OPUS: A Lightweight system for Observational Provenance in User Space

Nikilesh Balakrishnan, Thomas Bytheway, Ripduman Sohan, Andy Hopper
University of Cambridge, Computer Lab
{*firstname.lastname*}@cl.cam.ac.uk

Abstract

A variety of current provenance systems address the challenges of provenance capture, storage and query. However they require special setup and configuration, do not capture all I/O operations and limit themselves to specific specialised platforms. In this paper we propose the design of a data provenance capture and query tool called OPUS. OPUS works entirely in user space, is light-weight and requires minimum user intervention. OPUS is based on a formal model for versioning provenance objects that enables the succinct, complete representation of I/O operations in a manner that abstracts it from the details of the underlying operating system.

1 Introduction

The capture and use of provenance towards the goal of reasoning about the lineage of data has recently become a topical issue in computer science [2]. To this end, seminal efforts such as PASS [3], StoryBook [4] and Burrito [1] that focus on augmenting existing systems to support provenance have been well received.

While these proof-of-concept systems sought to demonstrate that it is possible to retrofit, capture and utilise provenance for useful purposes, it is our observation that one of the major barriers to the widespread adoption of provenance in current environments is the lack of a widely available, easily deployable, general purpose provenance capture system. In this paper we outline our plans for OPUS (Observed Provenance in User Space). OPUS is a lightweight general purpose provenance capture tool designed to integrate into and operate transparently in existing systems.

OPUS is inspired by previous work (in particular, PASS). It differs from them, however, in the following major dimensions:

Ease of Deployment: It is important that a general purpose provenance capture system be deployable in production systems without requiring operating system and library changes. Previous systems have relied either on custom kernels (e.g. PASS [3]) or bespoke kernel modules that communicate with specialised user space applications (e.g. StoryBook [4]). We observe that both approaches are unacceptable in production environments. OPUS is designed to drop into existing systems and operate entirely in user space to transparently capture provenance. It is expected that implementing OPUS entirely in this manner will aid portability across different operating environments.

Completeness: OPUS is designed to be *complete* in two dimensions:

(*i*) Runtime context collection: Existing systems record operations at system call level, associating them with changes in file data to create relevant provenance records. System call level records however disregard important runtime-specific context (e.g. library-to-system call mappings, thread call stack etc), potentially useful for facilitating construction of detailed provenance models. In OPUS we intend to explore the benefits of recording runtime context with regard to provenance lineage.

(*ii*) Operation completeness: To support a wide range of applications and provenance queries, OPUS captures *all* operations that access or mutate file system entities. In contrast, existing systems focus solely on major mutating operations. It is expected that complete operation capture will facilitate the creation of more detailed data provenance models.

Formal Provenance Object Versioning: Data provenance is commonly represented as state changes in entities. Entities are typically represented as objects and versioned in response to modifications by operations that change their state. Versioning can thereby be defined as the recording of object state at semantically relevant epochs. The goal of versioning is to associate object state with distinct, specific, time epochs in the object timeline. In this manner, versioning assists in reasoning about and optimising provenance queries.

Current provenance systems either perform no versioning or use one of two common versioning models: "version on write" or "open-to-close". Systems that perform no versioning associate all provenance for the entity's lifespan with a single object version. This can render provenance querying difficult to optimise as a potentially large amount of data needs to be traversed for queries. In contrast, versioning adds structure by segregating data associated with different entity versions. Versioning allows us to easily and efficiently ignore irrelevant data when traversing the provenance graph thereby

potentially speeding up queries.

Systems that utilise "version on write" can exhibit similar drawbacks as those that perform no versioning. This is especially true for write intensive processes as objects are versioned on every write resulting in long provenance version chains. Finally, there is no efficient way to model non-mutating operations such as *read* or *stat*.

The "open-to-close" versioning model addresses some of these drawbacks and provides some structure to the provenance data. However, the inherent limitation within "open-to-close" versioning is in its inability to express operations other than *open* or *close* operations.

We posit that the problems described with these versioning models can be solved by formalising the side-effects of operations with regard to object versioning. To this end, we introduce Provenance Versioning Model (PVM), an elementary model designed to enable provenance systems to (*i*) formalise the versioning side effects of *any* (possibly concurrent) noteworthy operations on file system entities in a clear and consistent manner and (*ii*) allow systems to abstract underlying I/O system semantics to provide consistent versioning semantics regardless of operating or file system.

