Journaling of Journal Is (Almost) Free

Kai Shen Stan Park* Meng Zhu
University of Rochester

Abstract

Lightweight databases and key-value stores manage the consistency and reliability of their own data, often through rollback-recovery journaling or write-ahead logging. They further rely on file system journaling to protect the file system structure and metadata. Such *journaling of journal* appears to violate the classic end-to-end argument for optimal database design. In practice, we observe a significant cost (up to 73% slowdown) by adding the Ext4 file system journaling to the SQLite database on a Google Nexus 7 tablet running a Ubuntu Linux installation. The cost of file system journaling is up to 58% on a conventional machine with an Intel 311 SSD.

In this paper, we argue that such cost is largely due to implementation limitations of the existing system. We apply two simple techniques—ensuring a single I/O operation on the synchronous commit path, and adaptively allowing each file to have a custom journaling mode (in particular, whether to journal the file data in addition to the metadata). Compared to SQLite without file system journaling, our enhanced journaling improves the performance or incurs minor (<6%) slowdown on all but one of our 24 test cases (with 14% slowdown in the exceptional case). On average, our enhanced journaling implementation improves the SQLite performance by 7%.

1 Introduction

Ensuring data consistency and durability despite sudden system crashes (due to power/battery outages and software panics) is a critical aspect of computer system dependability. Following such failures, the application and system data should be recovered from durable storage to a consistent state without unexpected data losses. With data-driven applications spreading from servers and desktops to resource-constrained devices and near-client cloudlets [17], the overhead of such protection is also an important concern. Many applications utilize lightweight databases and key-value stores to manage their data. For instance, the SQLite database [19] and Kyoto Cabinet [3] protect the consistency of their data through techniques such as rollback-recovery journaling or write-ahead logging. Transactions are flushed to durable storage at commit time to prevent the loss of data.

These data management applications run on traditional file systems. While file system journaling protects the file system structure and metadata, it lacks knowledge of the application's semantics to protect application data. Such *journaling of journal* appears to violate the classic end-to-end argument that the low-level implementation of a function (transactional data protection in this case) is incomplete and it can hurt the overall system performance [16, 21].

We argue that the cost of additional file system journaling is largely due to implementation limitations of the current system. Simple enhancements can be made following two principles—1) the critical path of a synchronous I/O commit should involve a single sequential I/O operation to the storage device, 2) different application-level log files should be allowed to use customized file system journaling modes that best match their respective access patterns. This paper describes the design and implementation of our techniques on the Ext4 file system, as well as a performance evaluation with the SQLite database on a Google Nexus 7 tablet and a conventional machine with a NAND Flash-based SSD.

We note that the journaling of journal can be avoided if the operating system provides a richer interface that allows application data protection semantics to be exposed to the OS. The OS can then provide unified data protection for both application data and the file system metadata. Examples of such enhanced OS data management interface include I/O transactions [13, 18], software persistent memory [4], and failure-atomic msync() [11]. In contrast to these approaches, we explore performance enhancements of the existing file system journaling with only minor addition to its semantics and usage, which can be more easily deployed in practice.

The issue of journaling of journal has been raised before, notably in the context of Android's employment of the SQLite database on smartphones [6, 7]. While our work was partially motivated by these studies, our own observation in limited experimental setups suggest that the performance of many typical Android smartphone workloads is dominated by network (particularly wide-area network) delay rather than storage I/O. Although we make no argument on the importance of smartphone storage I/O performance in this paper, we do caution that the performance findings of our work only applies to I/O-intensive workloads in server/cloud and client systems.

*Park is currently affiliated with HP Labs.

For clarity, we will call an application-level journal a *log file* in the rest of this paper and the term *journaling* will refer to the file system journaling by default.

2 Fast Journaling of Journal

When an application-level transaction commits, its REDO or UNDO information is synchronously written to a log file. The atomicity of such a commit is often ensured through a checksum code written together with the committed transaction. A rollback-recovery log records transaction UNDO information. Updates on the database file(s) occur after the log write but before the transaction completes. Alternatively, write-ahead logging records transaction REDO information. Since the REDO record contains sufficient information to keep the database up-to-date, updates to the database file(s) do not need to happen before the transaction commits. In fact they can be delayed until they are overwritten by future transactions and therefore never written to durable storage.

