Chapter 4

A Performance Study

Memory-mapped transactions are easy to use because the underlying system automatically handles I/O operations and concurrency control, but this convenience comes at a cost. In this chapter, I describe several experiments designed to measure the performance of memory-mapped transactions in Libxac. I use these results to construct approximate performance models for both nondurable and durable transactions whose working sets fit into main memory.

In Section 4.1, I describe the four different machines on which I tested the Libxac prototype: a 3.0 GHz Pentium 4, a 2-processor 2.4 GHz Xeon, a 4-processor, 1.4 GHz AMD Opteron, and a 300 MHz Pentium II.

In Section 4.2, I estimate the cost of executing a nondurable memory-mapped transaction by measuring the time required for expensive operations: entering and exiting a `SIGSEGV` handler and executing the `mmap` system calls. For a nondurable transaction that reads from R different pages and writes to W different pages, I estimate the additive overhead on a modern machine for executing the transaction is roughly of the form $aR + bW$, where $b \approx 2a$ and a is in the range of 15 to 55 μs. Using the advisory function to inform the runtime which pages a transaction accesses reduces this overhead on modern machines by anywhere from 20% to 50%.

Section 4.3 describes similar experiments for estimating the cost of executing a durable memory-mapped transaction. The single-most expensive operation for a durable transaction is the call to `fsync` that forces data to be written out to disk on a transaction commit. For durable transactions that write to only a few pages, I conjecture a performance model of the form $aR + bW + c$, where a is tens of microseconds, b is a few hundred microseconds, and c is fixed cost of anywhere from 5 to 15 ms.

Section 4.4 presents results from experiments designed to measure the potential concurrency of Libxac. On a multiprocessor machine, when two different processes ran independent nondurable transactions, Libxac achieved near-perfect linear speedup when the work done on the page touched by each transaction required time approximately two orders of magnitude greater than the overhead of accessing that page. With durable transactions, however, the same Libxac program did not exhibit any speedup on two processors, most likely because the time to synchronously write data to disk during a transaction commit represents a serial bottleneck.

Since the overhead of automatically detecting pages accessed is significant for nondurable transactions, and the synchronous writes to disk are a serial bottleneck for concurrent, durable transactions, the current Libxac prototype seems best suited for executing durable transactions on a single process. Improving the implementation to work with multiple disks may improve the performance of concurrent, durable transactions.

4.1 The Hardware

This section describes the four machines I used for testing the Libxac prototype.

1. A 3.0GHz Pentium 4 with hyperthreading and 2GB of RAM. The system has a RAID 10

Machine	Processor Speed	Time per Cycle
1	3.0 GHz	0.33 ns
2	2.4 GHz	0.42 ns
3	1.4 GHz	0.71 ns
4	300 MHz	3.33 ns

Table 4.1: Processor speeds and time per clock cycle for the test machines.

configured with 4 120GB, 7200-rpm SATA disks with 8MB cache and a 3ware 8506 controller. The drives were mounted with an ext3 filesystem. The system runs Fedora Core 2 with Linux kernel 2.6.8-1.521-smp.[1]

2. A machine with 2, 2.4GHz Intel Xeon processors and 1GB of RAM. The harddrive is a 80GB, 7200rpm Barracuda ATA IV with a 2MB cache. The drive was mounted with an ext3 filesystem. The system was running Fedora Core 1 with Linux kernel 2.4.22-1.2197.nptlsmp.

3. A machine with 4, 1.4GHz AMD64 Opteron processors and 16 GB of RAM. The system was running SUSE 9.1, with Linux kernel 2.6.5-7.147-smp. This machine has two different disks that I ran tests on:

   (a) A 73.4GB, 10,000rpm IBM Ultrastar 146Z10 with 8MB cache and an Ultra320 SCSI interface, mounted with the ReiserFS filesystem.

   (b) A 146.8GB, 10,000rpm IBM Ultrastar disk with 8MB cache and an Ultra320 SCSI interface, mounted with an `ext3` filesystem.

4. A 300 MHz Pentium II with 128 MB of RAM, running Redhat 8.0 with Linux Kernel 2.4.20. The machine has a 4.3 GB ATA disk drive.

The first three machines are relatively modern machines; the first is a standard single-processor machine while the second and third are more expensive multiprocessor machines. The fourth is a relatively old system with a slow processor and limited amount of memory. Testing LIBXAC on these different systems ensures I am not obtaining results that are specific to a single machine.

Each aspect of the system has a different effect on LIBXAC's performance. The type and number of processors affects how quickly transactions can execute and how much concurrency we can expect to get. See Table 4.1. The amount of RAM determines how many pages can be buffered in main memory. A nondurable transaction that accesses data that is entirely in memory should not incur the cost of disk writes. Changes made between Linux kernel 2.4 and 2.6 may affect the performance of system calls such as `mmap`. Finally, for durable transactions, the cost of `msync` and `fsync` depends on the rotational latency, seek times, and write speed of the disk drives.

