ZIL Performance: How I Doubled Sync Write Speed

Agenda

1. What is the ZIL?

2. How is it used? How does it work?

3. The problem to be fixed; the solution.

4. Details on the changes I made.

5. Performance testing and results.

*Press "p" for notes, and "c" for split view.

1 – What is the ZIL?

What is the ZIL?

- ZIL: Acronym for (Z)FS (I)ntent (L)og

  - Logs synchronous operations to disk, before `spa_sync()`

  - What constitutes a "synchronous operation"?

    - most *modifying* ZPL operations:

      - e.g. `zfs_create`, `zfs_unlink`, `zfs_write` (some), etc.

    - doesn't include non-modifying ZPL operations:

      - e.g. `zfs_read`, `zfs_seek`, etc.

When is the ZIL used?

- Always*

  - ZPL operations (`itx`'s) logged via in-memory lists

  - lists of in-memory `itx`'s written to disk via `zil_commit()`

  - `zil_commit()` called for:

    - *any* sync write

    - other sync operations (e.g. create, unlink), **and** `sync=always`

    - *some* reads (`sync=always` or `FRSYNC` set)

*Except when dataset configured with: `sync=disabled`

What is the SLOG?

- SLOG: Acronym for (S)eperate (LOG) Device

    - An SLOG is not necessary

    - An SLOG can be used to improve latency of ZIL writes

- Conceptually, SLOG is different than the ZIL

- ZIL is used, even if no SLOG attached

Why does the ZIL exist?

- Writes in ZFS are "write-back"

- Without the ZIL, sync operations inherit latency of `spa_sync()`[*]

    - `spa_sync()` can take tens of seconds (or more) to complete

- Further, with the ZIL, write amplification can be mitigated

- ZIL is essentially a performance optimization

[*]All operations inherit this latency, but only sync operations wait for completion

ZIL On-Disk Format

- Each dataset has it's own unique ZIL on-disk

- ZIL stored on-disk as a singly linked list of ZIL blocks (`lwb`'s)
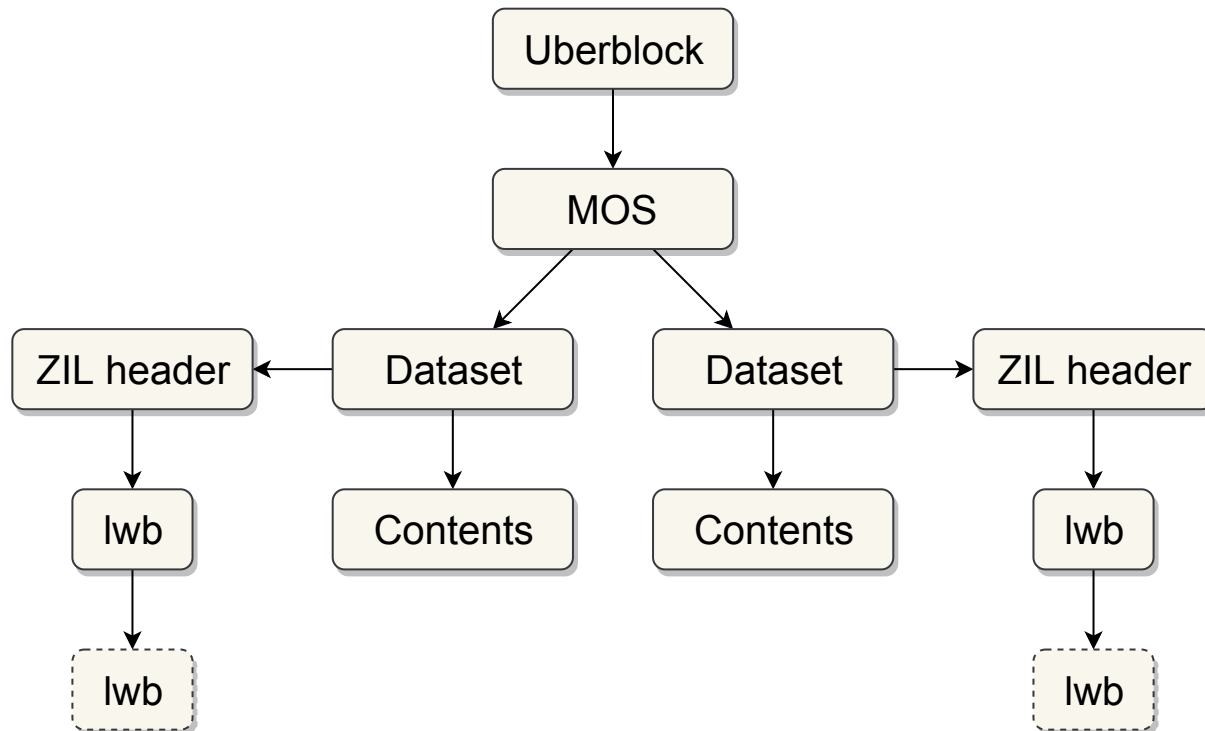
2 – How is the ZIL used?

How is the ZIL used?

- ZPL will generally interact with the ZIL in two phases:

  1. Log the operation(s) — `zil_itx_assign`

  2. Commit the operation(s) — `zil_commit`

Example: `zfs_write`

- `zfs_write → zfs_log_write`

- `zfs_log_write`

  `→ zil_itx_create`

  `→ zil_itx_assign`

- `zfs_write → zil_commit`

- Most ZPL operations have a corresponding `zfs_log_*` function

  - `zfs_log_create`
  - `zfs_log_remove`
  - `zfs_log_link`
  - `zfs_log_symlink`
  - `zfs_log_truncate`
  - `zfs_log_setattr`
  - ...

Example: `zfs_fsync`

- `zfs_fsync → zil_commit`

    - no *new* operations to log... no `zfs_log_fysnc` function

Contract between ZIL and ZPL.

- Parameters to `zil_commit`: ZIL pointer, object number

    - These uniquely identify an object whose data is to be committed

- When `zil_commit` returns:

    - Operations *relevant* to the object specified, will be *persistent* on disk

    - relevant – all operations that would modify that object

    - persistent – Log block(s) written (completed) → disk flushed

- Interface of `zil_commit` doesn't specify *which* operation(s) to commit

2 – How does the ZIL work?

How does the ZIL work?

- In memory ZIL contains per-txg `itxg_t` structures

- Each `itxg_t` contains:

    - A single list of sync operations (for all objects)

    - Object specific lists of async operations

Example: itx lists

sync list → itx S1 → itx S2

object A list → itx A1 → itx A2

object B list → itx B1
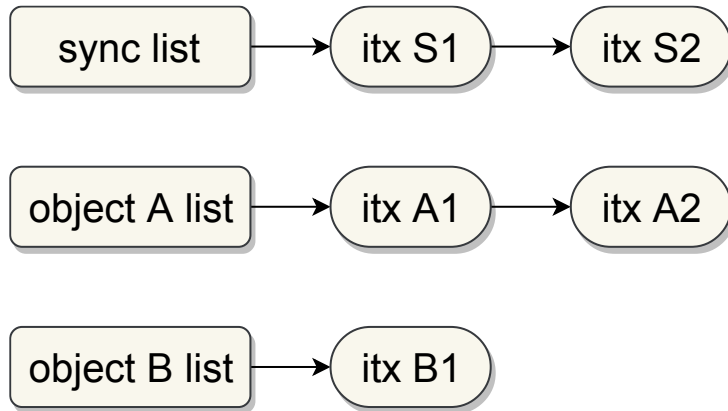
How are itx's written to disk?

- `zil_commit` handles the process of writing `itx_t`'s to disk:

How are itx's written to disk?