Write operations are particularly expensive on NAND Flash-based storage devices [1, 12]. File system journaling protects the file system structure and metadata which can incur additional I/O costs in two ways. First, it may increase the number of I/O operations. For instance, the Ext4 ordered-mode journaling only journals the file system metadata but for consistency, it must write the file data before journaling the metadata. Second, it may increase the write size. For instance, the Ext4 data-mode journaling may write twice as much as the application does (once to the journal and a second time to the main file system structure). Furthermore, the journal data writes can be substantially larger than the original data writes due to the minimal journal record granularity (e.g., a 4 KB page).

However, we show that the cost of file system journaling can be mitigated through simple implementation enhancements. Our techniques do not change the existing journaling semantics of protecting the file system metadata and require very little application modification.

2.1 Single-I/O Data Journaling

Under journaling of journal, the data management application protects the consistency of its own data while file system journaling protects the file system integrity. However, the journaling of both metadata and file data (e.g., Ext4 data-mode journaling) allows a single sequential I/O operation on the critical path of a synchronous I/O commit (a `fsync()`, `fdatasync()`, or `msync()`) and therefore may enhance performance.

To accomplish single-I/O data journaling, the operating system should avoid direct file data or metadata writes on the synchronous I/O commit path. This principle is compatible with data journaling semantics since the journaled content contains sufficient information to recover the committed data in the event of a crash. To ensure it on Linux/Ext4, we examined its I/O traces and inspected its file sync code paths which led us to identify a violation in its current implementation. Specifically, the `ext4_sync_file()` function flushes the dirty file buffer (through `filemap_write_and_wait_range()`) before writing the journal. Such flushes skip buffers ready for data journaling but still write previously journaled dirty data buffers (resultant from earlier transactions). These file buffer writes are unnecessary for the journaling semantics and they slow down the synchronous commit. We make corrections to avoid these writes under data journaling.

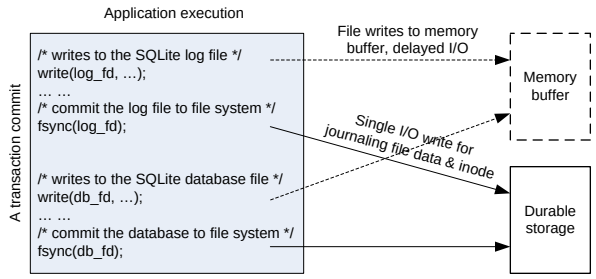

Figure 1: A simplified illustration of the journaling of journal during a transaction commit—file system journaling of the SQLite database that employs rollback-recovery logging.

The single I/O operation in the synchronous commit path only writes the file data and metadata to the file system journal. They would in theory still need to be asynchronously checkpointed to the main file system structure. In practice, the asynchronous checkpoints may never happen for data and log files when they are overwritten or deleted (for some UNDO logs) during continuous transaction processing. For instance, Figure 1 provides a simplified segment of system call trace during a transaction commit for the SQLite database that employs rollback-recovery logging. For the log file, the `fsync()` call would trigger a single I/O operation that commits the file data and metadata to the file system journal. Checkpoints to the SQLite log file itself are delayed and then typically canceled when the log file is deleted after the database transaction commits. Memory buffering and journaling for the database file writes behave similarly, in that the delayed checkpointing writes may be overwritten by a later transaction that writes to the same database location.

We also note that a traditional file system journaling transaction commit involves two write operations—first synchronously writing the transaction content and then writing a transaction commit record—to ensure atom-

icity. Alternatively, single-I/O atomic transaction commit can be accomplished by adding a checksum in the transaction block. In Ext4, this is supported by the `journal_async_commit` option. This option is not used widely in production systems due to a challenge of handling corrupted checksums in journal recovery [22]. There is also a concern in the case of ordered journaling that, with `journal_async_commit`'s removal of the I/O flush before writing the transaction commit record, the ordering between data writes and metadata journaling commits may not be enforced [23].

