1. Configurations

1.1. Hardware

System	HPE ProLiant DL380 Gen10
CPU	Intel Xeon Gold 6240M x 2 sockets (18 cores per socket; HT disabled by BIOS); one NUMA node per socket
DRAM	DDR4 2933MHz 192GiB/socket x2 sockets (32 GiB per channel x 6 channels per socket)
Optane PMem	Apache Pass, App Direct Mode, DDR4 2666MHz 1.5TiB/socket x 2 sockets (256 GiB per channel x 6 channels per socket; interleaving enabled)
PCIe SSD	Intel DC P4800X Series SSDPED1K750GA; connected to NUMA node #0

1.2. Software

Distro	Red Hat Enterprise Linux release 8.2 (Ootpa)
Linux kernel	4.18.0-193.el8.x86_64
gcc	8.3.1-5.el8
glibc	2.28-101.el8
PMDK	1.6.1-1.el8
VTune	Intel VTune Profiler 2021.2.0
PostgreSQL	8e4b332 (master @ Tue Mar 23 00:47:06 2021 +0100)

1.3. PostgreSQL installation

```
$ ./configure --enable-debug --prefix=$HOME/postgres/[snip] --with-extra-version=-[snip] [..]
$ make
$ make install-world
```

Each PostgreSQL is installed into separated directory under non-root `$USER`'s `$HOME/postgres/`, with an extra version string generated from commit ID to identify it after installation. The `--enable-debug` option is for analysis by VTune. The `install-world` target is for `pg_prewarm` extension.

There may be additional options in the above `[..]` on the certain conditions described in Section 1.4.

1.4. PostgreSQL per-condition setup

To compare performance between patchsets and/or customized configurations, I set up several conditions and give them names shown in the following table. Note that there are four variants for "SegmentBuffer" to see how and how much performance varies with initialization and recycle configurations of WAL segment files. Also note that the variants are displayed as in their short forms in the figures in Section 3.

IMPORTANT NOTES: After the tests and analyses reported in this document were done, I found that "Map WAL segment files on PMEM as WAL buffers" v1 patchset has an issue that a "wal" checkpoint due to the amount of WAL consumed since the last checkpoint will not be requested. This issue will be fixed in v2. Note that a "time" checkpoint will be requested normally.

Name	Patchset and/or customized configuration	wal_init_zero	wal_recycle	synchronous_commit
Original	No patchset or customized configuration	true	true	true
SimplePmdk	• **"Applying PMDK to WAL operations for persistent memory" 20210322** • Add `--with-libpmem` option to `./configure` • Amend `postgresql.conf` as follows: ➢ `wal_sync_method=pmem_drain`	true	true	true
SegmentBuffer	• **"Map WAL segment files on PMEM as WAL buffers" v1** • Add `--with-libpmem` option to `./configure` • Amend `postgresql.conf` as follows: ➢ `wal_pmem_map=true`	true	true	true
|-- (no-init-zero)	"	**false**	true	"
|-- (no-recycle)	"	true	**false**	"
`-- (no-both)	"	**false**	**false**	"
OneLargeBuffer	• **"Non-volatile WAL buffer" 20210322** • Add `--with-libpmem` option to `./configure` • Amend `postgresql.conf` as follows: ➢ `nvwal_path='/mnt/pmem0/pg_wal/nvwal` ➢ `nvwal_size=80GB`	true	true	true
UnloggedAsync	• No patchset • Add `--unlogged-tables` option to `pgbench -i`	true	true	**false**

1.5. Common postgresql.conf for all conditions

```
max_connections = 300
shared_buffers = 32GB
dynamic_shared_memory_type = posix
max_wal_size = 80GB
min_wal_size = 80GB
log_timezone = 'Asia/Tokyo'
datestyle = 'iso, mdy'
timezone = 'Asia/Tokyo'
lc_messages = 'C'
lc_monetary = 'C'
lc_numeric = 'C'
lc_time = 'C'
default_text_search_config = 'pg_catalog.english'
superuser_reserved_connections = 10
wal_level = replica
fsync = on
synchronous_commit = on
wal_sync_method = fdatasync
wal_recycle = on
full_page_writes = on
wal_compression = off
checkpoint_timeout = 12min
checkpoint_completion_target = 0.7
random_page_cost = 1.0
effective_cache_size = 96GB
logging_collector = on
log_rotation_size = 0
log_checkpoints = on
log_error_verbosity = verbose
log_line_prefix = '%t %p %c-%l %x %q(%u, %d, %r, %a) '
log_lock_waits = on
autovacuum = on
log_autovacuum_min_duration = 0
autovacuum_max_workers = 4
autovacuum_freeze_max_age = 2000000000
autovacuum_vacuum_cost_delay = 20ms
autovacuum_vacuum_cost_limit = 400
log_directory = '/dev/shm/pmem/tmp.XXXXXXXXXX'
```

1.6. Common environment variables for all conditions

```
export PGHOST=/tmp
export PGPORT=5432
export PGDATABASE="$USER"
export PGUSER="$USER"
export PGDATA=/mnt/nvme0n1/pgdata
export PGCTLTIMEOUT=86400
```

2. Methods

2.1. Performance test

Run the following steps for each condition in Section 1.4 and for every combination of s = 50 or 2000 and (c, j) = (8, 8), (18, 18), (36, 18), or (54, 18). Then plot "`latency average = __ ms`" as average latency and "`tps = __ (without initial connection time)`" as throughput for each condition to draw latency-versus-throughput curve to compare the performance between conditions.

In addition, for (c, j) = (36, 18) as nearly-saturated point, plot "`progress: __ s, __ tps` ..." for each condition to compare how and how much the throughput rises and falls over time.

1. Set environment variables as in Section 1.6.
2. Create a PMEM namespace on NUMA node #0. (`sudo ndctl create-namespace -f -t pmem -m fsdax -M dev -e namespace0.0`)
3. Make an ext4 filesystem on the PMEM namespace then mount it with DAX option. (`sudo mkfs.ext4 -q -F /dev/pmem0 ; sudo mount -o dax /dev/pmem0 /mnt/pmem0`)
4. Make another ext4 filesystem on PCIe SSD then mount it. (`sudo mkfs.ext4 -q -F /dev/nvme0n1 ; sudo mount /dev/nvme0n1 /mnt/nvme0n1`)
5. Make `/mnt/pmem0/pg_wal` directory for WAL and `/mnt/nvme0n1/pgdata` directory for PGDATA.
6. Run `initdb`. (`initdb --locale=C --encoding=UTF8 -X /mnt/pmem0/pg_wal` ...)
    i. On "OneLargeBuffer" condition, also give `-P` and `-Q` options to create a large buffer file. (... `-P /mnt/pmem0/pg_wal/nvwal -Q 81920`)
7. Edit `postgresql.conf` as in Section 1.5 and amend it as in Section 1.4.
8. Start `postgres` on NUMA node #0. (`numactl -N 0 -m 0 -- pg_ctl -l pg.log start`)
9. Create a database. (`createdb --locale=C --encoding=UTF8`)
10. Initialize tables for `pgbench`. (`pgbench -i -s __` ...)
    i. On "UnloggedAsync" condition, also give `--unlogged-tables` option.
11. Stop `postgres`. (`pg_ctl -l pg.log -m smart stop`)
12. Remount the two filesystems mounted at step 3 and 4.
13. Start `postgres` on NUMA node #0 again. (`numactl -N 0 -m 0 -- pg_ctl -l pg.log start`)
14. Run `pg_prewarm` extension for all the four `pgbench_*` tables.
15. Run `pgbench` on NUMA node #1 for 30 minutes. (`numactl -N 1 -m 1 -- pgbench -r -P 10 -M prepared -T 1800 -c __ -j __`)

