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	eb43bdb (master @ Tue May 25 19:44:55 2021 -0400)

1.3. PostgreSQL installation

```
$ ./configure --enable-debug --prefix=$HOME/postgres/[snip] --with-extra-version=-[snip] [..]
$ make world
$ 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 `world` and `install-world` targets are 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 two variants for "SegmentBuffer" to see how and how much performance varies with preallocation of WAL segment files. Also note that the variants are displayed as in their short forms in the figures in Section 3.

Name	Patchset and/or customized configuration
Original	No patchset or customized configuration
SegmentBuffer	• **"Map WAL segment files on PMEM as WAL buffers" v2, but *excluding* the last one patch "Preallocate and initialize more WAL if wal_pmem_map=true"** • Add `--with-libpmem` option to `./configure` • Amend `postgresql.conf` as follows: ➢ `wal_pmem_map=true`
`-- (prealloc)	**Ditto, but *including* the last one patch "Preallocate and initialize more WAL if wal_pmem_map=true"**
OneLargeBuffer	• **"Non-volatile WAL buffer" 20210525** • Add `--with-libpmem` option to `./configure` • Amend `postgresql.conf` as follows: ➢ `nvwal_path='/mnt/pmem0/pg_wal/nvwal` ➢ `nvwal_size=120GB`
UnloggedAsync	• No patchset • Amend `postgresql.conf` as follows: ➢ synchronous_commit=false • Add `--unlogged-tables` option to `pgbench -i`

1.5. Common postgresql.conf for all conditions

```
max_connections = 300
shared_buffers = 32GB
dynamic_shared_memory_type = posix
max_wal_size = 120GB
min_wal_size = 120GB