- `zil_commit` handles the process of writing `itx_t`'s to disk:

  1. find all relavant `itx`'s, move them to the "commit list"

Example: `zil_commit` Object B
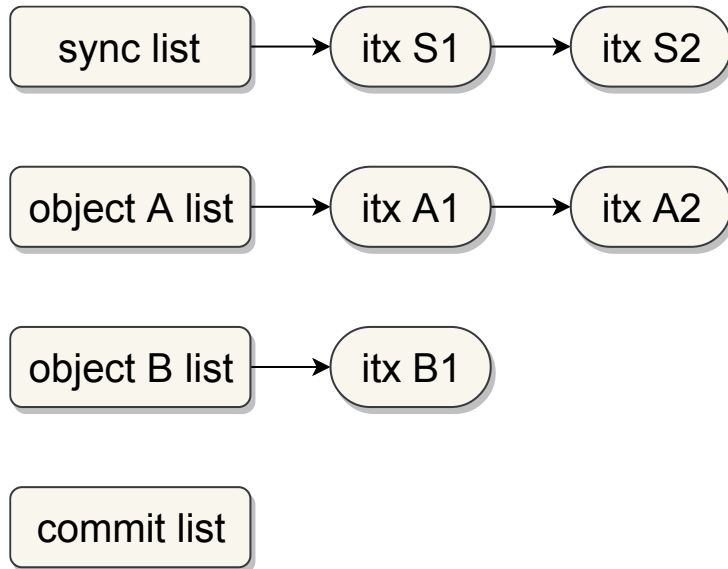
Example: `zil_commit` Object B

```
sync list ────▶ itx S1 ────▶ itx S2

object A list ────▶ itx A1 ────▶ itx A2

object B list ────▶ itx B1

commit list
```
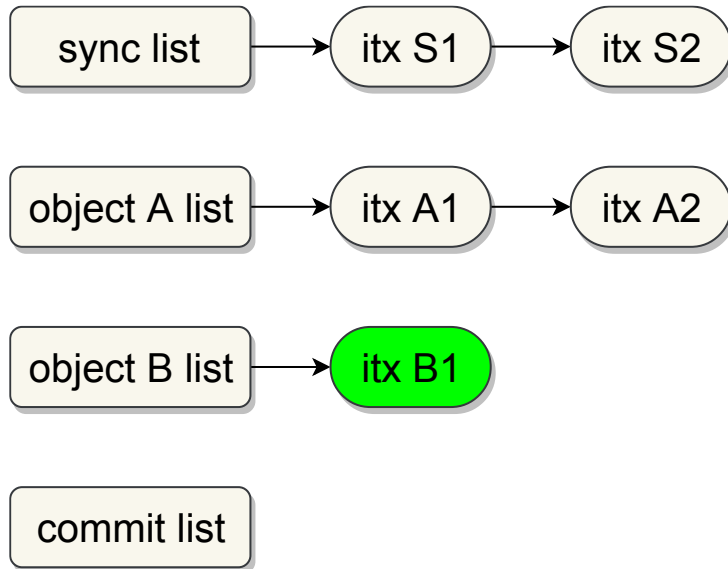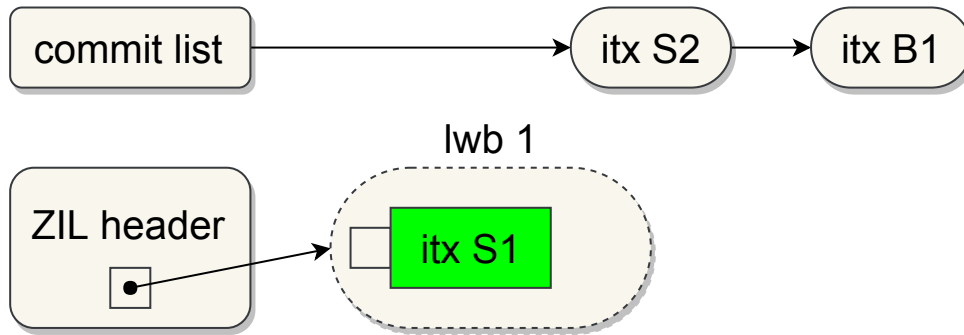
Example: `zil_commit` Object B
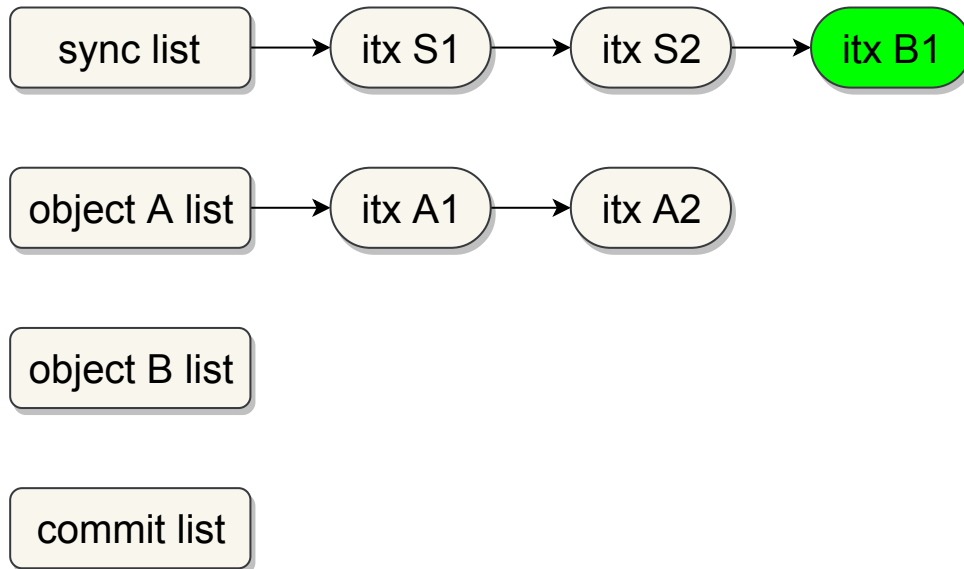
Example: `zil_commit` Object B
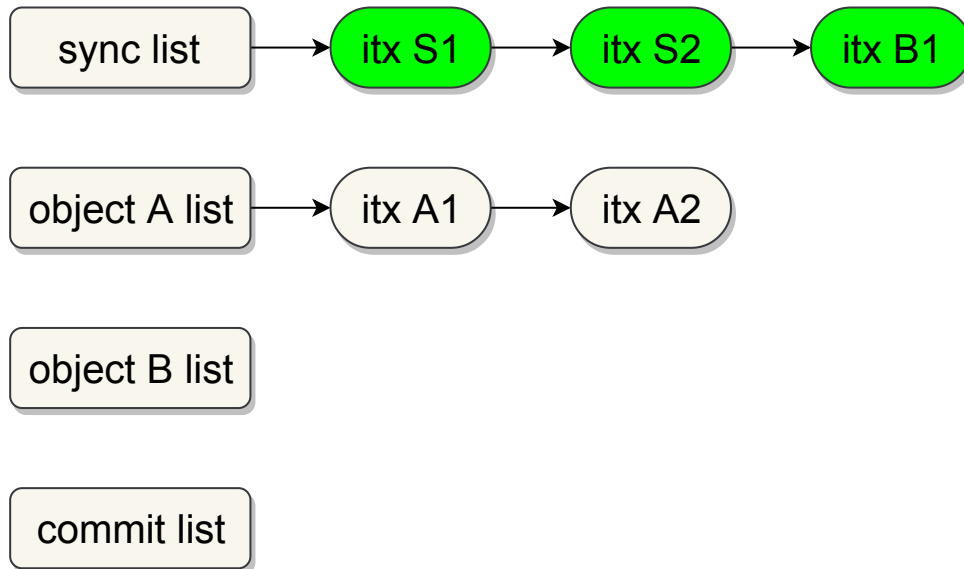
Example: `zil_commit` Object B
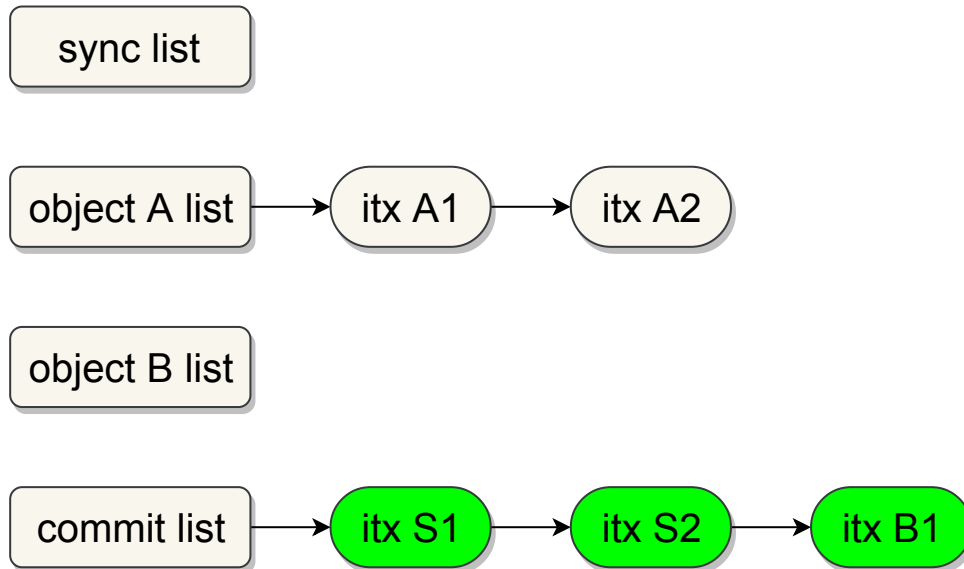
How are itx's written to disk?