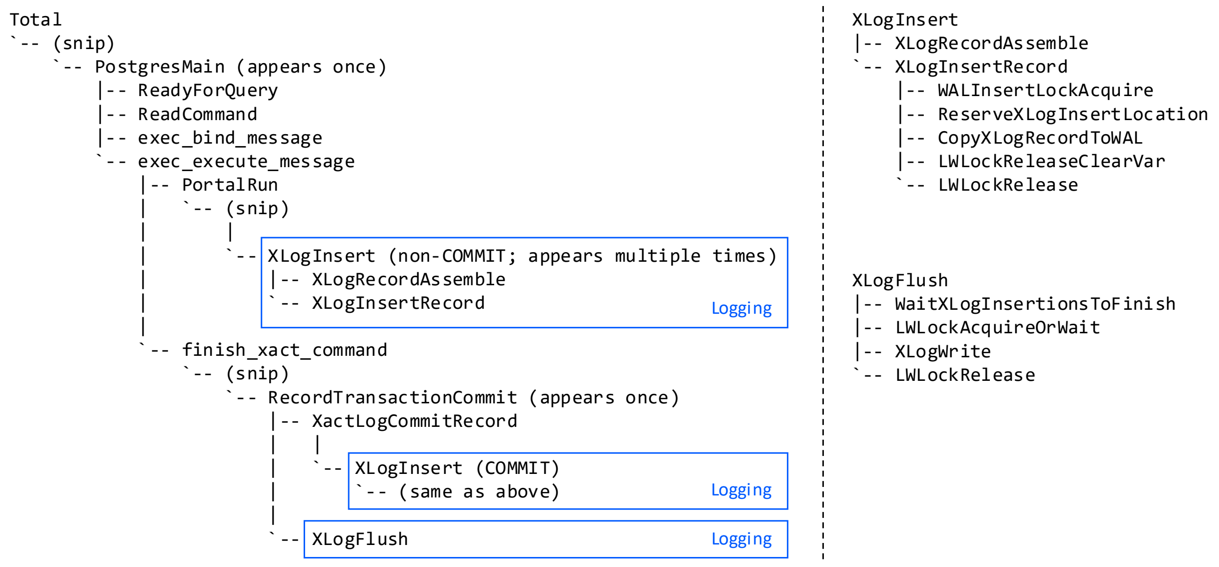
2.2. Performance analysis

Same as the performance test shown in Section 2.1, but step 13 and 15 are amended as follows to analyze `postgres` with VTune during benchmark.

13. Start `postgres` on NUMA node #0 again, with VTune started but analysis paused. Here, `postgres` command line is used instead of `pg_ctl` so as not to stop VTune due to termination of the main process of `pg_ctl`. (`vtune -collect hotspots -start-paused -finalization-mode=none -data-limit=0 -follow-child -call-stack-mode=user-plus-one -target-duration-type medium -knob sampling-mode=sw -knob enable-stack-collection=true -knob stack-size=0 -- numactl -N 0 -m 0 -- postgres`)
15. Resume VTune's analysis, then run `pgbench` on NUMA node #1 to send 2.7M transactions per client, that is, 97.2M transactions in 36-client total. After the benchmark finishes, stop VTune. (`vtune -command resume ; numactl -N 1 -m 1 -- pgbench -r -P 10 -M prepared -t 2700000 -c 36 -j 18 ; vtune -command stop`)

VTune reports how much CPU time `postgres` and its child processes took in total for each function. The following call graph is a part of what VTune told. Note that a few caller-callee relations look different from actual code, possibly due to optimization by compiler. Then I draw stacked bar charts with respect to total and logging, picking up the functions shown in the call graph that took much CPU time.

I made analyses for five conditions "Original," "SegmentBuffer," "SegmentBuffer (no-init-zero)," "OneLargeBuffer," and "UnloggedAsync" after seeing the results of performance tests in Section 3.1 and Section 3.2. Other three conditions were omitted because each of them didn't seem so worthy to be compared to the five conditions. See also discussions in Section 4.1 and Section 4.2.



```
Total
`-- (snip)
    `-- PostgresMain (appears once)
        |-- ReadyForQuery
        |-- ReadCommand
        |-- exec_bind_message
        `-- exec_execute_message
            |-- PortalRun
            |   `-- (snip)
            |       |
            |       `-- XLogInsert (non-COMMIT; appears multiple times)
            |           |-- XLogRecordAssemble
            |           `-- XLogInsertRecord                    Logging
            |
            `-- finish_xact_command
                `-- (snip)
                    `-- RecordTransactionCommit (appears once)
                        |-- XactLogCommitRecord
                        |   |
                        |   `-- XLogInsert (COMMIT)
                        |       `-- (same as above)             Logging
                        |
                        `-- XLogFlush                           Logging
```

```
XLogInsert
|-- XLogRecordAssemble
`-- XLogInsertRecord
    |-- WALInsertLockAcquire
    |-- ReserveXLogInsertLocation
    |-- CopyXLogRecordToWAL
    |-- LWLockReleaseClearVar
    `-- LWLockRelease


XLogFlush
|-- WaitXLogInsertionsToFinish
|-- LWLockAcquireOrWait
|-- XLogWrite
`-- LWLockRelease
```