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), (54, 18), or (72, 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 122880`)
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. Some 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 functions (blue-italicized), picking up the functions shown in the call graph that took much CPU time. Note, on "OneLargeBuffer," that WalSndWakeup appeared as RecordTransactionCommit's child (red-italicized) although it should be called by XLogFlush actually. In any case, however, I include WalSndWakeup into XLogFlush's chart.

```
Total
`-- (snip)
    `-- PostgresMain (appears once)
        |-- ReadyForQuery
        |-- ReadCommand
        |-- exec_bind_message
        `-- exec_execute_message
            |-- PortalRun
            |   `-- (snip)
            |       `-- XLogInsert
            |           (non-COMMIT; appears multiple times)
            `-- finish_xact_command
                `-- (snip)
                    `-- RecordTransactionCommit (appears once)
                        |-- XactLogCommitRecord
                        |   `-- XLogInsert (COMMIT)
                        |-- XLogFlush
                        |   |-- WaitXLogInsertionsToFinish
                        |   |-- LWLockAcquireOrWait
                        |   |-- XLogWrite
                        |   |   `-- pmem_flush
                        |   |       ("SegmentBuffer" variants)
                        |   |-- LWLockRelease
                        |   |-- pmem_flush ("OneLargeBuffer")
                        |   `-- WalSndWakeup
                        `-- WalSndWakeup
```

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


CopyXLogRecordToWAL
|-- GetXLogBuffer