- `zil_commit` handles the process of writing `itx_t`'s to disk:

    1. Move async itx's for object being commited, to the sync list

    2. Write all commit list `itx`'s to disk
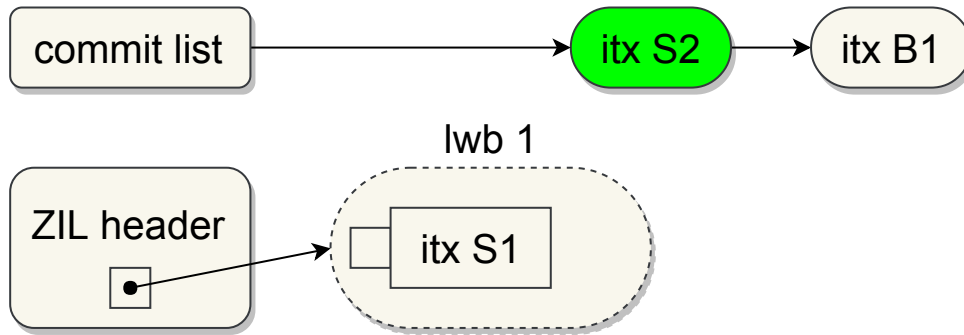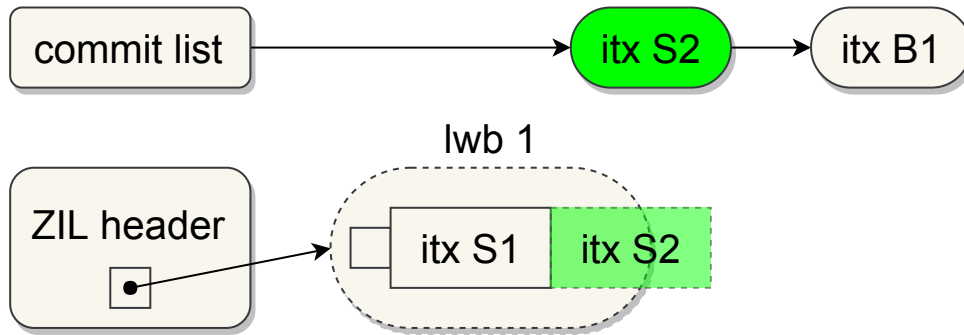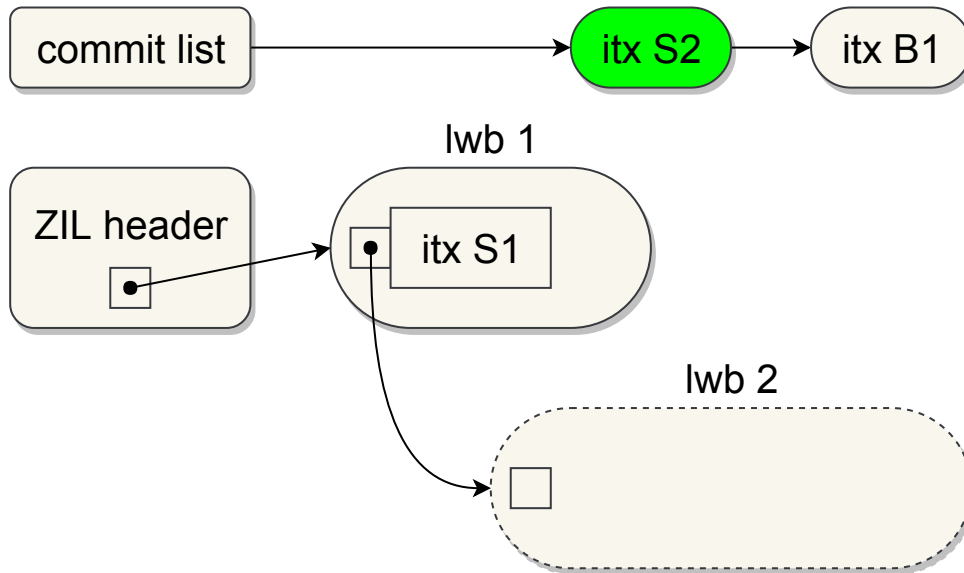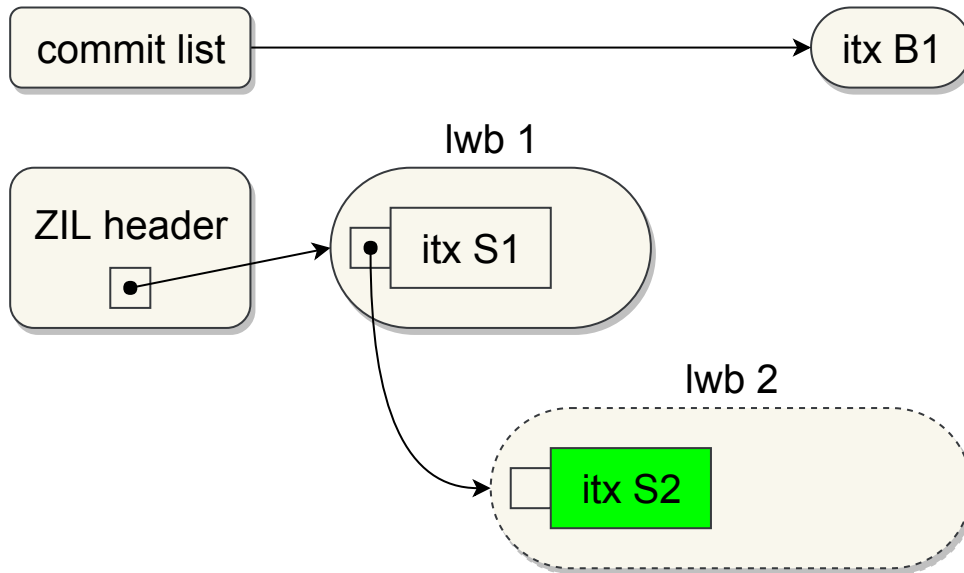
Example: `zil_commit` Object B

```
commit list ──▶ itx S1 ──▶ itx S2 ──▶ itx B1
```

