

Story Book: An Efficient Extensible Provenance Framework

R. Spillane, R. Sears*, C. Yalamanchili, S. Gaikwad, M. Chinni, and E. Zadok
*Stony Brook University and *University of California, Berkeley*

Abstract

Story Book is a general-purpose provenance system that treats dependencies between object and processes separately from information about the updates performed by the system. This allows it to provide high throughput updates and a set of application-independent queries over customizable provenance information.

Existing provenance file systems reside in kernel space, significantly constraining their implementations. Story Book’s user space design uses FUSE and simplifies integration of provenance data from multiple sources, and allows Story Book to allow application-specific extensions to intercept and log extra information about each request.

Unlike existing approaches, Story Book allows applications to specify provenance hooks that can be implemented independently of the original application’s source base. This paper presents four examples, one for provenance of `.txt`, `.docx`, and generic files in file systems, and the other for database queries.

Story Book’s provenance database makes use of existing user-space storage mechanisms, including high-throughput compression, application-aware durability, and user-level file systems. This allows Story Book to record provenance at high rates and to perform efficient queries without introducing complexities into the kernel.

We implemented application-specific extensions to Story Book in just a few days. Story Book stores provenance data in its database 4.5 times faster than PASS, another advanced provenance system. Story Book incurs only a 10% overhead over the existing message passing overhead of FUSE. When measured under worst case I/O-intensive metadata workloads, Story Book is nearly 2.5 times slower than vanilla `ext3`; however, most of this overhead is due to message passing with the FUSE user-level server.

1 Introduction

Existing provenance systems are designed to deal with specific applications and system architectures, and are thus inflexible and not readily portable to other systems. Story Book decouples the application-specific aspects of provenance tracking from dependency tracking, queries and other mechanisms common across prove-

nance systems. Story Book runs in user space, allowing it to use existing user space components to handle application data, and to use off-the-shelf storage libraries to implement its high-performance provenance storage system. Although implementing provenance (or any other) file system in user space incurs significant overhead, it can significantly reduce communication costs for Story Book’s application-specific provenance extensions, which may use of existing IPC mechanisms, or even run inside the process generating provenance data.

Story Book supports two types of application-specific extensions. The first brings provenance to new classes of systems: Story Book currently supports file system and database provenance and is designed to be extended to other types of systems, such as Web or email servers. The second class of extension increases the range of data types that Story Book understands; we currently support `.txt` and `.docx` files. For `.txt` files, Story Book stores the contents of the changes made by each application, while for `.docx` files it records changes to the metadata.

Story Book stores provenance data using database-style no-Force/Steal recovery, log-structured merge (LSM) trees [6] and compression. It is also able to leverage application-specific durability mechanisms, greatly reducing the number of force writes required for recoverable operation.

Our experiments show that Story Book’s storage and query performance are adequate for file system provenance workloads where each process performs a handful of provenance-related actions. However, systems that service many small requests across a wide range of users, such as databases and Web servers, generate provenance information at much higher rates and with finer granularity. As such systems handle most of today’s multi-user workloads, we believe they also provide higher-value provenance information than interactive desktop systems. This space is where Story Book’s extensible user-space architecture and high write throughput are most valuable.

The rest of this paper is organized as follows: Section 2 describes existing approaches to provenance, Section 3 describes the design and implementation of Story Book, and Section 4 describes our experiments and performance comparisons with PASS. We conclude in Section 5.

2 Background

A survey of provenance systems [9] provides a taxonomy of existing provenance databases for eScience. Applying the taxonomy shows that Story Book hard-codes a few aspects of provenance systems related to implementation strategies. The most fundamental is that we store annotations with each computation rather than attempt to derive process inputs from their outputs. While this limitation is fundamental to general-purpose file system provenance systems, some computations are invertible, allowing some existing provenance systems to delete derivable input data in order to conserve space.

A second hard-coded aspect of Story Book is the decision to store metadata information separately from application data. This is done partially out of necessity (since Story Book cannot change legacy formats)—and partially for performance reasons, since reducing the size of the dependency graph’s on-disk representation improves query performance. Story Book takes this idea a step further and stores extended records—which contain application-specific data—separately from the dependency graph; this improves disk locality during graph traversals.

Ignoring implementation details, Story Book is applicable across the application domains of most existing provenance systems. In particular, Story Book allows applications to determine the granularity of their provenance records, and whether to focus on information about data, processes or both. Similarly, Story Book’s implementation is independent of the format of extended records, allowing applications to use metadata standards such as RDF [15] or Dublin Core [2] as they see fit.

The primary limitation of Story Book compared to systems that make native use of semantic metadata is that Story Book’s built-in provenance queries would need to be extended to understand annotations such as “version-of” or “derived-from,” and act accordingly. Applications could implement custom queries that make use of these attributes, but such queries would incur significant I/O cost. This is because our schema places such application-specific annotations in extended records, which are potentially large and stored away from the rest of the provenance data.