2.2 File-Adaptive Journaling

While single-I/O journaling of both file data and metadata minimizes the number of I/O operations during a synchronous commit, it may write a larger volume of data than the original write. This is primarily due to the typical journaling implementation's use of whole pages for journaling records (including Ext4/JBD2). Under such an implementation, a small write of a few bytes still requires a 4 KB journal record. This presents an efficiency dilemma between the single-I/O full data/metadata journaling and ordered metadata-only journaling—the former allows a single I/O operation on the commit path but may write significantly more data. Note that the single-I/O data journaling described in Section 2.1 is an important foundation for this dilemma because otherwise the ordered journaling would almost always outperform the standard data journaling.

In earlier work, Prabhakaran et al. [14] have proposed an adaptive journaling approach that selects the best journaling mode for each transaction according to its I/O access pattern. Specifically, a transaction that would perform sequential I/O under ordered journaling mode should utilize ordered journaling. Such a transaction typically writes a contiguous set of data blocks without making any file metadata change. On the other hand, a transaction that contains multiple I/O segments or performs both data and metadata writes should use full data/metadata journaling to gain the efficiency of a single, sequential I/O operation.

However, per-transaction adaptive journaling may not be safe in the case of overwrites. Specifically, consider two transactions T_1 and T_2 (in that order) that overlap in their file data writes. Assume that the adaptive journaling system chooses the full data/metadata journaling for T_1 while it selects the ordered journaling for T_2. A crash-induced journal replay will apply the metadata and file data for T_1 but only the metadata for T_2, and thus incorrectly leaving T_1's file data write as the final state. This effectively reorders writes in the two transactions that would not have happened under the standard (non-adaptive) full data journaling or ordered journaling. This problem may be resolved by a recent technique [2, Section 4.3.6] that journals data buffer overwrites for transactions which utilize ordered journaling.

Another concern for per-transaction adaptive journaling is its implementation challenge. Specifically, since the relevant transaction characteristics for adaptive journaling decision (e.g., sequential I/O or not) is not known at the beginning of a transaction, writes at the early stage of a transaction would have to be done in a way that permits either data or ordered journaling. This presents new challenges to a typical journaling implementation that performs a write differently depending on the journaling mode. For instance, a write under data journaling must prohibit dirty data flushes until the transaction is committed to the journal while a write under ordered journaling flushes dirty data earlier. Furthermore, a write under data journaling records data writes in the journal while a write under ordered journaling does not.

To avoid these safety and implementation complications, we propose a coarse-grained (per-file) adaptive journaling approach. As long as all journal transactions to a given file follow a single journaling mode (either full data/metadata journaling or ordered metadata-only journaling), a crash-induced journal replay should always result in the correct final state for the file data. Furthermore, we let the application to set the journaling mode for a file according to its access characteristics. This approach works well for practical journaling of journal circumstances. Specifically, write-ahead log files are generally written sequentially with little metadata change and therefore more suitable for the ordered file system journaling. In contrast, a rollback-recovery log file deletes or truncates the transaction record at the end of each transaction and therefore triggers substantial metadata changes. Consequently it is more suitable for full data/metadata file system journaling.

The proposed file-adaptive journaling can be easily implemented in practice. Changes to Linux/Ext4 primarily include new file and inode flags set through `ioctl()` that indicate the desired journaling mode for each file. The journaling code will preform appropriate actions for a file according to its inode flag. For safety, changing the journaling mode for a file requires the flushing of all pending journal transactions. In practice, such flushing incurs little overhead if the journaling modes are set once at database open time for files that persist through a long work session (e.g., the write-ahead log files). Our changes to the SQLite database involve adding about 20 lines of C code right after each log file open.

There is one additional correctness concern for per-file adaptive journaling in the case of data reuses between files. Specifically, if a data block is freed from a data-journaled file (after transaction T_1) and then reused by an ordered-journaled file (followed with transaction T_2),

then the block may be journaled in T_1 but not in T_2. Therefore our earlier example of journal replay-induced write re-ordering may again appear. In fact, similar block reuse problems already exist in the standard ordered journaling, when a metadata block is freed from a file and then reused by another file as a data block [24]. The standard solution to such a problem is to use journal revoke records that instruct the journaling replay mechanism to avoid replaying the concerned blocks. This has not yet been implemented in our current prototype system.