Example: `zil_commit` Object B

commit list → itx S1 → itx S2 → itx B1

Example: `zil_commit` Object B

commit list → itx S1 → itx S2 → itx B1

lwb 1

ZIL header

Example: `zil_commit` Object B

commit list   →   itx S2 → itx B1

lwb 1

ZIL header

itx S1

Example: `zil_commit` Object B

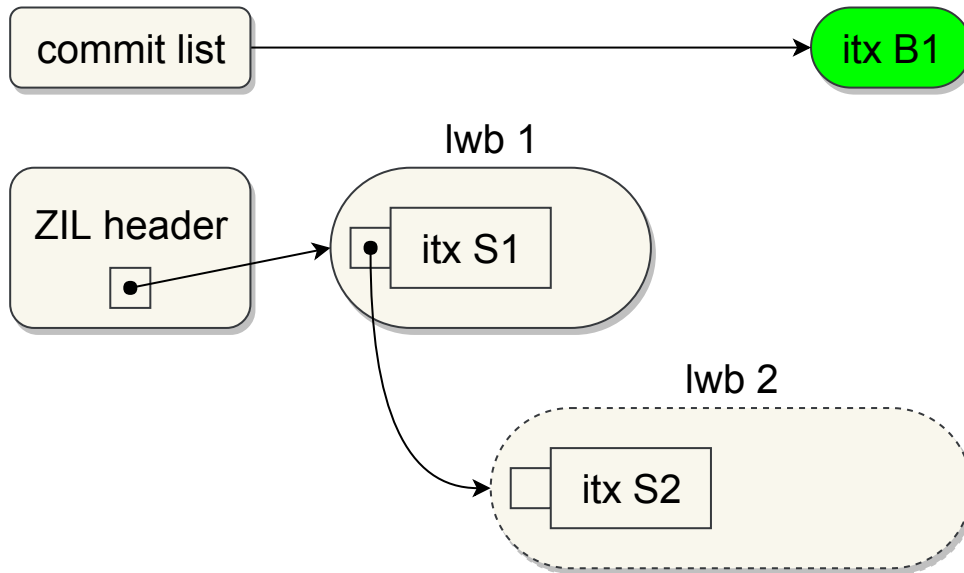# Example: `zil_commit` Object B
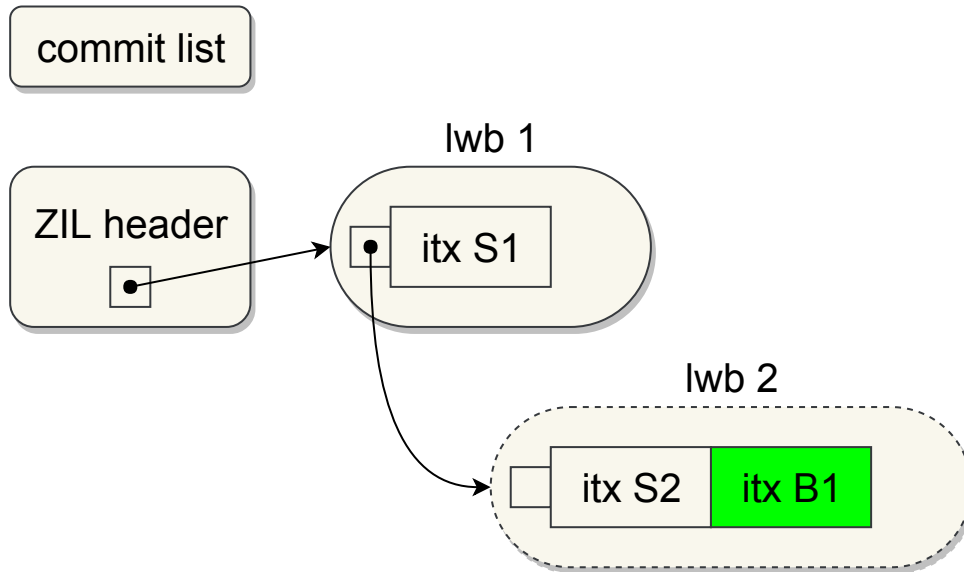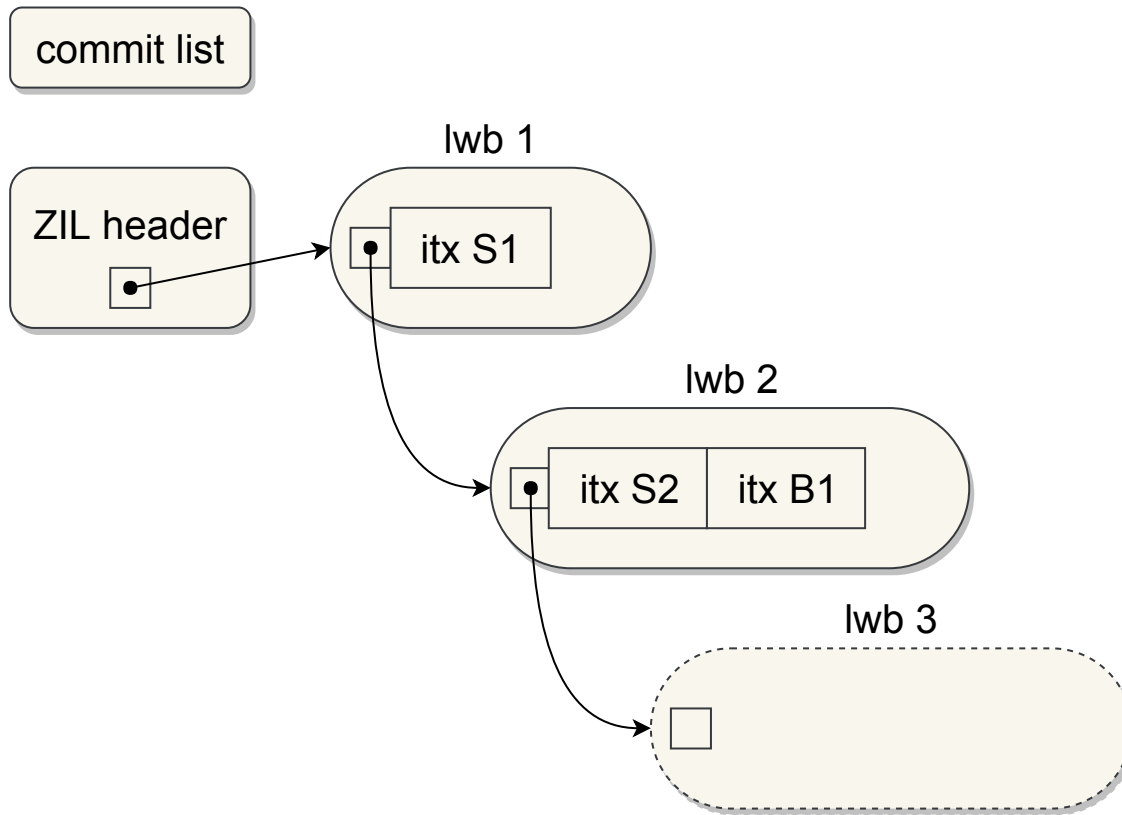
Example: `zil_commit` Object B

