Lightweight Provenance Service for High-Performance Computing

Dong Dai[1], Yong Chen[1], Philip Carns[2], John Jenkins[2], and Robert Ross[2]

[1]Computer Science Department, Texas Tech University {dong.dai, yong.chen}@ttu.edu
[2]Mathematics and Computer Science Division, Argonne National Laboratory {carns, jenkins, rross}@mcs.anl.gov

***Abstract*—Provenance describes detailed information about the history of a piece of data, containing the relationships among elements such as users, processes, jobs, and workflows that contribute to the existence of data. Provenance is key to supporting many data management functionalities that are increasingly important in operations such as identifying data sources, parameters, or assumptions behind a given result; auditing data usage; or understanding details about how inputs are transformed into outputs. Despite its importance, however, provenance support is largely underdeveloped in highly parallel architectures and systems. One major challenge is the demanding requirements of providing provenance service in situ. The need to remain lightweight and to be always on often conflicts with the need to be transparent and offer an accurate catalog of details regarding the applications and systems. To tackle this challenge, we introduce a *lightweight provenance service*, called *LPS*, for high-performance computing (HPC) systems. *LPS* leverages a *kernel instrument mechanism* to achieve transparency and introduces *representative execution* and *flexible granularity* to capture comprehensive provenance with controllable overhead. Extensive evaluations and use cases have confirmed its efficiency and usability. We believe that *LPS* can be integrated into current and future HPC systems to support a variety of data management needs.**

I. Introduction

Provenance, also commonly referred to as lineage, is metadata that describes the history of a piece of data [12]. Along with its graph representation, provenance indicates the relationships among all elements—data sources, processing steps, contextual information, and dependencies—that contribute to the existence of a piece of data [11, 20, 30]. An extensive number of data management functionalities rely on provenance [18, 37, 38]. Its importance has been well acknowledged, and various provenance systems have been investigated [9, 15, 33, 32, 35]. Nevertheless, a generic provenance service for highly parallel environments, such as high-performance computing (HPC) platforms, is still missing. The reason comes primarily from the challenging requirements of providing provenance service in these performance-critical environments.

First, as an always-on service, the provenance tracing and collection must be lightweight. It should achieve a low overhead (typically less than 1% slowdown of applications and less than a 1 MB memory footprint per core, according to discussions with domain scientists working on HPC clusters [7]). Most existing provenance systems, except those such as Zoom [9] and VisTrails [35] that work with specific workflow systems, experience significant overhead. For example, PASSv2 has up to 23% overhead in representative applications [33], and SPADEv2 reports 10% overhead for Apache production runs [22]. Such overhead is not acceptable for HPC platforms that run highly concurrent applications.

Second, because of the complexity of HPC platforms, the provenance tracing and collection must cover provenance generated from different physical locations (e.g., login nodes and compute nodes), about multiple execution abstractions (e.g., scientific workflows, multinode jobs, local processes), on both local and distributed data files. Many provenance systems, including PASSv2 [33], ES3 [21], TREC [39], and many others [8, 9, 25, 27, 35], are not designed for or are not capable of fulfilling these needs in HPC environments.

Third, as a default service in HPC platforms, provenance tracing is expected to be transparent and automatic. In other words, users should not need to modify their codes, link to nonstandard libraries, or set up environmental variables in order to enable provenance support. More aggressively, not only should the users not need to manually enable the provenance service, they even may not have the privilege to disable the provenance service if the collected provenance will be used for mission-critical tasks such as data auditing. Many cross-layer provenance systems normally require extensive interventions from users, limiting their participation willingness, especially for domain scientists [32, 28].

To tackle these challenges, we have designed and implemented a lightweight provenance service, called *LPS*, for HPC platforms. *LPS* combines a multitude of techniques. First, we enable transparent and automatic systemwide provenance tracing by instrumenting kernel-level events and efficiently aggregating them across the system. Second, we support flexible provenance granularities to strike a balance between comprehensive provenance and low overhead. Third, we introduce the concept of representative executions to compress the superfluous kernel events and further reduce the overheads of *LPS*. Our evaluation with various benchmark workloads and real-world scenarios shows that *LPS* introduces negligible overheads in general (less than 1% performance with coarse-grained granularity) with controllable memory consumption. The main contributions of this work are threefold:

- Design and implementation of a framework by instrumenting operating system kernel and leveraging environmental variables to collect systemwide provenance transparently.

- Introduction of a new methodology—unified provenance granularity—to achieve balance between overhead and provenance accuracy. Multiple granularities can be dynamically switched at runtime.
- Extensive evaluations to validate the *LPS* methodology and to verify its efficiency and usability.

The rest of this paper is organized as follows. Section II discusses the background and motivation of *LPS*. Section III analyzes the provenance granularity in HPC system and introduces the unified model to support multiple granularities in *LPS*. Section IV discusses the design and implementation details of *LPS*. Section V reports the evaluation results and analysis. Section VI summarizes our conclusions and briefly discusses future work.

II. Background and Motivation

We motivate this research based on the data management needs from real-world scenarios and their requirements for provenance information.

A. LPS *Provenance Graph Model*

Provenance can be modeled with the Open Provenance Model (OPM), which is a directed graph with three types of nodes: the data `Artifact`, indicating an immutable piece of state; the executing `Process`, indicating action or series of actions performed on or caused by the artifacts; and the controlling `Agent`, which is a contextual entity acting as a catalyst of a process. There are directed edges between different nodes representing the causal *dependencies* between the source and the destination [30].

OPM is generic, describing various types of provenance; but it is not particularly straightforward for HPC systems and users. In *LPS*, we follow the OPM concept but extend it to be more specific for HPC environments.

1) We define `User` and `User Group` node types, which belong to `Agent` as defined in OPM. We define dependencies `belongsTo` and `has` to connect these two node types.
2) We define `Workflow`, `Job`, and `Process` node types, which are executing `Process` as defined in OPM. The dependencies `belongsTo` and `has` connect nodes like `Workflow` and its `Jobs`; `parent` and `child` connect multiple `Process` nodes.
3) We define a `File` node type that belongs to `Artifact`. Since `Artifact` is immutable but files are always changing, each `File` node should represent a specific version (or snapshot) of the file.
4) We define more dependencies to represent meaningful relationships between node types; for example, `run/runBy` connect `User` and `Process` node, and `read/readBy` and `write/writeBy` connect `Process` and `File`.

In Figure 1, we show an example of provenance built by *LPS*. In this example, one user (*Joe*) from a user group (*Prj-1*) started multiple executions that read/write files. Because of the space limit, we do not label all the edges. Instead, different line patterns are used to denote different dependencies. File `/scratch/joe/ior.conf` was changed by *vi* and later read by the *mpi-ior job*; hence, two nodes are shown in the graph to denote two versions of the same file. We expect that more details about the interactions between processes and files will lead to more versions of the same file.

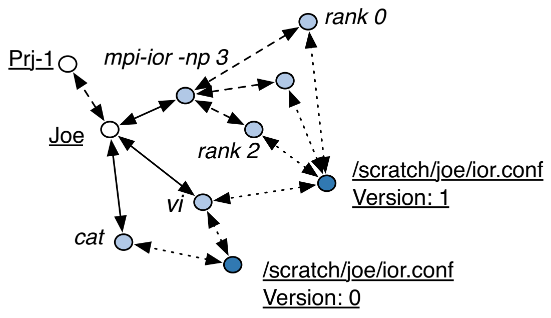

Figure 1: *LPS* provenance graph example.