2 PVM: A Provenance Versioning Model

This section outlines the provenance versioning model which attempts to address the versioning problems outlined in the previous section. We present the challenges facing the creation of an accurate and concise versioning model with reference to specific use cases. Full details of the model are available in a separate document[1].

Concurrent access to a shared data resource: In POSIX more than one process can access a given file system entity simultaneously. How can provenance be accurately represented in this scenario? To explain this in detail, we study an example using the open-to-close versioning model:

Let *P1* and *P2* be two processes accessing file F. Process *P1* first opens file F in write mode and obtains a local handle to it. At a later stage process *P2* opens file F in write mode and also obtains a local handle. This is illustrated in Figure 1(a), step 1, where both processes *P1* and *P2* are associated with file provenance object version *F0*. Operations such as *write, stat,* etc., made on file F by processes *P1* and *P2* result in provenance being associated with provenance object *F0*. Finally, when processes *P1* and *P2* perform a *close* on their local handle to file F, the provenance object *F0* is versioned to *F1* and *F2* respectively as shown in Figure 1(a), step 2 and 3.

Maintaining links between the various provenance object versions for file F allows us to easily traverse its provenance history. This example shows that using open-to-close versioning can be effective in representing simple concurrent access scenarios.

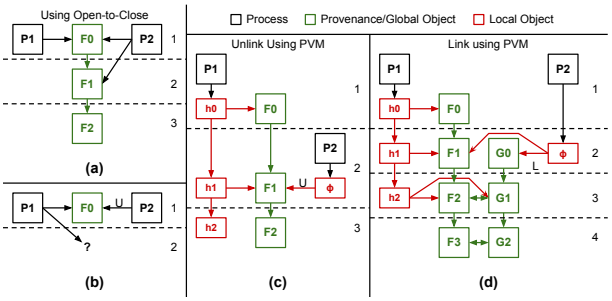

Figure 1: PVM case studies

We now try to use the open-to-close versioning model for a more complicated scenario as shown in Figure 1(b). This example is similar to the one in Figure 1(a) where two processes *P1* and *P2* access file F concurrently. However in step 1 of Figure 1(b), process *P2* unlinks file F while process *P1* still holds a local file handle to file F. This presents three problems, (*i*) The *unlink* operation causes the filename to disappear from the file system, yet using the open-to-close versioning model there is no clear way to create a new version for the provenance object of file F. (*ii*) What happens to subsequent writes made by process *P1*? From process *P1*'s perspective, the local handle to file F is still valid and writes to the local handle will return successfully. (*iii*) The *unlink* operation does not act on a local handle to file F since the operation takes a file path as argument. How can operations that do not explicitly act on file handles be represented in the provenance graph?

To solve the first problem we chose to abstract the operations that trigger versioning in the open-to-close model i.e. *open* or *close* into a more generic form of acquiring or releasing references to file system entities. Thus acquiring or releasing a reference to an entity will cause the provenance object for that entity to version. We will later describe how this concept is applied to our unlink example while explaining Figure 1(c).

The second problem manifests itself due to the existence of two perspectives, (*i*) a process's local view of the system and (*ii*) the global view of the system as seen external to any process. A process can only perceive the system through handles that it currently holds. We express this duality by introducing the concepts of *local* and *global* objects. Therefore we effectively replace the term 'provenance object' with 'global object'. Local objects represent the provenance of process local handles and allow us to associate provenance information to the point of interaction between a file and a process. This gives us the capability to associate provenance to a local object even if it is no longer bound to a global object (as in the unlink scenario).

Using the concepts introduced in solving the previous two problems, the third problem can be solved as follows: operations that do not explicitly use local file han-

dles (e.g. *unlink*, *rename*, *stat*) are represented using a virtual file handle referred to as ϕ. This virtual handle is used to associate the provenance information for any operation that does not normally utilise file handles.

