Hybrid Transaction Recovery with Flash

Tyler Harter and Heemoon Chae

December 19, 2010

Abstract

In order to satisfy the atomicity and durability requirements of the ACID properties, databases need to be robust against system crashes and other failures. Some early systems used shadowing to meet these requirements, but modern systems typically prefer ARIES-style write-ahead logs (WALs). We believe the emergence of flash has invalidated some, but not all, of the reasons ARIES is typically chosen over shadowing. Thus, we propose a modification to standard WALs that employs shadowing when appropriate. We implement both ARIES and hybrid prototypes and experimentally consider which system should be used for various types of workloads.

1 Introduction

In database systems, serializable transactions must have four properties: atomicity, consistency, isolation, and durability (ACID). The recovery component of a database management system is responsible for atomicity and durability. The recovery component will guarantee these properties even when transactions are aborted or the system crashes.

The two main recovery approaches that have been used are ARIES and shadowing. ARIES uses a write-ahead logging protocol for data recovery. To fulfill atomicity, it writes "before images" of updated pages to a log so that transactions that fail after just making partial changes can be undone. For durability, ARIES writes all the "after images" for the new data that is part of a transaction to a log so that modification to pages that have not yet been flushed from the buffer pool can be recovered. A significant disadvantage of ARIES logging is that there will often be a 200% overhead, as ARIES will have to write the new version of a page to its appropriate location, an old version of the page to the log (before image), and a new version of the page to the log (after image).

Shadowing is the second option for achieving atomicity and durability. Instead of performing in-place updates, it writes new versions of pages to new locations, leaving old versions intact. Atomicity is achieved since references to the page can be atomically changed from the old versions of the pages to the new versions of the pages. Durability is achieved because the actual data pages are forced to disk (this is in contrast with the ARIES approach which forces the log but not the data pages to disk). The advantage of shadow paging is that it requires less data transfer than a log writing scheme (one simply needs to write a new version of the page without recording before and after images).

Unfortunately, shadowing is rarely used by database management systems because of two major weaknesses. First, logical pages are written to new physical locations every time they are updated, so random updates will cause logically adjacent pages to become scattered at non-adjacent locations on disk. This results in poor performance for rotational disks which achieve much greater throughput for sequential workloads than they do for random workloads. Second, new data is written to new pages, so it is not possible to update less than an entire page. Thus, updaters must lock entire pages. This prevents multiple transactions from concurrently updating different records within the same page.

The first problem, data scattering, can be addressed with new hardware. With the emergence of

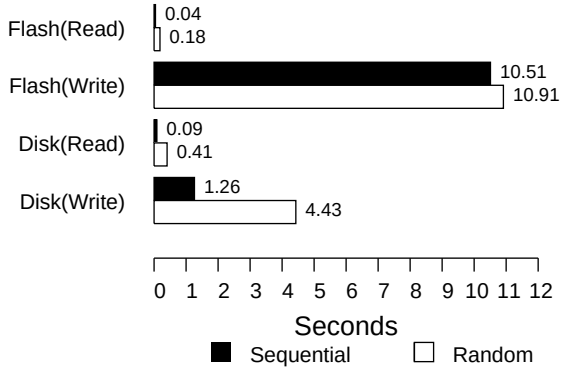

Figure 1: Random access for flash and disks. The times are for 1000-page accesses on a file (Ext2 file system). For larger files, the disk should have even worse random access times while the flash access times should remain relatively constant. Read access that is sequential is faster for flash, but this likely due to the prefetch behavior of the SSD.

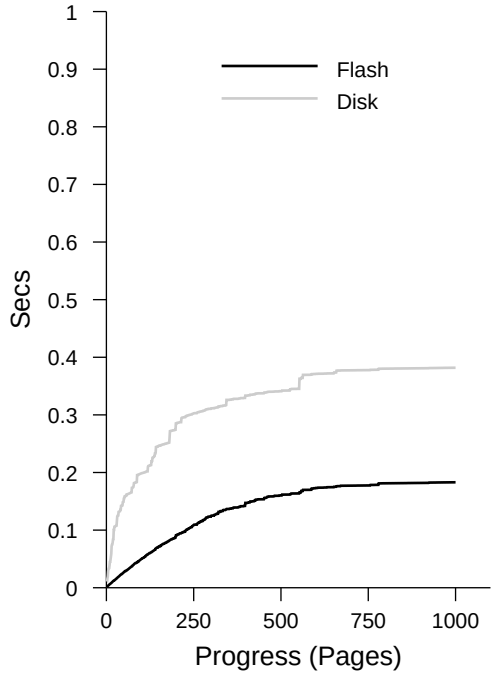

Figure 2: Random performance.

flash storage, random accesses are much faster than they used to be, so scattering is not nearly as concerning as it used to be. Since there is no actuator arm that has to physically move to the correct track in order to access data, the property of locality is not as relavent for flash as it is for disks. I/O operations, therefore, are not expensive like they are on hard disk drives. Figure 1 shows the relative time required for random and sequential access for both flash and disk.