Example: `zil_commit` Object B

Example: `zil_commit` Object B

Example: `zil_commit` Object B

commit list

ZIL header

lwb 1

itx S1

lwb 2

itx S2   itx B1

Example: `zil_commit` Object B

commit list

ZIL header

lwb 1

itx S1

lwb 2

itx S2 | itx B1
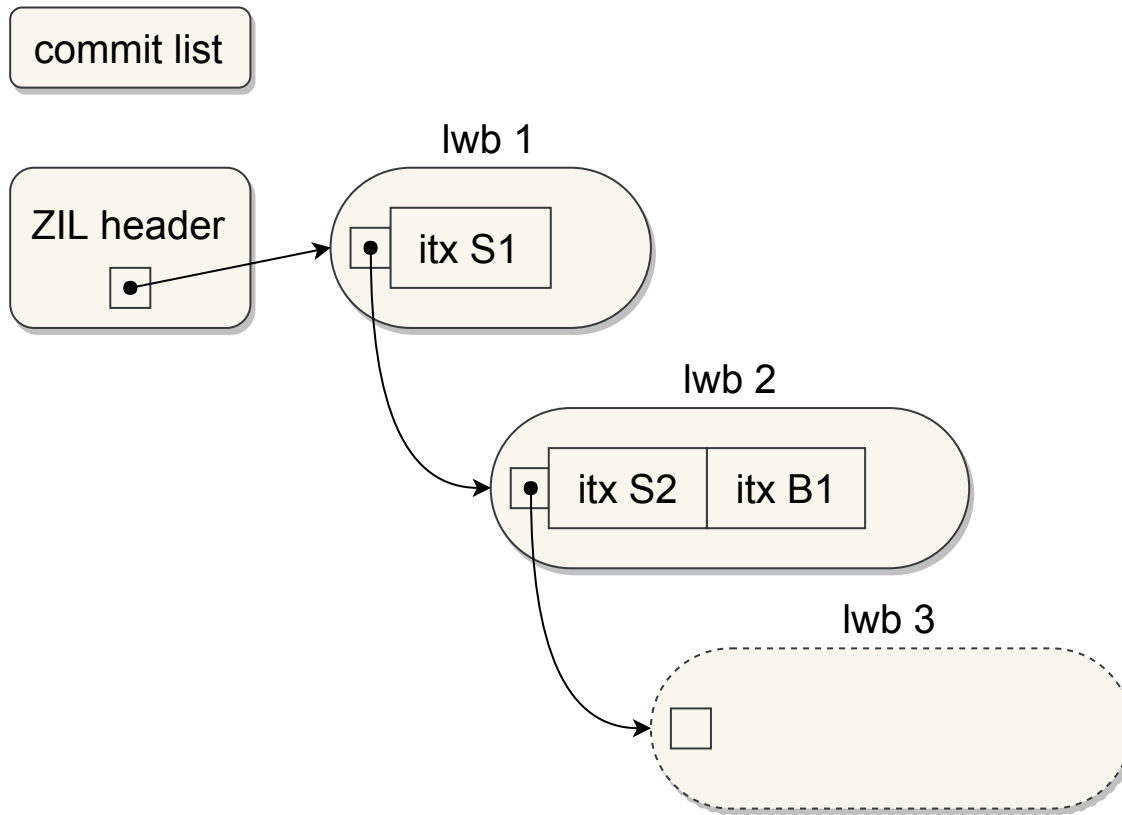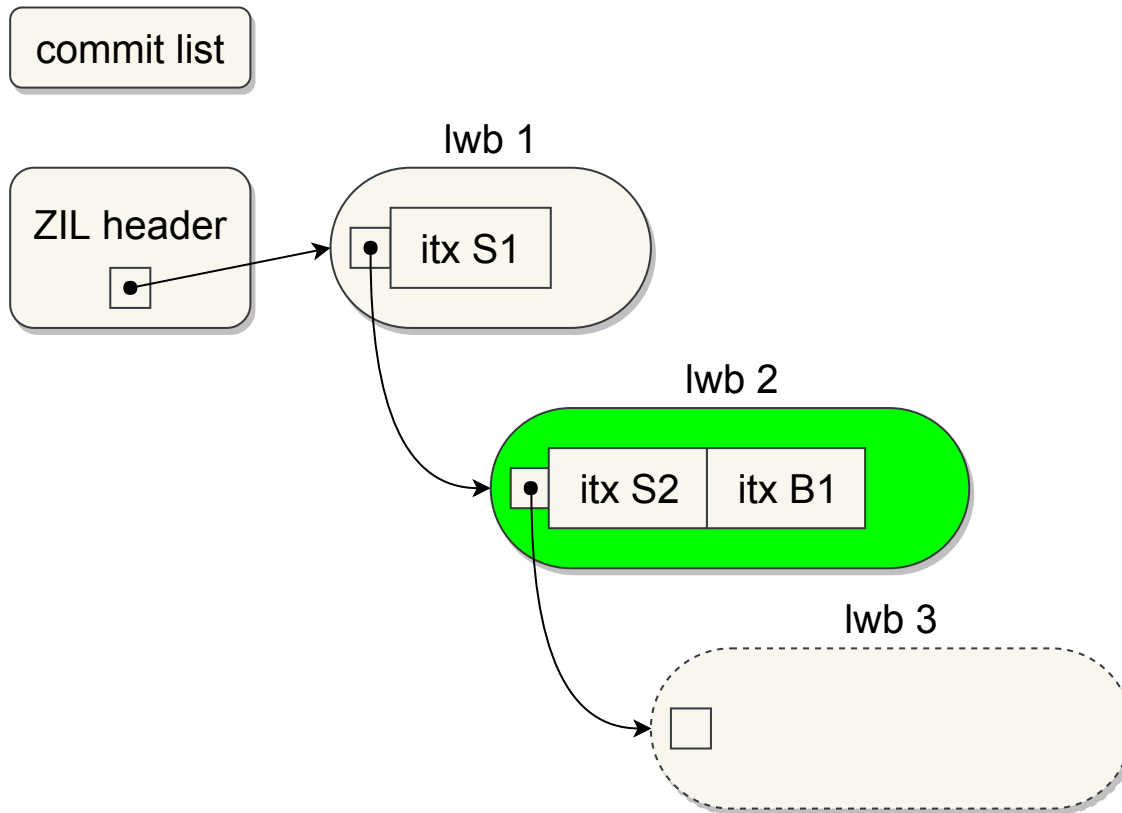
lwb 3

How are itx's written to disk?

- `zil_commit` handles the process of writing `itx_t`'s to disk:

    1. Move async itx's for object being commited, to the sync list

    2. Write all commit list `itx`'s to disk

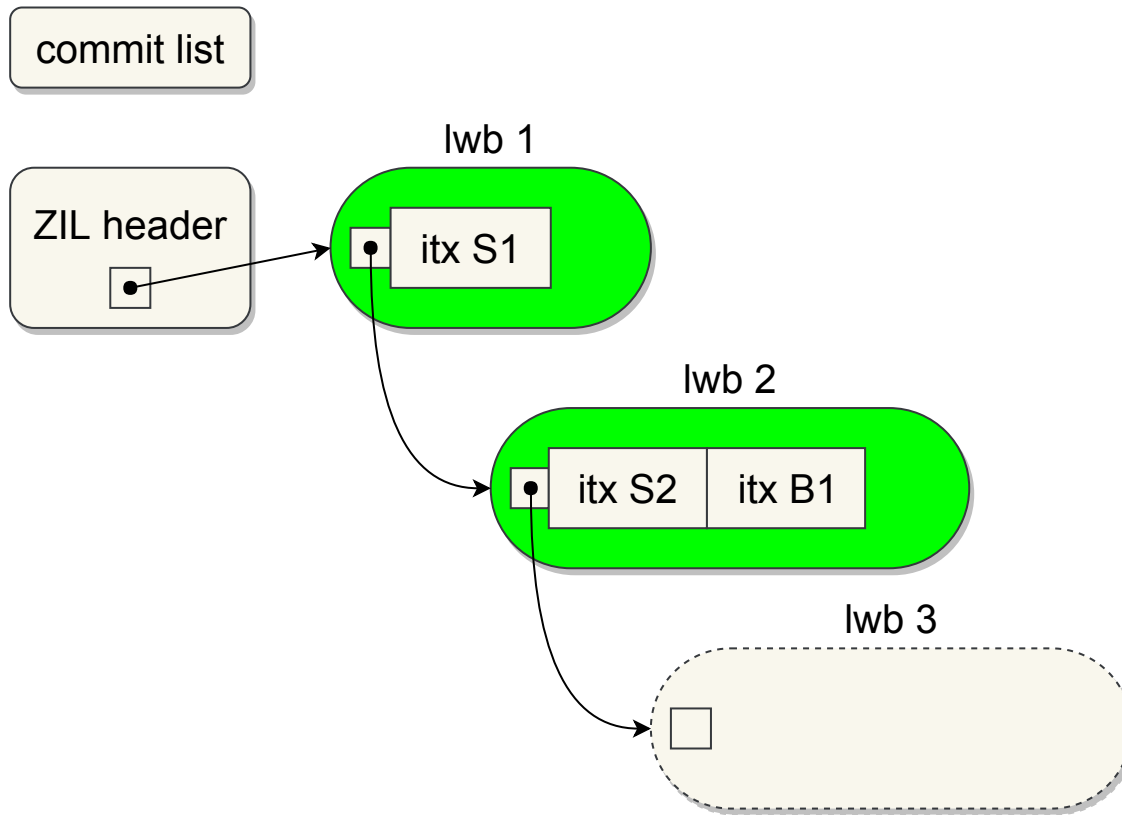   3. Wait for all ZIL block writes to complete

Example: `zil_commit` Object B

commit list

ZIL header

lwb 1

itx S1

lwb 2

itx S2 | itx B1

lwb 3

Example: `zil_commit` Object B

commit list

ZIL header

lwb 1

itx S1

lwb 2

itx S2 | itx B1

lwb 3

Example: `zil_commit` Object B

commit list

ZIL header

lwb 1

itx S1

lwb 2

itx S2 | itx B1

lwb 3

How are itx's written to disk?