We can now begin to describe the operations in Figure 1(c). In step 1, process *P1* opens file F causing the global object *F0* to be created, this also creates a local object *h0* which is bound to the global object *F0*. In step 2 the process *P2* performs an unlink operation on the file F causing a virtual handle ϕ to be created. The unlink operation first acquires a reference to file F causing the global object *F0* to version to *F1* and binds ϕ to *F1*. Then, the unlink operation removes file F from the filesystem's namespace. This causes local object h to lose its binding to the file F although this will not be reflected in the diagram until its next version. Finally the unlink operation releases its reference to the file F and its local object ϕ, thus causing *F1* to version to *F2*. In step 3, note that from process *P1*'s perspective the local object *h2* is still valid even though it is unbound from global object *F2*. Any operation performed by process *P1* on the local handle to file F will now be associated to the local object *h2*. Observe that the model has accurately captured all operations and their versioning side-effects in a complete and concise manner.

Hard and Soft links POSIX supports the concept of hard and soft links. Representing the provenance for both these link types poses two questions. (*i*) In case of a group of names representing hard links to a file, should the provenance information associated with any given name in the group be associated to all members in the group? (*ii*) Depending on the operations performed on soft links, the provenance information generated could be associated to either the soft link itself or to the canonicalised file the soft link references. How should our model demarcate this provenance information and associate it to the appropriate provenance objects?

In order to address the first question, we use the example in Figure 1(d) and provide a step by step explanation for the same. In step 1, process *P1* performs an open operation on file F, creating a global object version *F0*. During step 2, filenames F and G are hard linked. The provenance for the hard link operation is represented by acquiring a temporary local object ϕ and binding ϕ to global objects *F1* and *G0*. The global objects F and G are then marked as *equivalent*. The local object ϕ is released which causes the versioning of global objects *F1* to *F2* and *G0* to *G1* respectively. The side effects of marking global objects F and G equivalent results in all local objects associated to F being bound to G and vice versa, thereby unifying the provenance data associated with both F and G.

We see in step 3 that global object versions *F2* and *G1* are shown as being equivalent, represented by $\leftrightarrow$. The

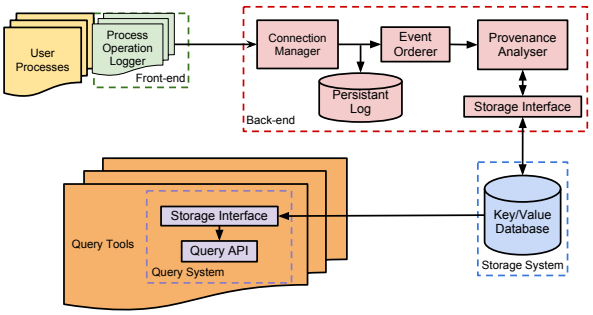

Figure 2: OPUS Architecture

local object *h2* used by process *P1* is now bound to both F and G. Finally as shown in step 4 when process *P1* releases its reference to file F, global objects *F2* versions to *F3* and *G1* versions to *G2*.

Soft links, however, are selectively canonicalised depending on operation. This selectivity allows us to capture and associate provenance information to the soft link itself or to the underlying physical entity as appropriate. For example the *open* operation applied to a soft link affects the underlying object whereas the *unlink* operation affects the soft link itself.

2.1 Operation Mapping

Based upon the objects, relations and conventions described above, a formal model for provenance object versioning, PVM has been established. PVM semantics involve a set of transformations that allow us to describe the state of a system at any given time. We expect that any operation can be defined in terms of these transformations. By defining operations in terms of these transformations, we can give a clear and consistent definition of an operation's versioning behaviour. Using PVM simplifies the design and implementation of provenance systems as only the transformations defined by PVM need to be implemented rather than every potential operation. The complete mapping of POSIX operations using PVM semantics is available in the PVM mapping document[1].

By using PVM we give ourselves a rigorous method for defining versioning for any operation, express concurrent interactions between applications over a shared data resource and have a consistent versioning semantic regardless of the underlying platform.

3 OPUS Architecture

OPUS is a single host provenance capture and analysis system for systems implementing the POSIX I/O interface. Similar to PASS it captures changes to file system entities. At its core OPUS employs our POSIX PVM mapping. We have mapped 180 libC calls into our PVM model giving us the equivalent of 54 system calls worth of capture coverage, the full details of which can be found in the PVM mapping document[1].