Lineage systems such as Trio [16] generally track provenance information automatically, but also attempt to reason about the quality or reliability of input data and processes that modify it over time. Introducing uncertainty into the data and operations complicates provenance data models and the semantics and execution of queries within lineage systems.

The provenance systems that Story Book targets track the execution of processes over application data. In such environments, it is reasonable to assume that provenance

data is completely reliable (applications were executed, or not). Furthermore, we expect that for most applications, it will be more natural for end-users to ask which version of an input was used rather than to calculate the probability that a particular file is corrupt.

2.1 Comparison to PASS

Existing provenance systems couple general provenance concepts to system architectures, leading to complex in-kernel mechanisms and complicating integration with application data representations. Architecturally, PASSv2 [5] is the closest system to Story Book. PASSv2 distinguishes between data *attributes*, such as name or creation time, and *relationships* indicating things like data flow, versioning and so on. Story Book performs a similar layering, except that it treats versioning information (if any) as an attribute. Furthermore, whereas PASSv2 places such mechanisms in kernel space, Story Book places these mechanisms in user space.

On the one hand, this means that user-level provenance inspectors could tamper with provenance information; on the other it has significant performance and usability advantages. In regulatory environments (which cope with untrusted end-users), we expect Story Book to run in a trusted file or other application server. The security implications of this approach are minimal: in both cases a malicious user would require root access to the provenance server in order to tamper with provenance information.

PASSv2 is a *system-call-level* provenance system, and requires that all provenance data to pass through relevant system calls. Application level provenance data is tracked by applications, then used to annotate corresponding system calls that are intercepted by PASSv2. In contrast, Story Book is capable of system call interception, but does not require it. This avoids the need for applications to maintain in-memory graphs of provenance information. Furthermore, it is unclear how system-call-level instrumentation interacts with applications that use write-back caching, which destroys the relationships between application-level operations and system calls. Story Book’s ability to track MySQL provenance shows that such systems are easily handled with our approach.

PASSv2 supersedes PASS, which directly wrote provenance information to a database. The authors of the PASS systems concluded that direct database access was neither “efficient nor scalable,” and moved database operations into an asynchronous background thread. Our experiments show that Fable, Story Book’s custom provenance database, provides considerably higher throughput than PASSv2’s background thread. Furthermore, Fable provides extremely low latency writes, (Section 3.2) allowing it to process provenance data syn-

chronously. This avoids the overhead of materializing then consuming an in-memory copy of the provenance log, and the complex consistency issues inherent in providing queries over stale data.

PASSv2 uses a recovery protocol called *write ahead provenance*, which is similar to Story Book’s file-system recovery approach. However, Story Book is slightly more general, as its recovery mechanisms are capable of entering into system-specific commit protocols. This is important when tracking database provenance, as it allows Story Book to use existing, inexpensive database replication techniques to atomically apply provenance records generated by database transactions.

PASSv2 supports a number of workloads currently unaddressed by Story Book, such as network operation and unified naming across provenance implementations. The mechanisms used to implement these primitives in PASSv2 follow from the decision to allow multiple provenance sources to build provenance graphs before applying them to the database. Even during local operation, PASSv2 must employ special techniques to avoid cycles in provenance graphs due to reordering of operations and record suppression. In the distributed case, clock skew and network partitions further complicate matters.

In contrast, Story Book appends each provenance operation to a unified log, guaranteeing a consistent, cycle-free dependency graph (see Section 3.8), but assumes a global ordering over operations.

3 Design and Implementation

Story Book has two primary scalability goals: (1) decrease the overhead of logging provenance information, and (2) decrease the implementation effort required to track the provenance of applications which use the file system or other instrumentable storage to organize their data.

We discuss Story Book’s application development model in Section 3.1. We describe the components Story Book adds to a typical operating system and what their roles are in Section 3.2. We explain Story Book’s indexing optimizations in Section 3.3 and recovery in Section 3.4. We discuss reuse of third party code in Section 3.5 and the types provenance that are recorded on-disk in Section 3.6. We explain how Story Book accomplishes a provenance query in Section 3.7 and describe Story Book’s ability to compress and suppress redundant provenance information in Section 3.8.

3.1 Application Development Model

Story Book provides a number of approaches to application-specific provenance. The most powerful, efficient and precise approach directly instruments applications, allowing the granularity of operations and actors

recorded in the provenance records to match the application. This approach also allows for direct communication between the application and Story Book, minimizing communication overheads.

However, some applications cannot be easily instrumented, and file formats are often modified by many different applications. In such cases, it makes sense to track application-specific provenance information at the file system layer. Story Book addresses this problem by allowing developers to implement *provenance inspectors*, which intercept requests to modify particular sets of files or other tracked objects. When relevant objects are accessed, the appropriate provenance inspector is alerted and allowed to log additional provenance information. Provenance inspectors work best when the intercepted operations are indicative of the application’s intent (e.g., an update to a document or a graph image represents an edit to the document or the graph). Servers that service many unrelated requests or use the file system primarily as a block device (e.g., Web servers, databases) are still able to record their provenance using Story Book’s file-system provenance API, though they would be better served by Story Book’s application-level provenance.