3 Experimental Evaluation

We implemented our journaling enhancements in the Linux 3.1.10 kernel and its Ext4 file system. We performed experiments on a Google Nexus 7 tablet with internal eMMC NAND storage. The Nexus 7 has a quad-core dynamic-frequency Tegra3/ARM processor, which was fixed to run at 1.0 GHz during our experiments. Our Nexus 7 is installed with the Ubuntu 12.10 Linux distribution. We also experimented with a conventional machine with an Intel 311 Flash-based SSD and two dual-core 2.0 GHz Intel processors. In each platform, we measured the transaction performance of the SQLite database (version 3.8.0.2).

We configured the system and application software to optimal-performance settings to establish a strong baseline. The Ext4 file system is mounted with `noatime,nodiratime,journal_async_commit` options. We also enable the `barrier=1` option to ensure the journal write ordering. We configured SQLite to use `fdatasync()` (instead of `fsync()`) for I/O commits. Note that `fdatasync()` will still write the dirty inode if the inode change affects the file system structural integrity (e.g., file size change).

3.1 Performance of File System Journaling

We compare the performance of different file system journaling approaches. Note that our file-adaptive journaling builds on our single I/O data journaling as described at the beginning of Section 2.2.

Our workloads run insert, update, and delete operations on a SQLite database table. Each record has an integer key and a 100-character value field. We test two transaction sizes—small, one-operation transactions and large, 1000-operation transactions. We also experiment with two application-level logging approaches in SQLite—write-ahead logging (WAL) and rollback-recovery logging. WAL is faster at transaction commits by issuing fewer `fdatasync()`'s (once vs. four times under the rollback-recovery logging), but it also has a number of practical disadvantages including poor support for reads and requiring periodic checkpoints [20].

Our test runs in rounds. Each round starts with an empty database table, inserts 10,000 records (10,000 one-operation transactions or ten 1000-operation transactions), updates the 10,000 records (in a random order that is likely different from the insert order), and deletes the 10,000 records (again in a random order). For each test case we run at least five rounds and report the average transaction response time.

Figures 2 and 3 illustrate performance results on our two platforms. Compared to no file system journaling, Ext4 ordered journaling has a worst-case slowdown of about 20%. Ext4 data journaling has a worst-case slowdown of about 73%. These indicate substantial costs of journaling of journal systems. Our single-I/O data journaling has enhanced the performance from the original Ext4 data journaling but it still suffers from a worst-case slowdown of 38%.

Our file-adaptive journaling has no more than 6% transaction slowdown compared to no file system journaling in all but one of our 24 test cases. It experiences 14% slowdown under the exceptional case of small (1 op/txn) update transactions with SQLite rollback-recovery logging on Nexus 7. A closer inspection finds that the overhead is due to page-granularity file system journal records and resultant larger write volume under full data/metadata journaling for the SQLite rollback-recovery log file. In one representative journal commit, nine small writes of a total 2,576 bytes turned into nine 4 KB pages (along with inode journaling and commit block, totaling 45,056 bytes) on the file system journal in Ext4/JBD2. We note that the page-granularity record is not a necessary design choice for file system journaling and previous work has shown that fractional-page journaling record is possible [5]. We expect that such an enhancement, if robustly incorporated into file system journaling, should further strengthen our argument that the journaling of journal is (almost) free.

We also find that our single-I/O data journaling and file-adaptive journaling achieve faster transaction responses than no file system journaling in some cases. This is most pronounced for large (1000 ops/txn) insert transactions with SQLite rollback-recovery logging on both machines. This is due to the substantial benefit of coalescing multiple piecemeal writes into a single I/O operation on the synchronous commit path. On average of all our test cases over different workload patterns and SQLite logging modes, our file-adaptive journaling in fact improves the performance of no file system journaling by 4% on Nexus 7, and by 9% on Intel 311.