3. Results

3.1. Performance test (s = 50)

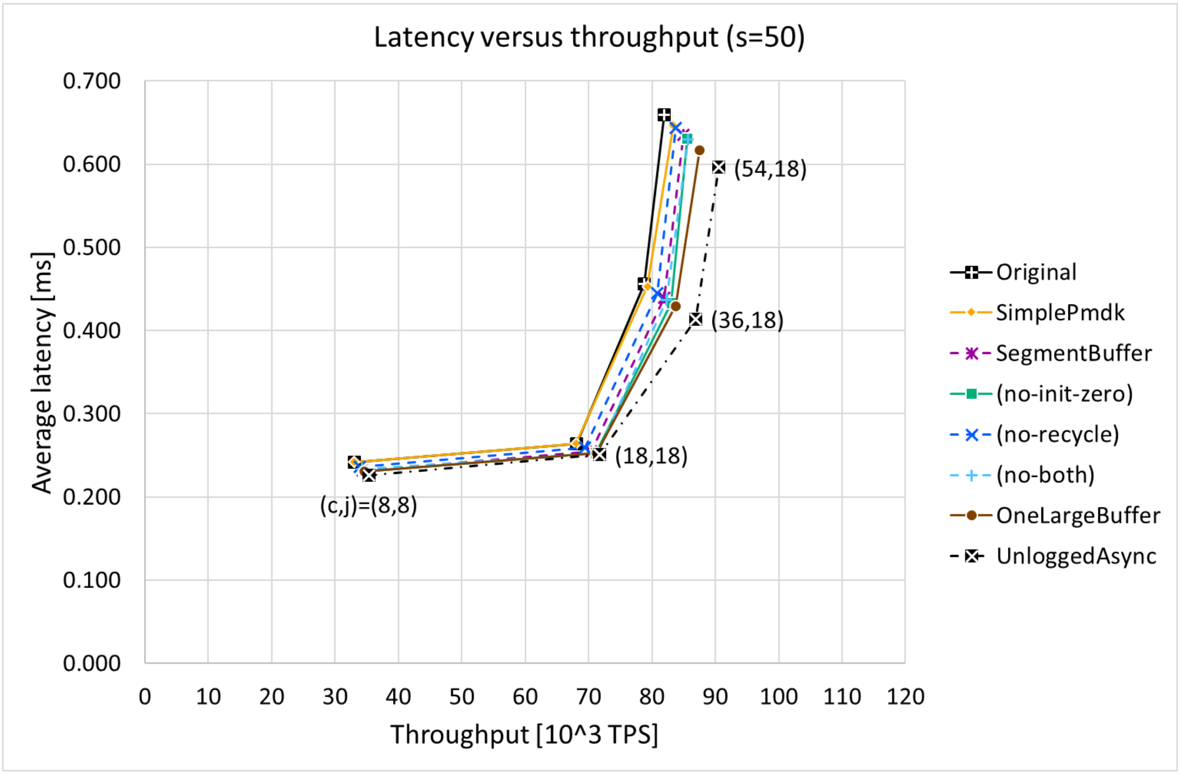


Figure 3.1-1 Latency versus throughput (s = 50) (lower-right is better)

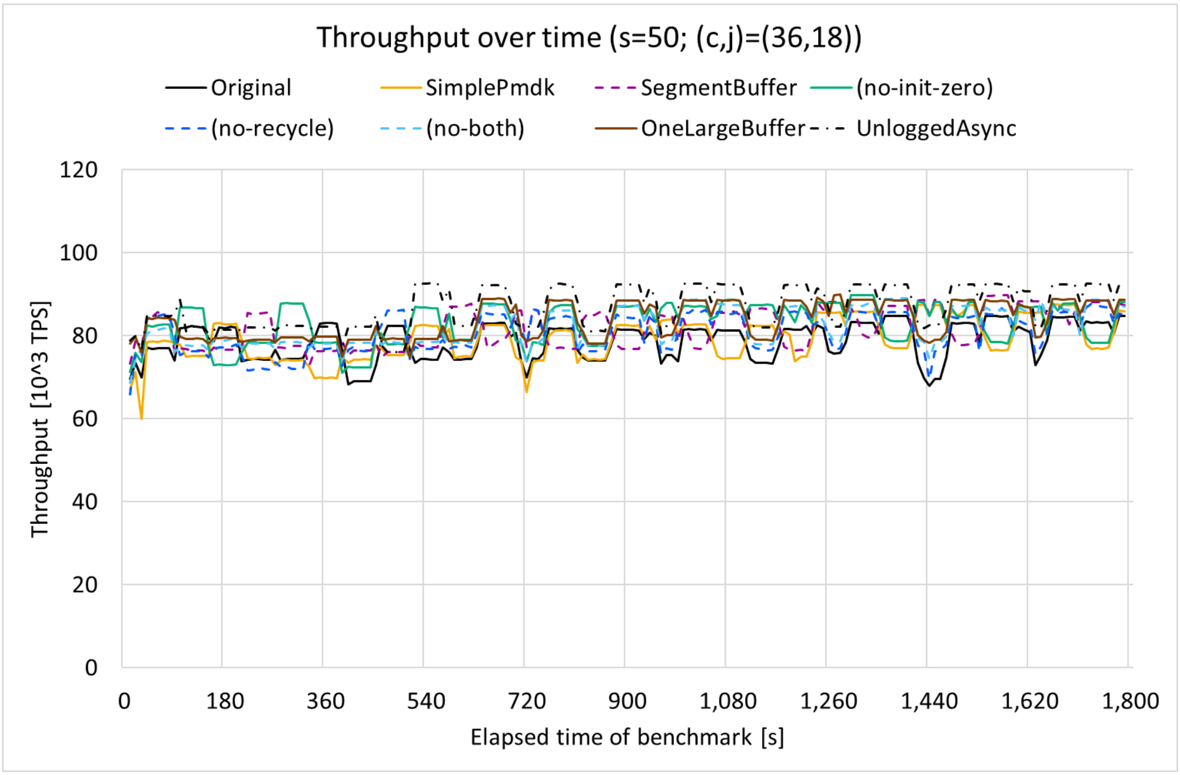


Figure 3.1-2 Throughput over time (s = 50) (higher is better)

