐ $null$
    $Put$(I) ⇐ $Predecessors$(i)
    while I ≠ $empty$ do
      x = $Dequeue$(I)
      if x ∉ Live then
        $Put$(C) ⇐ $Predecessors$(x)
      end if
    end while
  end for
```

no longer live at the start of Algorithm 2. Our scheme simply gives us the ability to restore the whole system to any state by retrieving older versions of all the files from the filesystem. If we now need to rollback a number of process vertices, we can simply identify the states of all the input modified since the target time and replace them with older versions of the files. Re-execution also becomes simpler since we no longer need to backtrack through the provenance graph to regenerate the inputs. However, this has a higher storage cost, associated with keeping older versions of the files.

IV. Evaluation

The purpose of our experiments was to assess the performance of our scheme under different sample workloads. We picked an Apache Web server compilation as the first workload used to validate our approach. Apache compilation was chosen as an example of a large-enough multistep process that could benefit from keeping soft-state and preserving useful work across several invocations. BLAST [1], our second sample, is a widely used bioinformatics program that involves significant integer computations, memory operations, and disk activity. We used a run of PostMark as the third sample workload for our system. PostMark [13] is a filesystem benchmark designed to replicate small file workloads, typical of email and other Web-based services. Similar to a filesystem hosting Web-based services, PostMark puts both file and metadata operations under heavy stress by creating a configurable number of random-sized files and performing a sequence of transactions on them.

All our experiments were performed on a single machine with a 2.8 GHz Intel Core2Duo processor with 3.5 GB memory. The performance of three approaches was compared. In all three approaches, we synthetically created a fault where we needed to re-execute starting from a particular point in time during the workflow. In the first approach, we used a naive re-execution scheme that involved restarting the computations from the beginning. In the second case, our provenance-based approach was used with a traditional filesystem. In the third approach, optimized re-execution was performed using the

TABLE I
Performance analysis

Apache	Operations	Improvement
Complete re-execution	63564	
Provenance-based re-execution	15701	75.3%
Snapshotting filesystem + Provenance-based re-execution	13595	78.61%
BLAST		
Complete re-execution	48602	
Provenance-based re-execution	9811	79.8%
Snapshotting filesystem + Provenance-based re-execution	8391	82.73%
PostMark		
Complete re-execution	57344	
Provenance-based re-execution	14305	75.05%
Snapshotting filesystem + Provenance-based re-execution	10031	82.5%

TABLE II
Storage overhead for provenance metadata

	Apache	BLAST	PostMark
Provenance-based re-execution	13 MB	8.7 MB	8.9 MB

NILFS [15] versioning filesystem. The resulting outputs were compared across all three approaches to verify the correctness of the system. The number of operations involved in each of the three approaches for a representative run is summarized in Table I.

The complete re-execution approach represents the worst-case scenario where the whole series of operations must be performed again and nothing is preserved from previous runs. The provenance subsystem recorded all the operations in each of the other approaches. The resulting size of the provenance graphs with over 50,000 vertices for each of our workloads clearly shows the inability to manually investigate the dependency graph to determine the optimal re-execution or rollback strategy. This necessitates the need for an automated solution to the problem.

Depending on the location of the fault, the SPADE-based re-execution strategy reduces the total work substantially by preserving the useful work done at an extremely fine granularity through the use of provenance graphs. In our synthetic re-execution scenario, there was a 75% improvement in the performance as compared to a naive restart approach, as measured by the total number of operations needed to produce correct output. While this result is dependent on the location of the synthetic fault, even if we were to redo the whole workflow from the very start, the use of our liveness and coloring algorithms would ensure the absolute minimal set of operations needed to ensure correctness of outputs. The use of a continuous snapshotting filesystem improves the efficiency of the system even further to 78% by making sure that inputs that are modified need not be regenerated since older versions can be recovered.

The results of our scheme were consistent across the various different workloads. The use of NILFS does come at a price, as significant storage overhead is incurred in maintaining a versioning filesystem. Benchmarking a standard installation of NILFS using PostMark with 1,000 random files between 10K and 10M in size with 1,000 operations (reads and writes) shows that NILFS performs at about two-thirds the throughput on an *ext4* partition. The workload also generated about 8 GB of overhead from older snapshots of the files. While an NILFS-based implementation would make re-executions and rollbacks faster, longer workloads and data pipelines would incur a large storage overhead.

V. Related Work

Workflow management systems such as Karma [23] collect workflow, process, and data provenance from scientific workflows in a service-oriented architecture. Workflows can be visualized as orchestrations of services passing data to other services. The workflow engine, services, and applications can generate provenance notifications in a distributed manner. A Karma Web service subscribes to and is responsible for storing the notifications in a central relational database. A Web service API and a visualization tool can be used to query workflow, process, and data provenance.