Error bars in Figures 2 and 3 show the standard deviations of results from multi-round tests in each case. Most tests show stable results. The most deviating performance results do not always reoccur under a given test condition, which hints at the possible effect of occasional

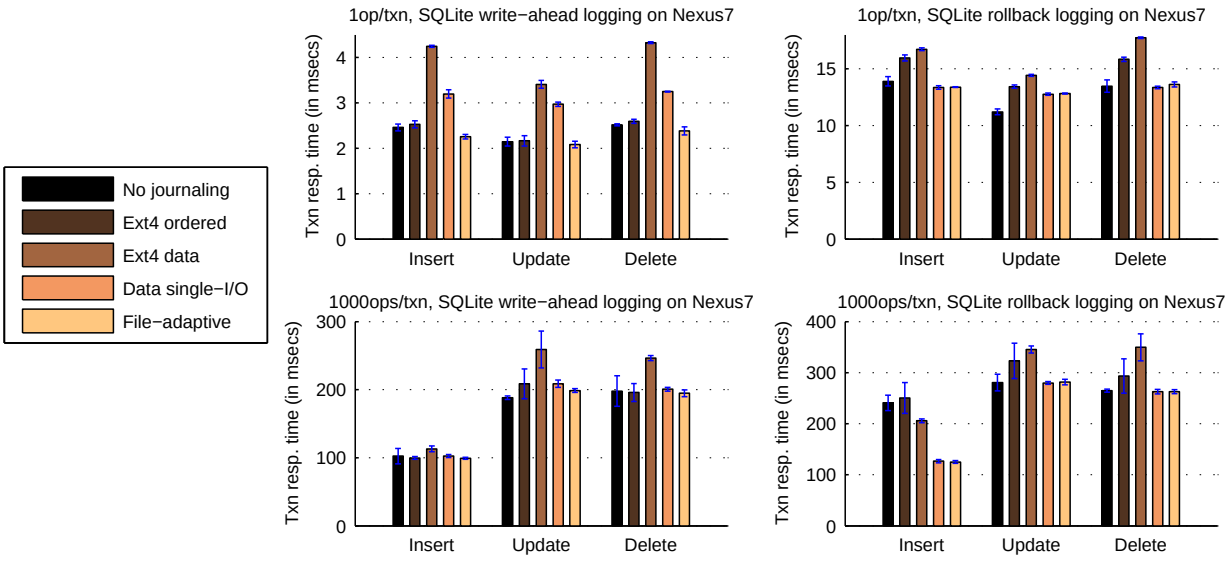

Figure 2: SQLite transaction response time under different file system journaling approaches on Nexus 7. The error bars show the standard deviations of results from multi-round tests in each case. The two columns show results for two SQLite logging modes (write-ahead logging and rollback-recovery logging). The two rows show results on two transaction sizes (1 operation per transaction and 1000 operations per transaction).

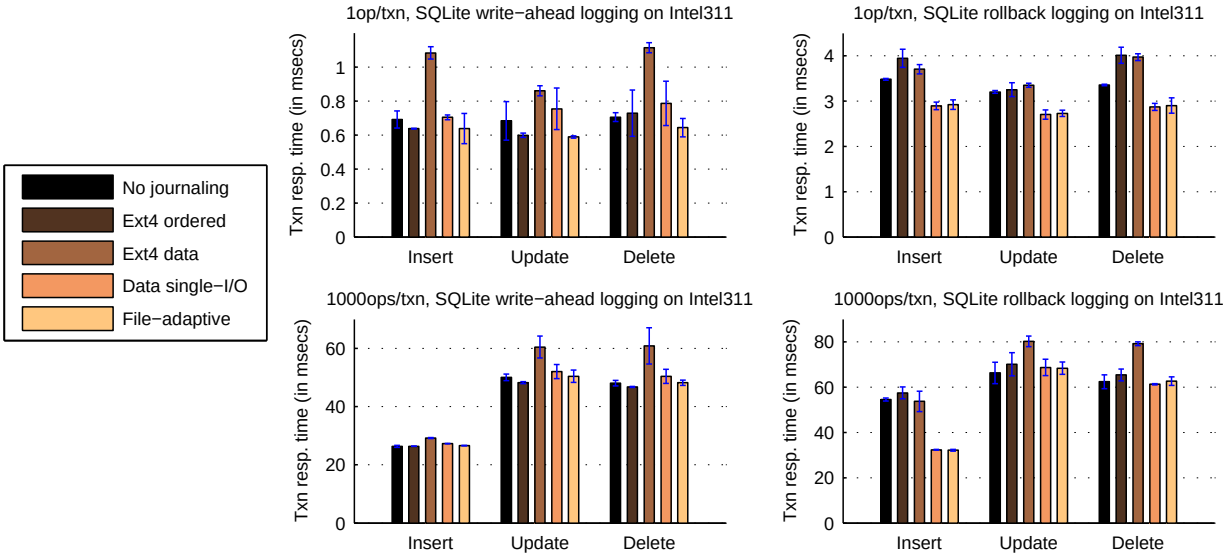

Figure 3: Performance results on a conventional machine with an Intel 311 SSD.

background Flash tasks such as garbage collection.