Unless otherwise noted, all times for nondurable transactions in Section 4.2 were measured by reading the processor's cycle counter twice using the `rdtsc` instruction, and recording the time difference. For durable transactions, in Section 4.3, I instead used `gettimeofday`. See Appendix C.1 for details on the resolution of the timers.

4.2 Performance of Nondurable Transactions

In this section, I present a rough performance model for small nondurable transactions. The additive overhead on a modern machine for executing a nondurable transaction that reads from R distinct pages and writes to W distinct pages is approximately $aR + bW$, where a is a constant between 10 and 60 μs and $b \approx 2a$. Using an advisory function to inform the runtime which pages a transaction accesses reduces this overhead by anywhere from 20 to 50%. I also describe several experiments that suggest that a significant fraction of the overhead for a transaction is spent entering and exiting the `SIGSEGV` handler, calling `mmap`, and handling page faults.

[1]The kernel was modified to install `perfctr`, a package for performance monitoring counters.

```
// Transaction that reads
//   from n pages
int* x = xMmap("input.db",
               n*PAGESIZE);
int i, value;
   ...

xbegin();
for (i = 0; i < n; i++) {
  value = x[i*PAGESIZE/sizeof(int)];
}
xend();
```

(a)

```
// Transaction that writes
//   to n pages
int* x = xMmap("input.db",
               n*PAGESIZE);
int i;
   ...

xbegin();
for (i = 0; i < n; i++) {
  x[i*PAGESIZE/sizeof(int)] = i;
}
xend();
```

(b)

Figure 4-1: A simple transaction that (a) reads from n consecutive pages, and (b) writes to n consecutive pages.

4.2.1 Page-Touch Experiment

For a transaction that reads from R pages and writes to W pages, I conjecture a performance model of $aR + bW$ and in this section I describe experiments designed to estimate the constants a and b. From the results, I conclude that for small nondurable transactions, a is on the order of tens of microseconds, and $b \approx 2a$. Whenever a transaction reads from or writes to a new page, the Libxac runtime must `mmap` the appropriate page in memory. During a transaction commit or abort, the runtime must change the memory protection of every accessed page back to no-access. The performance model is reasonable because the time for these operations is proportional to the number of pages accessed.

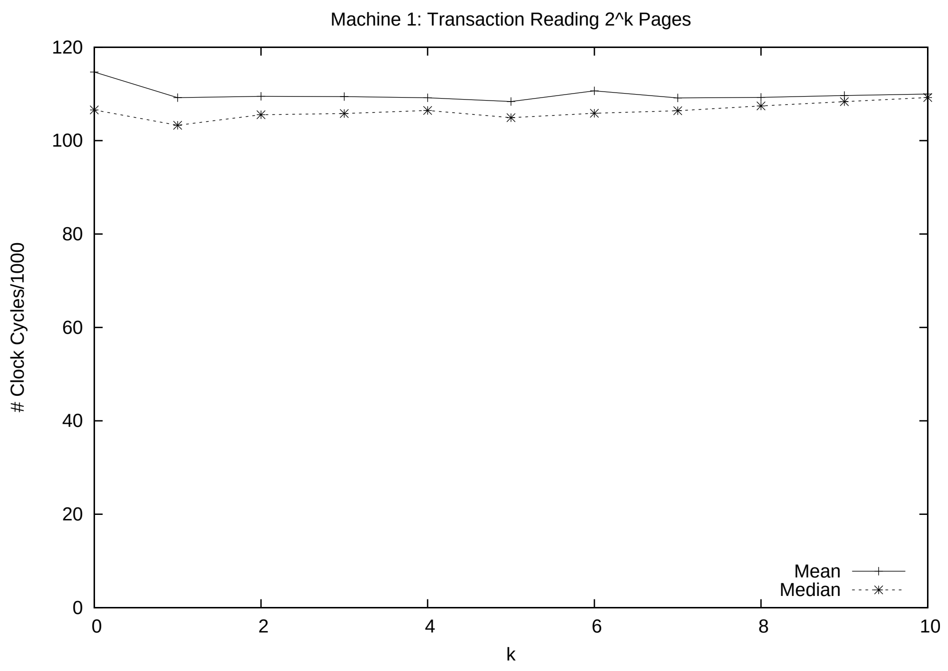
I tested the accuracy of this model with an experiment that ran the transactions shown in Figure 4-1. Transaction (a) reads from n consecutive pages, while (b) writes to n consecutive pages. Figure 4-2 shows the results on Machine 1 for transactions (a) and (b) as n varies between 1 and 1024. Transaction (a) took an average of about 110,000 clock cycles (37 μs) per page read. The data was less consistent for (b); for $n \geq 2^4$, the average time per page written was roughly 200,000 clock cycles.