|   |-- AdvanceXLInsertBuffer
|   |   |-- LWLockAcquire
|   |   `-- PmemXLogCreate
|   |       ("SegmentBuffer" variants)
|   `-- LWLockRelease
`-- __memmove_avx_unaligned_erms
```

3. Results

3.1. Performance test (s = 50)

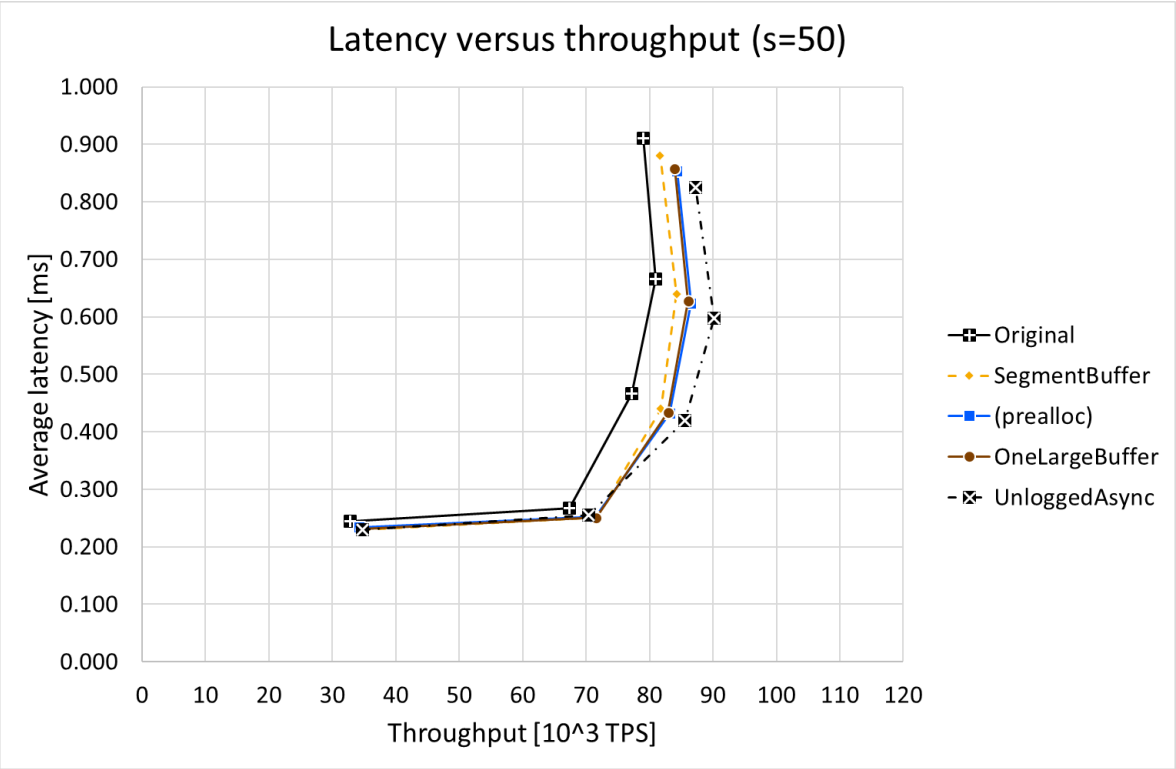

Figure 3.1-1 Latency versus throughput (s = 50) (lower-right is better)

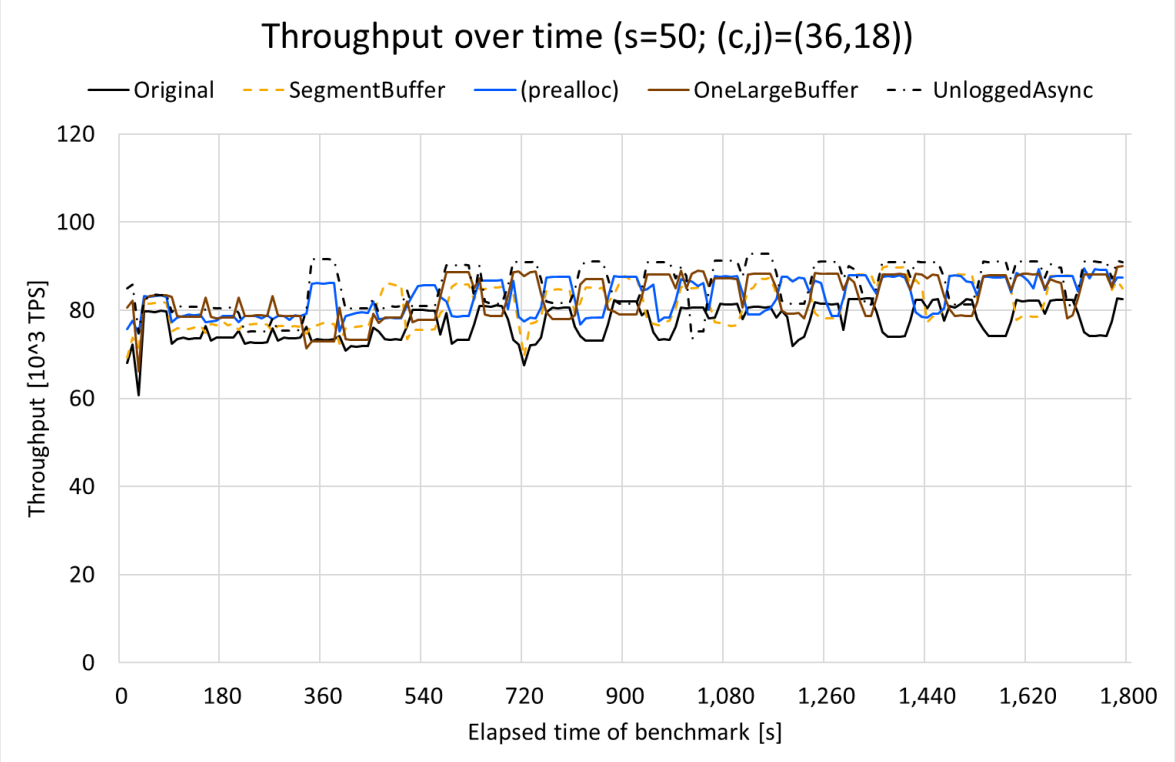

Figure 3.1-2 Throughput over time (s = 50) (higher is better)

3.2. Performance test (s = 2000)

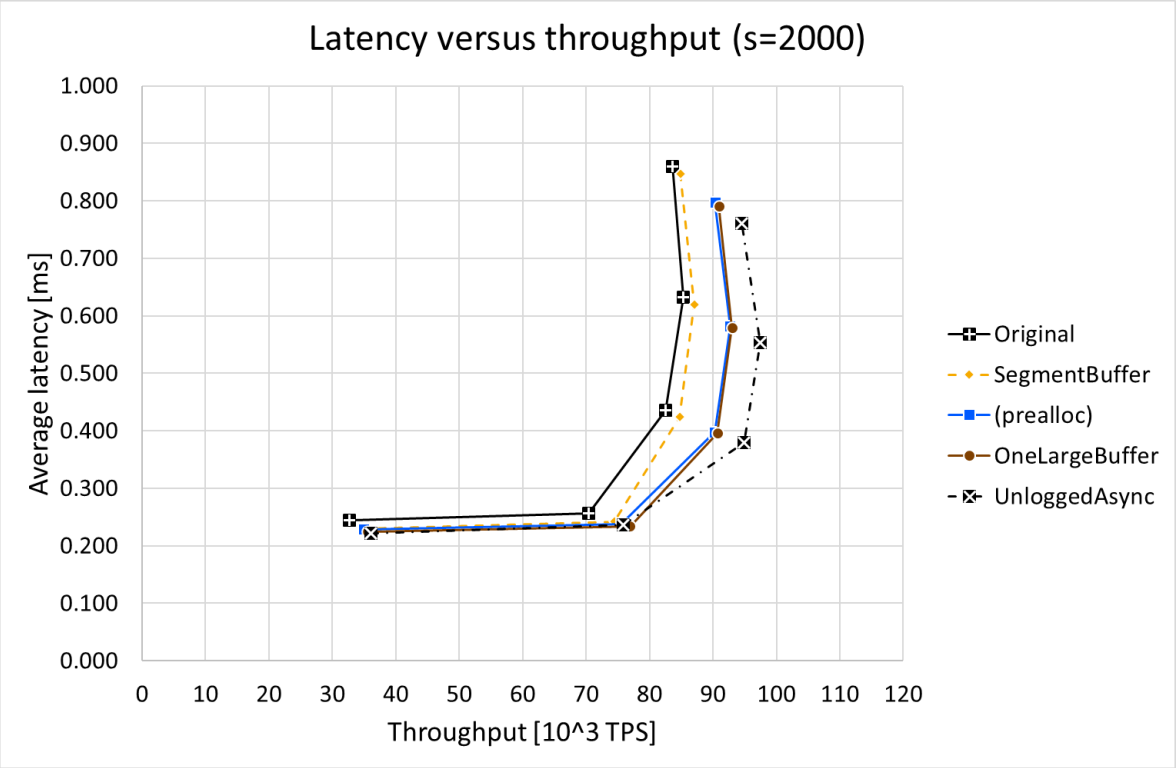

Figure 3.2-1 Latency versus throughput (s = 2000) (lower-right is better)

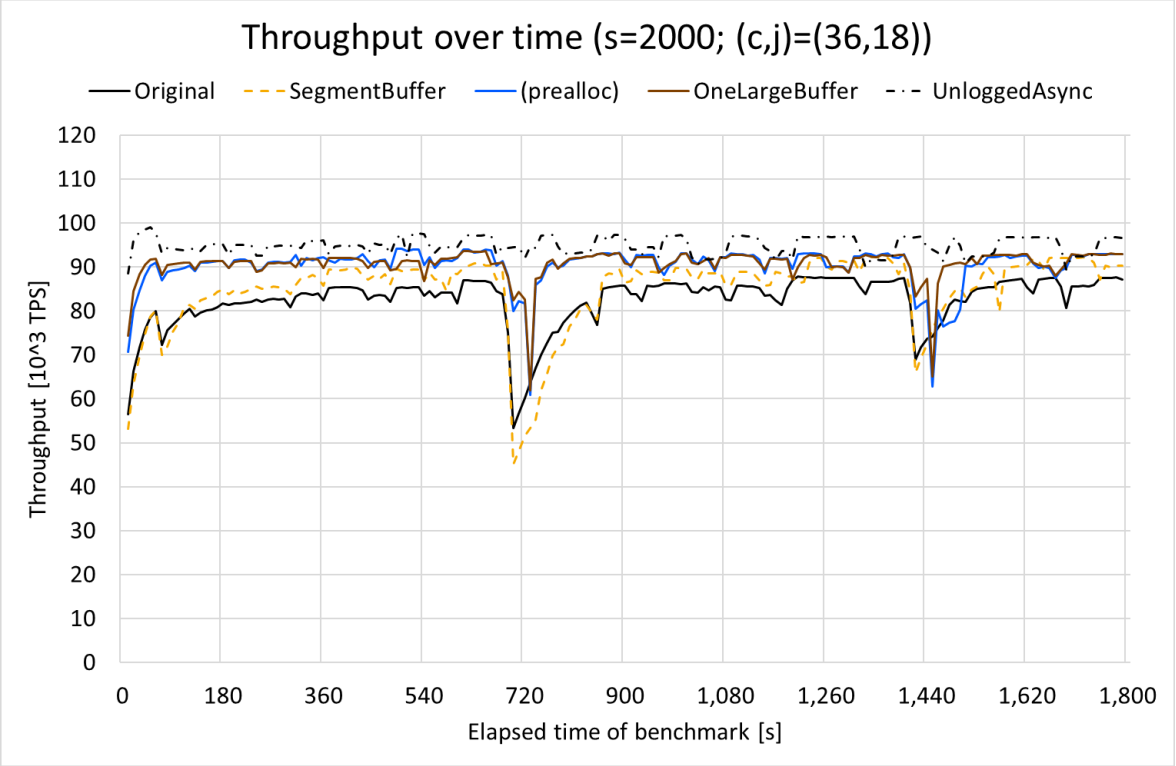

Figure 3.2-2 Throughput over time (s = 2000) (higher is better)

3.3. Performance analysis (s = 50)

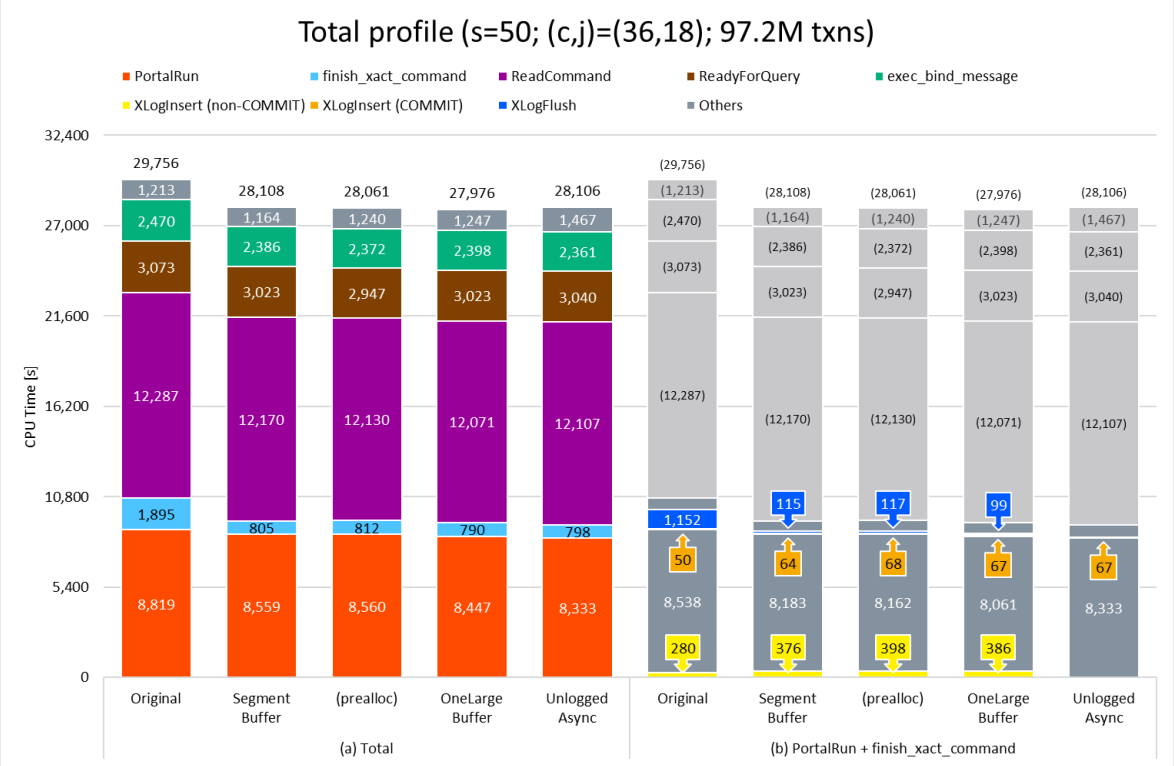

Figure 3.3-1 Total profile (s = 50) (lower is better)