B. Motivating Use Cases

With such a provenance graph built automatically in HPC environments, multiple useful data management tasks will be possible. For example, the graph can help validate computation results by tracing how a data file was generated and how relevant applications were executed, thus allowing us to rebuild a controlled environment (especially in an HPC environment) to repeat computation and validate the results. Another example is auditing the damages of shared datasets, which can happen from various sources such as intentional instrumentation, misconfigured applications, or just a buggy library. The provenance graph can trace activities on files and hence help identify both the source of the damages and the descendant files affected by the damaged files.

We demonstrate the usefulness of *LPS* with another commonly seen use case: analysis of computation results. Recently, we attempted to measure the I/O performance of a parallel file system by running a series of IOR benchmarks [36]. Because of system variability, the results were slightly different each time. Hence, many repetitions of the same test were conducted in order to generate multiple result files and `grep/awk` them to build a single data file for plotting. However, unexpected variations often occur, and one must identify the root cause of these variations. To do so, all IOR invocations need to be checked in order to confirm whether the same environment (environmental variables, configuration files, command lines, etc.) were used and whether they were executed in a time frame when the system was stable. A provenance graph similar to Figure 1 clearly helps such a task.

Although this particular example is system oriented, its pattern is common in the data-driven scientific discovery era. Scientists usually spend a considerable amount of time analyzing the massive amount of results generated from repeated runs of their simulations with different parameters, inputs, and algorithms. A tool such as *LPS*, which can automatically record how the results were generated and how applications were executed and will involve only a low overhead, will be highly valuable.

III. LPS Provenance Granularity

In Section I we discussed several critical requirements of a provenance service on highly concurrent environments. A significant challenge in meeting these requirements is the conflict between the need for systemwide, comprehensive provenance and low overhead in HPC environments. In *LPS*, we tackle this challenge by enabling the system to work under different granularities seamlessly.

A. Provenance Granularity

Granularity choice is involved in many aspects of provenance. For example, regarding the executions, one can capture provenance for workflows, jobs, processes, or threads; regarding the data, one can capture activities on the whole file or just a piece of the file [9, 35, 33, 22, 21]. In *LPS*, we fix the primitive of executions as the *local process* and the primitive of data as the *whole file*. Although one can always capture fine-grained provenance about entities such as threads or pieces of files, the scale of HPC applications and their complex I/O behaviors make this action impractical.

Even if the granularity of executions and data files is fixed, their interactions (file-accessing operations) still have multiple granularity choices. Figure 2 shows interactions between executions and files. There are three granularities on file accesses based on those interactions. The open/close granularity captures only the file open and close events and ignores all reads/writes between them. It may be inaccurate as users may not read/write a file immediately after opening it or close the file timely after accessing it. Also, it does not capture information like read/write size. The first/last granularity records the first and the last read or write accesses and pairs them, respectively. Its time range is always smaller than that of the open/close granularity and hence more accurate. But it does need to trace every read and write operation in order to determine the first and last one. The most accurate but also expensive granularity is read/write, which records each file access (both its start time and finish time) to distinguish concurrent accesses. The granularity of file accesses is important for answering queries relevant to data dependencies, which significantly affect the comprehensiveness of provenance. At the same time, the cost of capturing file accesses can take a major part of provenance tracking overheads because of the data-intensive nature of HPC applications and hence is critical to optimize.

Provenance systems normally choose one granularity based on their overhead requirements and stay with that one all the time [9, 35, 33, 22, 21]. This approach is problematic, however, since the chosen granularity could introduce high overhead in certain workloads or may not be accurate enough to support many use cases. For example, PASSv2 traces every read/write system call to obtain the read/write granularity. For

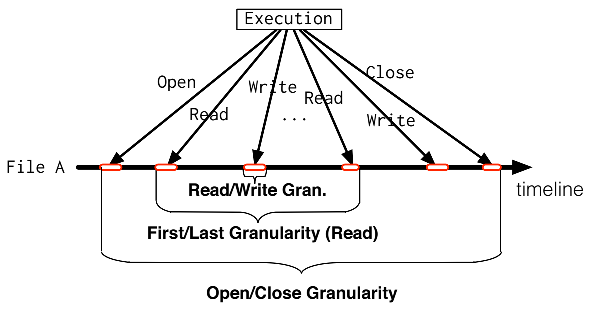

Figure 2: Typical interactions between execution and file.

certain workloads that conduct a large number of small I/O operations, however, it will experience a significant overhead (over 23%) even on a single server [33]. This is clearly not acceptable in HPC environment. On the other hand, some use cases, such as validating scientific results, prefer more accurate file access information, for example, whether a read from an application is “before” or “after” a write from other applications. If only open/close relationships are captured, an execution (E_2) that read before the FirstWrite but after the open of another execution (E_1) will be considered as dependent of E_1, which is not accurate. Although this inaccuracy is not catastrophic, since all collected provenance inevitably contain certain inaccuracy [12], it does introduce more false-positive data dependency, which may require the users’ domain expertise to judge or otherwise lead to more recomputations in use cases like validating the results. Hence, if the tracing overhead is acceptable, such inaccuracy should be minimized. It is worth noting that the inaccuracy is not equal to error of provenance. The false-positive dependency never misses real data dependency, which can lead to real problem and users can never correct it.

B. Unified Model for Multiple Granularities

In *LPS*, we support different granularities and allow the provenance granularity to be changed by users at runtime. Such changes are normally triggered by the overhead changes from running different workloads. As different granularities capture different file access operations, the provenance system needs to process them seamlessly.

To meet this need, we characterize different granularities into one unified representation. Specifically, in *LPS*, we consider that the file accesses operate on a time range of an immutable status of `File`, which represents a snapshot of the file and can be mapped to a node (with version) in the provenance graph. The immutable status of the file can be defined by a start event and an end event, determining the granularity. For example, the open/close granularity is defined as $T[e_{open},\ e_{close}]$, the first/last granularity is represented as $T[e_{FirstAccess},\ e_{LastAccess}]$, and the read/write granularity is defined as $T[e_{ReadStart},\ e_{ReadEnd}]$. Unlike other provenance systems that pair certain start and end events to form a fixed granularity (e.g., open/close), in *LPS* we allow any two events to be paired to form provenance. For example, an *Open* event can be paired with *LastAccess* instead of *Close*; similarly, *FirstAccess* can pair with *Close*, too. There are also illegal pairs. For example, we do not consider *Open* and *FirstAccess* as a meaningful granularity simply because there is no file access between these two events. The same reason is applicable to the case of events *LastAccess* and *Close*.

Using this model, *LPS* can smoothly process events of different granularities generated from different users and their allocated portion of compute nodes. It can also process the events if the granularity is forced to change while applications are running. For example, assume that the first/last granularity is used at the beginning. Later, if the overhead is intolerable, one can switch to an open/close granularity for lower overhead. Before switching, the system collects *FirstAccess* events and then collects *Close* events. With the unified model, *LPS* can pair them together meaningfully to $T[e_{FirstAccess},\ e_{Close}]$. If the first/last granularity is switched back later, one can still pair the remaining *Open* event with new *LastAccess* event. More details about the implementation are discussed in Section IV.