Systems such as the Chimera virtual data system [6] manage the provenance of processing and managing large distributed data sets in Grids. A Chimera virtual data catalog is used to store information about data objects, transformation types, application programs, and derivations (that include parameters, execution time, and program names, for example). A virtual data language interpreter translates user input into data definitions and query operations. The system can provide a graph of a series of programs or tasks that need to be performed to recreate a particular piece of data. A virtual data application can potentially combine information from Chimera as well as other Grid components to automate resource scheduling, synchronization, recreation of data, and data movement.

PASS [20], a provenance-aware storage system, is a high-performance provenance mechanism for local filesystems that automatically collects and maintains provenance at the operating system level. Follow-up work has focused on storing provenance in a highly available environment, such as a cloud [22]. The approach relies on the Amazon EC2 cloud using services such as S3, SimpleDB, and SQS. The authors also demonstrate the functionality of a distributed PASS by creating PA-NFS [21], an enhanced version of NFS capable of recording provenance information. Evaluations show little network overhead for maintaining and storing provenance.

The Time Aware Provenance (TAP) system [25] captures the distribution and causality of state updates. In addition to storing dependencies between existing tuples, TAP remembers dependencies that were present in the past at some point, allowing the system to be consistent and provide correct answers even while receiving new updates. The system also introduces several efficiency optimizations to answer distributed queries.

PERM [9] supports generating, querying, and storing provenance information in the context of relational databases. It computes the provenance of a query by applying a set of rewrite rules to augment the original query to annotate records or tuples with provenance information. The query rewriting mechanism is oblivious to the origins of the provenance metadata, which can potentially be added manually or through another provenance management system. Implemented as an extension of the Postgres database, the query rewriting operates on the internal tree representation of the query tree. Standard query optimization techniques implemented in Postgres can be applied to the execution of the transformed query. Provenance information can either be computed at query time or stored persistently for future use.

Annotation-based systems, such as DBNotes [4], also support tracking the lineage of annotated data through a relational database system. Each attribute value in a database can be annotated with a text notation and the lineage and attributes are propagated through an extension of the SQL language framework called pSQL. The annotations can provide an overall estimate on the quality of the database as well.

VI. Future Directions

Having successfully implemented our scheme on a single host, we are working towards extending our scheme to a distributed execution environment. Creating, aggregating, indexing, and querying provenance information in distributed environments adds several new dimensions to the problem. Remote job failures, network variations, network latency, and server outages are issues that need to be addressed.

The problem is also complicated by the fact that extant distributed filesystems trade consistency and availability. Ensuring that the output of a program that was run on one host is consistent with a copy used as an input on another host requires the distributed copies to be kept consistent. The weaker the consistency guarantee, the greater the availability of the data. However, the output of a chain of computation will be more likely to use stale data and produce an incorrect result. The stronger the consistency, the greater the assurance that the output will be correct. The tradeoff is that data is less available, which means that computations will take longer to complete.

Identifying objects uniquely across different administrative boundaries is difficult, though some of the issues related to global naming, indexing, and querying have been discussed in the context of provenance for sensor data storage [16], which is analogous to provenance in distributed settings.

Data integration also becomes important if provenance metadata is being generated across several heterogeneous entities in a distributed environment. Integration and data exchange is typically done through a set of rewrite queries that map one schema to another. Orchestra [10] is one such collaborative data sharing system that uses schema mappings, designed by the receivers, to share data across a diverse set of sources. Queries are performed on local copies of data that are updated periodically. The administrator at a peer publishes the local edit log and in the process invites updates from peers who have published their logs since the last update operation. Peers also specify local trust policies based on provenance information.

VII. Conclusions

Current execution environments do not provide fast re-execution and rollback functionality when no predefined workflows exist. In this paper, we presented an architecture that provides the functionality to produce both correct output and fast re-execution by leveraging the provenance of data. It propagates changes along implicit dependency graphs produced by a provenance subsystem. Our benchmark results show that our scheme is a feasible solution to the problem of effecting such tasks. The use of a ubiquitous provenance subsystem obviates the need to think in terms of workflow entities and design.

Despite the success of the scheme, there is an inherent tradeoff between availability of data and storage capacity. We explored this tradeoff using a versioning filesystem to reduce re-execution in exchange for higher storage overhead. The overhead of keeping more data becomes significant as the number of operations increases. Our scheme was able to preserve useful data transformations and processing across several different runs of representative workloads and helped minimize redundant computations. The use of provenance metadata in determining this minimal set of steps also ensured that correct output was produced and the system did not enter an inconsistent state.

VIII. Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant IIS-1116414. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Stephen Altschul, Thomas Madden, Alejandro Schaffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David Lipman, Gapped BLAST and PSI-BLAST: A new generation of protein database search programs, *Nucleic Acids Research*, Vol. 25(17), 1997.

[2] Stephen Burley, Andrzej Joachimiak, Gaetano Montelione, and Ian Wilson, Contributions to the NIH-NIGMS Protein Structure Initiative from the PSI production centers, *Structure*, Vol. 16, 2008.