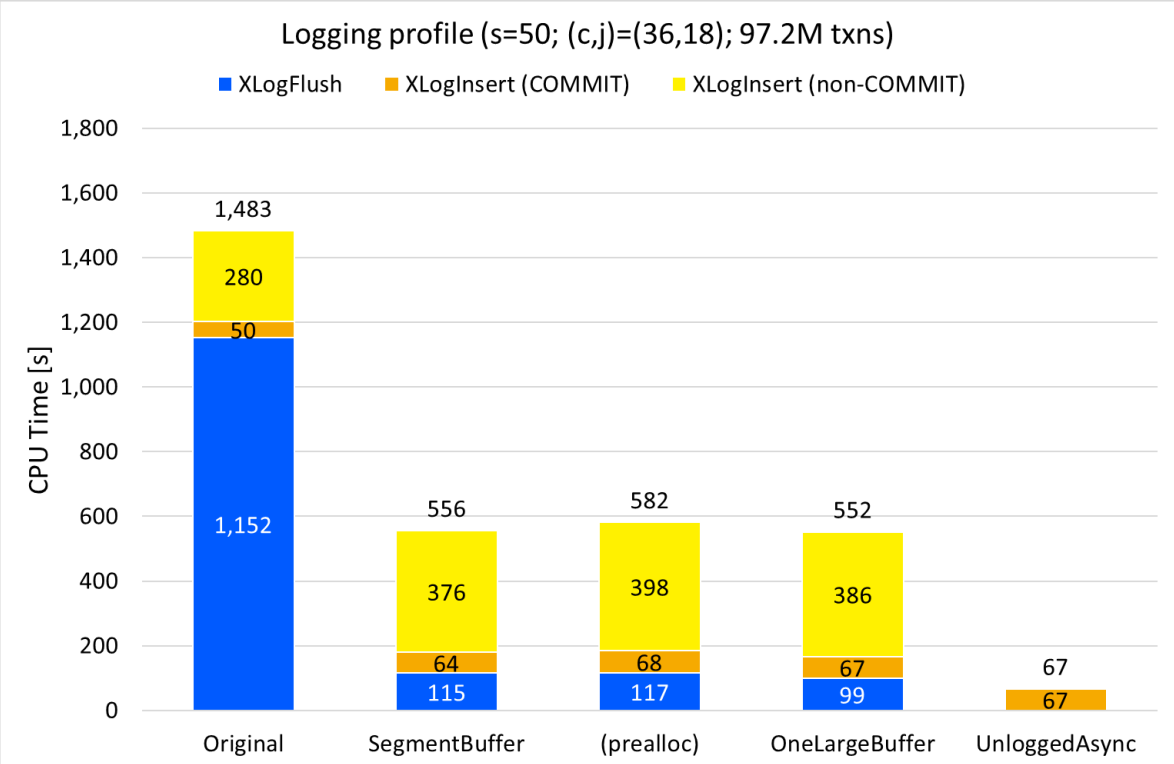

Figure 3.3-2 Logging profile (s = 50) (lower is better)

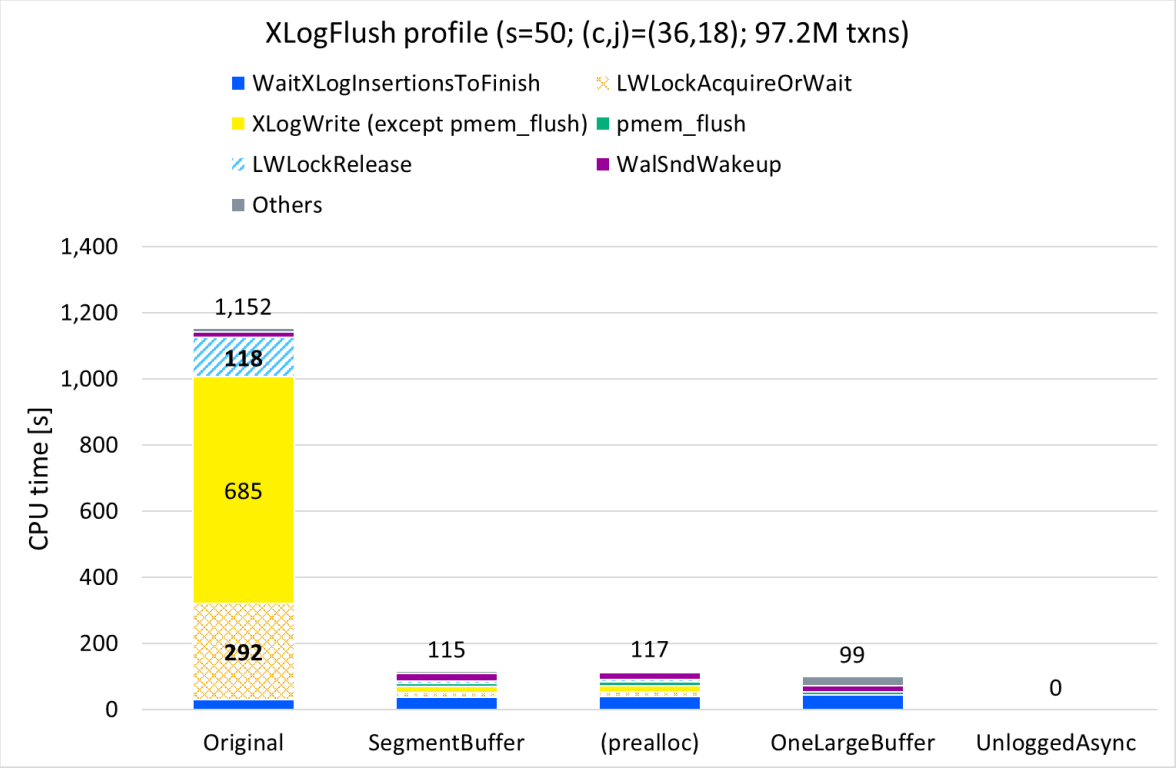

Figure 3.3-3 XLogFlush profile (s = 50) (lower is better)

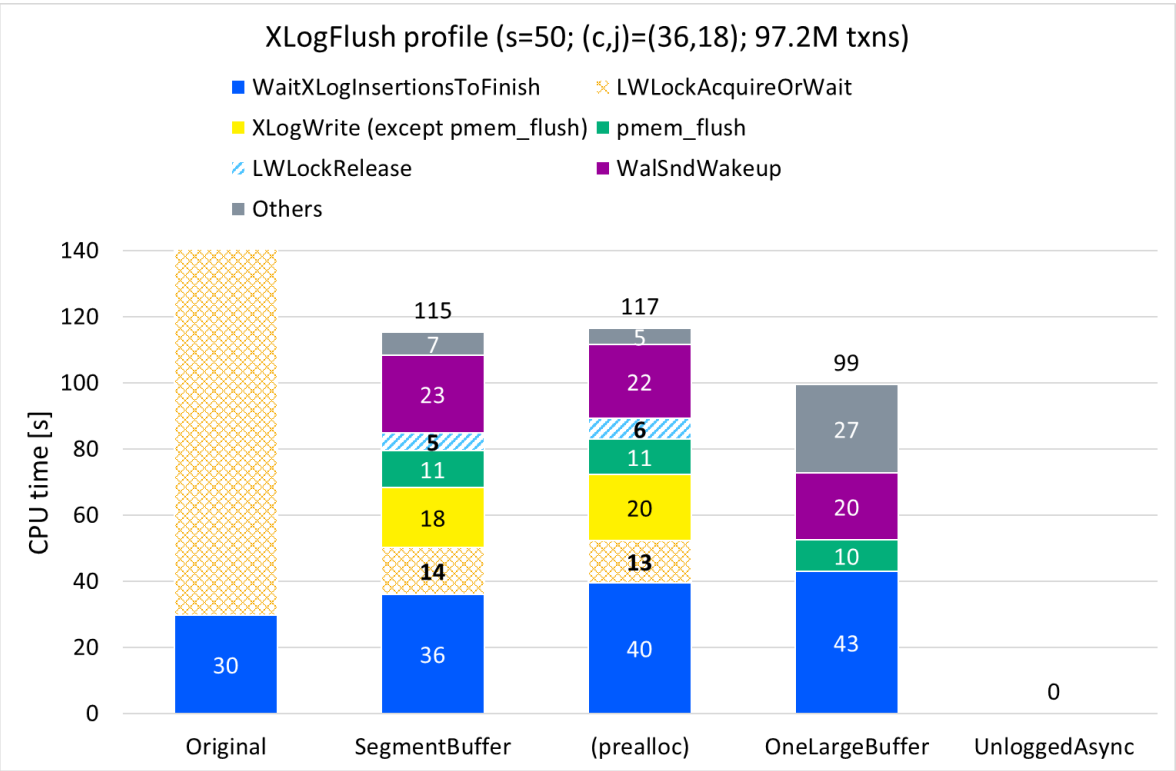

Figure 3.3-4 XLogFlush profile (s = 50) (zoom-in) (lower is better)

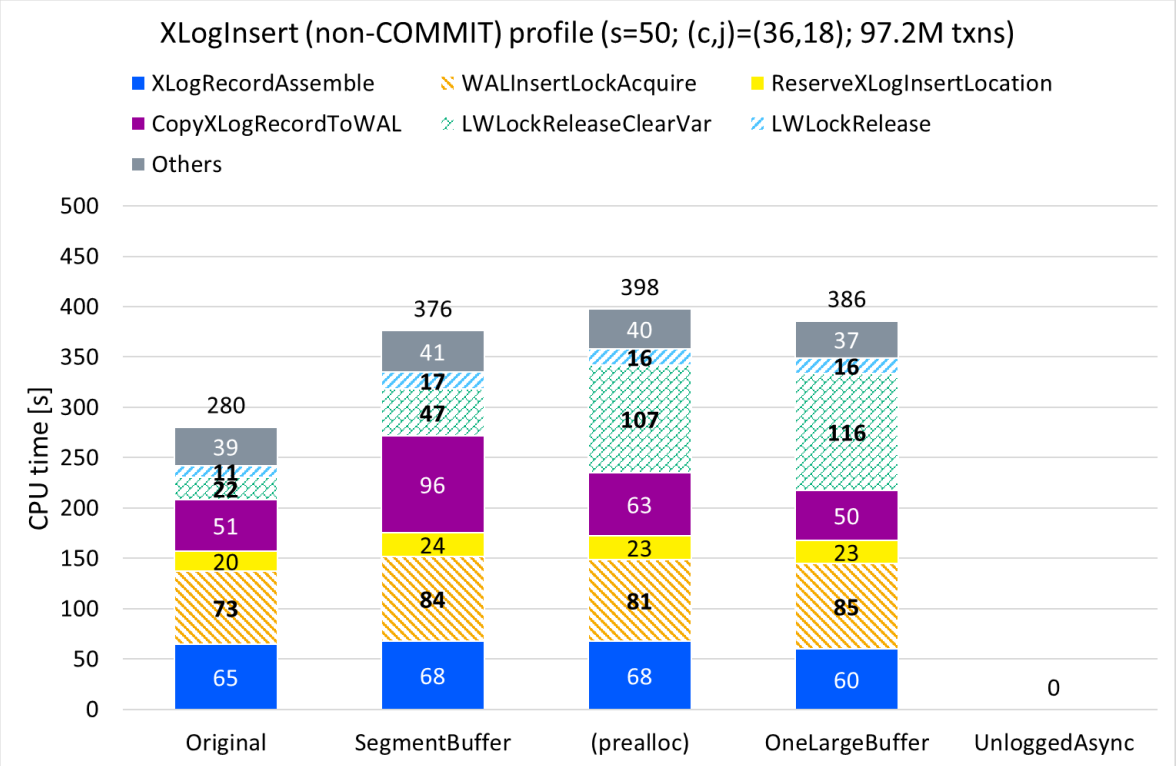

Figure 3.3-5 XLogInsert (non-COMMIT) profile (s = 50) (lower is better)

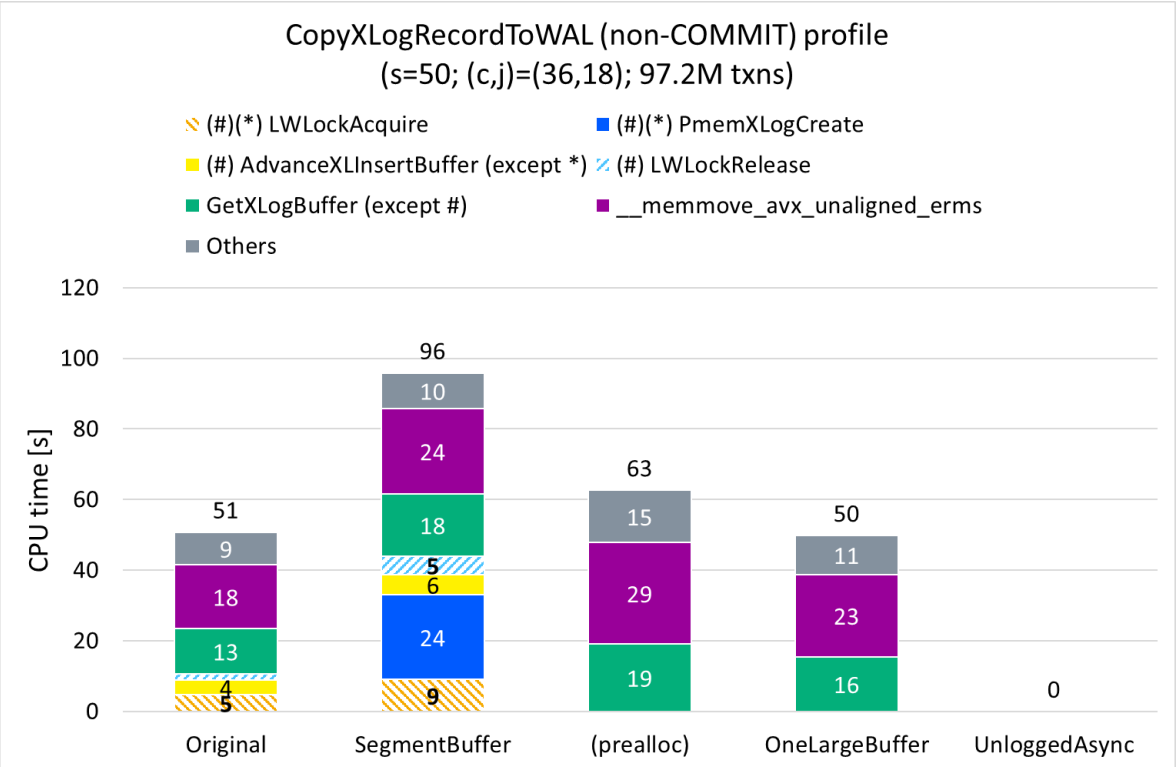

Figure 3.3-6 CopyXLogRecordToWAL (non-COMMIT) profile (s = 50) (lower is better)

3.4. Performance analysis (s = 2000)

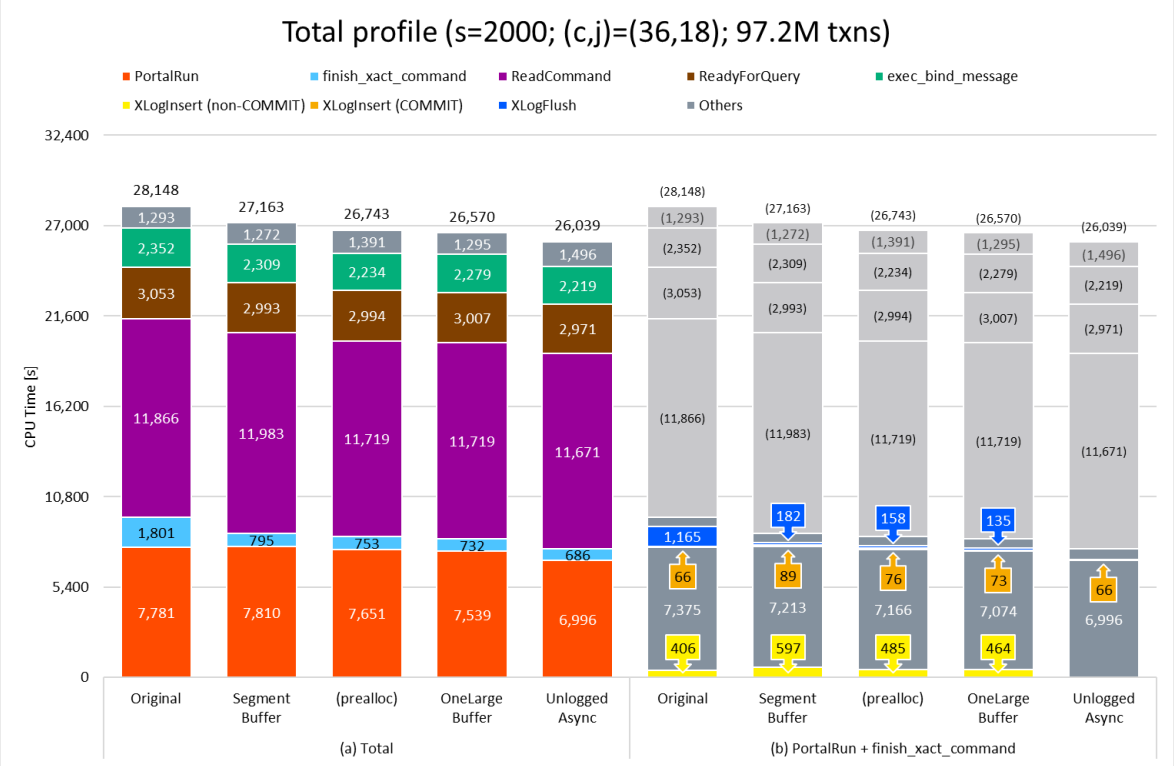

Figure 3.4-1 Total profile (s = 2000) (lower is better)

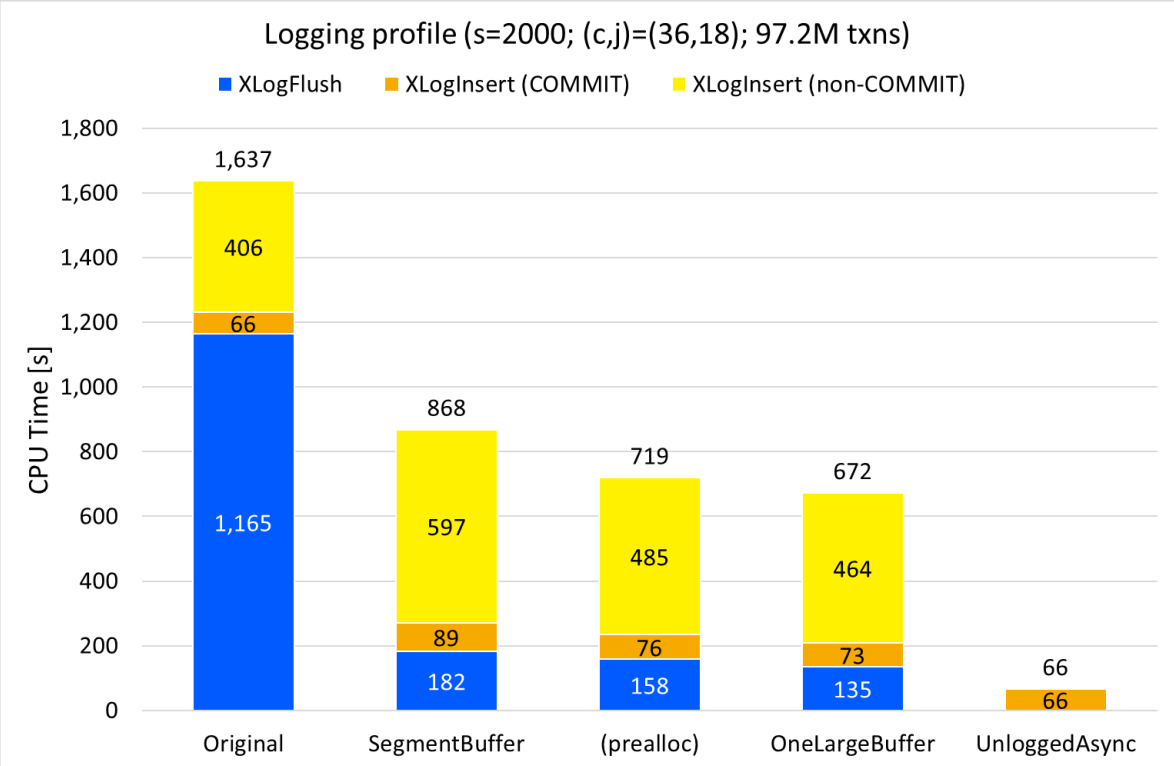

Figure 3.4-2 Logging profile (s = 2000) (lower is better)

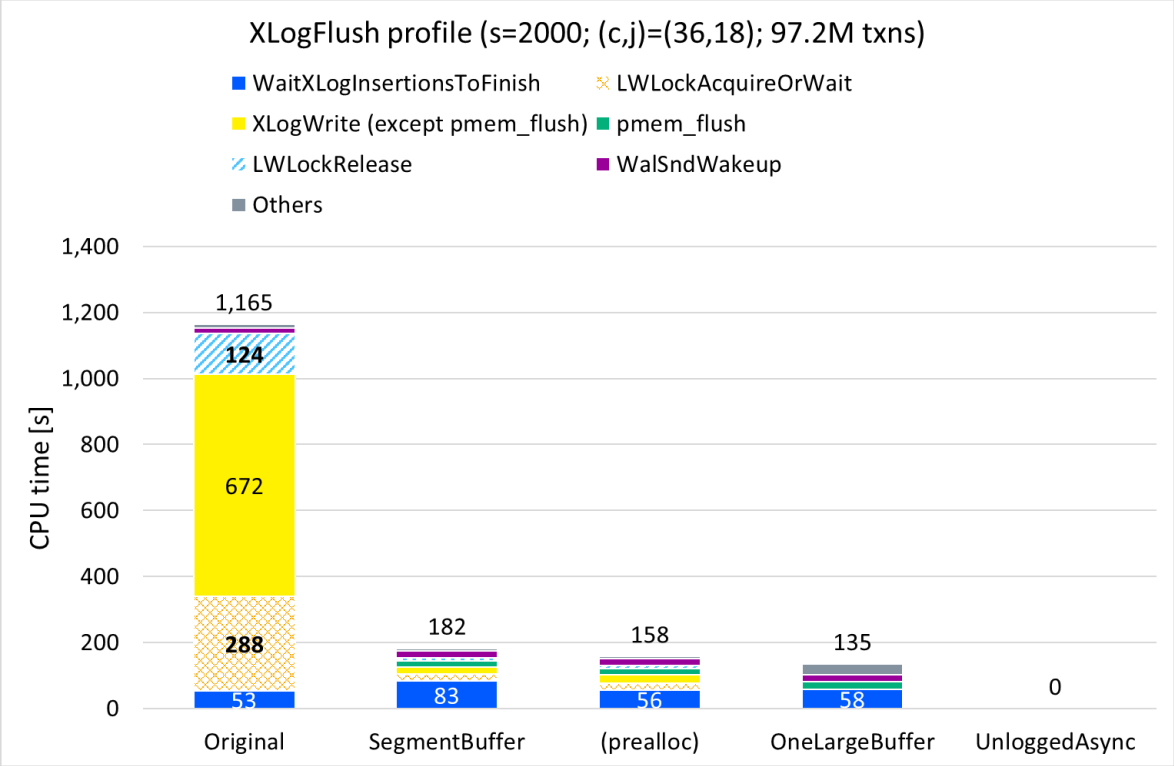