C. Discussion of Read/Write Granularity

Although read/write granularity captures the most accurate file access operations, its usage is actually very limited in HPC environments. The primary reason is that in such a highly parallel environment, even if we know the accurate data access order on a file through read/write granularity—for example, one execution (E_1) issued two writes to a file, and another execution (E_2) read the same file between those two writes—we still cannot tell the exact dependency because they may read/write different parts of the file. In fact, it is typical to have multiple executions issuing concurrent, independent I/O requests to different parts of a file in parallel applications [24]. Thus, the paid overheads for tracing each read/write operation and fusing them to form correct dependency may become insignificant. Moreover, the traced individual write/read is hard to be matched back to I/O operations that users can understand or repeat. This further diminishes the need for such fine-grained provenance. In many cases, it is enough to claim that E_2 depends on E_1 just by knowing that E_2 read after the first write of E_1. Such provenance is also helpful in many use cases, including determining that rerunning is needed in order to reproduce a given result. Given these reasons, we eliminate read/write granularity and keep only open/close and first/last as options for users in *LPS*.

IV. LPS Design and Implementation

Figure 3 shows the overall architecture of *LPS* and its integration with a typical HPC system. All compute nodes and login nodes in the HPC system run *Local LPS*, which takes charge of tracing and preprocessing local provenance-relevant events. In *LPS*, since we support multiple granularities, those tracers running on different compute nodes (allocated to different users) may be under different granularities, shown as different colors in Figure 3. All processed events are sent to a

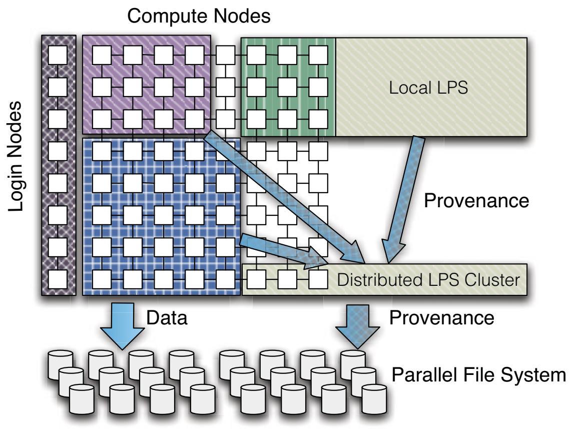

Figure 3: *LPS* overall architecture in HPC system.

distributed LPS cluster for further aggregation and storage. The size and running mode of the distributed cluster are configurable by administrators. *LPS* can run as a standalone cluster or utilize part of the existing compute nodes, as the figure illustrates.

For provenance storage, which is not the focus of this study, we deploy a distributed graph database specifically designed for storing a metadata graph such as provenance, following a previous study [17] and its open source release [16]. Specifically, we host a configurable number of RocksDB database instances to store the graph [3]. Each database instance stores part of the provenance graph as a single DB file, which is then persisted in the underlying parallel file system. The provenance graph is partitioned into these RocksDB database instances by using an incremental partitioning strategy, called DIDO [17].

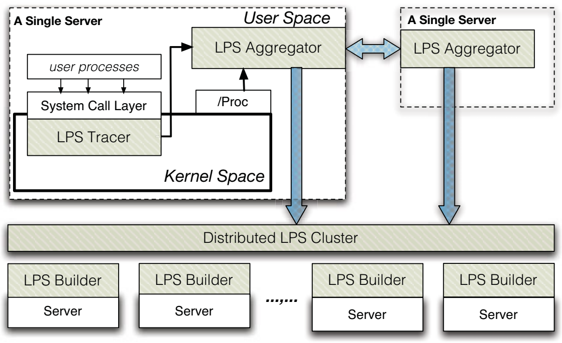

Figure 4: Detailed *LPS* architecture and components.

Figure 4 shows the key components of the *Local LPS* and the *Distributed LPS cluster*. Here, the *LPS Tracer* runs in the kernel space of each node to trace relevant events generated from users' activities. These events are the building blocks of provenance. The *LPS Aggregator* runs in user space of each node, receiving data from the tracer, caching them, pruning and aggregating them, and sending them to *LPS Builder* to build provenance. The *LPS Builder* runs in the user space of the remote *LPS* cluster. It takes charge of building the final provenance graph and storing them into backend graph databases.

A. LPS *Tracer*

Provenance covers a wide range of activities in the system. In general, three strategies, shown in Table I, are used to trace those activities.

TABLE I: Runtime activity tracing methods.

Method	Transparency	No Privilege	Dynamic
Provenance APIs	×	✓	×
Library Wrapper	×	✓	×
Kernel Instrumentation	✓	×	✓

The *Provenance APIs* method provides explicit APIs to users to trace interested activities [28]. It requires users to rewrite their applications and hence is not transparent. It also does not support dynamic changes on the provenance granularity at runtime. It normally does not need system privilege but is largely limited to the applications that are already integrated with those APIs.

The *Library Wrapper* method provides provenance-enabled libraries wrapping the original library to trace activities. It requires users to relink applications for static libraries or to set proper `LD_PRELOAD` for dynamic libraries. Hence, it is not transparent in general; however, system administrators can substitute the libraries systemwide to enable transparency, which in turn requires system privilege. Neither way supports dynamic changes on the provenance granularity. Darshan [5] is built by using this method.

The *Kernel Instrumentation* method tracks users' activities by hooking the operating system's system calls. It is transparent to users and maximizes the generality, since most applications activities can be identified through kernels. Although it requires system privilege to instrument the kernel, the risks have been proven controllable by many mature instrumentation tools (e.g., DTrace [23] and SystemTap [6]). This strategy has been widely used in provenance systems including PAASv2 [33], ES3 [21], and BURRITO [25]. Another advantage of kernel instrumentation is that it can change its instrumented system calls dynamically at runtime. This capabilitiy is necessary for *LPS* to support flexible provenance granularities. Hence, it is also the strategy that we use.

1) Systemtap Kernel Instrument.: LPS leverages kernel instrumentation to collect detailed runtime events to build provenance. Specifically, it uses Systemtap [6] to probe the kernel for three categories of activities.

- *Executions*. *LPS* probes the `execve`, `exit_group`, and `kprocess.create` kernel functions or system calls to collect information about processes creation, execution, and termination.
- *Files*. *LPS* probes `open/close/read/write` system calls to capture detailed information about file accesses. In addition, `mmap` and `pipe` are probed to get more complete file access activities.
- *Metadata*. *LPS* probes operations such as `rename`, `link`, `unlink`, and `delete` to maintain the mapping between provenance entities and files in the original file system.

For each event, a built-in function `timestamp()` of Systemtap is used to query a local time for an event automatically. The tracer also reads `/proc` to collect required runtime metadata such as environmental variables.

2) Dynamic Probe.: A major design feature in *LPS* is to allow the provenance granularity to be chosen or changed in each resource allocation (i.e., user-allocated compute nodes) as needed. Using Systemtap, this dynamical granularity changing turns to dynamically enable and disable `read/write` probes. Although SystemTap provides a *conditional probe* to allow users to define a condition to decide whether a probe should be activated or not [6], our experiments show that it introduces overheads even if the conditional probe is not enabled. Thus, instead of using conditional probes, we create two SystemTap scripts for the *LPS* tracer. The first one enables all default probe points, and the second one probes only `read/write` system calls. Both of them are active at runtime by default, but the second one can be deactivated by users. *LPS* sets a hook on a specific local file; and whenever users touch that file, the second script is executed. Similarly, if users touch another specific file, the second script is stopped. We also provide a tool to help users quickly touch all files across all nodes in their allocations.

