Appendix A

Restrictions to the Libxac Interface

A.1 Restrictions to Libxac

This appendix enumerates the various programming restrictions when using Libxac, and discusses potential improvements to the implementation.

1. Libxac's most significant restriction is that only one transaction per process is allowed. Libxac uses the no-access, read-only, and read/write memory protections provided by the `mmap` function. If the OS maintains a single memory map for each process rather than for each thread, we can not use memory protections to map a page read-only for one thread and read/write for a different thread on the same process. In this case, Libxac could not support multiple concurrent transactions on the same process without using a different mechanism for detecting the pages accessed by a transaction.

2. Libxac currently supports having only one shared memory segment. In other words, all concurrent processes that `xMmap` the same file must call `xMmap` with the same filename and length arguments. The motivation for this restriction is that programs can often simply make one call to `xMmap` to map one large file for the entire shared memory space.

   The implementation described in Chapter 3 could theoretically be extended to support the full functionality of normal `mmap` (i.e allowing multiple `xMmap` calls on different files, mapping only part of a file instead of the entire file, or mapping the same page in a file to multiple addresses). These extensions would require Libxac to maintain a more complicated map between the transactional page addresses returned by `xMmap` and the physical page address of files on disk, but these changes are, in principle, relatively straightforward. In this extension, accesses to pages in multiple files would be treated as though the multiple files were concatenated logically into one large shared file.

   This idea is different from the proposal of allowing a process access to multiple shared memory segments that are logically distinct in terms of the Libxac's atomicity guarantees. Having two distinct shared-memory segments (for example, A and B) has interesting semantics. Libxac would guarantee that transactions are serializable only with respect to operations on A, and also with respect to only operations on B. The serial order of transactions could be different for A and for B, however, so there is no guarantee of serializability if we consider all operations on A and B together.

3. The prototype arbitrarily sets the maximum size of the shared memory segment to 100,000 pages, and the maximum number of concurrent transactions to 16. This restriction allowed us to implement control data structures simply (and inefficiently) with large fixed-size arrays. Using dynamic control structures easily removes this limitation, and may also improve the caching behavior of the runtime system.

4. Linux limits a process to having at most 2^{16} different memory segments (virtual memory areas, or vma's in the kernel [10]) at any point in time. Whenever a transaction touches a page, LIBXAC calls `mmap` on that page and generates a new segment for that page. Thus, a single transaction can not possibly touch more than 2^{16} pages at once.

   One way of raising this limit, aside from modifying the Linux kernel, is to concatenate adjacent segments together when a transaction touches adjacent pages. This proposal does not fix the problem, as it is possible for a transaction to touch every other page. In that situation however, LIBXAC should escalate its concurrency control and work at a larger granularity, treating multiple pages as a single large segment when it detects a large transaction.

5. Every `xbegin` function call must be properly paired with an `xend`. The control flow of a program should not jump out of a transaction. If we also require that every `xbegin` must be paired with exactly one `xend`, then it is possible to detect unmatched function calls with compiler support.

6. As described in Section 3.2, recovery for durable transactions has not been implemented. Also, the structure of the log file is incorrect for transactions on multiple processes if transactions write data pages to the log that can be confused as metadata pages. Separating the metadata and actual data pages into separate files solves this problem and also facilitates further optimizations such as logging only page diffs.

7. LIBXAC does not provide any mechanism for allocating memory from the shared memory segment. This shortcoming is not technically a restriction on the implementation, but it is quite inconvenient if the programmer wishes to dynamically allocate and work with transactional objects. Providing a `malloc` function that allocates memory from the shared segment might lead to simpler user programs.

Appendix B

Transaction Recovery in Libxac

For durable transactions the LIBXAC prototype writes meta-data pages in the log and synchronously force changes out to disk. This in principle is enough to recover from a program crash and restore the user data file back to a consistent state, assuming the disk itself has not failed.

Although LIBXAC writes enough data to disk to do recovery, I have not yet implemented the recovery module for the prototype. In this section, I sketch a possible algorithm for transaction recovery when transactions execute on a single process.

LIBXAC's log files have the following structure:

- The `XCOMMIT` pages appear in the log in the order that transactions are committed.
- The `XBEGIN` page points to all pages belonging to T, and also to any spill-over pages for this storing this list. For any transaction, T, all spill-over pages and data pages written by T appear between the T's `XBEGIN` and `XCOMMIT` (or `XABORT`) meta-data pages.
- All pages in a new log file are initialized to all zeros before the log file is used.

Figure B-1 illustrates two example layouts for the log file when transactions are executing on a single process and two processes, respectively. At a high level, the recovery algorithm is as follows:

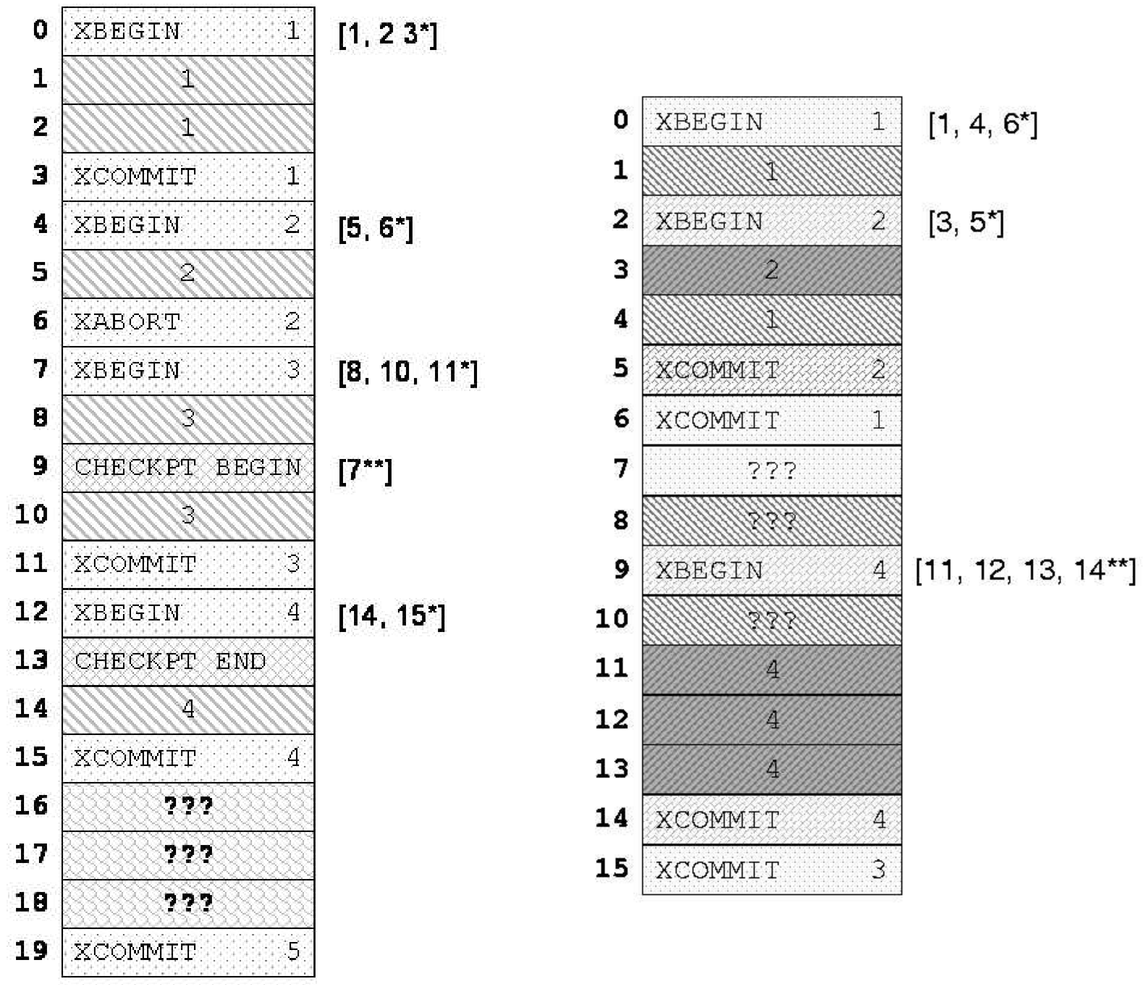
1. Go to log file containing the last `XCHECKPT_BEGIN` page that has a valid matching `XCHECKPT_END`.[1] Scan through the entire log starting at this point and compute which transaction each page in the log belongs to.

   We know that every valid page that comes after the `XCHECKPT_BEGIN` page is either (i) a meta-data page, (ii) a data page pointed to by an `XBEGIN` page in the list stored in the `XCHECKPT_BEGIN` page, or (iii) a data page pointed to by some `XBEGIN` page that comes after the `XCHECKPT_BEGIN` page. Therefore, we can match pages for all transactions after the last checkpoint. Any unmatched pages are considered to be invalid.

   For a transaction T, we may detect the following inconsistencies:

   - T has an `XBEGIN` page, but no `XCOMMIT` or `XABORT` page. This event means T had not completed at the time of the crash, or the `XCOMMIT` page did not make it out to disk.
   - T has both `XBEGIN` and `XCOMMIT` pages, but the checksum is wrong. Since writes of multiple pages are not guaranteed to happen atomically, the `XCOMMIT` page may get written to disk before one of the data pages. The checksum should detect this error.
   - T does not have an `XBEGIN` page. This situation can occur if a system crashes before the `XBEGIN` is flushed to disk. In this case, none of T's data pages will be pointed to by a valid transaction.

[1] We assume LIBXAC maintains a separate file on disk recording the location of all checkpoint meta-data pages. This file is synchronized a second `fsync` that occurs after the first `fsync` does the actual synchronization. Thus, the `XCHECKPT_END` page is not considered valid until this meta-meta-data appears on disk.



(a) (b)

Figure B-1: An example of a Libxac log file when transactions execute (a) on one process, and (b) on two processes.

In these three situations, the transaction T is considered to be aborted.

2. Once we have identified which transactions in the log were successfully committed, we can replay all those transactions in the correct serial order. This process is done by copying a transaction's pages into the original file. Alternatively, we could also attempting a more clever algorithm that works from the end of the log and only copies the latest version of each page back into the file.

Note that in the actual implementation, the recovery process must itself keep a log of its changes so we can restore the data to a consistent state in case of a crash during the recovery process.

In the example in Figure B-1 (a), scanning through the log file, we discover that the last valid checkpoint started at page 9. We only need to repeat transaction 3 because it is pointed to by the `XCHECKPT_BEGIN` page, and transaction 4 because it comes after that page. Transaction 5 attempted to commit, and the `XCOMMIT` page made it to disk. Its checksum will be incorrect however, as the corresponding `XBEGIN` page did not make it successfully to disk. Note that the system must have crashed before `fsync` returned. Thus, LIBXAC did not see transaction 5 finish its commit, and no other transaction could have read values written by 5.

The example in Figure B-1 (b) shows transactions executing concurrently on two processes. In this case, pages from different transactions can be interleaved in the log file. For this example, transaction 3 crashed while attempting to commit, and thus its `XBEGIN` page at page 7 did not successfully get written to disk. Transaction 4 that was executing on the other process did successfully commit however. Transaction 4 could not have read pages from transaction 3 because the `fsync` that was writing transaction 3's data did not succeed.