The sharp peak for transaction (b) in Figure 4-2 is somewhat typical behavior for long running Libxac programs. Although the average and median times to execute a small transaction are fairly stable, the maximum time is often one or more orders of magnitude greater than the average time. Since Linux does not generally provide any bound on the worst-case time for `mmap` or other system calls, this result is not too surprising. Also, because I measure real time in all experiments, any time when the test program is swapped out is also included. One of these phenomena may possibly explain the sharp peak.

4.2.2 Page-Touch with an Advisory Function

In this section, I present data that suggests that using the advisory function described in Section 2.3 for the page-touch experiment reduces the cost per page access on a modern machine by anywhere from 20 to 50%. Libxac detects which pages a transaction accesses by handling the `SIGSEGV` signal caused by the transaction first attempt to read from or write to a page. The advisory function, `setPageAccess` informs the runtime that a transaction is about to access a page. This optimization replaces the triggering and handling of a `SIGSEGV` with a function call.

Figure 4-3 exhibits the programs from the page-touch experiment, modified to call the advisory function before each page access. Figure 4-4 plots the average access times per page as n varies from 1 to 1024 on Machine 1. The advisory function reduced the average time per page read and write from 110,000 to about 60,000 clock cycles and from 200,000 to just under 100,000 clock cycles, respectively. Thus, on Machine 1, the advisory function cut down the per-page overhead by almost a factor of 2. Also, with the advisory function, the plots do not have any sharp peaks, suggesting



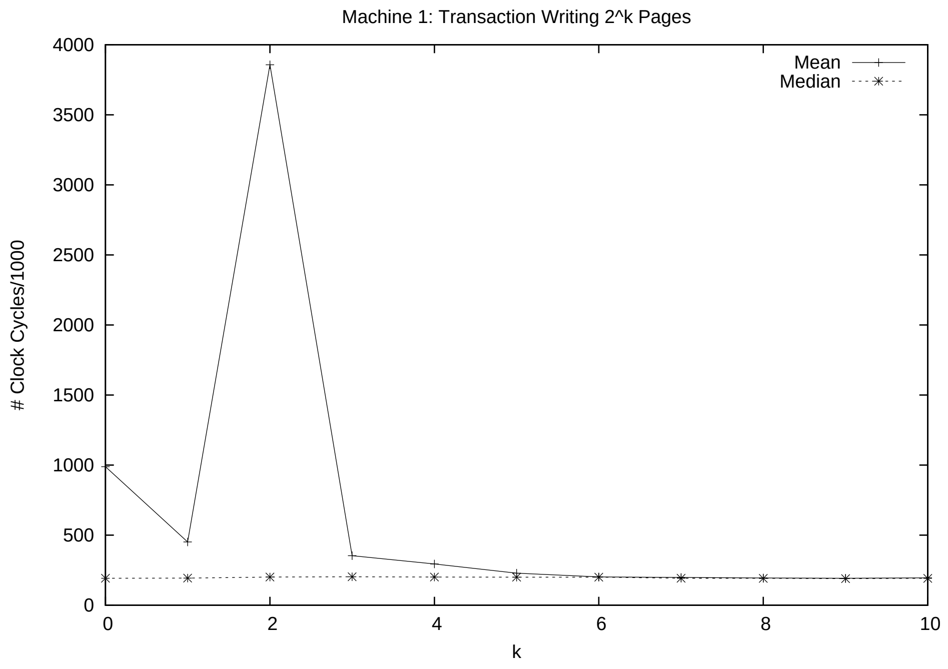


Figure 4-2: Average time per page to execute the transactions shown in Figure 4-1 on Machine 1. For each value of n, each transaction was repeated 1000 times.

```
// Transaction that reads
//   from n pages
int* x = xMmap("input.db",
               n*PAGESIZE);
int i, value;
   ...

xbegin();
for (i = 0; i < n; i++) {
  int j = i*PAGESIZE/sizeof(int);
  setPageAccess(&(x[j]),
                READ);
  value = x[j];
}
xend();
```

(a)

```
// Transaction that writes
//   to n pages
int* x = xMmap("input.db",
               n*PAGESIZE);
int i;
   ...

xbegin();
for (i = 0; i < n; i++) {
  int j = i*PAGESIZE/sizeof(int);
  setPageAccess(&(x[j]),
                READ_WRITE);
  x[j] = i;
}
xend();
```

(b)

Figure 4-3: The transactions in Figure 4-1 written with the advisory function. The transaction in (a) reads from n consecutive pages, the transaction in (b) writes to n consecutive pages.