Since the *LPS* tracer runs in kernel space, it increases overhead on kernel activities. In the *LPS* tracer implementation, we follow the principle of minimizing the time in kernel mode. The user-defined event-processing functions construct only needed metadata and send the data back to the user space *LPS Aggregator* for further processing. *LPS* also filters out useless events, for example, all processes that are part of the tracing service.

Through the tracer, we can capture detailed runtime events. However, this information is not enough since these events are still largely isolated. For example, users will not know whether a local operating-system-level process belongs to a certain distributed job because their runtime activities are collected at different locations: job submission activity is collected in login node, and process activities are collected in compute nodes. The *LPS builder* will fuse those isolated events to build the provenance.

B. LPS *Aggregator*

We utilize the `named pipeline` to connect the tracer and the aggregator on each node. The events collected in kernel space by the *LPS* tracer will be sent to such a pipeline waiting for processing. The *LPS* aggregator will continuously retrieve data from the same pipeline. The named pipeline is an in-memory data structure, and as such it is efficient for reading/writing. When remote builders are slow or the tracer generates too many events, the events will be buffered in the named pipeline temporarily. Doing so, however, increases its memory consumption. In most Linux deployments, the maximal size of named pipeline is limited (64 KB typically). In *LPS*, we manually set them to a larger value ($512 * Core$ KB). This number is chosen based on the limit of the memory footprint in *LPS*. Even with a larger pipeline size, however, *LPS* still may exhaust them if remote aggregators stall. Therefore, the aggregator continuously monitors the buffered events size. Once it uses over half the allocated memory, the aggregator starts to write collected events directly into local file system or remote parallel file system for backup. These events are processed after the remote *LPS* builder recovers.

1) Monitoring Overhead.: Each of the fired probes in SystemTap introduces a small overhead, and this overhead may itself be too high (over the 1% requirement). For example, if all reads/writes are probed and, at the same time, one application issues a large volume of small I/O requests (e.g., 1 million times of 1-byte writes), the probe overheads might become unacceptable. To avoid this situation, the *LPS* aggregator monitors the fired frequency of `read/write` events, with the help of the *LPS* tracer's SystemTap script. Specifically, in the second script that probes `read/write` system calls, we maintain a *counter* that increases every time a probe is fired. At the same time, a *timer* event fires every 1 ms to reset the counter to zero. After increasing the counter, if the tracer notices that the counter is too large, it notifies the aggregator, which then logs this event to notify users so that they can make a decision about switching granularity.

2) Pruning with Representative Executions.: The kernel instrumentation in the *LPS* tracer achieves transparency and flexibility but also elevates the noise by creating too many events that might not be interesting to users. In the *LPS* aggregator, we introduce *representative execution* to reduce the noise level and also help users better utilize the provenance.

Figure 5 shows an execution tree captured by the tracer after running an MPI program through `mpiexec -n 3 ./mpitest` in the local machine. In this figure, each node represents a process, labeled with an executable name and process Id. The directed edges indicate parent-to-child relationships. The MPI program is started by calling "mpiexec." The user's program (i.e., *mpitest*) is executed through "hydra_pmi_proxy," which is part of the MPICH runtime [24]. Some executions have `UNDEFINED` labels because they never call `execve`.

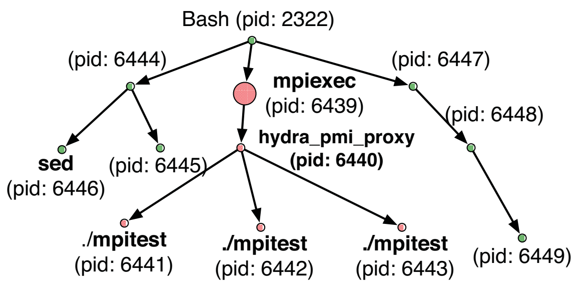

Figure 5: Example of a provenance graph (execution part).

In the example, of all the processes only a small part (i.e., nodes with red pattern) are of interest to users. The provenances—for example, who runs them, which files they accesses, their execution parameters, and environment variables—are important for understanding their executions and also for rerunning them. In *LPS*, we refer to them as the *representative executions*, which implies that they can represent the program without losing information about users' behaviors. Note that the metadata collected for nonrepresentative processes should be counted to its nearest parent representative execution.

The aggregator identifies representative executions for compressing the collected provenance. *LPS* relies on two observations in HPC platforms to deliver this information. First, in login nodes, users normally operate through *ssh* and *shell*. The child process of *shell* with the proper user command indicates the representative executions. Second, in compute nodes where parallel applications run, the processes are launched either through *ssh* similar to login nodes or through a specific runtime library such as MPI runtime, which means representative executions should be the child of the runtime process. Based on these two observations, *LPS* identifies representative executions from the real-time event streams generated by the tracer in an online manner.

Specifically, each time a process creation event arrives, *LPS* checks whether it is a child of a representative execution. If it is, this process is ignored, since its activities should be counted to its parent (i.e., the representative execution). If a process is not a child of any existing representative execution, then either it could be a representative execution or its parents have not been identified yet. In such a case, we need to buffer events about this process temporarily and wait for the determination. The `execve` system call is the place to determine a representative execution, which should have the shell name as the exec name or is a direct child of MPI runtime. The `exit` system call is the place to handle the processes that never call `execve`. They will have `UNDEFINED` as their *execname* when terminated. At that time, their parent should already be determined to be a representative execution or not, and buffered behaviors can be counted to its parent. In *LPS*, this on-line pruning procedure moves fast and typically does not need to buffer many events (as shown in Figure 12 in Section 5). In fact, most executions will first invoke `execve` and then create children processes. Thus, most new processes can be determined as children of a representative execution or not when created.

C. LPS *Builder*

The core task of the *LPS Builder* involves two parts. First, it fuses isolated execution events together; second, it builds correct data dependencies from data access events.

1) Fusing with Environmental Variables.: In general, connecting high-level execution abstractions such as workflows and jobs with processes requires a unique identifier shared across them. Normally, this needs knowledge from specific software. For example, how a job is scheduled into compute nodes is visible only in software such as a job scheduler (e.g., the Portable Batch System, or PBS, [10]). In *LPS*, to maximize the transparency, we utilize environment variables of processes to share such knowledge. Specifically, to map distributed jobs and local processes together, we require the job scheduler to expose semantic information through environmental variables to *LPS*. On the login node, the submitting process (e.g., "qsub") should create a `submitID` environment variable to denote the unique identifier of the job. Its value is assigned by the job scheduler. At the same time, in the compute nodes, each process should also be assigned to the same `submitID` environment variable to denote the job that current process belongs to. By matching these two Ids in the local *LPS* aggregator, we connect together the job and process metadata collected from different locations. In fact, existing HPC system software has partially done this. For example, in PBS, all processes running on compute nodes have an additional environment variable, `PBS_JOBID`, which is uniquely assigned by the scheduler to denote a new job. In the login node, the same Id is returned from the scheduler once the "qsub" command returns. Similar features can be found in COBALT [1] or Torque [4]. An I/O characterization tool such as Darshan also leverages these features to match jobs and their processes [13].