When files are even larger, the random performance gap between flash and disk becomes much greater because fewer request will be able to be satisfied by device caches. This is illustrated by figure 2 which shows elapsed time as a program progresses through a workload that involves randomly reading 1000 pages. Initially, the graph is steep, so each page read takes a long time. After the caches become warm, the read times becomes almost negligable for both devices. Before the cache becomes, warm, the disk line is much steeper, so it is clear that random accesses are much worse for disks.

Given that scattering is no longer much of an issue, the only significant disadvantage of shadowing relative to logging is limited concurrency. Therefore, we propose a hybrid recovery model in this paper that allows concurrency for sub-page updates but uses a more efficient shadowing approach for full page updates where concurrency is not possible anyway. In section 2 we describe our model. In section 3 we describe a prototype we implemented of the hybrid system, and in section 4 we report the results of running a variety of workloads on our prototype. Section 5 discusses related work and section 6 concludes.

2 A Hybrid Recovery Model

We see the two primary disadvantages of traditional shadowing as (1) scattering leading to slow random access and (2) an inability for multiple transactions to concurrently modify different parts of the same page. We suggest that (1) be addressed by using flash storage rather than hard disk storage. Flash is fast at random accesses, and in-place updates are slow, so shadowing is already employed by flash storage (either by a flash file system, or by a hardware

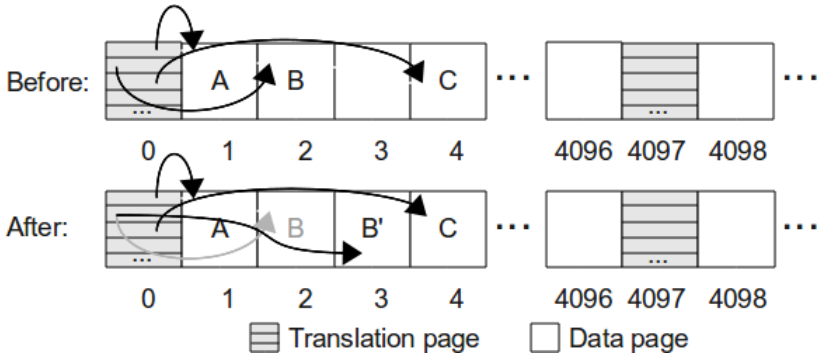

Figure 3: Translation table.

flash translation layer built into a SSD). We believe (2) can be addressed by using a hybrid between logging and shadowing. If (1) does not matter because flash is being used, shadowing clearly results in significantly less I/O in the case of whole page updates. In the other case where a page is only being partially updated, logging can be used instead of shadowing to allow concurrent access to different parts of the page.

Shadowing requires that logical pages can be stored at different physical locations, so our model requires a translation layer (section 2.1) that maps logical pages to physical pages within a file. We also require a write-ahead log (section 2.2) since we need to be able to handle sub-page updates and also atomically update translations.

2.1 Translation

In our hybrid system, there will be a translation layer between the database and file accesses. The translation layer will allow the database to request pages from files or specify updates to entire pages or parts of pages. The page numbers specified by the database will be logical page numbers, allowing the translation layer to change where the pages are physically stored. The translation layer will rely on a translation table that maps the logical page numbers used by the database to the physical page numbers that allow the pages to be accessed within the file.

The translations are stored in a physical array that contains the mappings. Each page of the array contains 1K translations since pages are 4KB and each logical page locations requires 4 bytes. The pages of the translation array are spaced at regular intervals within the file. This allows constant time lookups since translation pages are at known locations, but it also allows the translation table to be expanded as the file grows. Figure 3 shows the physical format that we use to provide translation. When the "B" page is updated, the new version is written to a new location and the pointer in the translation table is updated.

2.2 Write-ahead log

Our hybrid system also has an ARIES-style page-oriented log. The log records changes to the physical pages of the file. When part of a page is modified, the log will contain an "update" log entry with a before and after image of the modified data. When an entire page is updated, the log will just contain an update record for the translation page since a shadow copy of the data page will be written.

2.3 Commit and Abort

Figure 4 illustrates the differences between ARIES and the hybrid system upon commit. For ARIES, we first need to flush the update record with the before and after images of the data (about 8KB of I/O to the log). After writing the commit log entry, the actual new data (4KB) lazily replaces the old data when the