3.2 Cost of Transactional Data Protection

While we argue that properly adding file system journaling to applications that protect their own data incurs little additional cost, we are not suggesting that transactional data protection (ensuring data consistency and durability) over system failures is free. To reveal the cost of transactional data protection, we compare three system conditions on our experimental platforms—

- *Full protection*: SQLite write-ahead logging (application-level protection) combined with Ext4 file-adaptive journaling (file system protection).
- *Application-level protection*: SQLite write-ahead logging without any file system journaling.
- *No protection*: no SQLite logging or file system journaling; all writes are asynchronous and almost all operations are performed entirely in memory.

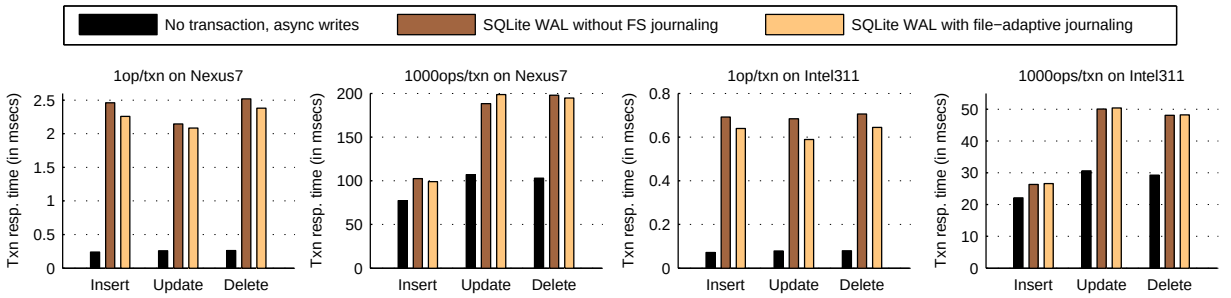

Figure 4: Cost of transactional data protection—performance comparison of application/system-level data protection mechanisms against the situation of no protection (effectively in-memory operations).

Results in Figure 4 show that in all cases, the full data protection with file system journaling adds almost no cost beyond application-level protection. Compared to no data protection with effectively in-memory operations, the transactional data protection and associated atomic, synchronous I/O adds substantial costs for small transactions (almost an order of magnitude slowdown). However, the costs are not excessive for large transactions (less than a factor of two). Larger transactions can be accomplished through careful application development and tuning, or system approaches that intelligently commit only when necessary [10].

4 Conclusion and Discussions

This paper argues that properly adding file system journaling to applications that protect their own data incurs minor additional cost. The large costs for journaling of journal on existing systems are primarily attributable to implementation limitations. We applied simple techniques that ensure a single I/O operation on the synchronous commit path and adaptively allow each file to have its custom file system journaling mode. We performed experiments for SQLite transactions on a Google Nexus 7 tablet and a conventional machine with an Intel 311 SSD. Compared to no file system journaling, our enhanced journaling implementation improves the performance or incurs minor (<6%) slowdown on all but one of our 24 test cases (with 14% slowdown in the exceptional case). On average, our enhanced journaling implementation improves the SQLite performance by 7%.

Our work contributes to the debate between using traditional vs. log-structured file systems [15] on NAND Flash-based systems. While a number of log-structured file systems (including NILFS [9] and F2FS [8]) have emerged recently and promise high performance on NAND storage, many systems have still chosen the Ext4 file system due to the concern of system maturity and robustness for production use. Our work suggests that such a choice does not necessarily carry significant performance costs.

Our proposed file-adaptive journaling is just one form of possible adaptive journaling approaches [14]. It has limited adaptation granularity (all transactions on a file must use a single journaling mode) but is effective for application-level log files that exhibit clear I/O patterns and journaling preferences. A broader use of adaptive journaling may require intelligent learning of the file access pattern and finer-grained control. Beyond adaptive journaling, journaling performance can be further enhanced by relaxing its ordering constraints [2].