For our evaluation of Story Book’s file system-level provenance, we implemented a `.txt` and a `.docx` provenance inspector. The `.txt` inspector records patches that document the changes that were made to a text file between the time it was opened and closed. The `.docx` inspector records changes to document metadata (e.g., author names, title) between the time it was opened and closed. We describe provenance inspectors in more detail in Section 3.6.3.

3.2 Architecture

Story Book stores three kinds of records: (1) *basic records* which are a record of a file open, close, read or write, (2) *process records* which are a record of a caller-callee relationship and (3) *extended records* which are a record of application-specific provenance information.

These provenance records may come from multiple sources. Story Book’s file system provenance inspectors use the FUSE library [13] to instrument file operations, while application-level provenance inspectors are implemented by instrumenting at the application level. For example, our MySQL [12] provenance system scrapes MySQL’s general query log. The provenance information generated from these sources is stored in *Fable*, Story Book’s provenance database.

Fable is built on top of Stasis [7] and is designed to efficiently service a number of predefined queries while providing extremely high write throughput. The basic and process record tables are stored using Stasis’ Rose [8] indexes, and we maintain a number of transactional hash indexes that are required by the provenance

queries.

Rose indexes are compressed log structured merge (LSM) trees that have been optimized for write-heavy replication environments. Durable hash index insertions are more expensive than Rose insertions, but we expect the hash table indexes to be small and infrequently updated (Section 3.6).

Rather than synchronously commit new basic, process and extended records to disk before allowing updates to propagate to the file system, we use Stasis’ non-durable commit mechanism and Valor’s write ordering [10] to ensure that write-ahead records reach disk before file system operations.

This allows Fable to use a single I/O operation to commit batches of provenance operations just before the file system flushes its dirty pages. Although writes to Stasis’ page file never block log writes or file system writeback, hash index operations may block if the Stasis pages they manipulate are not in memory. Unless the system is under heavy memory pressure, the hash pages will be resident. Therefore, most application requests that block on Fable I/O also block on file system I/O.

Figure 1 describes what happens when an application P writes to a document. The kernel receives the request from P and forwards the request to the FUSE file system (1). FUSE forwards the request to Story Book’s FUSE daemon (2) which determines the type of file being accessed. Once the file type is determined, the request is forwarded to the appropriate provenance inspector (3). The provenance inspector generates basic and extended log records (4) and schedules them for asynchronous insertion into a write-ahead log (5). Story Book’s database, Fable, synchronously updates (cached) hash table pages and Rose’s in-memory component (6), and then allows the provenance inspector to return to the caller P (7). After some time, the file system flushes its dirty pages (8), and Valor ensures that any scheduled entries in Fable’s log are written (9) before allowing the file system to flush (10).

Fable’s write-ahead log contains Stasis recovery information and basic records that have not yet been persisted by Rose, and can be periodically truncated. (Though due to a Stasis performance bug, our current implementation actually stores basic records in a separate log that is never truncated.) Pages in Stasis’ buffer manager are occasionally written to disk to enable truncation or respond to memory pressure (11).

3.3 Log Structured Merge Trees

Rose indexes are compressed LSM-trees that have been optimized for high-throughput and asynchronous, durable commit. Their contents are stored on disk in compressed, bulk loaded B-Trees. Insertions are serviced by inserting data into an in-memory red-black tree.

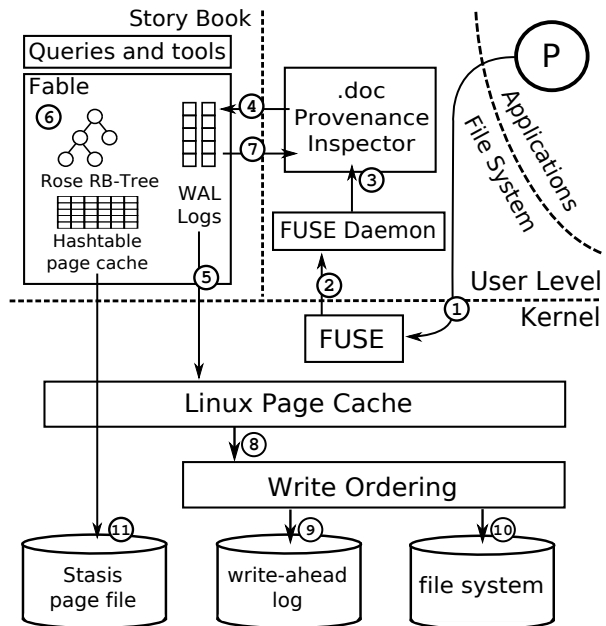


Figure 1: Story Book Architecture and write path

Once this tree is big enough, it is merged with the smallest of the on-disk trees and emptied. Since the on-disk tree’s leaf nodes are laid out sequentially on disk, this merge does not cause a significant number of disk seeks. Similarly, the merge produces data in sorted order, allowing it to contiguously lay out the new version of the tree on disk.