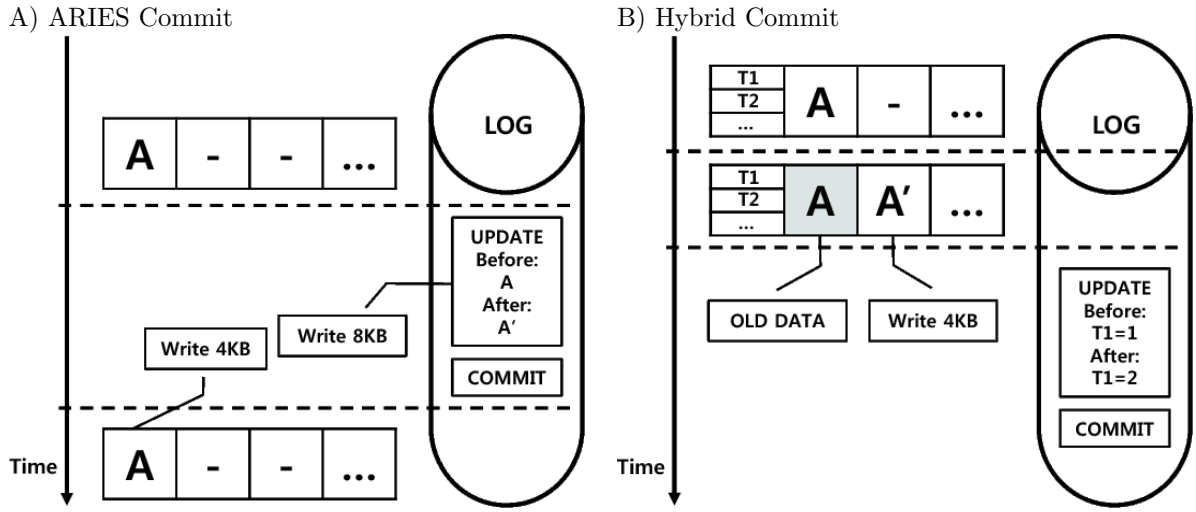

Figure 4: Commit Ordering

page is evicted from the buffer pool. For the hybrid system, we simply write the new data (4KB) to a new location and then flush the update record with the before and after locations of the updated page (negligable I/O).

Aborting in the hybrid system is especially fast because translations simply need to be updated in order to rollback to previous shadow copies. This involves minimal reading and writing.

2.4 Flushing

Table 1 compares the ordering and timing of flushes for ARIES and the hybrid system. In each case, the hybrid has at least as many flush steps as ARIES, but it still has two advantages. First, much less total data is flushed. Second, flushing has the side effect of cleaning pages in the buffer pool that were written as shadows.

3 Hybrid and ARIES Prototypes

In this section we discuss our Hybrid and ARIES prototypes. Section 3.1 describes the interface to the prototypes. Section 3.2 discusses the buffer pool manager we implemented, and section 3.3 discusses the write-ahead log. Section 3.4 describes the code base and also lists things that we left out of our prototypes.

3.1 Interface

The prototypes expose the interface described in table 2. Both prototypes share most of the code, allowing for more fair comparison, but there are a few differences. First, when running under hybrid mode, `lPageNum` for **trans_pinPage**, **trans_unpinPage**, and **trans_update** is translated to a physical page number via the translation table. When running under ARIES mode, there is no translation layer, so `lPageNum` is treated as a physical page number. The second difference occurs when **trans_update** is called with `length` equal to a full page. Under hybrid mode, the logical page `pageno` is assigned a new

System	ARIES			Hybrid		
Event	Commit	Memory log exhausted	Steal	Commit	Memory log exhausted	Steal
Actions	1) Flush log	1) Flush Log	1) Flush Log 2) Flush Page	1) Flush shadows 2) Flush log	1) Flush shadows 2) Flush log	1) Flush shadows 2) Flush log 3) Flush page (if !shadow)

Table 1: ARIES and hybrid flush behavior.

physical page number, but under ARIES the before and after images are written to the in memory log in the normal fashion. The third main difference is that the actions taken when **trans_commit** is called are different as described in section 2.4.

We did not distinguish between pinning new pages and pinning existing pages in our interface. This feature could readily be added as a convenience, but it is not strictly necessary as higher layers that use our system could keep track of file size. When a page that doesn't exist is requested, a new zero page is created and returned.

3.2 Buffer Management

For efficient buffer management, we used a clock-based page replacement algorithm that uses a reference bit to avoid replacing recently used pages. The clock algorithm keeps buffer pages as a circular list and uses a clockhand pointer to point out the buffer which can be flushed next. That is, if there is no empty page and one page must be flushed to a disk, the clock algorithm selects a victim page by the clockhand.

When running the hybrid prototype, the translation pages are managed by the buffer pool just like regular data pages. Due to the reference bit implementation and the heavy use of translation pages, translation pages are almost never chosen for replacement.

3.3 Write-ahead Log

Write-ahead logs have a memory region in addition to an on-disk region. The memory region is written to disk at important times as described in table 1. When a flush is necessary, our hybrid prototype system will first scan through all the entries in the in-memory log. For each whole page update entry, a shadow copy of the page will be written to storage. After all the shadow copies are written, the memory log can be written to the disk portion of the log and then be cleaned.

3.4 Implementation

Due to the significant code overlap between the ARIES and hybrid prototypes, we implemented both systems in a single code base and used a flag to indicate which mode to run in. A summary of the code is given in table 3.

The implementation is rather small (only about 1400 lines not counting test code and instrumentation) because there are a number of features that production systems would have that we did not implement. We list these features and discuss the impact of implementing them below:

- Garbage collection. Currently, new shadow pages are just written at the end of the file. In a production system, we would need to reuse old pages at some point. Pages could be reused by implementing a combination of threading and copying as described in the Log-Structured File System paper[5].