3.2. Performance test (s = 2000)

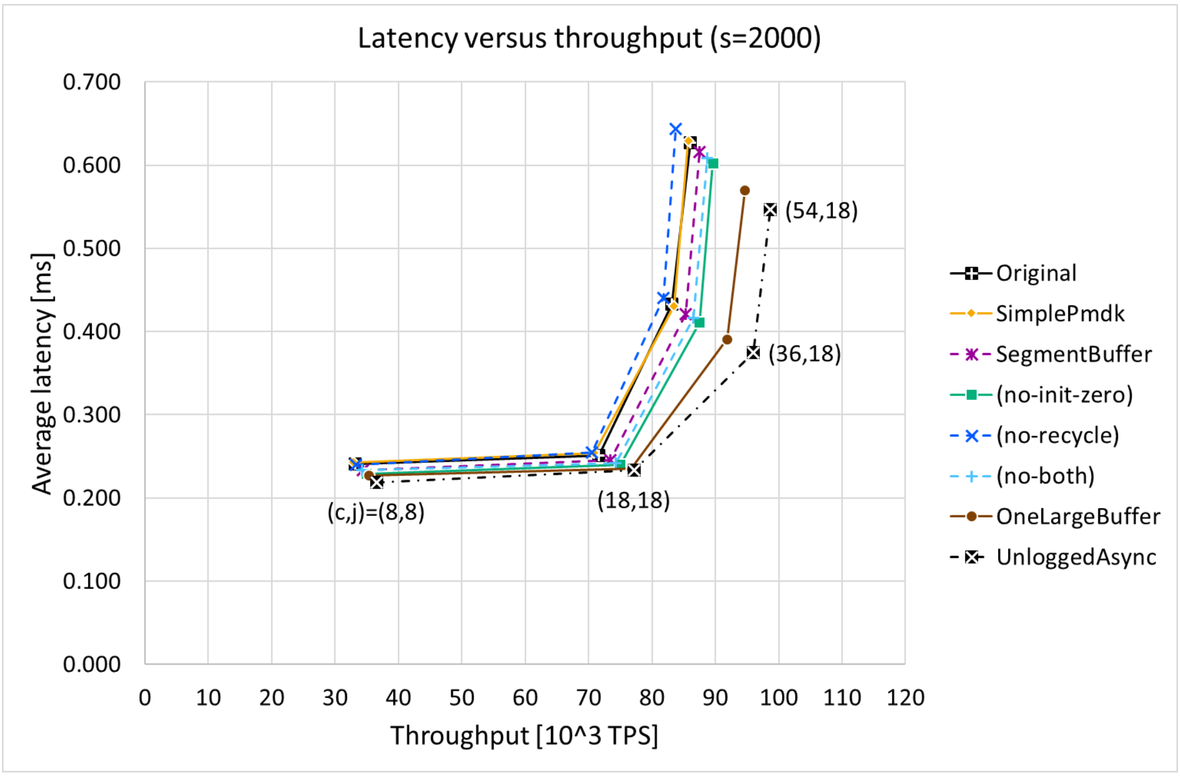


Figure 3.2-1 Latency versus throughput (s = 2000) (lower-right is better)

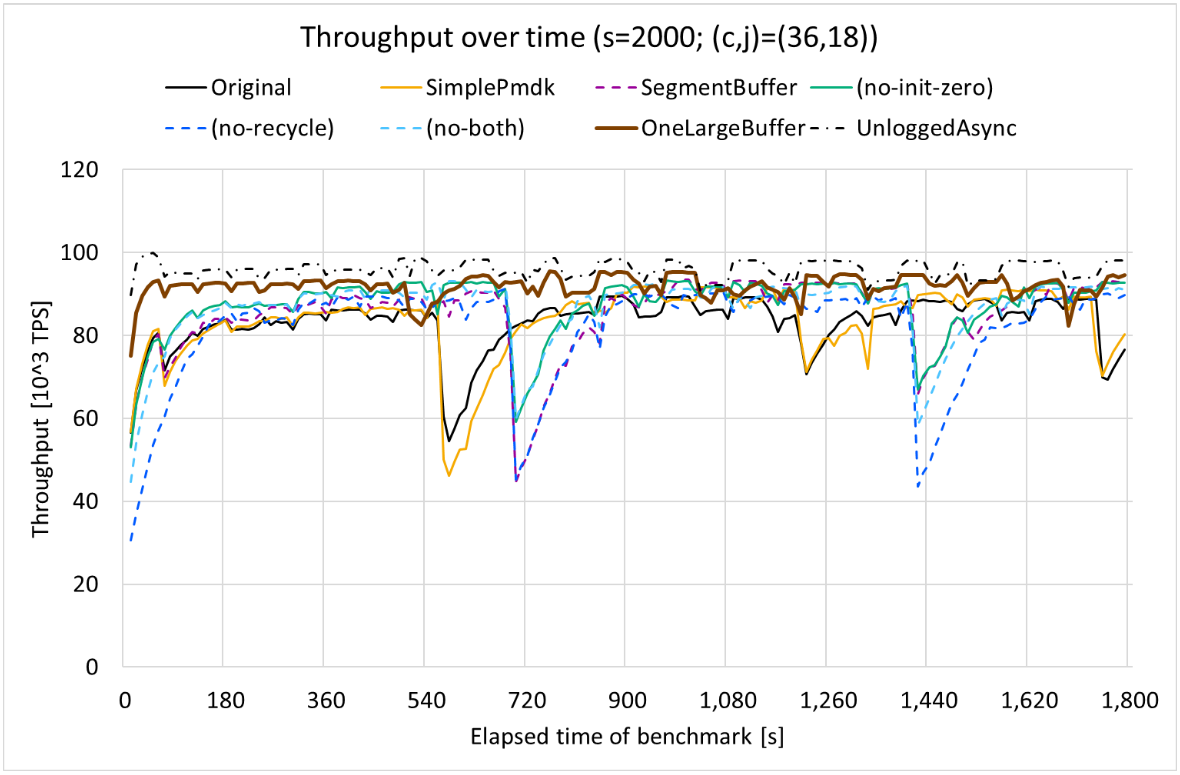


Figure 3.2-2 Throughput over time (s = 2000) (higher is better)