2) Building with Versioning.: To build correct data dependencies means to create necessary versions of a file and determine correct dependency between a process and a specific version of file. The basis of such a task is to sort I/O events on the same file in a chronological order and determine the version and dependency accordingly. Since *LPS* has multiple builders, to make sure that relevant events will always be assigned to the same builder, the *LPS* aggregator sends events on file (f_a) to the kth builder instance based on the hash value of the full file path ($k = hash(f_a)\%m$), where m equals the number of servers in the *LPS* cluster.

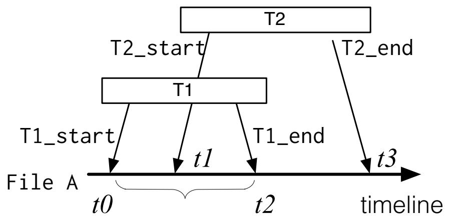

Figure 6: Open/close granularity example.

Since *LPS* supports multiple provenance granularities, we use the generic model to describe the algorithm. We assume that the file access granularity is $T[e_{start},\ e_{end}]$. Clearly, for m accesses to a file, there will be $T_1, T_2, ..., T_m$ received in the builder. If all of them are not overlapped regarding their time ranges, one can easily determine their dependencies: If T_i is write, it generates a new version of the file; if T_i is read, it depends on the previous write, that is, the newest version just created. If there are overlaps among T_is, however, the situation becomes more complicated. Figure 6 shows an example of this case, where T_2 starts at t_1, which is between the start (t_0) and end (t_2) of T_1. In this case, if T_2 is read and T_1 is write, then T_1 should create new version of file f_a, and T_2 should depend on the version that T_1 creates on t_2, instead of t_0. The reason is that T_2 may read data that was written near t_2. Similarly, if T_1 is read and T_2 is write, T_1 should depend on the version that T_2 creates at t_3. If T_1 and T_2 are both read or write, there is no dependency between them, although a new version of the file will still be created by writes. We can thus create a rule: For overlapped I/O, read always depend on the newest version that the overlapped writes create.

Implementing such rule requires efficiently detecting the overlapping and tracking the newest version created by overlapped writes. Note that the *LPS* builder does not directly receive file access activities as $T[e_{start},\ e_{end}]$. Instead, it receives individual events (e.g., e_{start} or e_{end}), which are mixed from different processes. The individual events can be matched together by comparing their unique source process Ids. Matched events can further be paired based on the unified granularity model introduced in Section III-B. For example, *OpenRead* event can be paired with *LastRead* or *CloseRead*. Similarly, *FirstWrite* can be paired with *LastWrite* or *CloseWrite*.

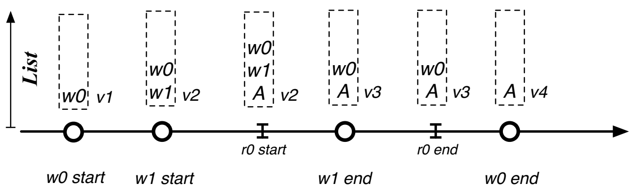

Figure 7: Building dependency with versions.

In the *LPS* implementation, on each builder we maintain a sorted list to buffer the access events for each file, as Figure 7 shows. Each list is sorted in chronological order, where the new event will be pushed into the beginning of the list. The version of the file increases whenever a write access arrives, regardless of whether it is the start event or the end event of a write. We label the latest version number on the right side of each list as new events arrive in Figure 7. The start event of a write is buffered in the sorted list (e.g., 'w_1 *start*'), waiting for its paired end event (e.g., 'w_1 *end*'), which will delete both of them from the list. The start event of a read (e.g., 'r_0 *start*') is also buffered in the sorted list, but it is not removed when its paired end event arrives. Instead, it only updates its order in the list based on the timestamps of the arrived end event. The read event can be removed only if it becomes the oldest element in the list, for example, when the "w_0 *end*" event arrives in Figure 7. In that case, all overlapped writes have finished, and the read (r_0) will depend on the last version (i.e., v_4) of the file.

3) Handling Clock Skew.: Note that, aggregating distributed events needs to consider the uncertainty from the imperfect clock synchronization in a distributed environment. Events collected from different nodes may vary a small amount of time. This might lead to incorrect order of file accesses from different servers [26]. To handle clock skew, we define a configurable constant as the maximal clock skew (i.e., c) for the whole HPC cluster. Typically, c can be as small as milliseconds in a production HPC systems through applying a clock synchronization protocols like NTP [29] or PTP [14]. Considering the clock skew, an event happening at t will not be considered as accurate anymore; instead, it will be turned into a time range as `[t-c, t+c]`. Note that, this uncertainty is only applied on distributed events, like data accesses on distributed files; not on the events only visible inside a single node. In Figure 2, we show the clock skew as the the red rectangles on the timeline. Considering the clock skew, when we pair two events to form a granularity unit, it should become $T[e_{start} - c, e_{end} + c]$ respectively.

V. Performance Evaluation and Analysis

We implement a prototype of *LPS* using MPI and SystemTap. To evaluate *LPS*, we build a typical HPC system using CloudLab APT cluster as the major evaluation platform. It provides bare-metal machines with root privilege to run SystemTap script [34]. Regarding the hardware, CloudLab has 128 nodes, and we allocated up to 45 nodes for this evaluation. To build a typical HPC environment, among these nodes we installed PVFS parallel file system on 5 dedicated nodes as a shared storage. Up to 32 nodes (256 cores) were allocated as compute nodes. The remaining nodes were reserved to construct the distributed *LPS* cluster. In all evaluations, if not explicated stated, we only used one of them for aggregating and storing the provenance. Each node has an 8-core Xeon processor, 16 GB RAM, and 2 TB local hard disk drives. All nodes are connected through a 10G Mellanox MX354A dual-port switch.

A. LPS *Tracer Performance and Analysis*

The *LPS* tracer introduces overhead onto an instrumented system call. Because it runs on every node, especially the compute nodes, this overhead is critical. In this series of evaluations, we carefully measured the overhead introduced by *LPS* tracer under different workloads. Specifically, we ran two benchmarks. First, we executed the IOR benchmark on a local `RamDisk` with different *blockSize* parameters (varying from 1K bytes to 64K bytes) [36]. By writing into RamDisk, we bypassed the disk I/O in order to minimize the possible variations. Second, we ran an `DD` command, `dd if=/dev/zero of=/dev/null bs=1 count=1M`, to further minimize the data copies (1 byte) in order to accurately measure the overhead of a system call itself due to instrumentation.

In Figure 8 we report the IOR results. We observe that for all workloads, the open/close granularity (only activate the first script) incurred negligible overhead. The reason is that during the IOR run, it opens/closes the test files several times (based on the *repetition* parameters). On the other hand, the first/last granularity did introduce noticeable overhead for some workloads, especially those having a large amount of small writes issued. If this overhead is unacceptable, a different granularity should be chosen. This result clearly shows the necessarily of supporting multiple granularities for different workloads.

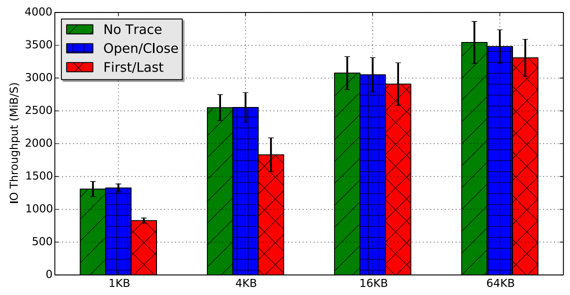

Figure 8: *LPS* local collector performance on IOR benchmark.