- `zil_commit` handles the process of writing `itx_t`'s to disk:

  1. Move async itx's for object being commited, to the sync list

  2. Write all commit list `itx`'s to disk

  3. Wait for all ZIL block writes to complete

  4. Flush VDEVs and notify waiting threads

2 – ZIL Block Sizing + Performance

# ZIL Block Sizing + Performance

- ZIL blocks must be "pre-allocated", due to on-disk format

    - Block size chosen at time of allocation

- Allocated block size can dramatically impact performance:

    - "too big" – wasted space

    - "too small" – too many (small) IOPs issued to disk

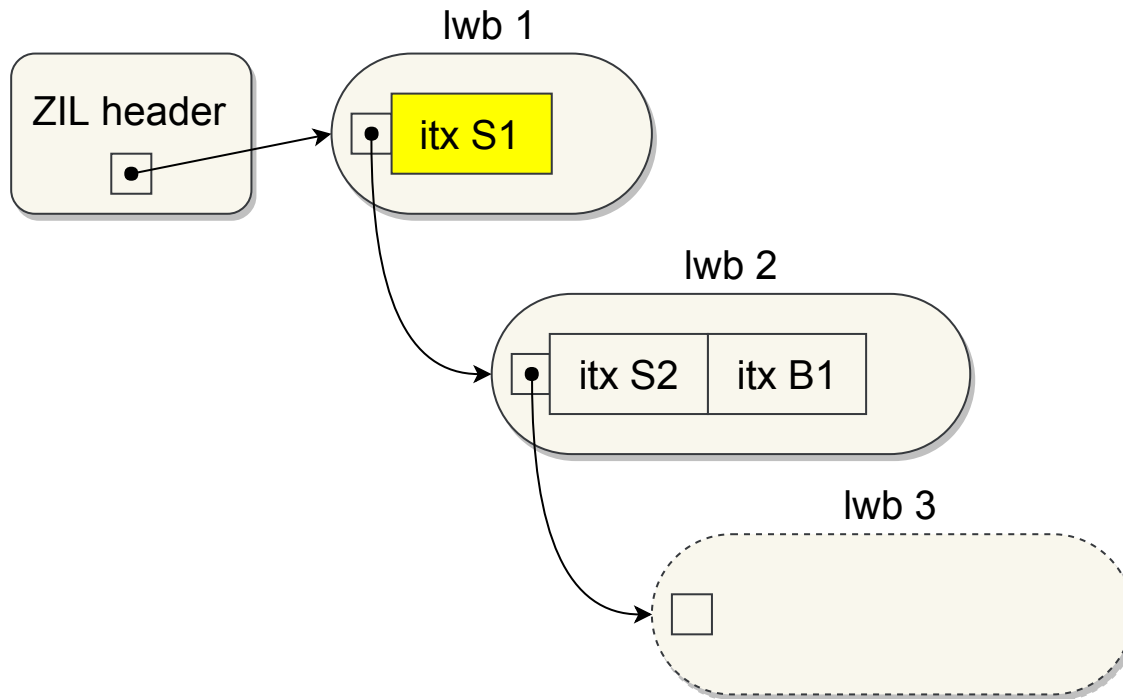   - "just right" – large IOPs filled with `itx`'s

3 – Problem

Problem

1. `itx`'s grouped and written in "batches"

    ◦ The commit list constitutes a batch

    ◦ Batch size proportional to sync workload on system

2. Waiting threads only notified when *all* ZIL blocks in batch complete
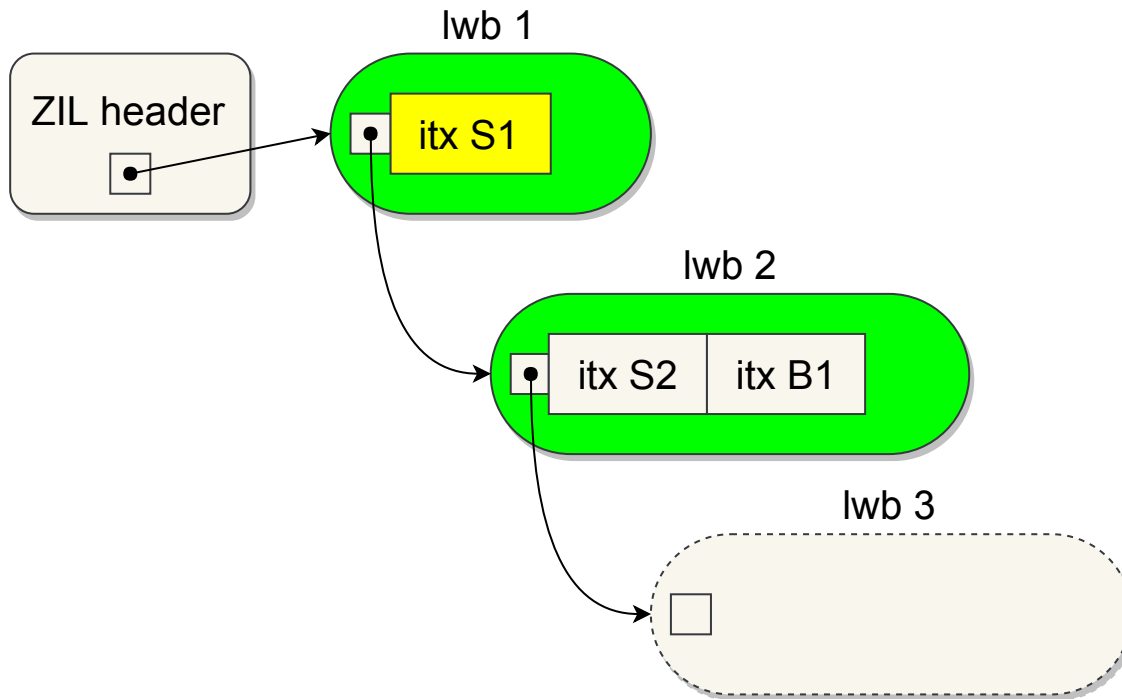
3. Only a single batch processed at a time

Example Batch
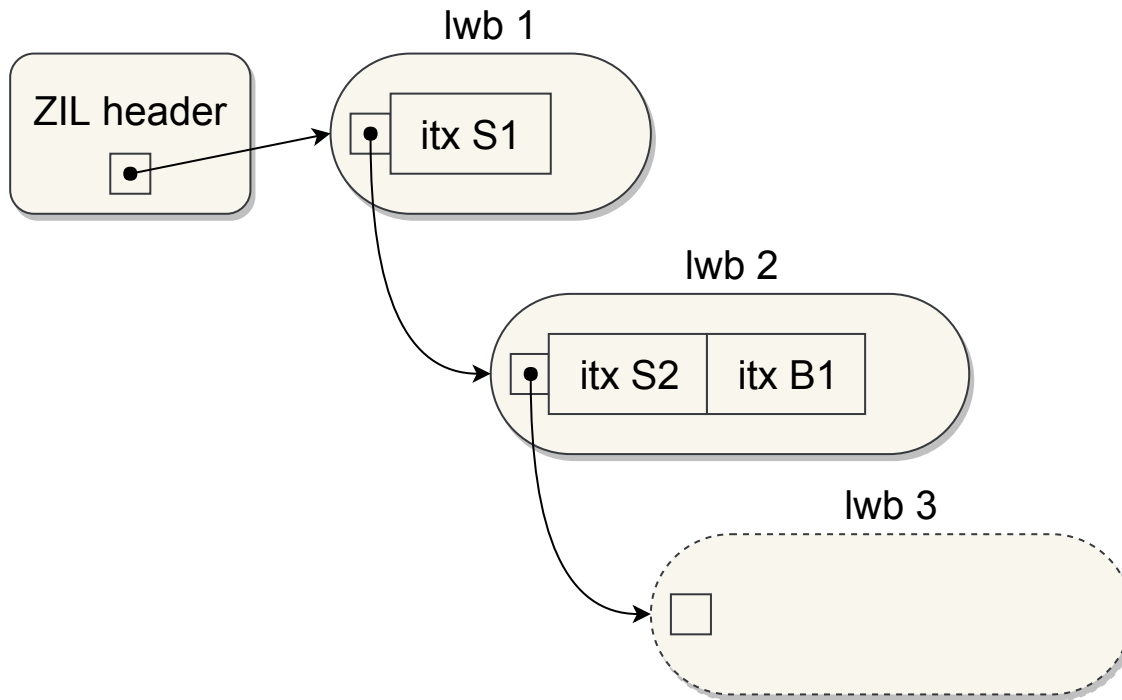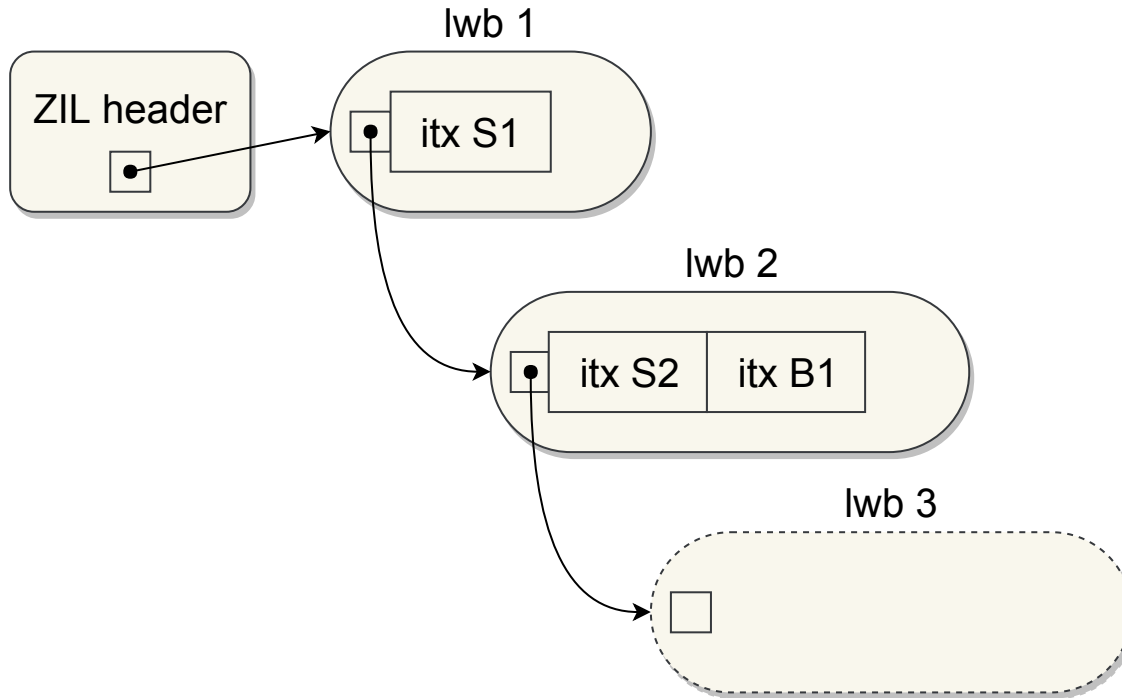
Example "itx S1"