Eventually the small on-disk tree component will become very large, causing merges with in-memory trees to become prohibitively expensive. Before this happens, we merge the smaller on disk tree with a larger on disk tree, emptying the smaller tree. By using two on-disk trees and scheduling merges correctly, the amortized cost of insertion can be reduced to $O(\log n * \sqrt{n})$, with a constant factor proportional to the cost of compressed sequential I/O. B-Trees’ $O(\log n)$ insertion cost is proportional to the cost of random I/O. The ratio of random I/O cost to sequential I/O cost is increasing exponentially over time, both for disks and for in-memory operations.

Analysis of asymptotic LSM-Tree performance shows that, on current hardware, worst-case LSM-Tree throughput will dominate worst-case B-Tree write throughput across all reasonable index sizes. However, in the best case for a B-Tree, insertions arrive in sorted order, allowing it to bulk load the data once with very good locality. However, as discussed below, provenance queries require fast lookup of basic records based on in-ode, not arrival time, forcing Story Book’s indexes to re-sort basic records before placing them on disk.

Sorted data compresses well, and Story Book’s tables have been chosen to be easily compressible. This al-

lows Rose to use simple compression techniques, such as run length encoding, to conserve I/O bandwidth. Superscalar implementations of such compression algorithms compress and decompress with GiB/s throughput on modern processor cores. Also, once the data has been compressed for writeback, it is kept in compressed form in memory, conserving RAM.

3.4 Recovery

As we see in Section 3.6, the basic records stored in Fable contain pointers into extended record logs. At runtime, Fable ensures that extended records reach disk before the basic records that reference them. Therefore, we know that all pointers from basic records encountered during recovery point to complete extended records, so recovery does not need to take any special action to ensure that the logs are consistent.

If the system crashes while file system (or other application) writes are in flight, Story Book must ensure that a record of the in-flight operations is available in the provenance records so that future queries are aware of any potential corruption due to crash.

Since Valor guarantees that Stasis' write ahead log entries reach disk before corresponding file system operations, Fable simply relies upon Stasis recovery to recover from crash. Provenance over transactional systems (such as MySQL) may make use of any commit and replication mechanisms supported by the application.

3.5 Reusing User-Level Libraries

Figure 1 shows that provenance inspectors are implemented as user-level plugins to Story Book's FUSE daemon. This permits the use of all the libraries the application itself links against in order to efficiently and safely extract the application-specific provenance. Our implementation of the `.docx` inspector is linked against the XML parser ExPat [3]. Story Book's FUSE-based design facilitates transparent provenance inspectors that require no application modification without forcing developers to port user-level functionality into the kernel [11]. We now discuss what kind of provenance Story Book writes to disk.

3.6 Provenance Schema

Story Book models the system's provenance using basic records, process records and extended records. To support efficient query, Fable must maintain several different Rose tables and persistent hash indexes. In this section we discuss the format of these three records. For each record type we explain how Story Book maintains the relevant tables and indexes.

3.6.1 Basic Records: Generic Provenance

To determine the provenance or history of an arbitrary file, Story Book must maintain a record of which processes read from and wrote to which files. Story Book clients achieve this by recording a basic record for every create, open, close, read and write to every file regardless of type. The format of a basic record is as follows:

GLOBAL ID The process ID of the process performing the file access. This ID is globally unique across reboots.

PARENT GLOBAL ID The global ID of the parent of the process performing the file access.

FILE INODE The inode number of a file or object id.

EXECUTABLE ID The global ID corresponding to the absolute path of the executable or other name associated with this process.

OPERATION Indicates if the process performed a read, write, open or close.

LSN The log sequence number of the record, used for event ordering and lookup of application-specific extended records

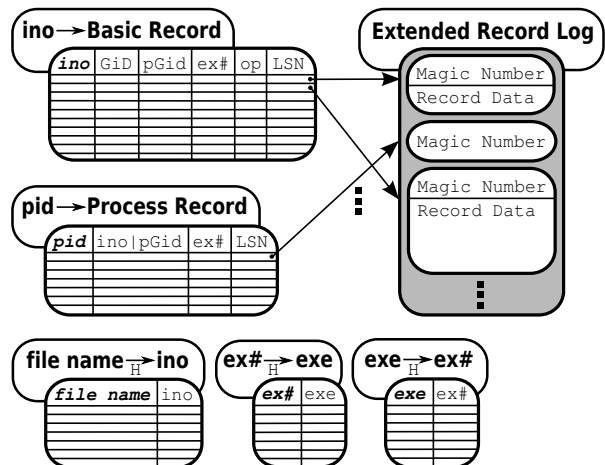


Figure 2: Story Book Database Layout. Table keys are bold. 'H' indicates the table is a hash table.

Figure 2 illustrates how Fable stores basic records. Executable paths are stored using two hash tables that map between executable ID's and executable names ('exe' to 'exe#') and ('exe#' to 'exe'). The first is used during insertion; the second to retrieve executable names during queries. File names are handled in a similar fashion. Fable stores the remaining basic record attributes in a Rose table, sorted by inode / object id.