When all transactions are executed on a single process (but checkpointing may be done by a different process), this recovery process works correctly because LIBXAC guarantees that the log will have at most two invalid data pages interleaved between valid log pages (the two pages reserved for `XCHECKPT_BEGIN` and `XCHECKPT_END`, in case the checkpointing process crashed). Since we record the pointers to the checkpoint meta-data pages in a separate file, we can always distinguish meta-data pages from data pages.

Unfortunately, this recovery algorithm does not always work when two or more processes execute transactions. In Figure B-1(b), when we see pages 9 and 14 in the log, these could either be meta-data pages for transaction 4, or they could theoretically be data pages for transaction 3 that crashed. This proposed recovery algorithm assumes that it is always possible to distinguish between data pages and meta-data pages. It seems quite unlikely that a programmer would accidentally execute a transaction that writes data pages that are exactly the meta-data plus data pages for a valid transaction. It is conceivable, however, that a programmer might call `xMmap` on a log file generated by LIBXAC, and perhaps copy a portion of this file as part of a transaction. In this case, the recovery process can no longer distinguish between data and meta-data.

When pages from transactions on different processes can be interleaved, it seems difficult to differentiate between data and meta-data without imposing additional structure on the log file. Possible solutions to this problem are to enforce some global structure in log (ex. all odd pages are meta-data, all even pages are data), to use separate log files for each process, or to use separate files for data and meta-data. The last two options are perhaps the most practical, although these implementations may require multiple disks to achieve good performance. Otherwise, the system may waste a significant amount of time doing disk seeks between two different files on disk.

Appendix C

Detailed Experimental Results

This appendix presents a more detailed description of some of the experiments described in Chapters 4 and 5. It also contains more detailed data collected from these experiments.

C.1 Timer Resolution

In this section, I describe an experiment to determine the resolution of the timers used in the empirical studies. Using the processor's cycle counter and `gettimeofday` is accurate at least for measuring times greater than 50 ns and 10 μs, respectively.

In all experiments, I measured the time for an event by checking the system time before and after the event and reporting the difference. For nondurable transactions, I typically used the processor's cycle counter, via the `rdtsc` instruction. For longer events, I used two calls to the `gettimeofday` function. To understand the resolution of this method, I measured the difference between two consecutive calls to check the timer, with no code in between. The results from repeating this experiment 10,000 times for `rdtsc` and `gettimeofday` are shown in Tables C.1 and C.2, respectively.

In Table C.1, the data on Machines 1 and 2 suggests that the delay when checking the cycle counter is about 100 clock cycles (less than 50 ns). The delay is even less for Machines 3 and 4. Although the maximum value on Machine 1 was about 12 μs, the 99th percentile was still under 50 ns, suggesting that this mechanism for estimating times is reasonably accurate for measuring times to within a few tenths of a microsecond, provided we make repeated measurements.

From the data in Table C.2, we observe that the delay between `gettimeofday` calls on all four machines was less than 7 microseconds for 99% of measurements. This data suggests `gettimeofday` has a resolution of a few microseconds.

On Machine 1, the maximum delay was approximately 200 μs. If we observe the distribution of delay times, as shown in Figure C-1, then we see that this was a rare event. This behavior is not surprising, as it is impossible to stop basic system processes during our experiments. An interrupt or other operating system process may have caused the timer code to get swapped out. These rare but expensive delays must be kept in mind when interpreting the experimental results.

Machine	Mean	St. Dev	Min	Median	99th Percentile	Max
1	99.5	458	92	92	112	35332
2	92.0	0.92	92	92	92	184
3	9.0	3.2	5	8	23	36
4	41.0	0.16	41	41	41	51

Table C.1: Delay (in clock cycles) between successive calls to timer using `rdtsc` instruction, 10,000 repetitions.

Machine	Mean	St. Dev	Min	Median	99th Percentile	Max
1	2.87	2.24	2.0	3.0	7.0	201.0
2	0.87	0.39	0.0	1.0	1.0	20.0
3	1.3	0.50	1.0	1.0	2.0	8.0
4	1.55	2.99	1.0	2.0	2.0	271.0

Table C.2: Delay between successive calls to `gettimeofday` (in μs), 10,000 repetitions.

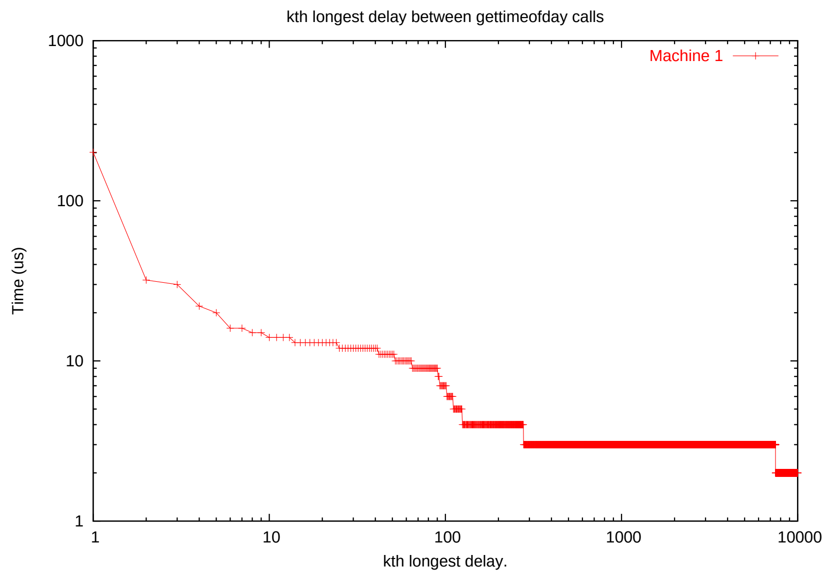


Figure C-1: Machine 1:Distribution of Delay Times Between Successive `gettimeofday` Calls.