3.3. Performance analysis (s = 50)

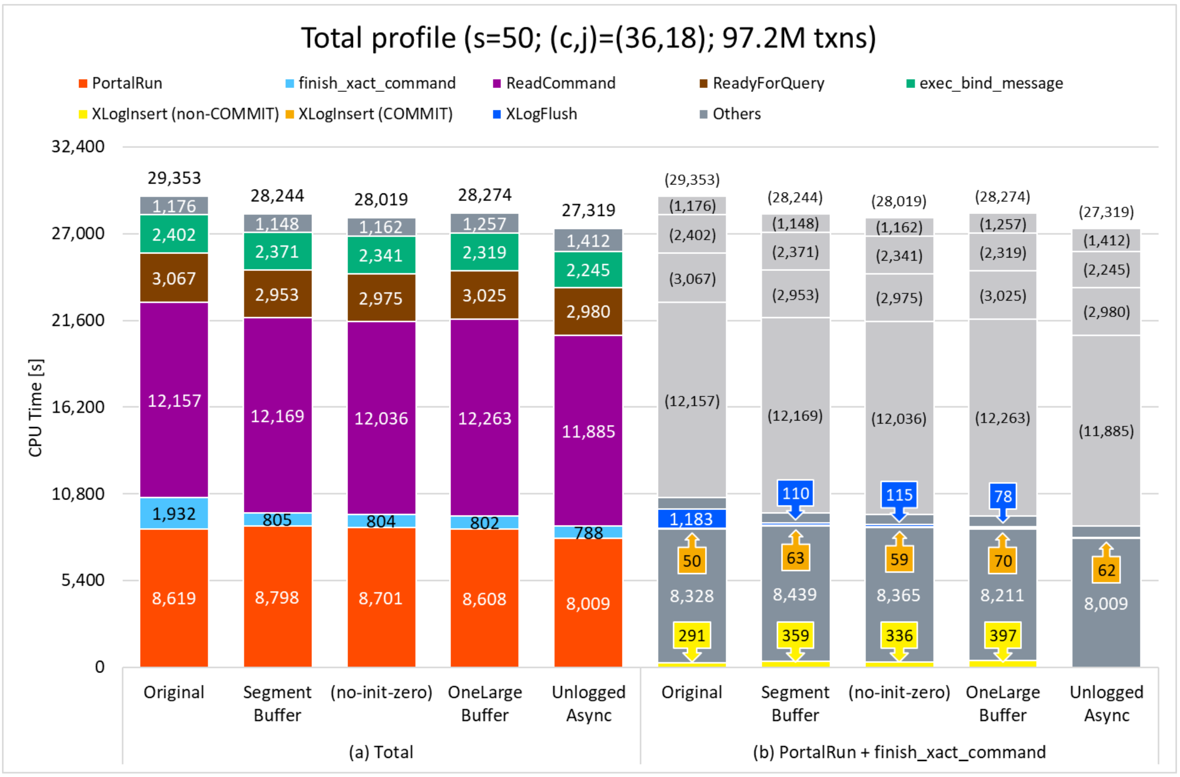


Figure 3.3-1 Total profile (s = 50) (lower is better)

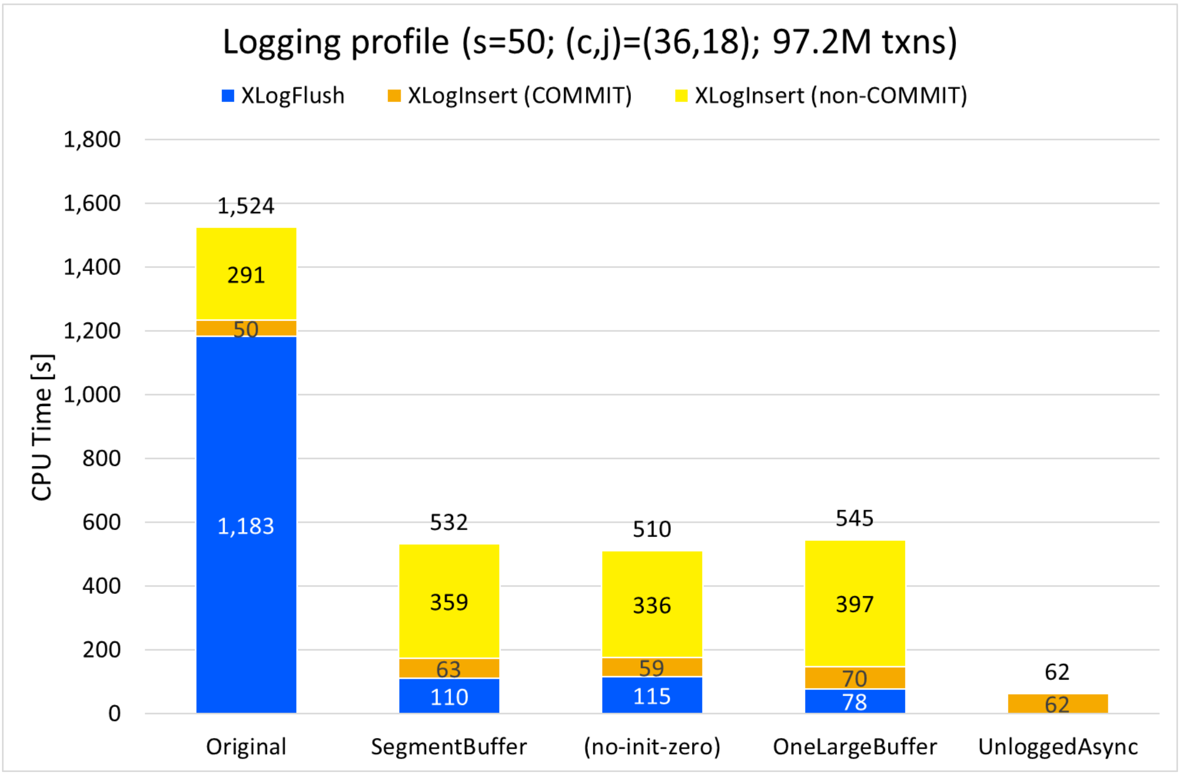


Figure 3.3-2 Logging profile (s = 50) (lower is better)

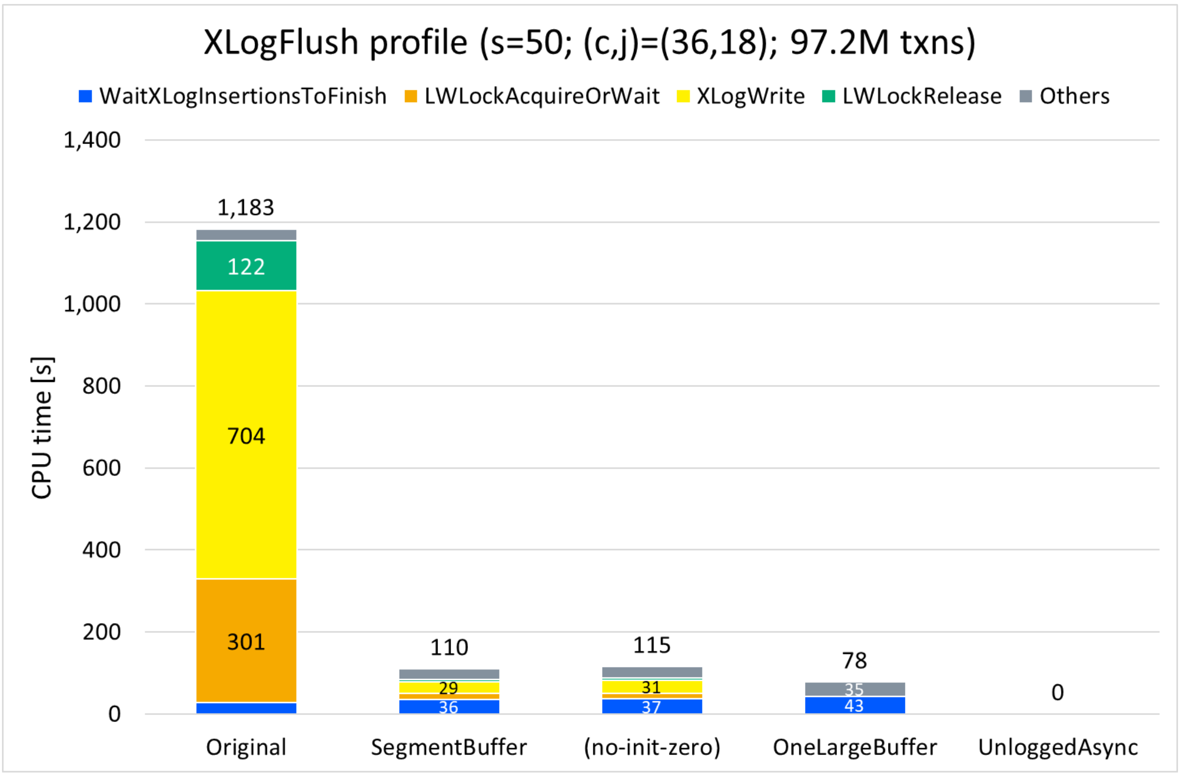


Figure 3.3-3 XLogFlush profile (s = 50) (lower is better)