3.6.2 Process Records: Parent-Child Provenance

When processes perform file operations Story Book records their executable name and process ID within a

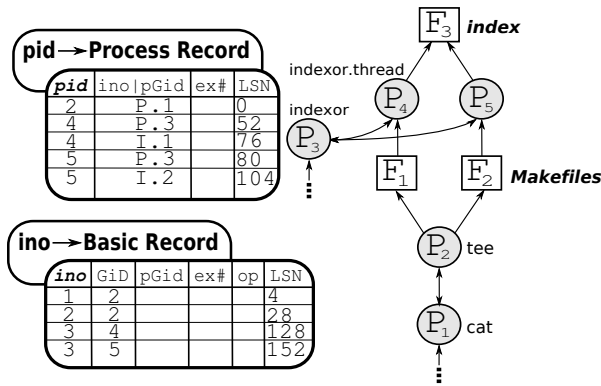


Figure 3: A provenance query. On the left is the data used to construct the provenance graph on the right. The query begins with a hash lookup on the file name’s inode whose provenance we determine (`< “index, ”3 >`).

basic record. However, Story Book cannot rely on every process to perform a file operation (e.g., the `cat` program in Figure 3), and must also record *process records* alongside basic records and application-specific provenance. Process records capture the existence of processes that might never access a file. The format of a process record is as follows:

GLOBAL ID The process ID of the process performing the file access. This ID is globally unique across reboots.

PARENT GLOBAL ID OR FILE INODE Contains either the global ID of the parent of this process or the inode of a file this process modified or read.

EXECUTABLE ID Global ID corresponding to the absolute path of the executable.

LSN The log sequence number of the record, used to establish the ordering of child process creation and file operations during provenance queries.

Before Story Book emits a basic record for a process it first acquires all the global IDs of that process’s parents by calling a specially added system call. It records a process record for every (global ID, parent global ID) pair it has not already recorded as a process record at some earlier time.

The exact mechanisms used to track processes varies with the application being tracked, but file system provenance is an informative example. By generating process records during file access rather than at `fork` and `exit`, file system provenance is able to avoid logging process records for processes that do not modify the files. This allows it to avoid maintenance of a provenance graph in kernel RAM during operation, but is not sufficient to reflect process reparenting that occurs before a file access or a change in the executable name that occurs after the last file access.

3.6.3 Extended Records: Application-Specific Provenance

Story Book manages application-specific provenance by having each application’s provenance inspector attach additional application-specific information to a basic record. This additional information is called an *extended record*. Extended records are not stored in Rose, but rather are appended in chronological order to a log that is never truncated. This is because their format and length may vary and extended records are not necessary to reconstruct the provenance of a file in the majority of cases. An extended record can have multiple application-specific entries or none at all. The format of an extended record is as follows:

MAGIC NUMBER Indicates the type of the record data that is to follow. For basic or process records with no application-specific data to store this value is 0.

RECORD DATA Contains any extended records’ application-specific data.

Each basic and process record must refer to an extended record because the offset of the extended record is used as an LSN. If this were not true Rose would have to add another column to the basic record table. In addition to an LSN it would need the offset into the extended record log for basic records which refer to application-specific provenance. Compressing the basic record table with this additional column would be more difficult and would increase Story Book’s overhead. Records that have no application-specific data to store need not incur an additional write since the magic number for an empty extended record is zero and can be recorded by making a hole in the file. Reading the extended record need only be done when explicitly checking for a particular kind of application-specific provenance in a basic record.

3.7 Provenance Query

Story Book acquires the provenance of a file by looking up its inode in the (`file name, ino`) hash table. The inode is used to begin a breadth first search in the basic and process record tables. We perform a transitive closure over sets of processes that wrote to the given file (or other files encountered during the search), over the set of files read by these processes, and over the parents of these processes. We search in order of decreasing LSN and return results in reverse chronological order.

3.8 Compression and Record Suppression

Rose tables provide high-performance column-wise data compression. Fable’s basic record table run length encodes all basic record columns except the LSN, which is stored as a 16 bit diff in the common case, and the opcode, which is stored as an uncompressed 8-bit value. The process record table applies run length encoding to

the process id, and ex#. It attempts to store the remaining columns as 16 bit deltas against the first entry on the page. The hash tables do not support compression, but are relatively small, as they only store one value per executable name and file in the system.

In addition to compression, Story Book supports *record suppression*, which ignores multiple calls to read and write from the same process to the same file. This is important for programs that deal with large files; without record suppression, the number of (compressed) entries stored by Fable is proportional to file sizes.

Processes that wish to use record suppression need only store the first and last read and write to each file they access. This guarantees that the provenance graphs generated by Story Book’s queries contain all tainting processes and files, but it loses some information regarding the number of times each dependency was established at runtime.

Unlike existing approaches to record suppression, this scheme cannot create cyclic dependency graphs, though it does force queries to treat suppressed operations as though they happened in race. Because Story Book dependency graphs can never create cycles, no special cycle detection or removal techniques are required.