In summary, the data suggests we can use `gettimeofday` to measure times longer than approximately 10 μs with reasonable accuracy provided the time interval is long enough or we do enough repetitions.

C.2 Page-Touch Experiments

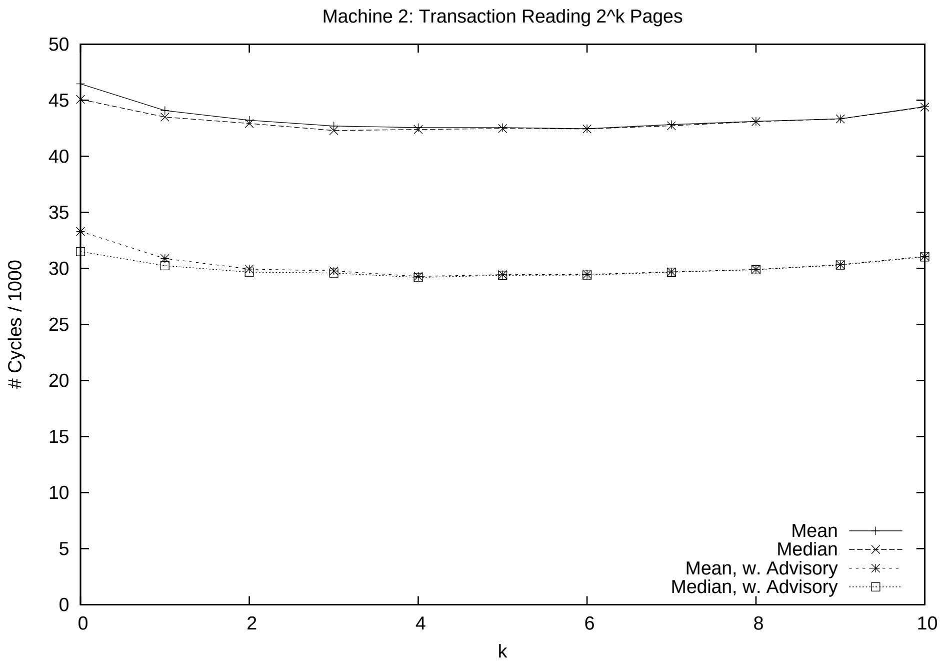
This section contains results from the page-touch experiments on Machines 2 through 4. These experiments were previously described in Section 4.2.1 and 4.2.2. Figures C-2 through C-4 plot the times per page read and write in the page-touch experiments with nondurable transactions.

C.3 Experiments on Various System Calls

In this section, I present detailed data from the microbenchmark experiments discussed in Section 4.2.3.

Memory Mapping and Fault Handlers

Table C.3 is a more complete version of Table 4.3.



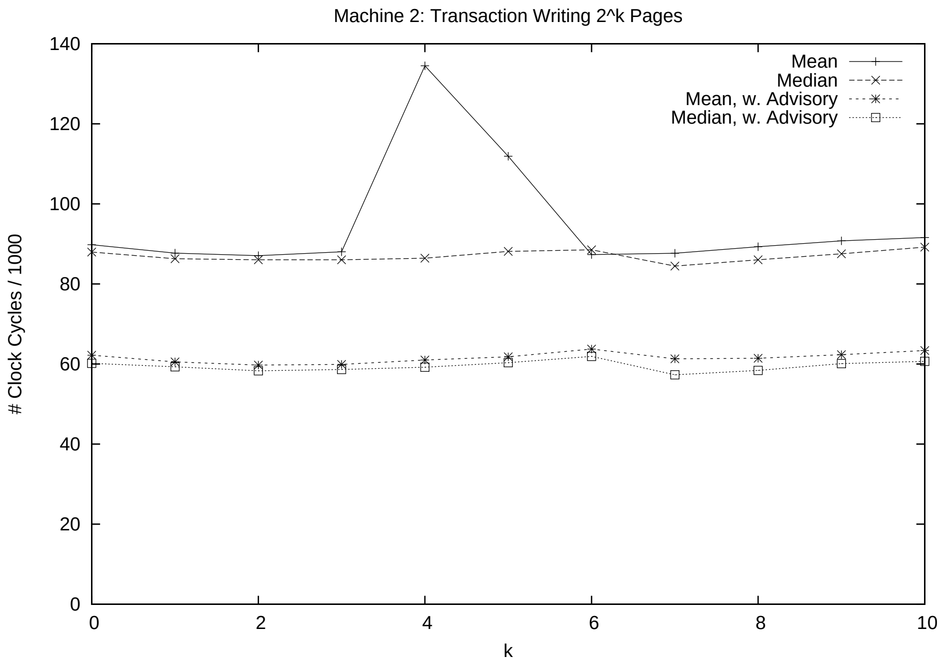
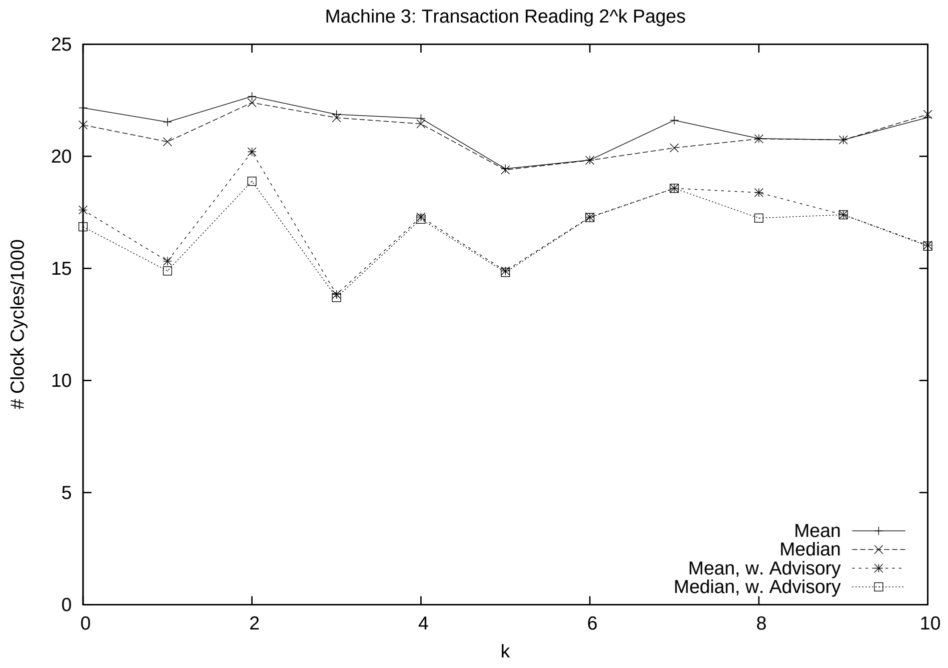