Machine	t_{read}			t_{write}		
#	Normal	With Adv.	% Speedup	Normal	With Adv.	% Speedup
1	110.0	58.5	47%	195.5	88.2	55%
2	44.4	31.1	31%	91.6	63.4	31%
3	21.7	16.0	26%	43.5	34.8	20%
4	16.1	10.1	37%	149.4.	143.8	4%

Table 4.2: Average # of clock cycles per page access for transactions touching 1024 pages, with and without the advisory function. Numbers are in thousands of cycles. Percent speedup is calculated as $100\left(\frac{\text{Normal - With Adv}}{\text{Normal}}\right)$.

that eliminating the need to handle SIGSEGVs reduced the variability of the experiment.[2]

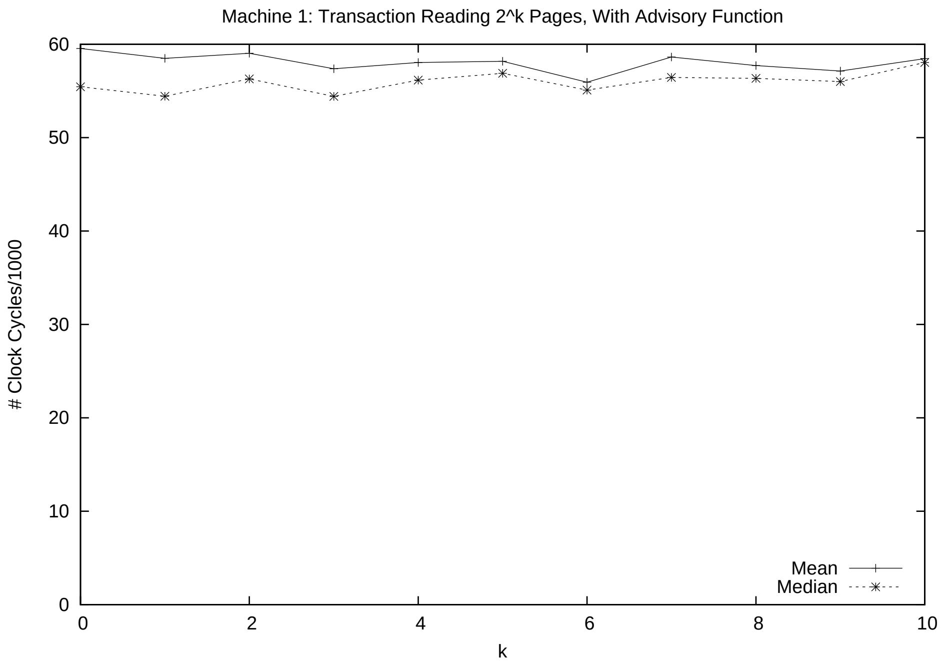
Table 4.2 summarizes the average access times per page for all machines for $n = 1024$. The advisory function improved performance on Machine 1 by approximately a factor of 2, but the improvement was less significant on the other machines. For example, on Machine 3, the speedup was only 26% and 20% for page reads and writes, respectively. One explanation that I discuss in Section 4.2.3 is that the cost of entering a SIGSEGV handler appears to be particularly expensive on Machine 1 compared to the other machines. The speedup for writes on Machine 4 was only 4%. Since Machine 4 has only 64 MB, it is possible that this experiment no longer fits into main memory. Machine 4's slow processor speed might be another factor.

Converting the time per page read from clock cycles to microseconds, we find that a, the time per page read, ranges from 15 μs on Machine 3 to 54 μs on Machine 4. On all but Machine 4, b, the time per page write, is slightly less than $2a$.

4.2.3 Decomposing the Per-Page Overhead

In this section, I attempt to account for the per-page overheads observed in Sections 4.2.1 and 4.2.2 by estimating the times required for individual runtime operations. The experimental results suggest that in most cases, over half the overhead can be explained by the time required to handle a SIGSEGV signal, to call mmap, to copy a page on a transaction write, and the time to handle a page fault. I conjecture that most of the remaining time is spent handling cache misses and additional page faults.

[2]Similar plots for other machines are shown in Appendix C.2.