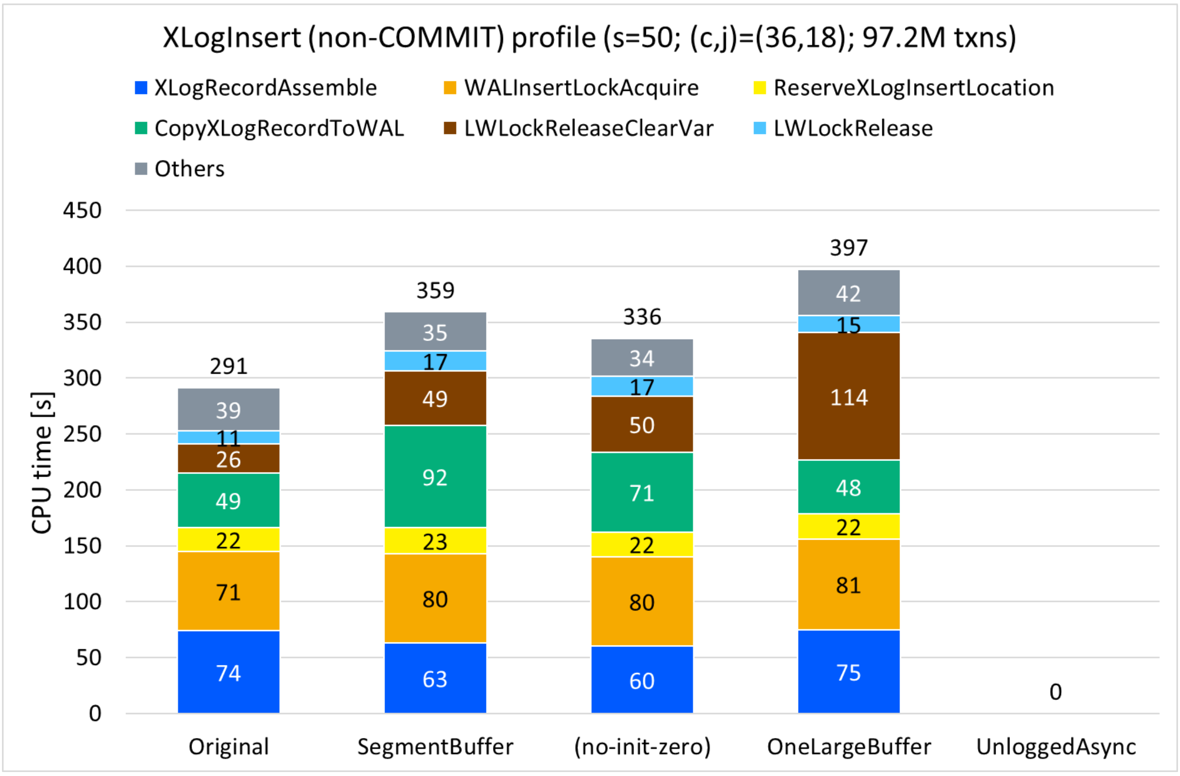


Figure 3.3-4 XLogInsert (non-COMMIT) profile (s = 50) (lower is better)

3.4. Performance analysis (s = 2000)

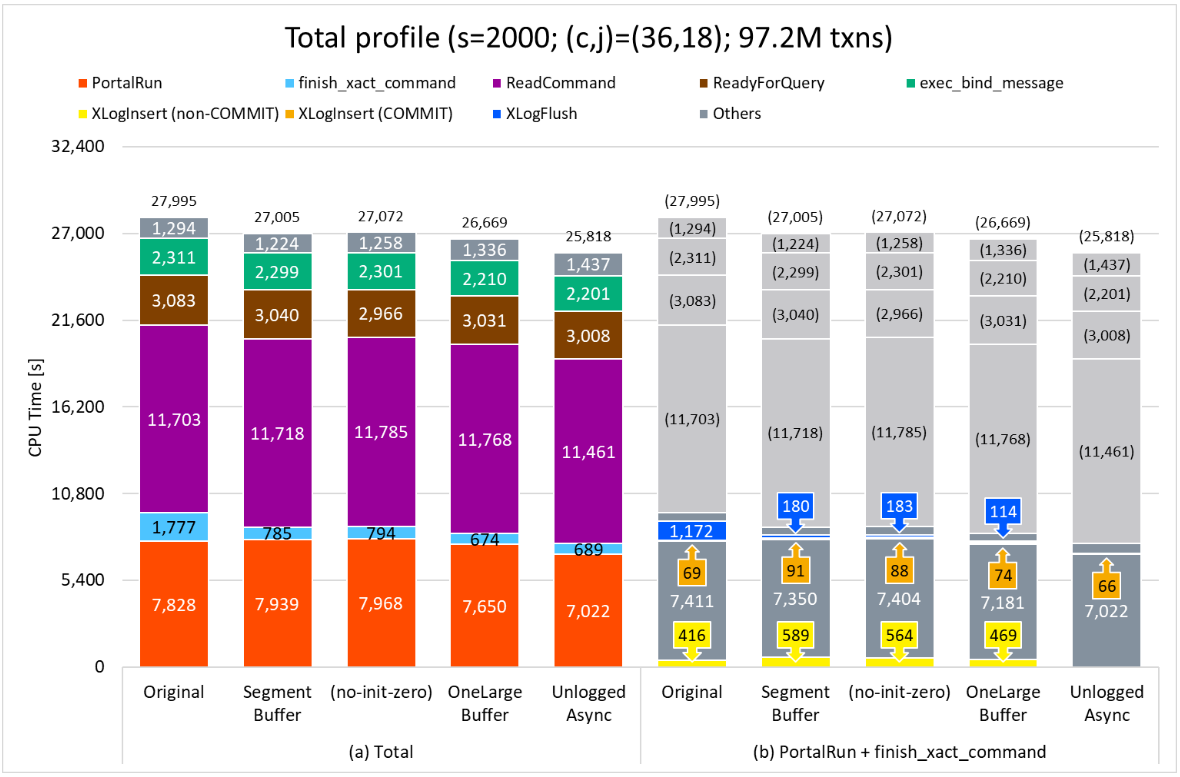


Figure 3.4-1 Total profile (s = 2000) (lower is better)