Figure C-2: Average time per page to execute the transactions shown in Figure 4-3 on Machine 2. For each value of n, each transaction was repeated 1000 times.



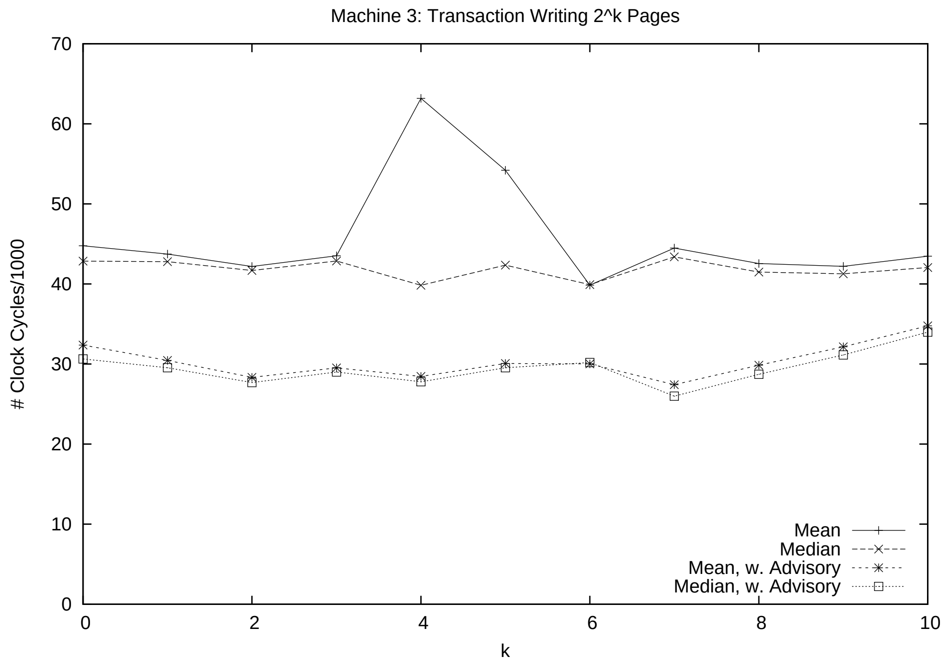
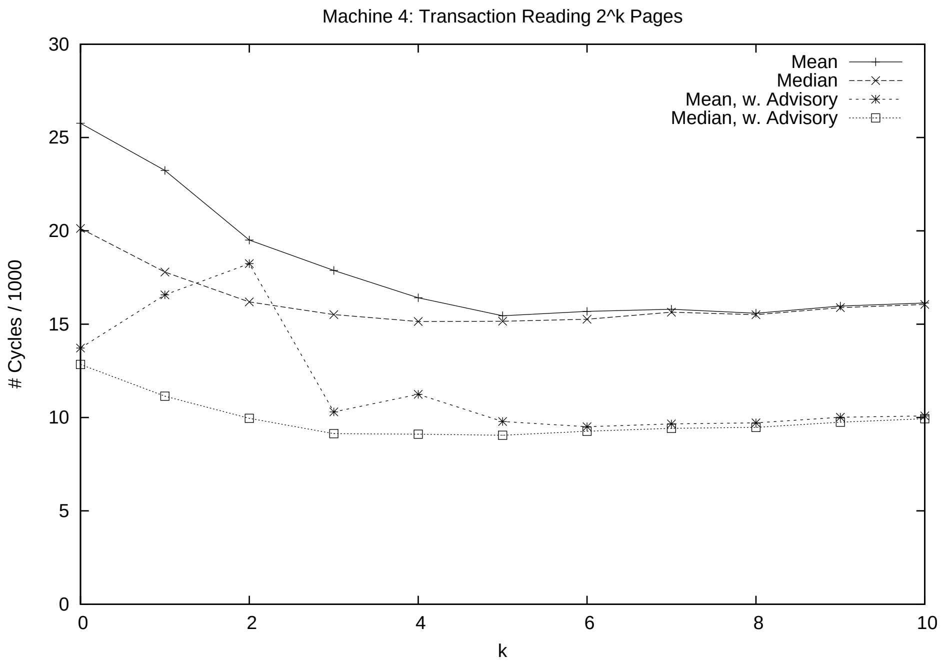


Figure C-3: Average time per page to execute the transactions shown in Figure 4-3 on Machine 3. For each value of n, each transaction was repeated 1000 times.