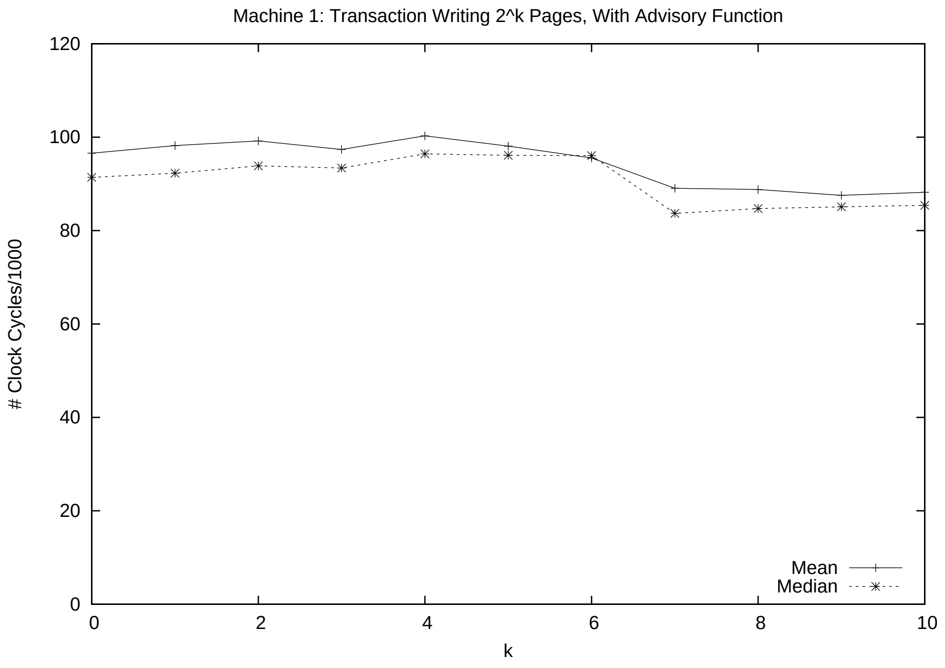


Figure 4-4: Average time per page to execute the transactions shown in Figure 4-3 on Machine 1. For each value of n, each transaction was repeated 1000 times.

Machine	Entering `SIGSEGV`	Exiting `SIGSEGV`	`mmap`	
#	Handler (Cycles)	Handler (Cycles)	Clock Cycles	μs
1	32,216	29,323	15,156	5.05
2	8,032	9,723	10,054	4.19
3	3,489	4,745	3,228	2.31
4	3,078	3,140	2,282	7.61

Table 4.3: Number of clock cycles required to enter `SIGSEGV` handler, call `mmap`, and exit handler (average of 10,000 repetitions).

Memory Mapping and Fault Handlers

As I described in Section 3.1, the runtime detects which pages a transaction accesses by mapping pages with no-access or read-only memory protections and handling `SIGSEGV` signals. I describe results from an experiment that measured the time required for the three expensive operations in this process: entering the fault handler, calling `mmap`, and exiting the handler.

In the experiment, I ran a loop that captures the basic actions that the runtime executes when a transaction tries to read from or write to a new page for the first time. The times required for the loop operations give a lower bound on the overhead of a nondurable transaction. Starting with a memory-mapped file that is initially unmapped, I timed 10,000 repetitions of the following loop:

1. Attempt to increment the first `int` stored on the first page of the file. Measure the time required to enter the `SIGSEGV` handler for this operation.

2. Inside the `SIGSEGV` handler, `mmap` the same page with read-write protection. Record the time for this operation.

3. Measure the time required to exit the `SIGSEGV` handler.

4. Unmap the first page in the file and repeat.

Table 4.3 reports the average number of clock cycles required to do these operations on each machine. The main observation is that entering and exiting fault handlers and the `mmap` system call all took several thousand to tens of thousands of clock cycles. These times are one or two orders of magnitude greater than a cache miss that takes a hundred clock cycles. Of the three modern machines, Machine 1, the Pentium 4 required the most time for these operations. In summary, a transaction must incur at least several microseconds in overhead for every new page accessed.

Although I present only average values here, there is some variability in the measured values. For example, on Machines 2 and 3, the maximum time for an `mmap` operation was about 5 times the average value. See Table C.3 in Appendix C for more complete statistics for this experiment.

First Access to an `mmaped` Page

Linux does not actually bring an `mmap`'ed page into memory until it is first accessed. The time to handle this page fault represents another expensive component of nondurable transactions. In this experiment, I `mmap`'ed a single page of a file with read/write protection and then recorded the time required to increment the first integer on the page for the first time. Table 4.4 shows that these times are comparable to the time to enter and exit a fault handler.

Test of `memcpy`

Before a transaction writes to a page for the first time, Libxac first makes a copy of the old page. In this experiment, I measured the time required for a `memcpy` between two 4K character arrays. In Table 4.5, the first value is the time for the program's first `memcpy`, the second is the average time

Machine	Avg. # of Clock Cycles	σ	Time in μs
1	16,851	945	5.6
2	5,310	139	2.2
3	3,311	211	2.3
4	3,995	1,133	13.3

Table 4.4: Clock cycles required to write to a page for the first time after memory mapping that page. Each experiment was repeated 5 times.

Machine	First `memcpy`	Avg. of Ops 2 through 5	Avg. of 1000 Ops.
1	44,067	2,493	1,122
2	19,497	1,718	1,052
3	9,578	676	608
4	7,324	3,320	932

Table 4.5: Clock cycles required for a `memcpy` between two 4K character arrays in memory.

of the 2nd through 5th `memcpy`'s, and the third is the average time for 1,000 `memcpy`'s between the two arrays.[3]

The data suggests that optimizing the LIBXAC runtime to avoid cache misses and page faults could significantly improve performance. For the first two machines, a `memcpy` between two pages that are already in cache (the third value) costs approximately an order of magnitude less than the `mmap` from Table 4.3. The difference is less for the other two machines, but a `memcpy` is still faster than an `mmap`. On the other hand, a `memcpy` when the arrays are not cached is a factor of about 2 to 3 more expensive than an `mmap` call.

Predicting Per-Page Overhead

I use the data in Tables 4.3, 4.4, and 4.5 to try explain the per-page overheads observed in the page-touch experiments from Sections 4.2.1 and 4.2.2.