Figure 3.4-3 XLogFlush profile (s = 2000) (lower is better)

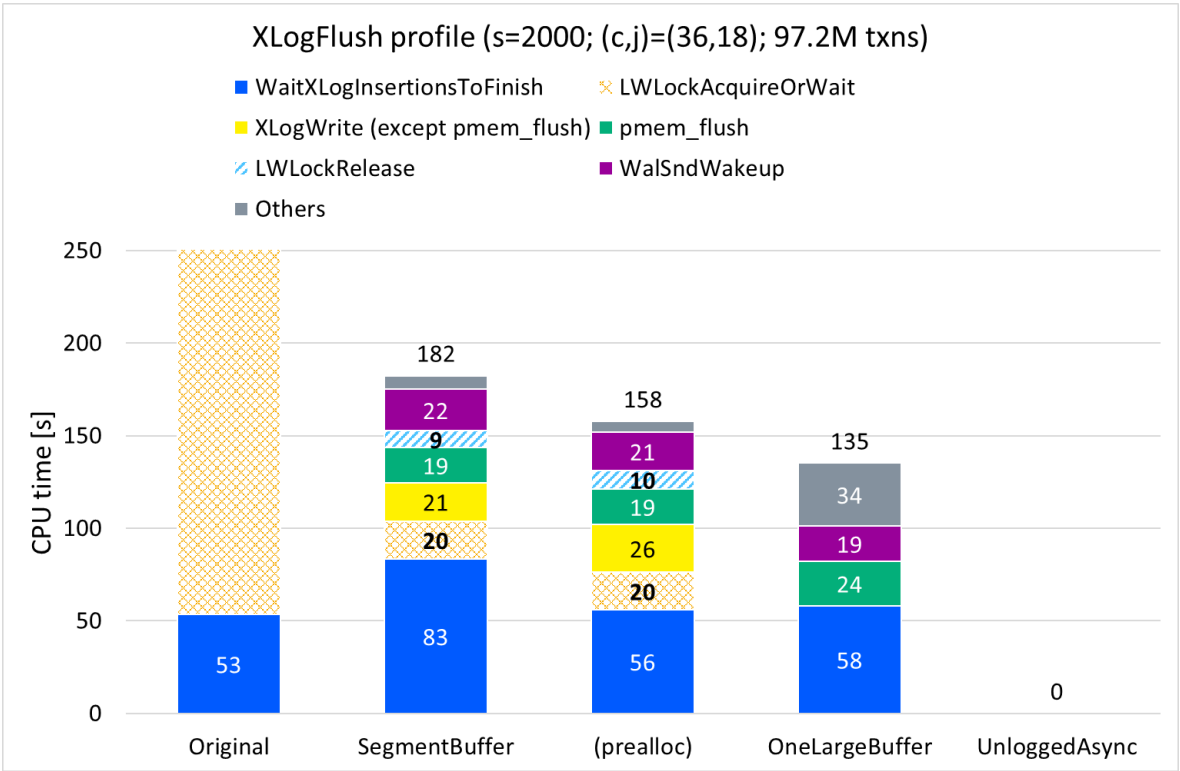

Figure 3.4-4 XLogFlush profile (s=2000) (zoom-in) (lower is better)

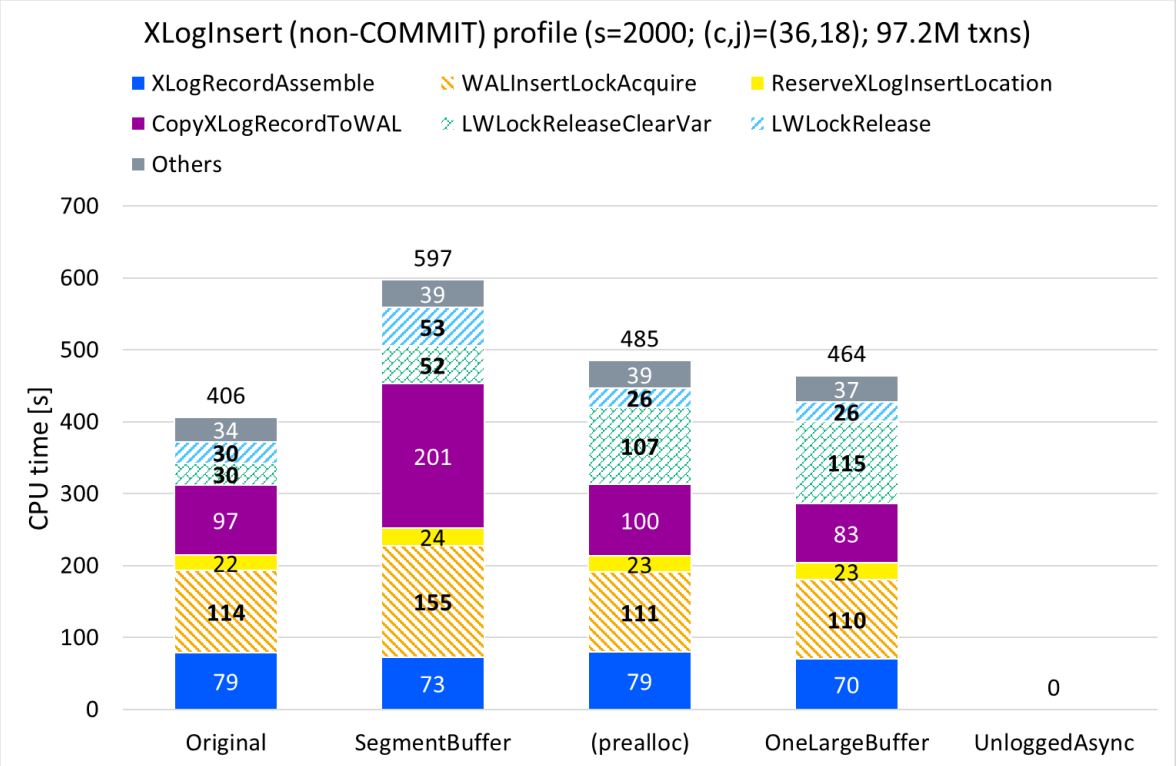

Figure 3.4-5 XLogInsert (non-COMMIT) profile (s = 2000) (lower is better)

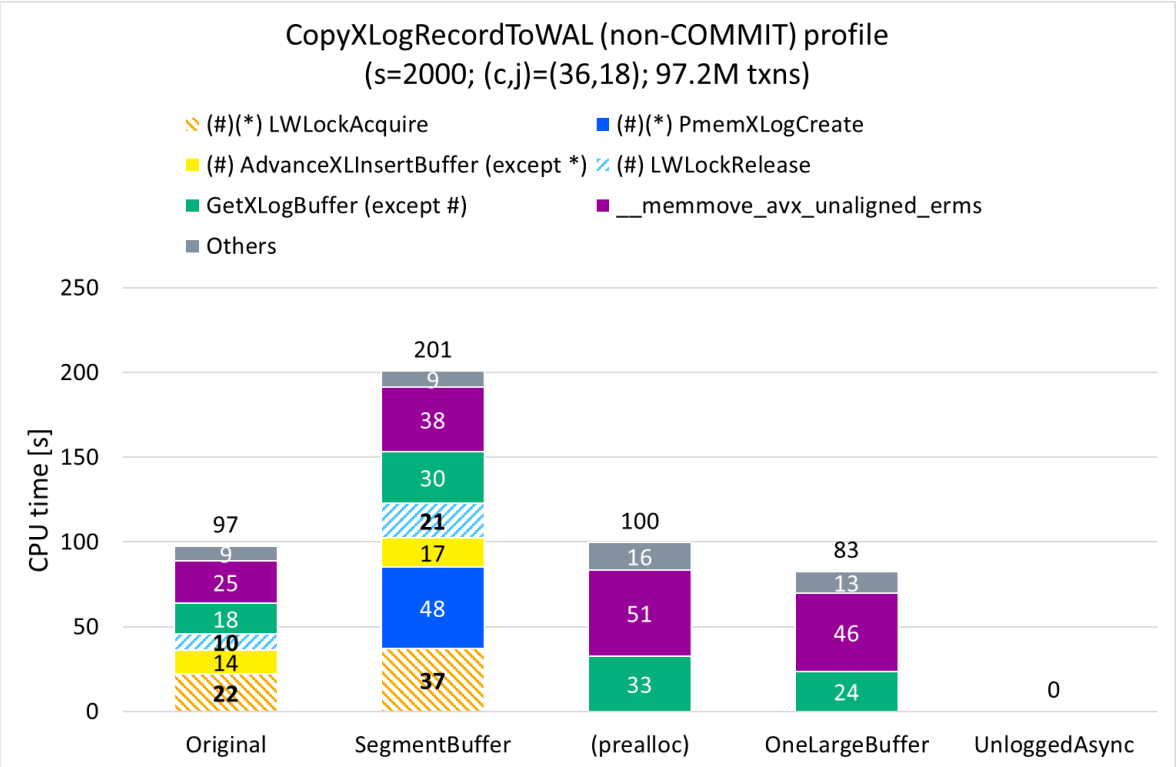

Figure 3.4-6 CopyXLogRecordToWAL (non-COMMIT) profile (s = 2000) (lower is better)

4. Discussions

4.1. Performance results with or without preallocation of WAL

As shown in Figure 3.1-1 and Figure 3.2-1, “SegmentBuffer (prealloc)” got better throughput and average latency than “Original” and “SegmentBuffer,” and got as much performance as “OneLargeBuffer.” Both of “SegmentBuffer (prealloc)” and “OneLargeBuffer” preallocate WAL during startup, so preallocation looks helpful for high performance.