[3] Scott Callaghan, Philip Maechling, Ewa Deelman, Karan Vahi, Gaurang Mehta, Gideon Juve, Kevin Milner, Robert Graves, Edward Field, David Okaya, Dan Gunter, Keith Beattie, and Thomas Jordan, Reducing time-to-solution using distributed high-throughput mega-workflows - Experiences from SCEC CyberShake, *4th IEEE International Conference on e-Science*, 2008.

[4] Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya, DBNotes: A Post-it system for relational databases based on provenance, *ACM SIGMOD International Conference on Management of Data*, 2005.

[5] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei-hui Su, Karan Vahi, and Miron Livny, Pegasus: Mapping scientific workflows onto the Grid, *Grid Computing*, 2004.

[6] Ian Foster, Jens Vockler, Michael Wilde, and Yong Zhao, Chimera: A virtual data system for representing, querying, and automating data derivation, *Scientific and Statistical Database Management Conference*, 2002.

[7] Ashish Gehani, Minyoung Kim, and Tanu Malik, Efficient querying of distributed provenance stores, *8th ACM Workshop on the Challenges of Large Applications in Distributed Environments*, 2010.

[8] Ashish Gehani and Dawood Tariq, SPADE: Support for provenance auditing in distributed environments, *13th ACM/IFIP/USENIX International Conference on Middleware*, 2012.

[9] Boris Glavic and Gustavo Alonso, Perm: Processing provenance and data on the same data model through query rewriting, *25th International Conference on Data Engineering*, 2009.

[10] Todd Green, Gregory Karvounarakis, Zach Ives, and Val Tannen, Provenance in ORCHESTRA, *IEEE Data Engineering Bulletin*, Vol. 33(3), 2010.

[11] Israel Hernandez and Murray Cole, Reliable DAG scheduling on Grids with rewinding and migration, *1st International Conference on Networks for Grid applications*, 2007.

[12] David Hollingsworth, The Workflow Reference Model (v1.1), *Workflow Management Coalition*, Document TC00-1003, 1995.

[13] Jeffrey Katcher, PostMark: A new file system benchmark, Technical Report TR3022, Network Appliance, 1997.

[14] Sven Kohler, Sean Riddle, Daniel Zinn, Timothy McPhillips, and Bertram Ludaescher, Improving workflow fault tolerance through provenance-based recovery, *23rd International Conference on Scientific and Statistical Database Management*, 2011.

[15] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai,The Linux implementation of a log-structured file system, *ACM SIGOPS Operating System Review*, Vol. 40(3), 2006.

[16] Jonathan Ledlie, Chaki Ng, David Holland, Kiran-Kumar Muniswamy-Reddy, Uri Braun and Margo Seltzer, Provenance-aware sensor data storage, *1st IEEE International Workshop on Networking Meets Databases*, 2005.

[17] Paolo Missier, Stian Soiland-Reyes, Stuart Owen, Wei Tan, Alexandra Nenadic, Ian Dunlop, Alan Williams, Tom Oinn, and Carole Goble, Taverna, reloaded, *22nd International Conference on Scientific and Statistical Database Management*, 2010.

[18] Luc Moreau, Ben Clifford, Juliana Freire, Yolanda Gil, Paul Groth, Joe Futrelle, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, Yogesh Simmhan, Eric Stephan, and Jan Van den Bussche, The Open Provenance Model core specification (v1.1), *Future Generation Computer Systems*, 2010.

[19] Norbert Podhorszki, Bertram Ludaescher, and Scott Klasky, Workflow automation for processing plasma fusion simulation data, *2nd Workshop on Workflows in Support of Large-Scale Science*, 2007.

[20] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer, Provenance-aware storage systems, *USENIX Annual Technical Conference*, 2006.

[21] Kiran-Kumar Muniswamy-Reddy, Uri Braun, David Holland, Peter Macko, Diana Maclean, Daniel Margo, Margo Seltzer, and Robin Smogor, Layering in provenance systems, *USENIX Annual Technical Conference*, 2009.

[22] Kiran-Kumar Muniswamy-Reddy, Peter Macko, and Margo Seltzer, Provenance for the Cloud, *8th USENIX Conference on File and Storage Technologies*, 2010.

[23] Yogesh Simmhan, Beth Plale, Dennis Gannon, and Suresh Marru, Performance evaluation of the Karma provenance framework for scientific workflows, *1st International Provenance and Annotation Workshop*, 2006.

[24] Jia Yu and Rajkumar Buyya, A taxonomy of scientific workflow systems for Grid computing, *Journal of Grid Computing*, 2005.

[25] Wenchao Zhou, Ling Ding, Andreas Haeberlen, Zachary Ives, and Boon Thau Loo, TAP: Time-aware provenance for distributed systems, *3rd USENIX Workshop on the Theory and Practice of Provenance*, 2011.