For transaction (a) in Figures 4-1 and 4-3, recall that every time a transaction tries to read from a page for the first time, the runtime must handle a `SIGSEGV` and change the memory map from a page mapped as no-access to a (possibly different) page mapped read-only. At least one page fault occurs when the transaction actually reads from the page. Finally, when the transaction commits, the memory protection on that page must be changed back to no-access. Thus, transaction (a) involves handling one `SIGSEGV`, two calls to `mmap` and one page fault. Using the advisory function eliminates the cost of entering and exiting the fault handler.

On a given machine, let $t_{\text{segvEnter}}$ and t_{segvExit} be the average time to enter and exit a `SIGSEGV` handler, respectively, and let t_{mmap} and $t_{\text{pageFault}}$ be the average times to call `mmap` and to handle a page fault, respectively. Then, based on this model, a lower bound on t_{read} and t'_{read}, the time to read a new page inside a transaction, without and with the advisory function, is[4]

$$\begin{aligned} t_{\text{read}} &\geq t_{\text{segvEnter}} + t_{\text{segvExit}} + 2t_{\text{mmap}} + t_{\text{pageFault}} \\ t'_{\text{read}} &\geq 2t_{\text{mmap}} + t_{\text{pageFault}} \end{aligned} \tag{4.1}$$

When a transaction writes to a page, the runtime handles two `SIGSEGV`'s: the first for reading the page, and the second to copy the page and `mmap` the new copy with read/write protection. Thus, a page write requires one extra fault handler, an additional call to `mmap`, and one `memcpy`.[5] Using

[3]The first and second values are averages from 5 repetitions of the experiment. The third is an average from 1000 repetitions of the experiment. See Table C.4 for detailed data.

[4]To get a more accurate lower bound, I actually subtract $t_{\text{timerDelay}}$, the delay in our timer, once from every measured value measured (in this case, $5t_{\text{timerDelay}}$).

[5]For t_{memcpy}, I use the value the smallest value, value 3, from Table 4.5.

Machine	t_{read}				t_{write}			
#	Est.	Actual	Δ	% Diff.	Est.	Actual	Δ	% Diff.
1, no Adv.	108.2	110.0	1.8	**1.6%**	185.7	195.5	9.8	**5.0%**
1, w Adv.	46.9	58.5	11.6	19.8%	47.9	88.2	40.3	45.7%
2, no Adv.	43.2	44.4	1.3	**2.8%**	72.0	91.6	19.6	**21.3 %**
2, w. Adv.	25.4	31.1	5.7	18.2%	26.5	63.4	26.5	58.2 %
3, no Adv.	18.0	21.7	3.8	**17.4%**	30.0	43.5	13.5	**31.0 %**
3, w. Adv.	9.7	16.0	6.3	39.3%	10.3	34.8	24.4	70.3 %
4, no Adv.	14.6	16.1	1.6	**9.7%**	23.9	149.4.	125.6	**84.0 %**
4, w. Adv.	8.4	10.1	1.7	16.4%	9.3	143.8.	134.5	93.5 %

Table 4.6: Average # of clock cycles per page access for transactions touching 1024 pages. All numbers are in thousands of clock cycles.

the advisory function eliminates the entering and exiting of the `SIGSEGV`, and also one call to `mmap`, as the runtime does not need to map the original version of the page as read-only first. Thus, I estimate t_{write} and t'_{write}, the time required for a transaction to write to a new page, without and with the advisory function, as

$$\begin{aligned} t_{\text{write}} &\geq 2t_{\text{segvEnter}} + 2t_{\text{segvExit}} + 3t_{\text{mmap}} + t_{\text{pageFault}} + t_{\text{memcpy}} \\ t'_{\text{write}} &\geq 2t_{\text{mmap}} + t_{\text{pageFault}} + t_{\text{memcpy}} \end{aligned} \tag{4.2}$$

After repeating the transactions in Figures 4-1 and 4-3 1,000 times, with $n = 1024$, I obtained the data shown in Table 4.6. I use the average values from Table 4.3 as estimates for $t_{\text{segvEnter}}, t_{\text{segvExit}}$, and t_{mmap}, and I use the data in Table 4.4 and 4.5 as estimates for $t_{\text{pageFault}}$ and t_{memcpy}, respectively.

On modern machines, a majority of the overhead for a nondurable transaction comes from handling the `SIGSEGV` signal, calling `mmap`, copying the page, and handling page faults. On Machines 1, 2, and 4, the lower bound in (4.1) accounted for all but 10% of t_{read}. The bound in (4.2) was less accurate for predicting t_{write}, however. Generally, the predictions were most accurate for Machine 1, which has the greatest overhead in clock cycles for the fault handler and the `mmap` operations. With the exception of t_{write} on Machine 4, the predictions for t_{read} and t_{write} account for at least 50% of the time spent per page access.