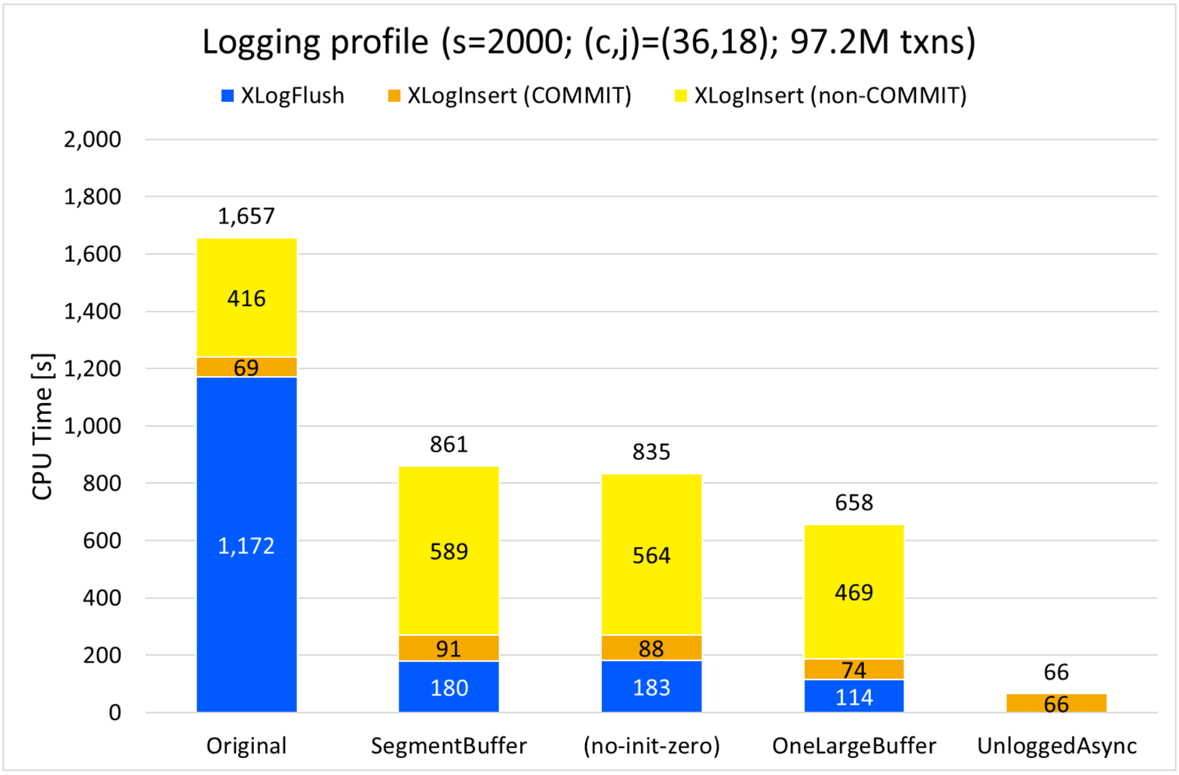


Figure 3.4-2 Logging profile (s = 2000) (lower is better)

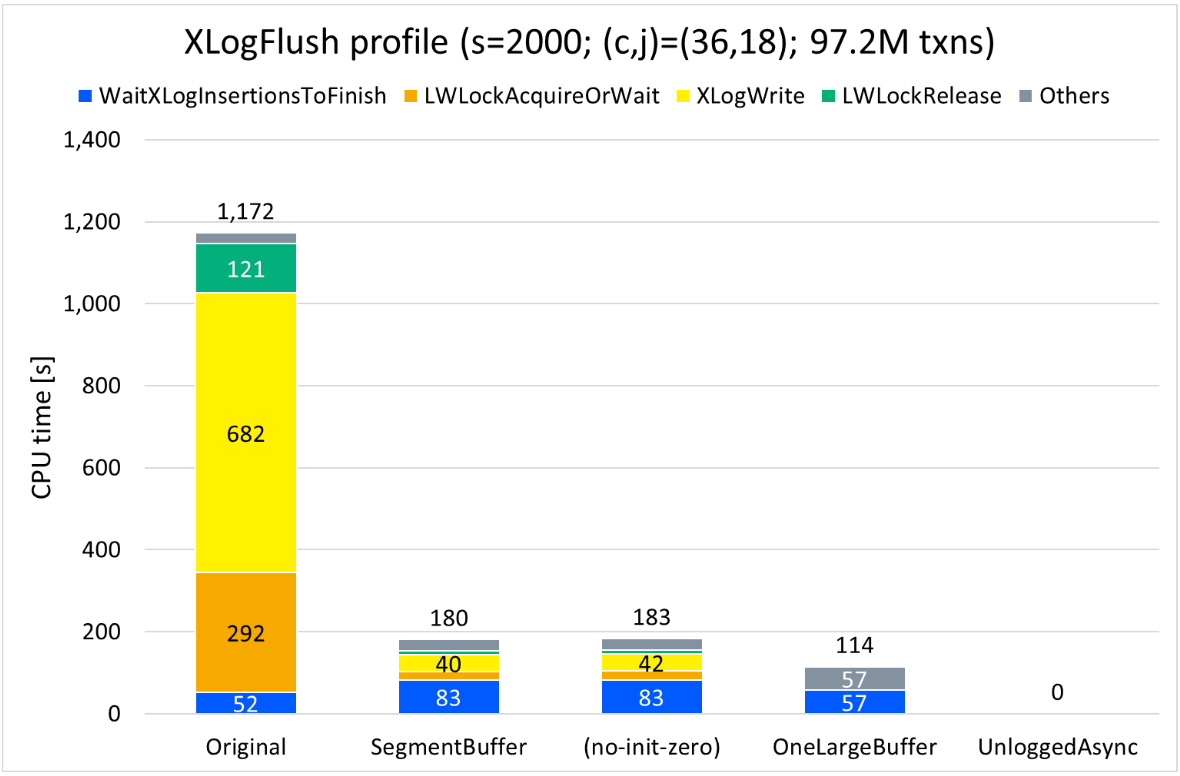


Figure 3.4-3 XLogFlush profile (s = 2000) (lower is better)