Example "itx S1"

Implications

1. `zil_commit` latency proportional to system workload, *not* disk latency

2. Disk "anomalies" → larger batches → increased `zil_commit` latency

3. New calls to `zil_commit` wait for "current" batch, *and* "next" batch
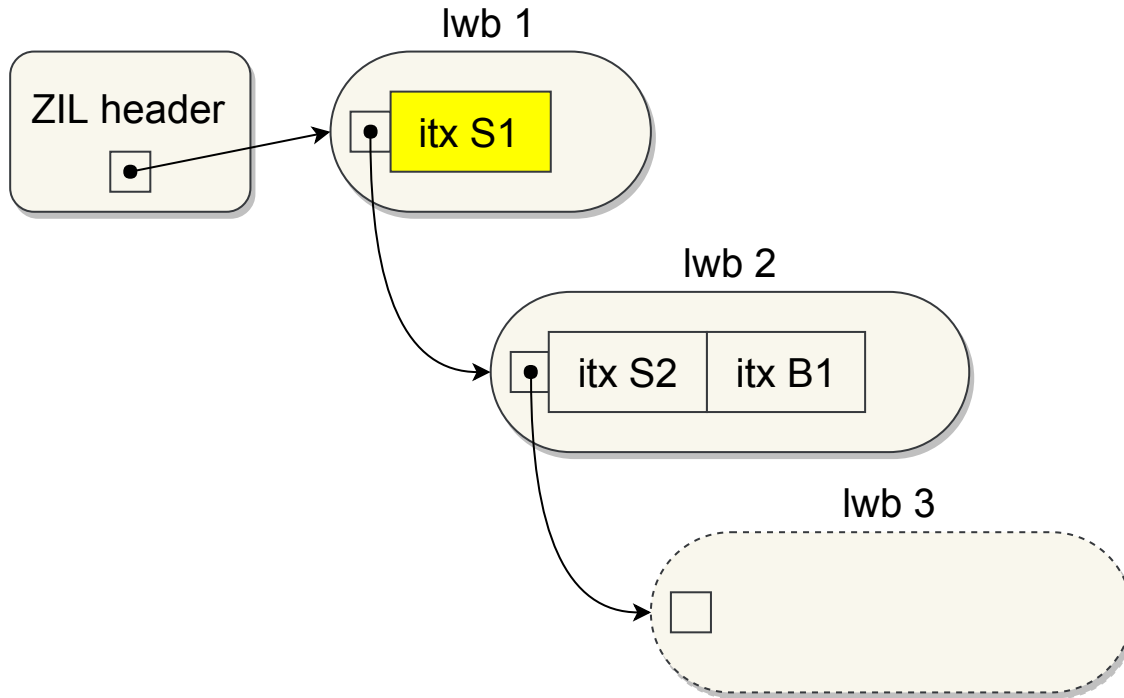
3 – Solution

Solution

- Remove concept of "batches":

  1. Allow `zil_commit` to issue new ZIL block writes immediately

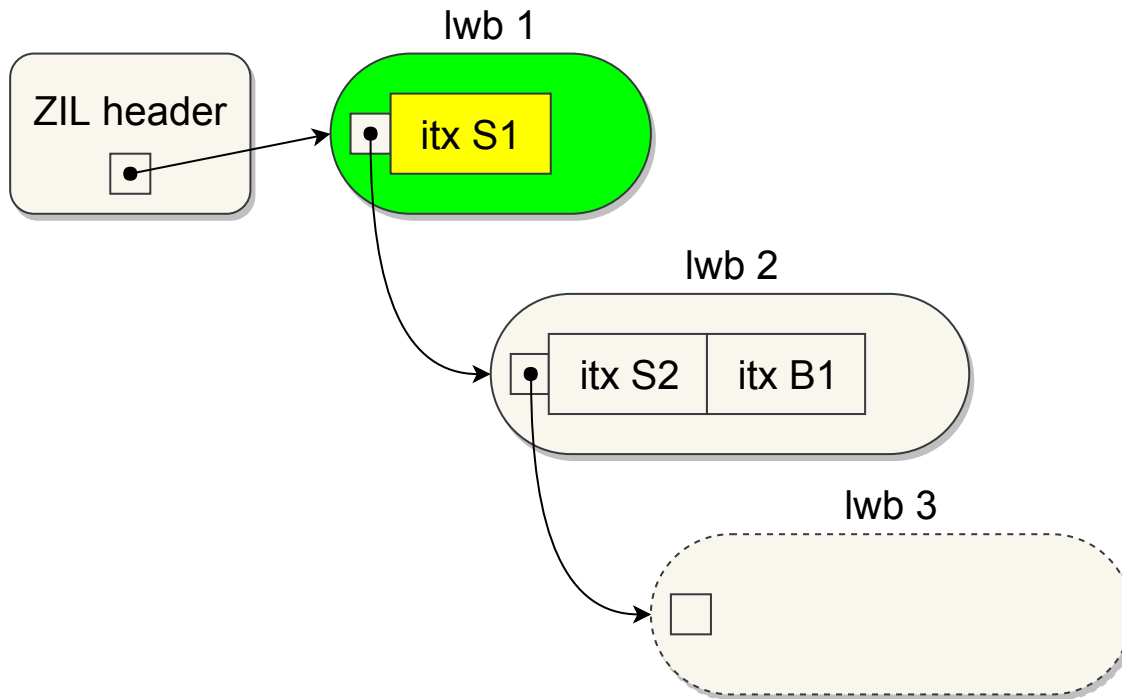 2. Notify threads immediately when *dependent* `itx`'s on disk

Example "Batch"