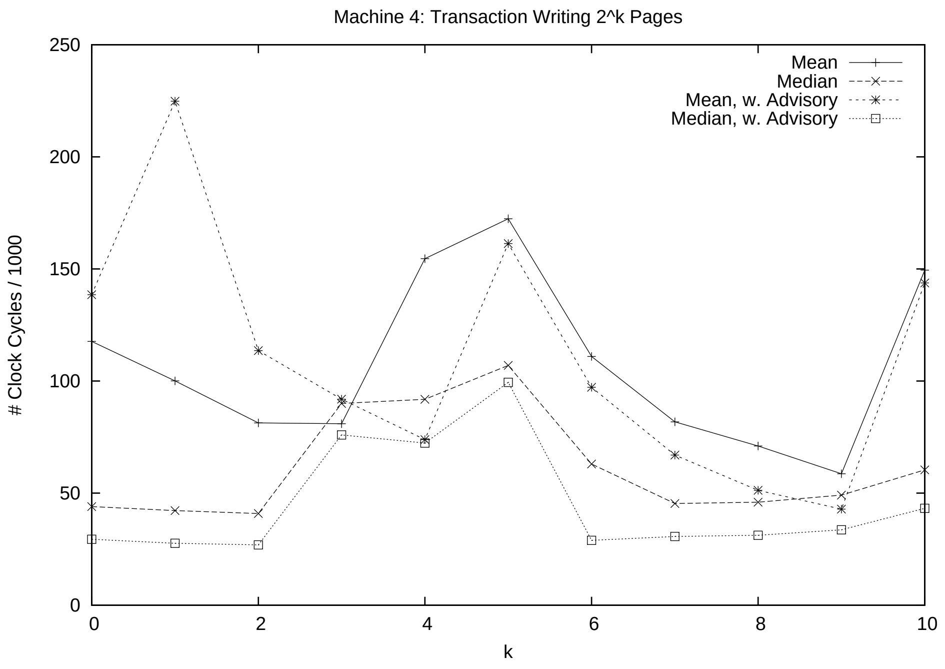


Figure C-4: Average time per page to execute the transactions shown in Figure 4-3 on Machine 4. For each value of n, each transaction was repeated 1000 times.

Operation	Mean	St. Dev	Min	Median	99th %tile	Max
1: Entering `SIGSEGV`	32,216	11,268	33,320	33,956	54,772	648,236
1: `mmap`	15,156	2,747	13,916	14,580	26,672	70,748
1: Exiting `SIGSEGV`	29,323	9,288	27,408	28,428	44,996	635,340
2: Entering `SIGSEGV`	8,032	546	7,884	8,032	8,132	45,720
2: `mmap`	10,054	639	9,920	10,024	10,380	49,520
2: Exiting `SIGSEGV`	9,723	830	9,632	9,704	9,872	46,960
3: Entering `SIGSEGV`	3,489	182	3,238	3,466	3,741	13,190
3: `mmap`	3,228	512	2,892	3,205	3,420	13,335
3: Exiting `SIGSEGV`	4,745	110	4,377	4,709	4,844	9,204
4: Entering `SIGSEGV`	3,078	613	2,971	3,051	3,155	30,968
4: `mmap`	2,282	711	2,142	2,259	2,340	56,322
4: Exiting `SIGSEGV`	3,140	537	3,055	3,114	3,267	21,255

Table C.3: Timing data for entering `SIGSEGV` handler, calling `mmap`, and leaving handler, 10,000 repetitions. All times are processor cycles.

Operation	Mean	St. Dev	Min	Median	99th %tile	Max
1: `memcpy`	1,122,167	129,167	1,064,328	1,076,620	1.803,956	2,204,000
2: `memcpy`	1,052,253	7,379	1,046,292	1,052,316	1,086,384	1,106,784
3: `memcpy`	608,482	8,049	605,017	608,112	612,237	791,724
4: `memcpy`	931,823	98,636	925,048	925,089	953,525	4,037,042

Table C.4: Clock cycles to do 1,000 calls to `memcpy` between two 4K character arrays in memory, 1,000 repetitions. times are in μs.

Test of `memcpy`

Table C.4 is a more detailed look at the time required to do `memcpy` between two arrays 1,000 times (i.e. the last column of Table 4.5).

C.4 Durable Transactions

All times in this section of the appendix were measured using `gettimeofday` as the clock.

Page-Touch Experiments

Table C.5 is a more complete version of Table 4.7 from Section 4.3.

Synchronizing a File

Table C.6 is a more complete version of Table 4.8.

Write Speed

This benchmark measures the time to write 10,000 pages to a file, 1 page at a time, using the `write` system call. This operation was repeated 1000 times. The results from this test are shown below in Table C.7.

The data suggests that `lseek` is actually lazy, with the disk head not actually moving until an I/O operation executes. This test was done when the write-cache on all machines was enabled.