To accurately measure the overhead introduced by probing every read/write in the *LPS* tracer, we ran the `DD` benchmark. It issues 2 million I/O requests (reads and writes) in total, each of which contains 1 byte to minimize the data copies. In this way, we can measure the time cost of each `read/write` system call before and after probing. We show the results in Figure 9. In this figure, the y-axis is the time cost of each `read/write` system call in nanoseconds. We show three cases for comparison. First, with no probe at all, the average response time for a read or write was around 100 ns. Second, if we probe only the read/write system calls but do not collect any events, it introduced around a 300 ns overhead. This is the overhead coming from SystemTap itself. Third, if we collect all *LPS* events, another 100 ns overhead was introduced. Note that this is an extreme case designed for quantifying the overheads of kernel instrumentation. Such I/O behaviors barely show in real HPC applications. Instead, lower overheads as shown in Figure 8 are more common.

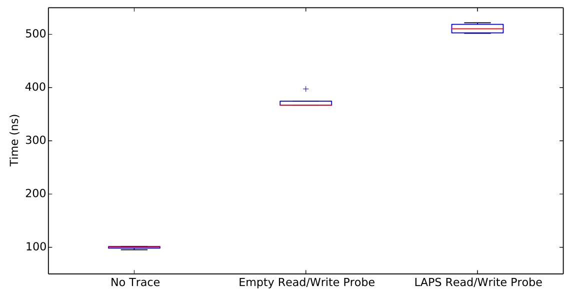

Figure 9: Read/write probe overhead measurement.

B. LPS *Aggregator Performance and Analysis*

The main memory consumption of *LPS* on compute nodes comes from the pipeline. Its size is highly relevant to how fast the events are produced by the tracer and also how fast they are consumed by the aggregator. In this section, we first measure the speed of event generation. The speed of consumption is relevant to the remote *LPS* builder and hence is discussed in the next section. Note that since first/last and open/close generate the same number of events, we used the open/close script in this evaluation. We also used the same IOR settings as the previous one to build I/O-intensive workloads. Figure 10 plots the generated events after a 200-second run of the I/O-intensive workloads. The maximal event generation speed is around 40 events per second, which leads to limited memory consumption since the size of each event is typically less than 100 bytes. We note that the IOR benchmark was run on `RamDisk`, representing the maximal I/O performance achievable. Real HPC applications will generate far fewer events.

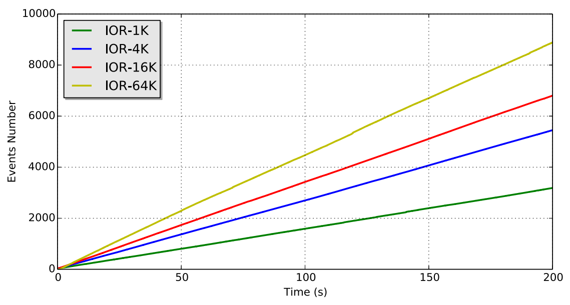

Figure 10: Event generation speed under different workloads.

The *LPS* aggregator monitors a *counter* to identify whether the current granularity should be changed. Figure 11 shows how this counter value changes under different workloads (using the same IOR settings). The results show that for the workload "IOR-64K" the aggregator monitors the minimal counter values and that for the workload "IOR-1K" the counter values are significantly larger, occasionally reaching 7K. Recall the results shown in Figure 8, where "IOR-1K" introduces significant overhead but "IOR-16K" has only minor overhead. Thus, we can have an empirical threshold for our testing platform: Once the counter value exceeds 1500, the aggregator should notify users to switch to coarser granularity.

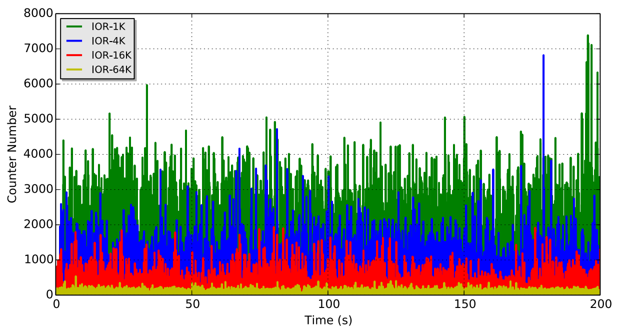

Figure 11: Counter value changes under different workloads.

In Figure 12 we also show the pruning performance by plotting sampled statistical metrics (only about execution-relevant events) in *LPS*. The workloads include MPI programs and data manipulation. In this plot, we show the changing number of received total execution relevant events (on the right y-axis) and the number of running processes in the system, the number of buffered processes due to pruning, and the number of final representative executions (all on the left y-axis). As we can observe from this figure, the total number of collected events (processes relevant) increases steadily. The buffered processes and representative executions, however, are kept at a constantly small number and drop to 0 or 1 when the system goes to idle most of the time. This observation indicates a small memory footprint of *LPS* in handling the process-relevant events, in addition to the limited memory consumption in the pipeline.

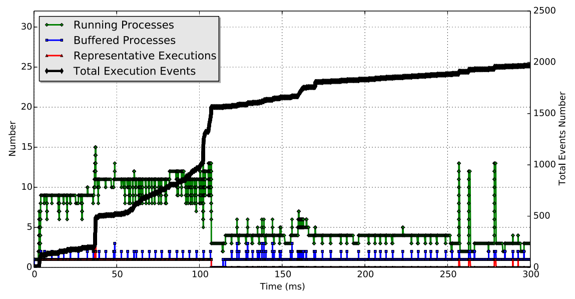

Figure 12: *LPS* local aggregator statistic metrics.

C. LPS *Builder Performance and Analysis*

In *LPS*, all events after being processed by aggregators are sent to the remote builder for final processing and persistence. Hence, the builders are expected to process events from all login nodes and compute nodes in an HPC cluster, potentially leading to a scalability bottleneck. We evaluate the scalability of *LPS* builder in this section. Since the builder runs on dedicated nodes, we do not intend to keep a small memory footprint as we strive to achieve on compute nodes.

We note that although our evaluation platform is not large (45 nodes in total), we can still scale up to conduct serious pressure tests on the *LPS* builder by running IOR on `RamDisk` instead of the much slower parallel file systems. Specifically, we forced the test file (in the RamDisk of each node) written by the IOR benchmark to be considered as a shared file. Hence, all compute nodes send file access events to a single builder for testing its scalability. Considering the speed of RAM, this series of tests actually delivers an extreme number of I/O events, similar to a fully loaded medium-scale to large-scale supercomputer (around 30 GB/s I/O throughput). One builder and up to 256 compute cores were used in this evaluation.

Figure 13 reports the results. We use the number of write operations per second per node to show the IOR performance. From these results, we conclude that (1) the open/close granularity still achieved the negligible overhead compared with the original system and (2) the first/last granularity experienced significant overhead. Although the process number increased,

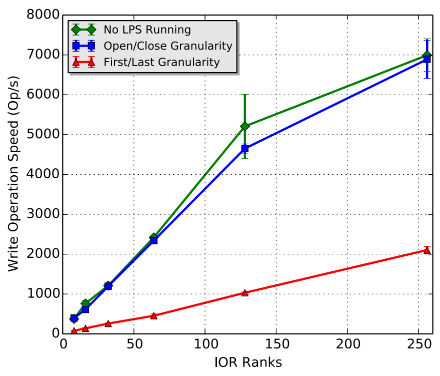

Figure 13: *LPS* builder scalability.