Note that this scheme never increases the number of records generated by a process, as we can interpret sequences such as “open, read, close” mean a single read. Record suppression complicates recovery slightly. If the sequence “open, read, ..., crash” is encountered, it must be interpreted as: “open, begin read, ..., end read, crash.” Although this scheme requires up to two entries per file accessed by each process, run length encoding usually ensures that the second entry compresses well.

4 Evaluation

Story Book depends on fast write performance to maintain a low overhead on top of the operating system and other applications. Story Book must index the provenance records it stores to provide querying capabilities on its data. Therefore, we focus on Story Book write throughput in Section 4.2, a traditional file system workload in Section 4.3, and a large application in Section 4.4. We measure provenance read performance in Section 4.5.

4.1 Experimental Setup

We used four identical machines, each with a 2.8GHz Xeon CPU and 1GB of RAM for benchmarking. Each machine was equipped with six Maxtor DiamondMax 10 7,200 RPM 250GB SATA disks and ran CentOS 5.2 with the latest updates as of September 6, 2008. All benchmarking systems ran Linux 2.6.25. To ensure a cold cache and an equivalent block layout on disk, we ran each iteration of the relevant benchmark on a newly for-

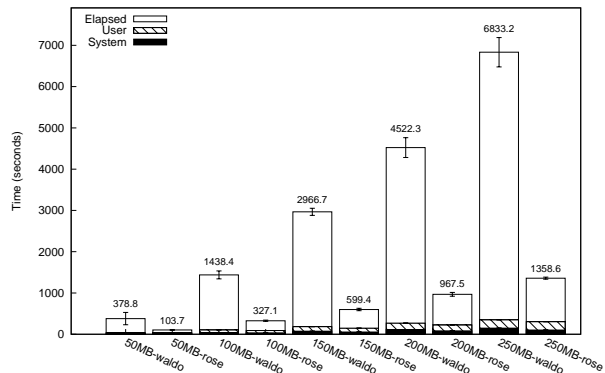


Figure 4: Performance of Waldo and Story Book’s insert tool on logs of different sizes.

matted file system with as few services running as possible. For query benchmarks, database files are copied fresh to the file system each time. We ran all tests at least four times and computed 95% confidence intervals for the mean elapsed, system, user, and wait times using the Student’s-*t* distribution. In each case, unless otherwise noted, the half widths of the intervals were less than 5% of the mean. Wait time is elapsed time less system and user time and mostly measures time performing I/O, though it can also be affected by process scheduling.

(The sources for our kernel patch, our modified kernel, our benchmarks, and our past results will be made available upon request to the Program Chairs.)

Provenance Workloads. The workload we use to evaluate our throughput is a shell script which creates a file hierarchy with regular files as leaf nodes and all other nodes being directories. The workload we use to evaluate our query performance is a shell script which creates a file hierarchy with regular files as leaf nodes and then spawns processes that randomly read from and write to these regular files. These scripts were designed to generate a highly dense provenance graph to obtain an accurate upper bound on Story Book’s query performance and demonstrate its write performance.

4.2 Provenance Throughput

PASSv2 consists of two primary components: (1) an in-kernel set of hooks that log provenance to a write-ahead log, and (2) a user-level daemon called Waldo which reads records from the write-ahead log and stores them in a Berkeley Database (BDB) database. The complex kernel components of PASSv2 make it difficult to reliably complete I/O intensive system benchmarks, so we compare only the second component. Waldo’s log is built from successful runs of PASSv2 on exactly the same workload that we use for Story Book. We then measure the time it takes to store the entire generated

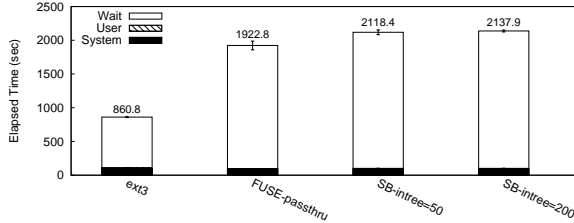


Figure 5: Performance of Story Book compared to *ext3* and *FUSE* for I/O intensive file system workloads.

write ahead log in BDB using Waldo and in Rose using a tool similar to Waldo. The Waldo tool reads records off of the write-ahead log for Waldo and inserts them into BDB. Our tool does the same but uses Story Book’s write-ahead log and it inserts records into Rose.

Our results in this benchmark show the advantage of Rose’s bulk-loading log structured merge tree: we can avoid random I/O penalties by merging new records in-RAM with the database on disk in a series of large serial transfers. For this experiment, we use Waldo’s default settings and Rose is configured with an in-memory tree of 5 MiB and a buffer size of 80 MiB. For a 250 MiB log, Waldo incurs a 402% overhead over Rose when inserting a write-ahead log. Waldo incurs a 339% overhead over Rose for a 100 MiB log with a half-width of 6%. As the amount of data increases, Rose’s ability to exploit its more efficient flushing mechanism enables it to achieve higher throughput.