Machine	Mean	σ	Min.	Median	99th %tile	Max
1, Page Read	38.6	65.0	34.6	35.5	52.7	2044.5
1, Page Read w. Adv.	22.4	75.0	18.1	18.8	36.0	2032.2
1, Page Write	1569.3	4146.0	560.4	731.3	9578.2	106,560
1, Page Write w. Adv.	1297.4	2808.0	524.4	842.5	8645.2	54,514
3(a), Page Read	15.4	33.1	14.1	14.4	14.7	1062.6
3(a), Page Read w. Adv.	14.0	32.5	11.8	13.0	13.5	1039.4
3(a), Page Write	235.0	276.3	108.7	126.0	1231.71	1800.13
3(a), Page Write w. Adv.	179.4	173.9	102.9	114.7	822.3	1612.6
3(b), Page Read	16.4	10.6	15.5	15.8	30.2	344.6
3(b), Page Read w. Adv.	15.1	25.2	12.7	13.8	25.9	636.2
3(b), Page Write	226.8	297.6	103.9	121.2	1510.3	1798.0
3(b), Page Write w. Adv.	182.2	203.8	97.8	115.3	1072.4	1674.3

Table C.5: Average Access Time (μs) per Page, for Transactions Touching 1024 Pages.

Operation	Mean	St. Dev	Min	Median	99th Percentile	Max
1: `msync`	8	1	7	8	16	27
1: `fsync`	13575	184338	2439	8015	12672	5836456
2: `msync`	11	3	8	9	19	25
2: `fsync`	5137	36919	294	2663	41061	1156245
3(a): `msync`	5	1	4	5	6	12
3(a): `fsync`	4794	24018	735	4002	7000	761552
3(b): `msync`	5	1	4	5	6	12
3(b): `fsync`	4698	23090	818	3949	6925	731910
4: `msync`	33	6	28	31	43	111
4: `fsync`	3531	46,721	592	632	32,576	1,460,662

Table C.6: Timing data for calling `msync` and `fsync` on a 10,000 page file with a random page modified, 1000 repetitions. All times are in μs.

Operation	Mean	St. Dev	Min	Median	99th %tile	Max
1: `lseek`	6	2	5	6	13	62
1: `write`	103,446	244,907	90,564	92,088	113,437	5,615,347
2: `lseek`	4	13	2	3	4	406
2: `write`	151,895	166,318	119,006	119,571	1,193,881	1,278,444
3(a): `lseek`	3	1	2	2	4	9
3(a): `write`	63,465	4,036	62,992	63,270	64,268	190,798
3(b): `lseek`	3	1	2	3	4	9
3(b): `write`	92,507	70,402	78,171	83,658	586,612	837,660
4: `lseek`	9	5	7	8	9	103
4: `write`	2,168,185	504,761	1,423,656	2,521,593	2,893,774	3,070,492

Table C.7: Time to write 10,000 pages to a file, 1,000 repetitions. All times are in μs.

Machine	Hash	Mean	σ	Min	Median	99th %	Max
1	SHA1	85.6	16.5	81.7	82.9	118.6	685.6
1	MD5	36.3	10.1	34.4	35.1	67.3	680.5
2	SHA1	84.5	2.3	83.6	84.2	86.4	177.5
2	MD5	35.5	5.7	34.4	35.5	35.9	582.0
4	SHA1	105.2	555.9	52.2	52.3	69.2	6,069.2
4	MD5	54.6	412.2	25.9	25.9	30.6	6,050.1

Table C.8: Time to compute SHA1 and MD5 hash functions on a single page. All times are in thousands of clock cycles.

Checksum Calculations

Table C.8 shows the time required to calculate the SHA1 and MD5 hash functions on a single page. On Machines 1 and 2, the average time for MD5 is 12 and 15 μs, respectively. This data suggests that the overhead of computing a checksum of each page a transaction writes on a transaction commit is not too expensive for small durable transactions.

C.5 Concurrency Tests

Table C.9 shows the average time required per nondurable transaction for the concurrency tests described in Section 4.4. This table represents a more complete version of Table 4.9. For transactions on two processes, I report the number of aborted transactions on each process. All times in this section are measured with `gettimeofday`.

Similarly, Table C.10 is a more complete version of the data for concurrency tests for durable transactions, originally presented in Table 4.10.

C.6 Search Trees using Libxac

I use `gettimeofday` as the timers for all experiments on the LIBXAC search trees. For insertions done on a single process, I measure and record the time required for every insertion. To provide a comparison on two processes, I use a call to `fork`, and did 125,000 insertions on each process. To estimate the time required for each insertion, I record the time to complete each insertion and compare that to the time before the call to `fork`. Note that this introduce a slight bias in favor of the single process because I am also including the time required to do the `fork` operation. On the other hand, each insertion on two processes requires only one call to `gettimeofday` instead of two.

Insertions as Nondurable Transactions

This section describes the details of the experiments done on the LIBXAC search trees and on Berkeley DB's B-tree and presents a complete table of results for nondurable and durable transactions. See Section 5.4 for details.

With the LIBXAC versions of the B^+-tree and the CO B-tree, I measured the time to insert 250,000 elements. Each search tree had 512-byte data blocks, each indexed by a 64-bit key. For the B^+-tree, the blocksize was 4K. The keys for the inserted elements were chosen at random using the `rand` function, with each insert being a separate transaction. I tested 2 versions of the B^+-tree: one unoptimized implementation, and one that uses the advisory function. I only tested an unoptimized CO B-tree.

For the Berkeley DB B-Tree, I used the `DB_AUTO_COMMIT` feature to automatically make each `put` operation on the B-tree its own transaction. On Machines 1 and 3, I ran Berkeley DB version 4.2 (`-ldb-4.2`). Machines 2 and 4 had Berkeley DB version 4.1 (`-ldb-4.1`). On each machine, the cache size was set to be 2 caches, 1 MB in size.