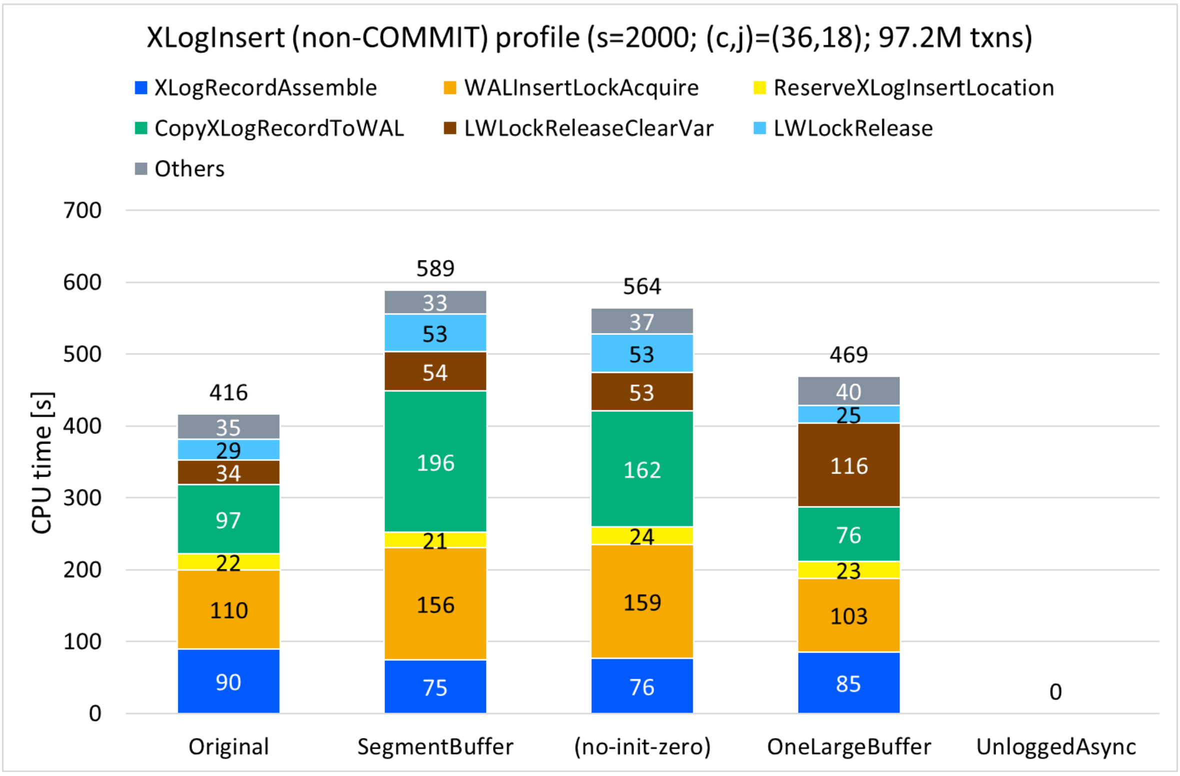


Figure 3.4-4 XLogInsert (non-COMMIT) profile (s = 2000) (lower is better)

4. Discussion

4.1. Four variants of "SegmentBuffer"

As shown in Figure 3.1-1 and Figure 3.2-1, "SegmentBuffer (no-init-zero)" got the best throughput and average latency of the four variants. So each of `wal_init_zero=false` and `wal_recycle=true` looks helpful for performance. Of course, it can also break reliability of WAL not to zero-initialize the segment files if copy-on-write (CoW) filesystems are not used, and the ext4 used this time is not CoW one. So we should be careful to use `wal_init_zero=true`.

4.2. "SegmentBuffer" versus "OneLargeBuffer"

Also as shown in Figure 3.1-1 and Figure 3.2-1, "OneLargeBuffer" got better throughput and average latency than any four variants of "SegmentBuffer," especially in the case of s = 2000. So the patchset of "SegmentBuffer" variants needs to be improved more.

By the way, "SimplePmdk" got only as much performance as "Original." So the patchset of "SimplePmdk" is little effective on the configurations at this time.

4.3. Falls of throughput and recycle of WAL segment files

Figure 3.1-2 and Figure 3.2-2 tell that throughput fell down at some time points during benchmark, and the degree of the falls in the case of s = 2000 were greater than that of s = 50. Server logs tell that checkpoints started at those time points. So the falls look due to full-page write to WAL. Note that the time lag of the falls is because of the different reasons of checkpoint, that is, "wal" or "time." See also "IMPORTANT NOTES" in Section 1.4.

As shown in Figure 3.2-2, there were two or three throughput falls in the entire period of 30-minute benchmark, and the degree of the fall of the first one between 540-720 second was greater than that of the second one between 1080-1440 second, except on "SegmentBuffer (no-recycle)" and "SegmentBuffer (no-both)" conditions. From this, recycling WAL segment files looks helpful for throughput. Moreover, if WAL segment files could be recyclable in advance, that is, if an adequate amount (probably `wal_min_size`) of WAL segment files could be pre-allocated during startup, the first few falls of throughput could be improved, at the price of longer startup time.

4.4. CPU time of XLogFlush

As shown in Figure 3.3-1, Figure 3.3-2, Figure 3.4-1, and Figure 3.4-2, CPU time of XLogFlush on each condition of "SegmentBuffer," "SegmentBuffer (no-init-zero)," or "OneLargeBuffer" got smaller than that of "Original," while XLogInsert time became a bit larger. To sum up them, total CPU time decreased. This looks consistent with performance improvement.

In regard to XLogFlush, Figure 3.3-3 and Figure 3.4-3 tell that CPU time of XLogWrite dropped significantly or even completely. This is a positive effect of persistent WAL buffers on PMEM. On "Original" condition, inserted (that is, memory-copied) WAL records need to be written out of volatile WAL buffers into segment files to be durable. In contrast, on "SegmentBuffer" variants or "OneLargeBuffer," inserted records already on PMEM so they only need to be flushed out of CPU cache into PMEM. The latter is simpler than the former so it leads to improvement of CPU time.

In addition, each CPU time of LWLockAcquireOrWait or LWLockRelease is also reduced. This looks to come with the improvement of XLogWrite. Note that difference between “SegmentBuffer” variants and “OneLargeBuffer” is in which function cache-flush is done: XLogWrite on “SegmentBuffer” variants and XLogFlush on “OneLargeBuffer.” The patchset of “SegmentBuffer” variants for now cache-flushes WAL records conservatively in XLogWrite which is called from inside of a critical section, LWLockAcquireOrWait or LWLockRelease still appear in the analysis results. This may be removed in the future because cache-flushing WAL records can be safely done in parallel without locks, as it is done so in the patchset of “OneLargeBuffer.”

4.5. CPU time of XLogInsert

In regards to XLogInsert, Figure 3.3-4 and Figure 3.4-4 show that CPU time of CopyXLogRecordToWAL on “SegmentBuffer” variants got larger than that of “Original.” This is a negative effect of WAL buffers on slow memory. Because Optane PMem is slower than DRAM, it takes more time to memory-copy WAL records into the buffers on Optane PMem than those on DRAM. This also looks to cause WALInsertLockAcquire, LWLockReleaseClearVar, and LWLockRelease in XLogInsert, and WaitXLogInsertionsToFinish in XLogFlush to take more time.

As difference of CPU time of CopyXLogRecordToWAL between “SegmentBuffer” and “SegmentBuffer (no-init-zero),” whether initializing WAL segment files affects that CPU time. It may be reduced by pre-allocating WAL segment files.

CopyXLogRecordToWAL need to be investigated more deeply, but haven’t been done so yet. It’s a future work.