OPUS runs entirely in user space thereby simplifying its implementation and setup. It also allows for a faster development cycle as there are a plethora of tools and components available for user space application development. OPUS is designed to always be active as a system service and is expected to capture and analyse provenance data for all applications in the system.

OPUS has two major components the front-end and the back-end. The front-end is a thin light-weight library that captures raw provenance data propagating it to the back-end. The back-end receives raw provenance data, logs, orders and analyses provenance information before persisting it to the database as provenance objects. The various sub-components in the front-end and back-end can be replaced or extended due to their modular nature.

The system's major components as illustrated in the Figure 2 are described below:

Process Operation Logger: The process operation logger component is a Linux DSO implemented in C++ invoked using the linker's LD_PRELOAD[2] feature. The preliminary version of the operation logger will override the application symbol table during startup enabling OPUS to intercept and capture provenance at the C library level. In the future we expect to augment this functionality to intercept statically compiled executables by implementing binary rewriting at application load time. Using this approach, provenance capture becomes unobtrusive and seamless from a user's perspective. Captured information is relayed to the back-end for further processing.

Persistent Log File: The Persistent log file stores raw provenance data received from front-end processes by persisting data to stable storage. The persistent log file provides the ability to reconstruct provenance trails in the event of system or process crashes. This is achieved by maintaining periodic checkpoint indexes for rollback. The persistent log file further enables offline processing.

Event Orderer: Maintaining a correct view of the ordering of events in the system is critical especially in cases where two or more processes share entities. Thus, the event orderer in the back-end ensures provenance data being forwarded to the analyser is ordered correctly.

Provenance Analyser: The provenance analyser is the heart of the OPUS back-end. It will take ordered provenance data as input, apply the transformations defined for a given operation as specified in our POSIX PVM mapping and convert the provenance data into provenance objects and relations. The processed provenance data is then persisted using the storage system.

Storage System: The storage system will use levelDB[3], a key/value database to store provenance objects. A storage interface will be used by various components in the system to store and retrieve provenance objects from the database.

Query System: The query system will provide a query API with a range of functionality to retrieve provenance objects using the storage interface. Query tools ranging from command line to GUI driven can use the query API in order to access the database and retrieve provenance information.

4 Open Questions

Deploying OPUS as an always-available aspect of system infrastructure in production environments will require addressing the following open issues:

Intra-host Provenance Capture And Analysis: In order to precisely capture data provenance in distributed computing environments it will be necessary to capture and analyse data provenance across hosts. We anticipate achieving this will require extending the PVM to model distributed systems such that intra-host data and analysis algorithms can be extended across hosts.

Performance: The concept of OPUS as a core, conceptual component of system infrastructure has the implication that the spatial and temporal overheads of logging provenance information needs to be minimised in order to ensure adequate performance. To this end we are exploring the concept of hardware-accelerated provenance capture.

5 Conclusion

In this paper we have provided the motivation for a provenance versioning model (PVM). We also described the high level design for our provenance system OPUS which internally uses PVM semantics in order to version objects.

Acknowledgements

We acknowledge George Coulouris, Lucian Carata and Sherif Akoush for their advice on shaping this paper.

References

[1] GUO, P. J., AND SELTZER, M. Burrito: wrapping your lab notebook in computational infrastructure. In *Proceedings of the 4th USENIX conference on Theory and Practice of Provenance* (Berkeley, CA, USA, 2012), TaPP'12, USENIX Association.

[2] KAVANAGH, J., AND HALL, W. Grand challenges in computing research 2008. In *Grand challenges in computing research 2008* (United Kingdom, 2008), GCCR'08, UK Computer Research Committee.

[3] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. Provenance-aware storage systems. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference* (Berkeley, CA, USA, 2006), ATEC '06, USENIX Association.

[4] SPILLANE, R., SEARS, R., YALAMANCHILI, C., GAIKWAD, S., CHINNI, M., AND ZADOK, E. Story book: an efficient extensible provenance framework. In *First workshop on on Theory and practice of provenance* (Berkeley, CA, USA, 2009), TAPP'09, USENIX Association.

Notes

[1] `http://www.cl.cam.ac.uk/research/dtg/www/research/`
[2] `http://linux.die.net/man/1/ld`
[3] `http://code.google.com/p/leveldb/`