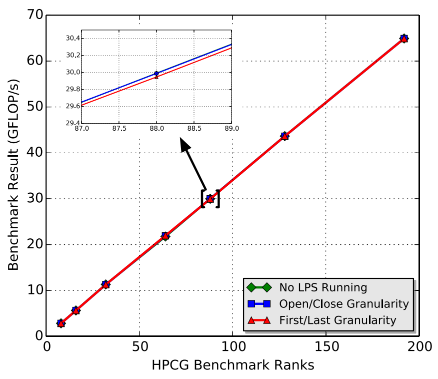

Figure 14: *LPS* performance with HPCG benchmark.

however, the first/last granularity is still scalable. The low speed of first/last granularity comes from the tracer overhead on each compute node, and the stable scalability comes from the fact that it generates the same number of events without overflooding the system.

D. Overall LPS *System Performance Evaluation*

In this subsection, we present the overall performance and overhead analysis of *LPS*. In this series of tests, we used an *LPS* deployment that includes 32 compute nodes (total 256 cores), 5 storage servers, and 1 dedicated server acting as the *LPS* cluster. We chose three different benchmarks indicating different scenarios of performance-oriented applications in HPC. Specifically, we used the High Performance Conjugate Gradients (HPCG) benchmark to represent the balanced data- and CPU-intensive workloads [19]; we used IOR to simulate the growingly critical data-intensive workloads [36]; and we used the MDTest to conduct evaluations representing metadata-intensive applications [31]. We believe the selected benchmarks represent typical workloads seen in most scientific applications.

In Figure 14 we report the overhead comparison of the HPCG benchmark tests. In these tests we ran HPCG with an increasing number of processes (from 8 to 196) and measured its reported performance in GFlops. The same benchmark was run with all granularities. Since the performance reported in GFlops by HPCG is stable in all runs, we do not plot the error bars. As the figure shows, the difference among runs is too minor to be noticeable. We also plot a zoomed-in version of a small range, from 29 GFlops to 30 GFlops, to identify the difference. As the zoomed-in figure shows, the example without *LPS* running achieved the best performance, although with less than 0.1% difference compared with other cases.

In Figure 15 we report the results of the IOR benchmark tests with each supported granularity. For the benchmark, we ran with different numbers of processes; the total number of processes in the experiment was 256. Each process ran on a single core; and each issued 50,000 rounds of random 4 Kbyte writes. Such concurrent I/O processes hit the limits of our PVFS cluster. Each IOR run was repeated 10 times, and we plot the mean throughput and the standard deviation. From this figure, we can make two observations: (1) open/close granularity barely introduces an overhead (around 0.1%) for such I/O-intensive tests; and (2) first/last granularity introduces noticeable overheads compared with the original system. However, the overhead is largely covered by the deviation, indicating that it is also minor in most cases. The maximal observed overhead was still less than 1% in our evaluations.

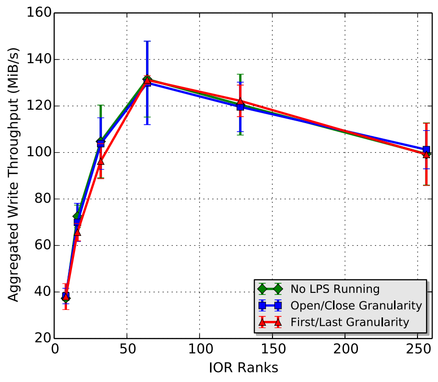

Figure 15: *LPS* performance with IOR benchmark.

In Figure 16 we report the results of the MDTest cases with different granularities. MDTest is a benchmark that measures the performance of various metadata operations. In this figure, we report several representative operations, including “Directory Creation/Removal” and “File Creation/Read.” The benchmark was conducted multiple times, and we plot their mean values in the figure. From the results, we can easily observe that for most of these metadata operations, except “File Read” operations, *LPS* does not introduce noticeable overhead. The large overhead of reading files is due mostly to the read operations issued by MDTest hitting the I/O buffer, which may lead to high pressure on *LPS* with first/last granularity. We have discussed this result in previous evaluations.

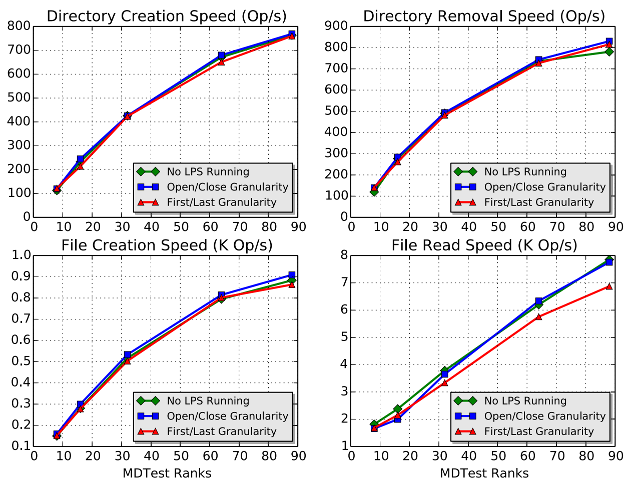

Figure 16: *LPS* performance with MDTest benchmark.

E. Real-World Use Case Support

In this subsection, we show an example of how *LPS* supports data management tasks such as the computation results analysis described in Section II.

Figure 17 shows a fragment of the *LPS* provenance graph built after running the IOR benchmarks using the same configuration file multiple times (visualized using d3.js [2]). In this figure, the light blue dots represent processes, and the deep blue dots represent files. Note that to save the space, we eliminated irrelevant file accesses by the emacs and IOR processes. Also, we utilized the open/close file access granularity in this example.

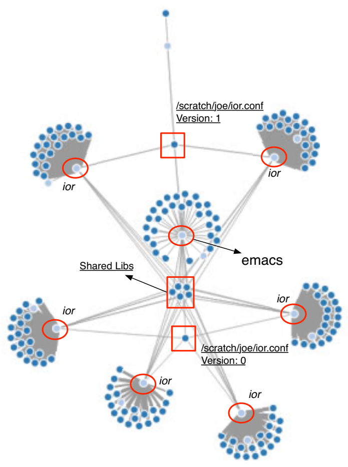

Figure 17: *LPS* support of computation results analysis.

As we can see here, users ran the same IOR benchmarks multiple times trying to obtain an accurate I/O performance. This is a common practice to smooth the system variations in performance evaluations. The key is that all the IOR runs should use the same configuration file so that their results are comparable. As we highlight in the figure, however, *LPS* reveals that two IOR runs were actually based on a different version (version 1) of the configuration file, which was created by an emacs process. This indicates that someone may accidentally modify the configuration file. By looking at this figure, one can easily realize that the average of these IOR results will not correctly reflect the real system performance. Moreover, by tracking back the emacs process, one will be able to identify who made such a modification and whether the consequence is severe. This example demonstrates how the computation results analysis use cases can be supported with our *LPS* system on HPC platforms.