4.3 Parallel Postmark

We measured Story Book’s performance in a worst case scenario where there is a large amount of I/O that consists of provenance generating metadata operations. We parallelized postmark [4] by spawning multiple threads to perform Postmark transactions. Each thread is its own process. We initialized Postmark’s number of directories to 890, its number of files to 9,000, its number of transactions to 3,600, its average size of file to 190 KiB, its size of read or write to 28KB, and its number of threads to 30. These numbers are based on the arithmetic mean of file system size from Agarawal’s 5-year study [1]. We scaled Agarawal’s numbers down by 10 as they represented an entire set of file system modifications, but the consistent scaling across all values maintains their ratios with respect to each other. We configured the buffer size of Rose to be 39 MiB as this provides a good cache hit rate. In Figure 5 *FUSE-passthru* is a pass-through *FUSE* file system that simply forwards requests from the kernel to the underlying file system. It measures just the overhead of message passing to *FUSE*. We ensured our benchmark is I/O intensive so as to accurately measure the effect of recording provenance when the system is

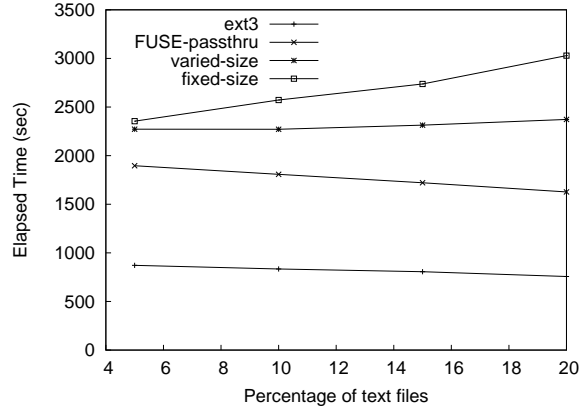


Figure 6: Performance of Story Book compared to *ext3* and *FUSE* for I/O intensive file system workloads.

under heavy load. *Fuse-passthru* refers to a *FUSE* file system that simply forward requests to the file system. *SB-intree-50* refers to Story Book running the same benchmark with its in-memory tree component set to 50 MiB, and the other labels for Story Book are set accordingly. Story Book has a 10% overhead on top of *FUSE-passthru*. *FUSE-passthru* has such a high overhead (123%) on top of *ext3* because each thread must synchronously wait on every message sent to the file system and therefore the *FUSE* daemon. The performance of *FUSE-passthru* establishes an upper bound on Story Book’s performance. Story Book’s overhead on top of *ext3* is 146% for an in-memory tree of 50 MiB. Story Book’s overhead with an in-memory tree of 50 MiB and 200 MiB is statistically indistinguishable.

In another benchmark shown in Figure 6 based on Postmark we measure the cost of storing application-specific provenance in the form of external records. We increase the percentage of *.txt* files in the working set from 5% to 20%. These numbers are based on the cumulative distribution function in Agarawal’s 5-year study. Since Story Book tracks full version differences for *.txt* files it must perform a diff and several hashes after each close of each file, and then append an extended record with this information to Story Book’s extended record log. *Varied-size* represents a Story Book system performing the benchmark with all *.txt* files equal to 8 KiB in size, and all other files uniformly distributed between 180 KiB and 200 KiB. *Fixed-size* represents a Story Book system performing the benchmark with all files uniformly distributed from 180 KiB to 200 KiB. All other systems are run on a working set equivalent to that used in *varied-size*. We see that because an increasing percentage of files is becoming smaller the creation phase of Postmark finishes more quickly and *ext3* and *FUSE-passthru* consequently start doing better. Both *varied-size* and *fixed-size* start doing worse be-

	Regular Response Time (s)	Story Book Response Time (s)
Delivery	69.967	73.483
New Order	60.6262	65.5924
Order Status	59.2852	64.3684
Payment	58.8362	63.9602
Stock Level	68.6076	75.189
Throughput (tpmC)	167.528	160.694

Table 1: Average response time for TPC-C Benchmark

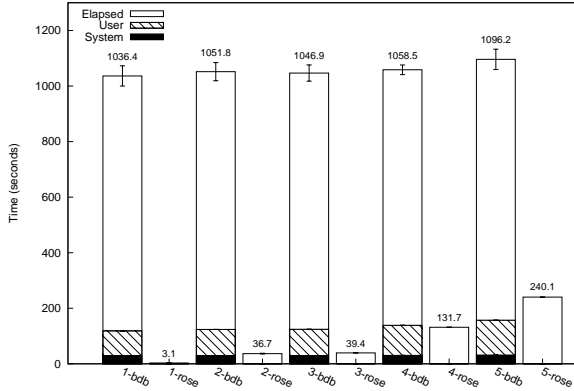


Figure 7: Performance of a provenance query.

cause each Postmark appends around 28 KiB on average and Story Book must still perform diffs and checksums, but Story Book does better with the smaller .txt files in varied-size than in the larger .txt files of fixed-size

4.4 MySQL TPC-C

We also measure the provenance logging performance of an application that does not rely on the file system for storing user data. In this benchmark the size of Rose’s in-memory tree (5 MiB) and buffer (39 MiB) is 44 MiB. We implemented a tool to read the general query log that is provided by MySQL. The tool notes who read from and wrote to what tables and translates this into Story Book’s provenance schema by treating tables as files, and user queries as processes (named by their query text). We measure the performance impact of our provenance tracking system on MySQL using the TPC-C [14] benchmark. We set up TPC-C with 50 warehouses and 20 clients. In Table 1, we notice that the response time for all table accesses has increased by an average of 8% and our total throughput has dropped by 4%. Rapid numbers of queries induce additional memory overhead and additional write-ahead log writes since Story Book is also logging provenance data in addition to what work MySQL is doing.