API Call	Description
`int` **trans_start**`();`	Start a new transaction and return the transaction number that can be passed to future calls to perform operations from this transaction.
`void` **trans_commit**`(int transid);`	Commit the transaction, making it durable.
`void` **trans_abort**`(int transid);`	Undo the transaction.
`void` **trans_readonly**`(int transid);`	Indicates the end of a read-only transaction. The transaction is deleted without writing anything to the log.
`void` **trans_update**`(int transid, int pageno, char *page, int offset, int length, char *data);`	Update a page as part of the transaction by writing `data` to `page` at offset `offset`.
`char *` **trans_pinPage**`(int transid, int lPageNum);`	Pin the specified logical page. It is necessary to know which transaction is responsible for pinning in case it is beyond the end of the logical file, in which case this call could generate an update to the translation table. The buffer manager will return a pointer to the page data either from its cache or by fetching the page from disk.
`void` **trans_unpinPage**`(int transid, int lPageNum);`	Unpin the specified page, allowing it to be flushed to disk.

Table 2: Prototype Interface

Component	Lines of Code
Transaction Management	231
Buffer Management	582
Write-ahead Log	592
Instrumentation & Statistics	421
Test Code	452
Total	**2278**

Table 3: Source code.

- Recovery. Although we have implemented the ability to abort transactions, we have not implemented a recovery mechanism to handle system crashes at arbitrary times. However, implementing this would be straightforward and would not require any modifications to existing code, so the lack of this feature has no influence on our experiments.

- Cleaner thread. Currently dirty pages are flushed when stolen. Most real databases have a separate thread that actively cleans the buffer pool. We would expect the ARIES prototype to have better performance if this were implemented as described in section 4.5.

- Group commit. Some databases block when transactions commit in the hope that other transactions will also commit and the flushes can be merged. The Hybrid system would gain the most from this feature, as grouping transactions makes total I/O a more important factor and the number of flushes a less important factor, and the Hybrid system is a clear winner over ARIES in terms of total I/O.

4 Evaluation

We ran a variety of workloads on our prototypes to evaluate their performance under various circumstances. We instrumented our prototypes in order get statistics about flushing behavior, I/O, progress at regular intervals, and overall performance.

Variable	Values		
Type	Commit	Abort	*Read*
Page	Full	*Partial*	
Sequentiality	Sequential	Random	
# of Transactions	1	N	

Table 4: Workload Variables. Our experiments consisted of every combination of the above four variables. The italicized values indicate scenarios where the hybrid system offers no advantage over ARIES.

4.1 Workloads and Configuration

We evaluated our prototypes on a variety workloads which varied along four dimensions. A summary of these dimensions is in table 4. Our hybrid system is designed to be faster when full pages are being written. The hybrid system does nothing to improve reads or partial page writes, but we believe the slowdown in these cases that is induced by the translation overhead of the hybrid is worth the speedup that can be gained for full page writes.

We ran our experiments using both flash storage and a traditional disk on machines configured as described in table 5:

Config	System 1	System 2
Storage Type	5400RPM HD	Flash SSD
Device	Seagate Barracuda 7200.7	OCZ Technology SATA II Core Series
Capacity	80GB	64GB
OS	Ubuntu 10.04	
File System	Ext2	
Memory	1GB	
CPU	3GHz	

Table 5: Test Machines

We originally ran our experiments on top of a Ext4 file system. Ext4 is a journalling file sytem, however, so we switched to Ext2 to reduce noise as Ext does not employ a journal. A journal is also somewhat redundant since we are testing recovery systems which are by definition supposed to operate on top of unreliable systems. We did not observe drastic changes in the relative performance between ARIES and the hybrid after switching to Ext2. We were surprised, however, that the SSD well outperformed the disk overall when we were using Ext4, but after switching to Ext2, the disk was faster overall. We are not familiar with the Ext2 and Ext4 implementations, but we speculate that journals require additional seeks since data must be written to both a final destination as well to a journal. The additional seeks likely caused our disk to perform slower than the SSD even though it has faster sequential throughput.

Although we had 1GB of RAM available to us, we restricted the memory logs and buffer pools for our prototypes to 30 pages (4KB) each. We chose to limit the memory so that we could run a large variety of experiments under conditions where resources are limited in a reasonable amount of time (running experiments that fully utilized 1GB of RAM would have taken very long to run, so we would have had to limit the variety of workloads).

Our decision to split the 60 pages we allocated to our prototypes evenly between the buffer pool and memory log is justified by figure 5 where we varied the allocation ratio between these two subsystems for a representative workload. We observed that 30/30 is a reasonable split (the graph is relatively flat for both systems and both storage devices for moderate ratios).

4.2 CPU Overhead

Using an additional translation layer requires more CPU work for every page access (there will also be additional I/O when translation pages are first read, but the heavy access to translation pages virtually guaruntees they will remain in the buffer pool). However, the additional CPU load is moderate. In figure 6, we measured the amount of CPU time as a difference between total time and I/O time for each of our experiments. Although our prototype is simplistic and uses iteration in places where a hash table would be better, the CPU use is still relatively small (less than 15% in the worst cases). We also observe that the hybrid only stresses the CPU moderately more than ARIES.