4.2. Checkpoint with or without preallocation of WAL

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.

As shown in Figure 3.2-2, there were two throughput falls in the entire period of 30-minute benchmark on “Original” and “SegmentBuffer.” The first fall around 720 seconds was larger (deeper and longer) than that of the second one around 1440 seconds. This looks due to WAL recycle, that is, it takes less time to recycle existing WAL segment files during the second falls while it takes more time to prepare new files during the first one.

There were also two falls on “SegmentBuffer (prealloc)” and “OneLargeBuffer,” but those falls are smaller (shallower and/or shorter) than the previous two. This looks to tell that preallocating WAL is helpful for stable performance.

4.3. 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 (prealloc),” 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, Figure 3.3-4, Figure 3.4-3, and Figure 3.4-4 tell that CPU time of XLogWrite dropped significantly or completely. This is a positive effect of persistent WAL buffers on PMEM. On “Original,” 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 are 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 the difference between “SegmentBuffer” variants and “OneLargeBuffer” is in which function cache-flush is done: XLogWrite on “SegmentBuffer” variants and XLogFlush on “OneLargeBuffer.” “SegmentBuffer” variants cache-flush records and update shared variables with WALWriteLock held, so LWLockAcquireOrWait and LWLockRelease still appear in the analysis results. On “OneLargeBuffer,” WALWriteLock was not held during cache flush any more, but a spin-lock was required for updating shared variables. This possibly causes the increase of CPU time indicated by “Others” sub-bar, compared to “SegmentBuffer” variants.

4.4. CPU time of XLogInsert

In regards to XLogInsert, Figure 3.3-5 and Figure 3.4-5 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.

The two figures also show that CPU time of CopyXLogRecordToWAL on "SegmentBuffer (prealloc)" got smaller than that of naïve "SegmentBuffer." See the next section for details.

4.5. CPU time of CopyXLogRecordToWAL

Figure 3.3-6 and Figure 3.4-6 are breakdown of CopyXLogRecordToWAL. LWLockAcquire, PmemXLogCreate, AdvanceXLInsertBuffer, and LWLockRelease completely dropped on "SegmentBuffer (prealloc)." This shows why performance got better: offloads of WAL initialization. Please note that "initialization" here includes not only clearing buffer pages and/or segment files, but also advancing LSNs with the pages and putting headers onto the pages.

On "Original," WAL buffers (`pages` and `xlblocks`) are initialized for new records in two ways. First, initialization just before insertion in cases of buffer full. Second, periodical initialization by walwriter, one of postgres' background processes. The former is on a critical path, and appears in the two figures as AdvanceXLInsertBuffer and the two light-weight lock functions for WALBufMappingLock. In addition, on "SegmentBuffer," WAL segment files for new records should exist at insertion time, and they will be created and cleared by PmemXLogCreate if they do not exist at that time yet. On those conditions, the size of the WAL buffers managed by `xlblocks` is not so large: at most one segment (typically 16MiB) on "Original" or exactly one segment on "SegmentBuffer."

On "SegmentBuffer (prealloc)," my patchset introduces two changes. One is that the size of the WAL buffer pages managed by `xlblocks` grows from one segment to `min_wal_size`. The other is that the WAL buffers are initialized also at startup. By those changes, WAL initialization on "SegmentBuffer (prealloc)" can be summarized as follows. At startup, the WAL buffers and the underlying segment files are initialized for the next `min_wal_size`. After startup, the walwriter periodically initializes them, that is, advances `xlblocks`, allocates and memory-maps a new segment file, clears that file, and puts segment and page headers onto that file. There is no turn for the initialization on the critical path, so the four functions does not appear in the two figures.

On "OneLargeBuffer," the four functions does not appear in the figures due to offloads of WAL initialization, too. However, how "OneLargeBuffer" allocates and initializes WAL is different from how "SegmentBuffer (prealloc)" does. On "OneLargeBuffer," WAL buffers are on single large file on PMEM. The file is located at `nvwal_path` and its size is `nvwal_size` which can be dozens of GiB. The file is allocated at `initdb` and the buffers on it are initialized at startup time. After startup, the buffers are initialized for new records in bulk at end of checkpoints, not periodically by walwriter. This works well and is more time-efficient than "SegmentBuffer (prealloc)" if bulk initialization runs enough ahead of record insertions. If not so, however, the initialization will be caught up and may block the insertions for a longer time than segment-by-segment initialization on "SegmentBuffer (prealloc)." This may lead to a temporary and extreme fall of performance.

5. Conclusions and future works

5.1. “SegmentBuffer (prealloc)” versus “OneLargeBuffer”

To summarize, “SegmentBuffer (prealloc)” looks the most reasonable way of all. It’s compatible to the existing WAL due to use of segment files, but achieves as high performance as “OneLargeBuffer.”

5.2. Relation between startup time and WAL initialization

WAL initialization at startup looks to help high and stable performance. However, startup time must get longer than before. It’s a future work to examine how much time initialization at startup takes and how the relation between the time and the size of WAL is.

5.3. Performance change in long term

In this report, benchmark duration for performance test was 30 minutes long. However, longer benchmark should be also required to examine performance more deeply, so it is one of the other future works. As mentioned in Section 4.2, performance looks to change whether there are recyclable WAL segment files or not. Even if only a small amount of WAL is initialized at startup, after enough time has passed, there will be a sufficient amount of recyclable segment files. So an impact of WAL initialization at startup may get small with time.

5.4. Analysis of WAL initialization

On “SegmentBuffer (prealloc),” WAL initialization is offloaded from a critical path to startup and/or walwriter. I analyzed the initialization on the critical path in this report, but did not do the others yet. This is also a future work.