VI. Conclusion and Future Work

Provenance has proved useful in many use cases that are critical to scientific discoveries, such as identifying the data sources, parameters, processing or assumptions behind a given result; auditing data usage to locate datasets that are derived from problematic inputs; understanding the detailed process that how different input data are transformed into outputs; or improving the system performance by analyzing past data access behaviors. Even though provenance is important and highly valuable, however, it is challenging to trace and build provenance metadata efficiently in highly parallel architecture.

In this research, we have introduced *LPS*, a lightweight provenance service for HPC systems. The key contributions and unique features of *LPS* include the following: (1) a system-level transparent tracing methodology designed for HPC environment; (2) a unified provenance granularity model introduced to support flexible granularity control and a balance between overhead and accuracy; and (3) a representative execution concept introduced to compress superfluous events for achieving the low overhead requirement. We have also conducted comprehensive evaluations, and the results support these design objectives. We consider that *LPS* and its associated design and methodologies provide a provenance solution for performance-critical parallel architectures and systems, including the performance-centric HPC systems.

Our future work focuses on two areas. First, we will develop an automatic granularity-switching mechanism to further enhance the transparency of *LPS*. Second, we will work on performance optimizations and on extending *LPS* to better support various scientific workflow systems.

Acknowledgment

We are thankful to the anonymous reviewers for their valuable feedbacks and suggestions that improved this paper significantly. This material is based upon work supported by the U.S. Department of Energy, Office of Science, under Contract No. DE-AC02-06CH11357; and by the National Science Foundation under grant CNS-1162488, CNS-1338078, IIP-1362134, and CCF-1409946.

References

[1] Cobalt web page. http://trac.mcs.anl.gov/projects/cobalt.

[2] Data-driven documents. https://d3js.org/.

[3] Rocksdb. http://rocksdb.org/.

[4] Torque resource manager. http://www.adaptivecomputing.com.

[5] FTP Site: Darshan Data. ftp://ftp.mcs.anl.gov/pub/darshan/data/, 2013.

[6] Conditional probes. https://www.sourceware.org/systemtap/, 2016.

[7] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, et al. The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 154–165. IEEE Press, 2014.

[8] A. M. Bates, D. Tian, K. R. Butler, and T. Moyer. Trustworthy Whole-System Provenance for the Linux Kernel. In *USENIX Security Symposium*, pages 319–334, 2015.

[9] O. Biton, S. Cohen-Boulakia, and S. B. Davidson. Zoom* Userviews: Querying Relevant Provenance in Workflow Systems. In *Proceedings of the 33rd international conference on Very large data bases*, pages 1366–1369. VLDB Endowment, 2007.

[10] B. Bode, D. M. Halstead, R. Kendall, Z. Lei, and D. Jackson. The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters. In *Usenix, 4th Annual Linux Showcase & Conference*, 2000.

[11] P. Buneman, S. Khanna, and T. Wang-Chiew. Why and Where: A Characterization of Data Provenance. In *Database Theory ICDT 2001*, pages 316–330. Springer, 2001.

[12] L. Carata, S. Akoush, N. Balakrishnan, T. Bytheway, R. Sohan, M. Selter, and A. Hopper. A Primer on Provenance. *Communications of the ACM*, 57(5):52–60, 2014.

[13] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross. Understanding and Improving Computational Science Storage Access through Continuous Characterization. *ACM Transactions on Storage (TOS)*, 7(3):8, 2011.

[14] K. Correll, N. Barendt, and M. Branicky. Design Considerations for Software Only Implementations of the IEEE 1588 Precision Time Protocol. In *Conference on IEEE*, volume 1588, pages 11–15, 2005.

[15] Y. Cui, J. Widom, and J. L. Wiener. Tracing the Lineage of View Data in a Warehousing Environment. *ACM Transactions on Database Systems (TODS)*, 25(2):179–227, 2000.

[16] D. Dai. GraphMeta Prototype. http://discl.cs.ttu.edu/gitlab/dongdai/graphfs, 2015.

[17] D. Dai, Y. Chen, P. Carns, J. Jenkins, W. Zhang, and R. Ross. GraphMeta: A Graph-Based Engine for Managing Large-Scale HPC Rich Metadata. In *Proceedings of the IEEE Cluster 2016*, 2016.

[18] D. Dai, Y. Chen, D. Kimpe, and R. Ross. Provenance-Based Object Storage Prediction Scheme for Scientific Big Data Applications. In *2014 IEEE International Conference on Big Data*, pages 271–280. IEEE, 2014.

[19] J. Dongarra, M. A. Heroux, and P. Luszczek. HPCG Benchmark: A New Metric for Ranking High Performance Computing Systems.

[20] J. Freire, D. Koop, E. Santos, and C. T. Silva. Provenance for Computational Tasks: A Survey. *Computing in Science & Engineering*, 10(3):11–21, 2008.

[21] J. Frew and P. Slaughter. ES3: A Demonstration of Transparent Provenance for Scientific Computation. In *Provenance and Annotation of Data and Processes*, pages 200–207. Springer, 2008.

[22] A. Gehani and D. Tariq. SPADE: Support for Provenance Auditing in Distributed Environments. In *Proceedings of the 13th International Middleware Conference*, pages 101–120. Springer-Verlag New York, Inc., 2012.

[23] B. Gregg and J. Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, 2011.

[24] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, 1999.

[25] P. J. Guo and M. Seltzer. BURRITO: Wrapping Your Lab Notebook in Computational Infrastructure. In *TaPP*, 2012.

[26] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[27] S. Ma, X. Zhang, and D. Xu. ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In *NDSS*, 2016.

[28] P. Macko and M. Seltzer. A General-Purpose Provenance Library. In *TaPP*, 2012.

[29] D. L. Mills. Internet Time Synchronization: The Network Time Protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, 1991.

[30] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, and J. Myers. The Open Provenance Model Core specification (v1. 1). *Future Generation Computer Systems*, 27(6):743–756, 2011.

[31] C. Morrone, B. Loewe, and T. McLarty. mdtest HPC Benchmark. 2010.

[32] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. L. MacLean, D. W. Margo, M. I. Seltzer, and R. Smogor. Layering in Provenance Systems. In *USENIX Annual Technical Conference*, 2009.

[33] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer. Provenance-Aware Storage Systems. In *USENIX Annual Technical Conference, General Track*, pages 43–56, 2006.

[34] R. Ricci and E. Eide. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *; login:*, 39(6):36–38, 2014.

[35] C. Scheidegger, D. Koop, E. Santos, H. Vo, S. Callahan, J. Freire, and C. Silva. Tackling the Provenance Challenge One Layer at a Time. *Concurrency and Computation: Practice and Experience*, 20(5):473–483, 2008.

[36] H. Shan and J. Shalf. Using IOR to Analyze the I/O Performance for HPC Platforms. In *CUG'07*, 2007.

[37] C. T. Silva, J. Freire, and S. P. Callahan. Provenance for Visualizations: Reproducibility and Beyond. *Computing in Science & Engineering*, 9(5):82–89, 2007.

[38] Y. L. Simmhan, B. Plale, and D. Gannon. A Survey of Data Provenance in e-Science. *ACM Sigmod Record*, 34(3):31–36, 2005.

[39] A. Vahdat and T. E. Anderson. Transparent Result Caching. In *USENIX Annual Technical Conference*, 1998.