Machine	Test #	Avg. Time per Xaction (μs)	Standard Dev. (μs)	Speedup
1	A, 1 proc.	32.2	0.27	
1	A, 2 proc.	30.7	0.36	1.05
1	B, 1 proc.	33.6	0.25	
1	B, 2 proc.	32.2	0.87	1.04
1	C, 1 proc.	1,453	4.90	
1	C, 2 proc.	1,460	56.0	1.00
2	A, 1 proc.	26.2	0.10	
2	A, 2 proc.	23.0	0.15	1.14
2	B, 1 proc.	28.3	0.26	
2	B, 2 proc.	24.2	0.48	1.17
2	C, 1 proc.	1,787	30.3	
2	C, 2 proc.	903	36.3	1.98
3(a)	A, 1 proc.	22.9	0.63	
3(a)	A, 2 proc.	24.3	1.14	0.94
3(a)	B, 1 proc.	28.1	0.42	
3(a)	B, 2 proc.	27.5	1.05	1.02
3(a)	C, 1 proc.	2,259	3.90	
3(a)	C, 2 proc.	1,132	1.82	2.00
3(b)	A, 1 proc.	24.3	1.36	
3(b)	A, 2 proc.	24.9	1.07	0.98
3(b)	B, 1 proc.	28.2	1.67	
3(b)	B, 2 proc.	26.3	0.74	1.07
3(b)	C, 1 proc.	2,248	2.74	
3(b)	C, 2 proc.	1,130	1.30	1.99
4	A, 1 proc.	109	3.39	
4	A, 2 proc.	185	13.1	0.59
4	B, 1 proc.	121	3.39	
4	B, 2 proc.	190	11.5	0.64
4	C, 1 proc.	10,487	8.51	
4	C, 2 proc.	10,565	8.86	0.99

Table C.9: Concurrency tests for nondurable transactions. Times are μs per transaction.

Machine	Test #	Mean Time per Xaction (μs)	Standard Dev. (μs)	Speedup
1	A, 1 proc.	8,231	234	
1	A, 2 proc.	8,531	3500	0.96
1	B, 1 proc.	8,868	46.5	
1	B, 2 proc.	8,878	3133	1.00
1	C, 1 proc.	9,125	65.6	
1	C, 2 proc.	10,042	1452	0.91
3(a)	A, 1 proc.	6,116	8.23	
3(a)	A, 2 proc.	6,162	95.3	0.99
3(a)	B, 1 proc.	6,113	3.28	
3(a)	B, 2 proc.	6,201	93.8	0.98
3(a)	C, 1 proc.	6,315	6.57	
3(a)	C, 2 proc.	6,626	464	0.95
3(b)	A, 1 proc.	6,210	23.1	
3(b)	A, 2 proc.	6,264	64.8	0.99
3(b)	B, 1 proc.	6,215	18.1	
3(b)	B, 2 proc.	6,213	20.1	1.00
3(b)	C, 1 proc.	6,388	16.2	
3(b)	C, 2 proc.	6,619	438.4	0.97

Table C.10: Concurrency tests for durable transactions. Times are per transaction.

To make transactions nondurable, for the machines Berkeley DB 4.2, I used the `DB_TXN_NOT_DURABLE` flag to turn off durability. For Machines 2 and 4, finding an appropriate point of comparison with Berkeley DB was challenging, as using Berkeley DB version 4.1 seemed to cause some incompatibility with this flag. Instead, I used the flag `DB_TXN_NOSYNC` on Machines 2 and 4.

Durable Insertions with Write-Cache Enabled

Table C.12 shows the average times per durable insert when the write-caches on the harddrives on all machines were enabled. It is unclear how to interpret these numbers, as these transactions are not strictly durable. The main point is, however, that the performance of Libxac search trees and Berkeley DB are still comparable under different hardware settings.

Machine	Search Tree	# Proc.	Avg. Time (μs) per Insert	# Aborts
1	B^+-tree, no adv.	1	411	–
1	B^+-tree, no adv.	2	488	59,992, 56,193,
1	B^+-tree, w. adv.	1	240	–
1	B^+-tree, w. adv.	2	236	27,753, 28,041
1	CO B-tree, no adv.	1	490	–
1	CO B-tree, no adv.	2	455	3,370, 2,876
1	Berkeley DB	1	37	–
1	Berkeley DB	2	29	–
2	B^+-tree, no adv.	1	244	–
2	B^+-tree, no adv.	2	191	32,270, 33,397,
2	B^+-tree, w. adv.	1	189	–
2	B^+-tree, w. adv.	2	152	31,491, 27,238
2	CO B-tree, no adv.	1	260	–
2	CO B-tree, no adv.	2	189	3,733, 5,785
2	Berkeley DB	1	24	–
3(a)	B^+-tree, no adv.	1	266	–
3(a)	B^+-tree, no adv.	2	264	31,128, 27,250
3(a)	B^+-tree, w. adv.	1	232	–
3(a)	B^+-tree, w. adv.	2	229	26,877, 25,497
3(a)	CO B-tree, no adv.	1	338	–
3(a)	CO B-tree, no adv.	2	337	3,345, 5,166
3(a)	Berkeley DB	1	26	–
3(a)	Berkeley DB	2	20	–
4	B^+-tree, no adv.	1	18,019	–
4	B^+-tree, w. adv.	1	17,408	–
4	CO B-tree, no adv.	1	2,286	–
4	Berkeley DB	1	393	–

Table C.11: Time to do 250,000 nondurable insertions into LIBXAC search trees.

Machine	B^+-tree	CO B-tree	Berkeley DB B-Tree
1	2.6	2.2	2.1
2	6.0	4.8	4.6
3(a)	2.7	2.1	6.5
3(b)	2.0	1.7	14.1

Table C.12: Time to do 250,000 durable insertions on a single process into the various search trees, with write-caches on the harddrives enabled. All times are in ms.