4.5 Provenance Read

To evaluate Story Book’s query performance, we inserted the exact same record format used by Story Book into a BDB database. Our BDB database is organized to closely mimic our Story Book database so that BDB can exploit any available performance advantages from maintain simpler on-disk data structures. In this benchmark the size of both BDB’s cache and Rose’s in-memory tree (5 MiB) and buffer (39 MiB) is 44 MiB. We created our Rose database by running a provenance workload on Story Book. We created our BDB database by replaying our write-ahead log (which wasn’t truncated) into our BDB back end. We randomly selected a sequence of 500 files from our working set used to generate our databases. We perform just the first 100 queries in our first data point, then the first 200 queries and so on.

As the size of the database grows, the number of random seeks followed by a serial read increases. Since the cache is much smaller than the size of the dataset, performance is closely related to the layout of the data on disk so we see large I/O wait times. Rose competes favorably with BDB because its data is stored sequentially by key and can be transferred into RAM with a serial read, as is BDB’s. However Rose stores its data in a compressed, fragmentation-free format which allows it to read it in at a higher throughput for these query sets. The size of the dataset read into BDB was 1.664 GiB, and resulted in a BDB database size of 2.6 GiB. Rose’s final database size was 598 MiB due to compressing records before writing them out to increase throughput. When performing the entire sequence of 500 queries, we see BDB is 4.5 times slower.

5 Conclusions

Our design of Story Book focused on minimizing the amount of state held in RAM and on optimizing writes. We are able to compress records to efficiently store exact provenance information, and support a straightforward record-suppression scheme that does not maintain a graph in RAM. Our efficient index structures allow us to provide superior query performance despite the simplicity of our provenance data model. This was made possible by applying log structured merge trees and compression techniques to logging provenance data and providing a flexible application development framework.

Story Book’s primary overhead comes from its FUSE filesystem instrumentation layer. Our experiments show that Story Book’s underlying provenance database provides much greater throughput than necessary for most filesystem provenance workloads. This, coupled with Story Book’s extensibility, allow it to be applied in more

demanding environments such as databases and other multiuser applications.

References

- [1] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. In *FAST '07: Proc. of the 5th USENIX conference on File and Storage Technologies*, pp. 3–3, Berkeley, CA, USA, 2007.
- [2] Dublin Core Metadata Initiative. Dublin core. dublincore.org, 2008.
- [3] J. Clark et al. ExPat. expat.sf.net, Dec 2008.
- [4] J. Katcher. PostMark: A new filesystem benchmark. Technical Report TR3022, Network Appliance, 1997. www.netapp.com/tech_library/3022.html.
- [5] K. Muniswamy-Reddy, J. Barillari, U. Braun, D. A. Holland, D. Maclean, M. Seltzer, and S. D. Holland. Layering in provenance-aware storage systems. Technical Report TR-04-08, Harvard University Computer Science, 2008.
- [6] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996.
- [7] R. Sears and E. Brewer. Stasis: Flexible Transactional Storage. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, Nov. 2006.
- [8] R. Sears, M. Callaghan, and E. Brewer. Rose: Compressed, log-structured replication. In *Proc. of the VLDB Endowment*, volume 1, Auckland, New Zealand, 2008.
- [9] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A survey of data provenance in e-science. *ACM SIGMOD Record*, 34(3):31–36, Sept. 2005.
- [10] R. P. Spillane, S. Gaikwad, E. Zadok, C. P. Wright, and M. Chinni. Enabling Transactional File Access via Lightweight Kernel Extensions. In *Proc. of the Seventh USENIX Conf. on File and Storage Technologies*, San Francisco, CA, Feb. 2009. to appear.
- [11] R. P. Spillane, C. P. Wright, G. Sivathanu, and E. Zadok. Rapid file system development using ptrace. In *Proc. of the Workshop on Experimental Computer Science, in conjunction with ACM FCRC*, page Article No. 22, San Diego, CA, Jun. 2007.
- [12] Sun Microsystems. MySQL. www.mysql.com, Dec 2008.
- [13] M. Szeredi. Filesystem in Userspace. <http://fuse.sf.net>, Feb. 2005.
- [14] Transaction Processing Performance Council. TPC Benchmark C, Standard Specification. www.tpc.org/tpcc, 2004.
- [15] W3C. Resource description framework. w3.org/RDF, 2008.
- [16] Jennifer Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *Proc. of the 2005 CIDR Conf.*, 2005.