The lower bounds are even less successful for predicting the overheads when the advisory function is used. Equation (4.1) accounts for at least 50% of t'_{read}, but equation (4.2) significantly underestimates t'_{write}. One reason for this discrepancy is that Equations (4.1) and (4.2) assume the time spent updating control data structures is negligible. Although the computation required to modify the transaction state or maintain the consistency tree is not significant for a small transaction running on a single process, any cache misses or page faults incurred when accessing the control data structures may noticeably increase the overhead. For example, when running the transaction in Figure 4-1(a) with $n = 1$ on Machine 3, I noticed that the LIBXAC runtime spent 21,000 cycles inside the fault handler, but actually spent about 11,500 cycles adding the first page to the transaction's readset. Since this operation only involves adding an element to an empty list stored in an array, this operation is probably expensive only because a page was not cached in main memory.

I conjecture that most of the unexplained overhead is due to poor caching behavior. LIBXAC's data structures for a transaction's readset and writeset and for the transaction page tables are both implemented as large, fixed-size arrays. These data structures are unlikely to exhibit significant locality, since they are stored in a 59 MB control file. This phenomenon may be even more noticeable on Machine 4, which is only 300 MHz and has only 64 MB of RAM. With a more efficient implementation of the control structures, the discrepancies may be lower.

Although Equations (4.1) and (4.2) underestimate the overhead, for small transactions that fit into memory, the simple linear performance model of $aR + bW$ should still be reasonable. Failing to account for all the page faults or cache misses should only increase the constants a and b.

Machine	Avg. Time (μs)	
#	Page Read	Page Write
1, no Adv.	39	1,569
1, w. Adv.	22	1,297
3(a), no Adv.	15	235
3(a), w. Adv.	14	179
3(b), no Adv.	16	227
3(b), w. Adv	15	182

Table 4.7: Average Access Time (μs) per Page, for Transactions Touching 1024 Pages.

4.3 Durable Transactions

In this section, I attempt to construct a performance model for durable transactions and test this model by repeating some of the experiments described in Section 4.2 for durable transactions.[6] Durable transactions incur extra overhead when they commit because the runtime must synchronously force data to the log file on disk using an `fsync`. This disk write should add both a large constant overhead to all transactions for the latency of accessing disk, and a roughly constant overhead per page written for updating the log file. Thus, I conjecture that a small durable transaction that reads from R pages and writes to W page incurs an additive overhead of $aR + bW + c$, where a is tens of microseconds, b is hundreds to a few thousand microseconds, and c is between 5 and 15 milliseconds.

Page Touch Experiments

Table 4.7 presents data from page-touch experiments for durable transactions, when $n = 1024$.[7] Since each transaction runs for tens to hundreds of milliseconds, the latency of accessing disk is not significant and I can use this data to construct slight over-estimates of a and b.

From the data, we see that the per-page overhead for nondurable and durable read-only transactions are roughly the same. Since `fsync` for a read-only transaction writes only meta-data to disk, the latency to access disk is amortized over 1024 pages. The time required per page write, however, increases by several orders of magnitude. Machine 1, which only has SATA drives, performs page writes 6 or 7 times slower than Machine 3, which has SCSI drives. It is possible, however, that other effects may explain this difference.

Synchronizing a File

To estimate c, the latency of a write to disk, I measured the time required to synchronize a memory-mapped file after modifying a one randomly selected page. This benchmark repeats the following loop:

1. Pick a page uniformly at random from the 10,000 page memory-mapped file. Increment the first `int` on that page.[8]

2. Measure the time for an asynchronous `msync` on that page in the file.

3. Measure the time for an `fsync` on the file.

The time required for the `msync` operation was on average less than 10 μs on all machines. In Table 4.8, I report the times required to do `fsync`. The average time for to write a page out to disk

[6]For these tests, I disabled the write-cache to ensure that `fsync` actually writes all data to disk. I omit results from Machine 2, because I had insufficient permissions to disable the write-cache.

[7]Table C.5 gives a more complete version of this table.

[8]I had arbitrarily set the default size for Libxac's log files to 10,000 pages.

Operation	Mean	St. Dev	Min	Median	99th %tile	Max
1: `fsync`	**13.6**	184.3	2.4	8.0	12.7	5,836.5
3(a): `fsync`	**4.8**	24.0	0.7	4.0	7.0	761.6
3(b): `fsync`	**4.7**	23.1	0.8	3.9	6.9	731.9

Table 4.8: Time required to call `msync` and `fsync` on a 10,000 page file with one random page modified, 1000 repetitions. All times are in ms.