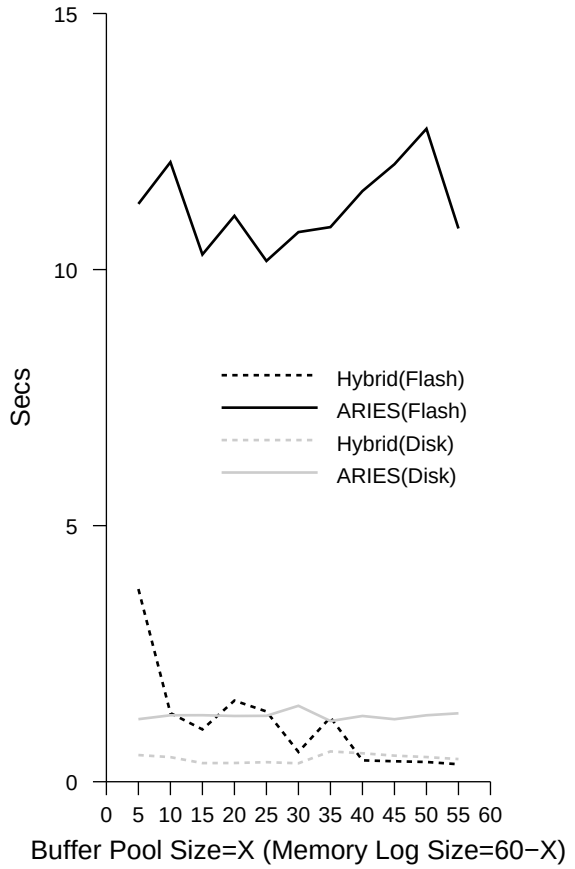

Figure 5: Buffer Pool / Memory Log Ratio. We had initially suspected that different ratios might be optimal for the hybrid than for ARIES, but this indicates that the systems are relatively insensative to this variable for moderate values.