Our cost evaluation has targeted the application response time. At the same time, we recognize that the employment of full data and metadata journaling tends to write more to a NAND Flash device and increase its wear. We explained in the paper that such increase of the write volume is primarily due to the page-granularity records in the journaling file system implementation. This can be potentially mitigated by a fractional-page journaling implementation [5].

Experimental work in this paper has focused on systems with NAND Flash-based storage. While much of our design rationale should also apply to mechanical disks, the resulted quantitative benefits might be different. In particular, since seek time dominates the performance of a mechanical disk, our single-I/O data journaling (Section 2.1) may produce larger performance gains than on Flash storage. On the other hand, the benefit of write size reduction under file-adaptive journaling (Section 2.2) might be much smaller on mechanical disks.

Acknowledgments This work was supported in part by the National Science Foundation grants CCF-0937571, CNS-1217372, CNS-1239423, and CCF-1255729, and by a Google Research Award. We thank Ted Ts'o for clarifying relevant issues in the Ext4 file system and its journaling support. We also thank the anonymous FAST reviewers and our shepherd Florentina Popovici for comments that helped improve this paper.

References

[1] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of Flash memory based solid state drives. In *ACM SIGMETRICS*, pages 181–192, Seattle, WA, June 2009.

[2] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic crash consistency. In *SOSP'13: 24th ACM Symp. on Operating Systems Principles*, pages 228–243, Farmington, PA, Nov. 2013.

[3] FAL Labs. Kyoto Cabinet: a straightforward implementation of DBM. `http://fallabs.com/kyotocabinet/`.

[4] J. Guerra, L. Mármol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei. Software persistent memory. In *USENIX Annual Technical Conf.*, Boston, MA, June 2012.

[5] A. Hatzieleftheriou and S. V. Anastasiadis. Okeanos: Wasteless journaling for fast and reliable multistream storage. In *USENIX Annual Technical Conf.*, Portland, OR, June 2011.

[6] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won. I/O stack optimization for smartphones. In *USENIX Annual Technical Conf.*, San Jose, CA, June 2013.

[7] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting storage for smartphones. In *FAST'12: 10th USENIX Conf. on File and Storage Technologies*, San Jose, CA, Feb. 2012.

[8] J. Kim. F2FS: Introduce Flash-friendly file system, Oct. 2012. `https://lwn.net/Articles/518718/`.

[9] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai. The Linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, July 2006.

[10] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. *ACM Trans. on Computer Systems*, 26(3), Sept. 2008.

[11] S. Park, T. Kelly, and K. Shen. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *EuroSys'13 Conf.*, Prague, Czech Republic, Apr. 2013.

[12] M. Polte, J. Simsa, and G. Gibson. Comparing performance of solid state devices and mechanical disks. In *3rd Petascale Data Storage Workshop*, Austin, TX, Nov. 2008.

[13] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating system transactions. In *SOSP'09: 22th ACM Symp. on Operating Systems Principles*, pages 161–176, Big Sky, MT, Oct. 2009.

[14] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *USENIX Annual Technical Conf.*, Anaheim, CA, Apr. 2005.

[15] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. on Computer Systems*, 10(1):26–52, Feb. 1992.

[16] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. on Computer Systems*, 2(4):277–288, Nov. 1984.

[17] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for VM-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, Oct. 2009.

[18] R. Sears and E. Brewer. Stasis: Flexible transactional storage. In *OSDI'06: 7th USENIX Symp. on Operating Systems Design and Implementation*, Seattle, WA, Nov. 2006.

[19] SQLite. `http://www.sqlite.org/`.

[20] SQLite Write-Ahead Logging. `http://www.sqlite.org/wal.html`.

[21] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.

[22] T. Ts'o. What to do when the journal checksum is incorrect, May 2008. `http://lwn.net/Articles/284038/`.

[23] T. Ts'o. Personal communication, Jan. 2014.

[24] S. Tweedie. EXT3, journaling filesystem. In *Ottawa Linux Symposium*, July 2000. `http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html`.