Machine	Test	Mean Time per Xaction		Speedup
		1 proc.	2 proc.	
2	A	26.2	23.0	1.14
2	B	28.3	24.2	1.16
2	C	1,786	903	1.98
3(a)	A	22.9	24.3	0.94
3(a)	B	28.1	27.5	1.02
3(a)	C	2,259	1,132	2.00
3(b)	A	24.3	24.9	0.98
3(b)	B	28.2	26.3	1.07
3(b)	C	2,248	1,130	1.99

Table 4.9: Concurrency tests for nondurable transactions. Times are μs per transaction. Speedup is calculated as time on 1 processor over time on 2 processors.

seems to be between 5 and 15 ms on average. Again, the harddrives on Machine 3 are faster than on Machine 1. The maximum time required for an `fsync` is 0.8 seconds on Machine 3, and almost 6 seconds on Machine 1. This result is consistent with the fact that the operating system does not provide any guarantees on the worst-case behavior of system calls. Expensive operations occur, but only infrequently.

4.4 Testing Concurrency in Libxac

I conclude this chapter by describing several experiments that test Libxac's performance when executing independent transactions on two concurrent processes. In an experiment for nondurable transactions, the system achieved near-linear speedup when the work done on a page touched by a transaction was about two orders of magnitude greater than the overhead of accessing that page. The same program did not exhibit significant concurrency with durable transactions, however, most likely because the writes to disk during a transaction commit represents a serial bottleneck.

Test Programs

Figure 4-5 shows the transactions used to test the concurrency of Libxac. The single-process version of Test A executes a simple transaction 10,000 times, while the two-process version runs 5,000 independent transactions on each process.[9] Since Test A does little work incrementing the first integer on a page, I also tested two other versions of this program, shown in Figure 4-6. Test B increments every integer on the page inside the transaction, and Test C repeats B's loop 1000 times. In Figure 4-6, I only show the single-process version, as the two-process version is similar.

[9]Although this code does not show it, I use the advisory function for these concurrency tests.

Test A

Single process version:

```
for (i = 0; i < 10000; i++) {
  xbegin();
    x[0]++;
  xend();
}
```

Two process version:

Process 1

```
for (i = 0; i < 5000; i++) {
  xbegin();
    x[0]++;
  xend();
}
```

Process 2

```
for (i = 0; i < 5000; i++) {
  xbegin();
    x[PAGESIZE/sizeof(int)]++;
  xend();
}
```

Figure 4-5: Concurrency Test A: Each transaction increments the first integer on a page 10,000 times.

Test B

```
xbegin();
for (j = 0;
     j < PAGESIZE/sizeof(int);
     j++) {
  x[j]++;
}
xend();
```

Test C

```
xbegin();
for (k = 0; k < 1000; k++) {
  for (j = 0;
       j < PAGESIZE/sizeof(int);
       j++) {
    x[j]++;
  }
}
xend();
```

Figure 4-6: Concurrency Tests B and C: Test B increments every integer on the page. Test C repeats the transaction in Test B 1,000 times. I omit the outermost for-loop, but as in Figure 4-5, each transaction is repeated 10,000 times.

Machine	Test	Mean Time per Xaction		Speedup
		1 proc.	2 proc.	
3(a)	A	6.12	6.16	0.99
3(a)	B	6.11	6.20	0.99
3(a)	C	6.32	6.63	0.95
3(b)	A	6.21	6.26	0.99
3(b)	B	6.22	6.21	1.00
3(b)	C	6.39	6.62	0.97

Table 4.10: Concurrency tests for durable transactions. Times are milliseconds per transaction.

Nondurable Transactions

Table 4.9 exhibits the results with nondurable transactions for Tests A, B, and C.[10] From this data, we can make several observations. First, there is no significant speedup on Machines 1 or 4. Since both machines have a single processor, this result is not surprising. The slight speedup for Tests A and B on Machine 1 may be due to the Pentium 4's hyperthreading.

Machine 2, which has 2 processors, exhibited speedup on all three tests, ranging from about 12% for Test A to almost 50% is Test C. On the other hand, Tests A and B actually run faster on one process than on two on Machine 3. Test C, which performs a lot of work on one page, does manage to achieve near-perfect linear speedup. It is unclear whether the differences between Machines 2 and 3 are due to the different architectures or due to some other factor.

These results suggest that on Machine 3, LIBXAC only exhibits concurrency for independent transactions if each transaction does significantly more work per page than the overhead for a page access. Thus, the prototype implementation of LIBXAC may not be efficient for small concurrent nondurable transactions.

Durable Transactions

Table 4.10 presents the data for Tests A, B, and C, repeated with durable transactions. For a durable transaction, the cost of forcing data out to disk at a transaction commit is significant. These results are consistent with the data from Table 4.8 for the times required to complete `fsync` on the various machines.

In these experiments, we do not observe any speedup on the multiprocessor machines. Since we are running transactions using only one disk, the `fsync` is likely to be a serialization point. Thus, it may not be possible to achieve significant speedup with only one disk without an implementation that supports group commits, i.e., committing multiple transactions on different processes with the same synchronous disk write. One possible reason for observing slowdown is that having multiple processes accessing the same log file simultaneously may cause slightly more disk head movement compared to having a single process access the file.

[10] See Table C.9 for more detailed data.