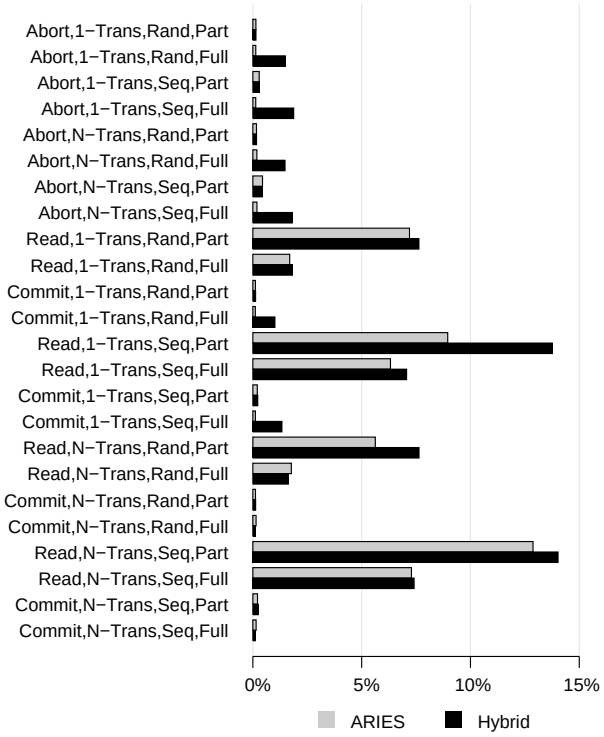

Figure 6: CPU Time

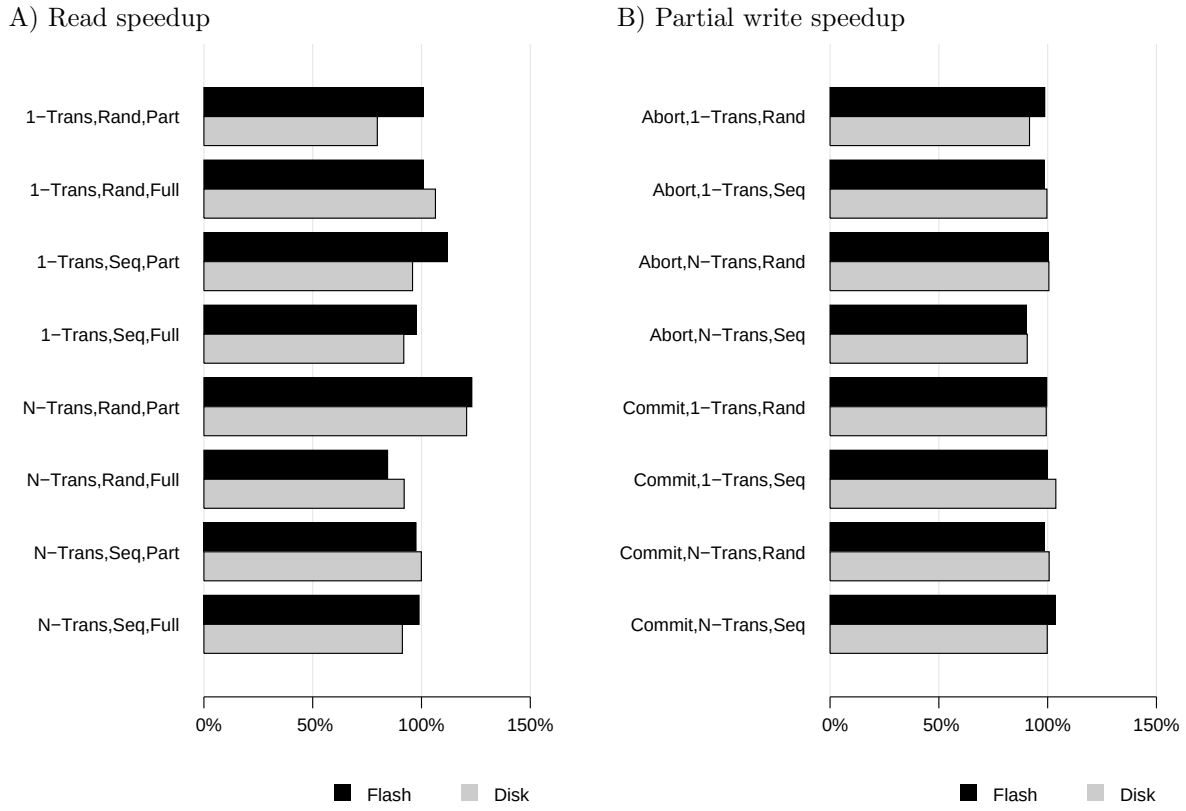

Figure 7: Speedup for regular workloads.

4.3 Partial Page and Read Workloads

Although our hybrid prototype is not designed to provide any gains for read workloads or workloads that update only certain records in a page, it is important to make sure that we d not have to pay a huge performance penalty in these cases. Figure 7 shows the speedup for these cases. Fortunately, the additional layer of translation introduces negligible overhead (indeed, due to moderate noise in the data, the hybrid even has positive speedup in some cases).

4.4 Full Page Writes

The hybrid system is optimized for full page writes. The speedups for the 8 workloads we ran in this category are illustrated in figure 8. There are several interesting trends in this data:

Abort performance. We observed extreme speedups for the abort cases since aborting simply involves reverting a version number for the hybrid system as opposed to reverting entire pages. For disk, we observed between 3x - 8x performance improvement among the various experiments. For flash, the improvement was much greater: 13x - 33x. This is due to the abort access pattern which involves accessing the log file backwards, starting with the most recent record at the end of the log, and working back toward the beginning file, executing undo's until the transaction is completely undone. This type of reverse access pattern is slow for hard drives as the disk platter will

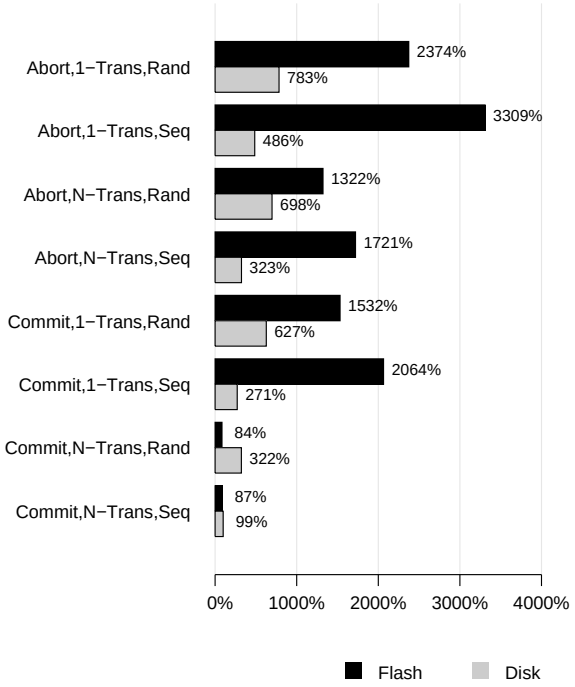

Figure 8: Full Page Writes

have to rotate each time unless the device automatically prefetches the data before the actual request location. Thus, even though the amount of I/O is reduced drastically, the random accesses still prevent the disk from realizing the same kind of performance gains as flash.

Workload affinity. The hybrid system helps disks most when the workload is random, but it helps SSDs most when the workload is sequential. The trend is moderate for flash but very strong for disks. 100% of our 16 experiments in figure 8 support this observation.

If we fix workload type (commit or abort) and number of transactions (1 or many), we are left with two workloads, one of which is random and one of which is sequential. Flash will have a greater speedup for the sequential workload than for the random workload, but the hard drive will have a greater speedup for the random workload than for the sequential workload. For the hard drive case, the drastic difference is to be expected because using out-of-place updates allows the system to turn random writes into sequential writes. Some file systems, such as LFS [5] use shadowing for this reason alone even when atomicity is not a concern.

The reason for the flash trend is less obvious since the performance gains should be the same regardless of whether the transactions are sequential or random. Even though the time spent writing data should be reduced by the same amount for both the sequential and random case, less time will be spent initially reading the data in the sequential case (due to SSD or HD prefetching). Thus, the same absolute write performance improvement will be felt relatively more for the workloads where less time is spent reading due to the sequential pattern.

Transaction size. All of our experiments show that when the transaction size varies and other variables are held constant, hybrid provides greater performance boosts when there is only one large transaction. Indeed, we were initially a bit surprised that performance is actually worse for a few of the workloads that involved many transactions that each update a single page. To explore the impact of varying the transaction size variable, we did writes to 1000 pages and evenly divided the work between a number of transactions, each of which operates on 1 to 20 pages. We see in graph A of figure 9 that after transactions start touching more than a single page the hybrid system quickly becomes much faster. This trend is due to the large number of flushes causes by ARIES in our implementation. ARIES flushes every time, so the hybrid has better performance except when a transaction is commited after every page is written. In that case, the hybrid has to flush nearly as often as ARIES, so their is no advantage. The frequent flushing behavior of ARIES is exlained in section 4.5.

4.5 Buffer Pool Behavior

We observed slow behavior for large, single transaction writes when ARIES is used. We discovered that the slow behavior was due to flushing before each page is written. The problem is that the large write fills the buffer pool with dirty pages. Each time a

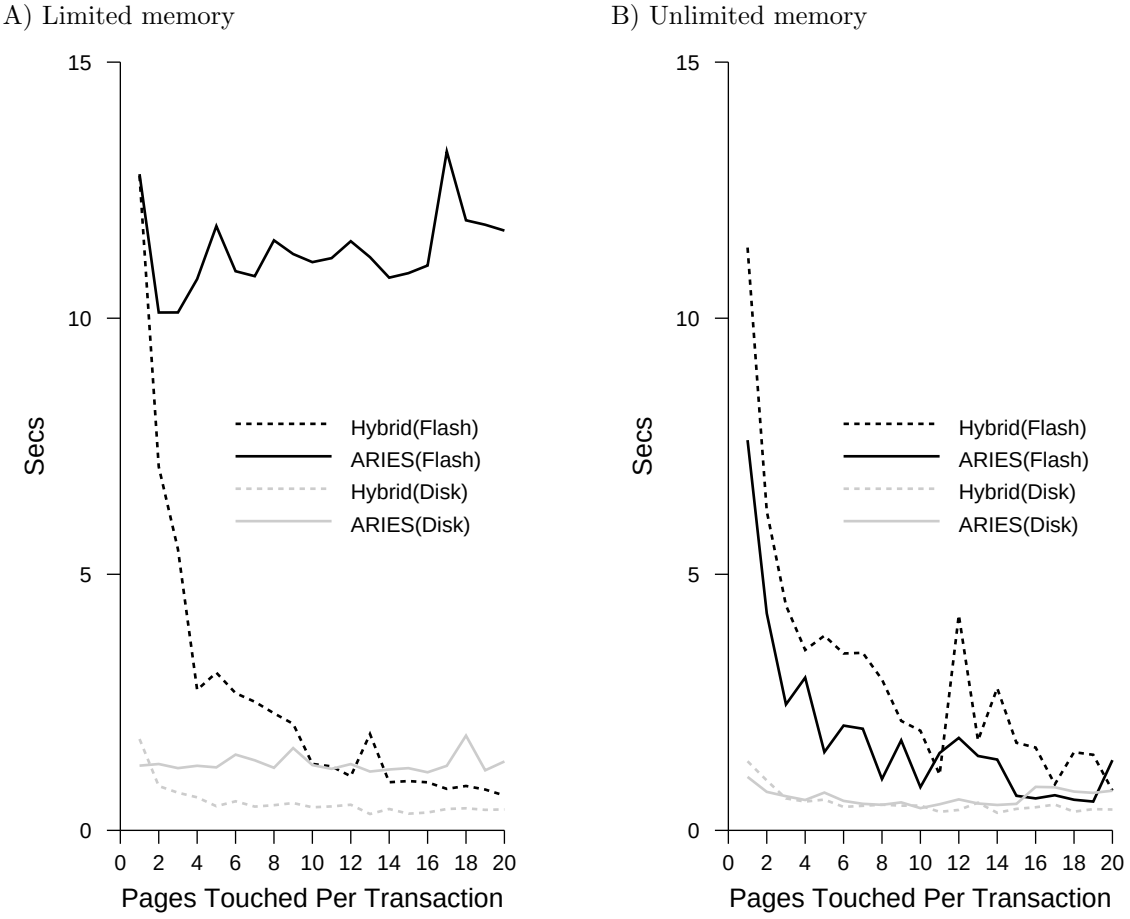

Figure 9: For all of these workloads, we sequentially wrote 1,000 pages

page is stolen, it is necessary to flush the log and flush the page before the memory can be used. This happens constantly because a page has to be stolen for each new page that is updated. In contrast, when the hybrid system is used, shadow copies are necessarily written to disk before the log is flushed. This has the convenient side effect of cleaning the entire buffer pool. The buffer pool behavior of the two prototypes can be compared in figure 10. For ARIES, after the pool becomes full, it stays full, but the hybrid system cleans the pool after each flush, resulting in the zig-zag pattern.

Although ARIES would be faster if a separate cleaning thread were implemented, the cleaning thread would also induce some additional overhead. When shadowing is used, a cleaning thread would likely be of little value (assuming most updates are to full pages). Thus, it would be interesting to compare the performance of ARIES with a cleaning thread to the hybrid system without a cleaning thread.

Since the performance gains were largely due to frequent flushing because of a full buffer, we tried rerunning the experiment with unlimited memory to see if the hybrid system would still be better. The

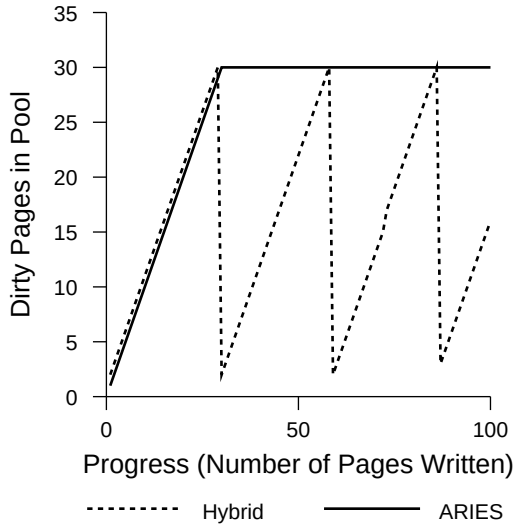

Figure 10: Buffer Pool Behavior

results in graph B of figure 9 show that there is little performance difference, but ARIES seems to slightly outperform the hybrid system overall. These results would seem to indicate that the frequency of flushes is a much more important performance factor than the total amount of I/O.

5 Related work

Straightforward shadowing has been used in some early database systems such as System R [2]. Such basic shadowing approaches are not typically used by modern databases for the reasons discussed in this paper.

Shadowing is also a technique used by some file systems such as LFS [5]. LFS uses shadowing for updates so that it can turn all updates to various parts of various files into large, sequential write operations. Although LFS would seem to be sacrificing read performance (since data will naturally be more fragmented) for better write performance, the LFS researchers argued that as main memories become larger, write performance is more important since reads can be satisfied more often by large caches.

The attractiveness of shadowing file systems for use with flash is evidenced by file systems such as JFFS2 [6, 7] and other literature [4]. In-place updates are already very expensive in flash (since it necessary to re-write the entire block that contains the page being updated), so file systems such as JFFS2 do out-of-place writes for performance reasons.

Using a write-ahead log on flash has been discussed in [3]. In this work, the practicality of using flash to absorb bursts of updates and then migrating the data to disk at times of lesser load is considered. If our system were to utilize both flash and disk, we believe it would actually be better to use the disk for logging and flash for actual data. Logs are written sequentially, so flash has almost no advantage over disks since arbitrarily fast sequential write throughput can be achieved with disks via RAID.

Perhaps most similar to our work is ZFS [1]. ZFS is intended for use with traditional hard drives, but it also uses a shadowing/logging hybrid approach for storing and updating data. Under normal circumstances, ZFS simply using shadowing. Since all the pages used by ZFS are part of a large hierarchical structure, modification to any page requires changes to the parent of the page (so that it points to the new version of its child). This recursively affects all the ancestors of a modified page. This guarantees atomicity because none of the new versions of the pages will be reachable until the root page of the file system is updated. Since every update to any page in the file system results in an update to the root as well as many other updates, it is not feasible to immediately write changes to disk, so ZFS keeps all changes in a buffer that is infrequently flushed to disk. To provide durability to applications that need it, ZFS can flush a log of all the system calls that were made since the last flush of the file system hierarchy.

6 Conclusion

Shadowing and logging are two different ways databases can provide atomicity and durability. Although shadowing results in less I/O than logging for some workloads, logging has historically been favored by production databases because logging pro-

vides greater concurrency than shadowing and also results in less data scattering. Scattering is not as relevant if a database is using flash storage. Furthermore, we believe using a shadowing/logging hybrid system can provide the advantages of shadowing without restricting the concurrent execution of transactions. We implemented simple ARIES and hybrid prototypes and found that a hybrid system yields drastic performance gains for most full page write workloads.

References

[1] N. L. Beebe, S. D. Stacy, and D. Stuckey, "Digital forensic implications of zfs," *Digital Investigation*, vol. 6, no. Supplement 1, pp. S99 – S107, 2009, the Proceedings of the Ninth Annual DFRWS Conference. [Online]. Available: http://www.sciencedirect.com/science/article/B7CW4-4X1HY5C-F/2/0cc08faa24b186b9dc0d4f98b98d4ed7

[2] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger, "The recovery manager of the system r database manager," *ACM Comput. Surv.*, vol. 13, no. 2, pp. 223–242, 1981.

[3] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating server storage to ssds: analysis of tradeoffs," in *Proceedings of the 4th ACM European conference on Computer systems*, ser. EuroSys '09. New York, NY, USA: ACM, 2009, pp. 145–158. [Online]. Available: http://doi.acm.org/10.1145/1519065.1519081

[4] Numonyx, "Flash file systems overview." [Online]. Available: www.numonyx.com/ Documents/ WhitePapers/ Flash_file_ systems_WP.pdf

[5] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, 1992.

[6] D. Woodhouse, "JFFS: The journalling flash file system." Ottawa Linux Symposium, 2001.

[7] ——, "The journalling flash file system," 2001, slideshow used with presentation at Ottawa Linux Symposium: http://sourceware.org/ jffs2/ jffs2-slides-transformed.pdf.