lwb 1

ZIL header

itx S1

lwb 2

itx S2 | itx B1

lwb 3

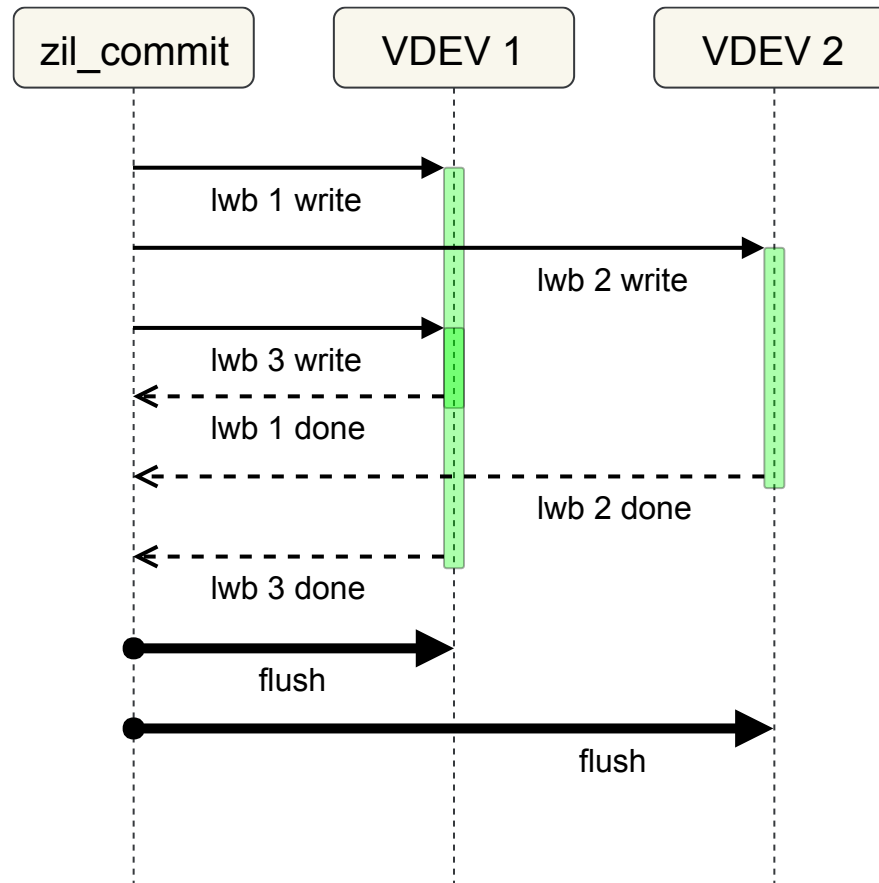# Example "itx S1"

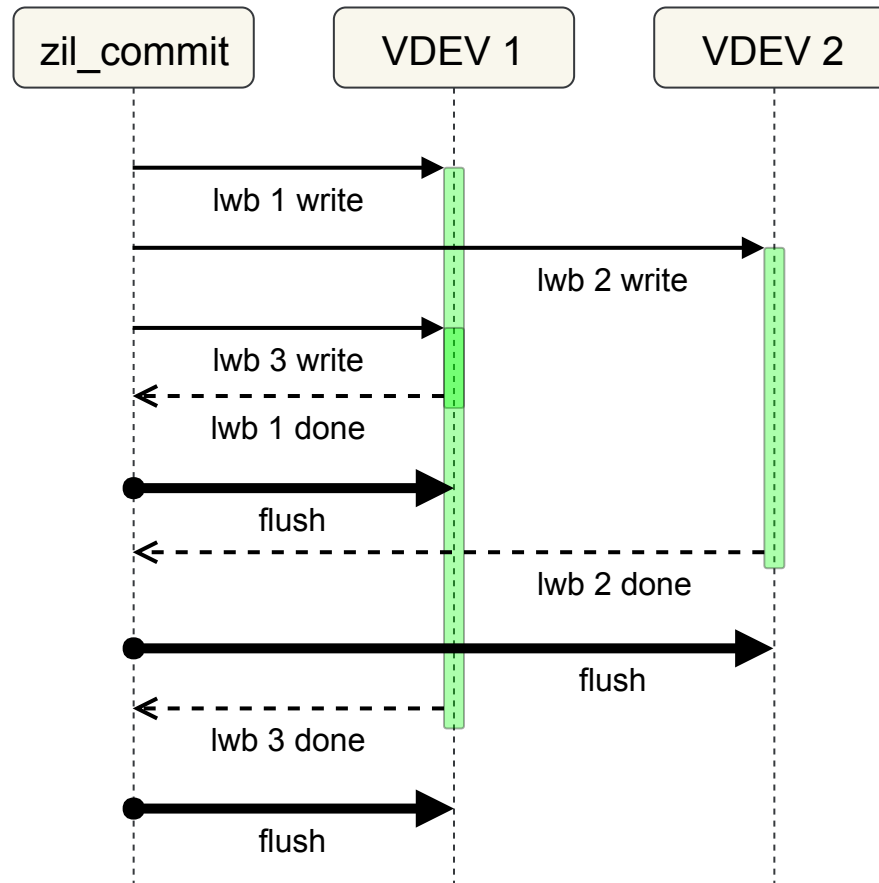# Example "itx S1"

4 – Changes to VDEV Flush

Details

- A ZIL block is not "persistent" until the VDEV is flushed

- Prior mechanics:

  - Single VDEV flush for each VDEV, after batch completes

  - 1 flush per many lwb's

- New mechanics:

  - VDEV flush issued after each ZIL block written

  - 1 flush per 1 lwb

Example: Before

Example: After

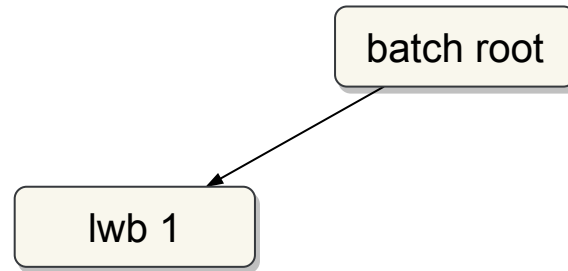# 4 – Changes to ZIL Block ZIO Tree

Details

- ZIL blocks issued to disk using ZIOs

- Prior mechanics:

  - "root" ZIO created for each batch

    - "write" ZIOs, for all `lwb`'s in batch, are children of root ZIO

  - "flush" ZIOs issued separately after root ZIO completes

- New mechanics:

  - "root" ZIO created for each lwb

    - "write" and "flush" ZIOs are child of root ZIO

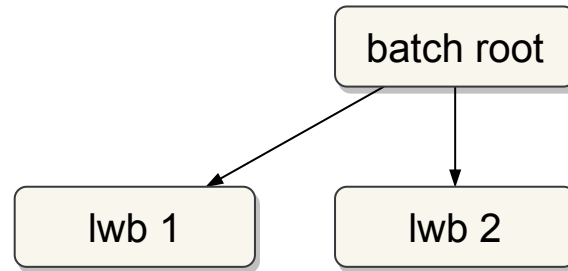 - "next" lwb root ZIO become parent of "current" lwb root ZIO
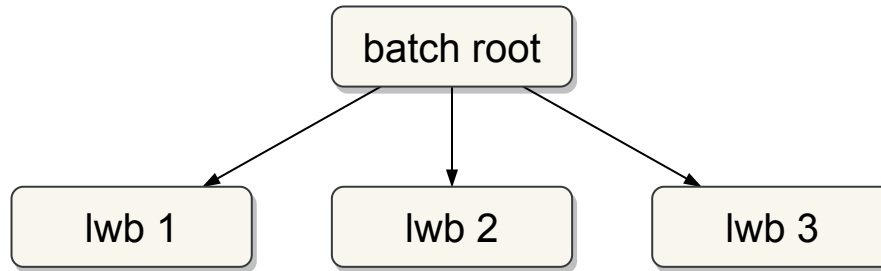
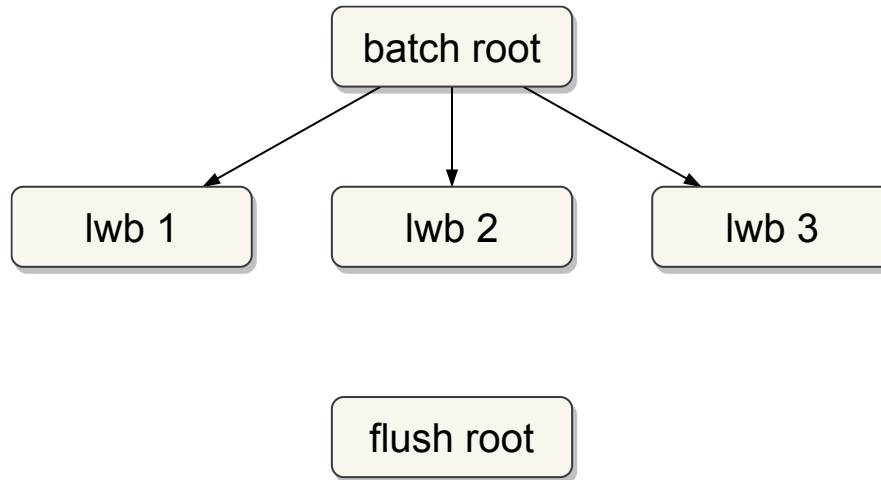# Example: Before

batch root

Example: Before

batch root

lwb 1

Example: Before

```
            ┌──────────────┐
            │  batch root  │
            └──────────────┘
               ╱          │
              ╱           ▼
             ▼       ┌──────────┐
    ┌──────────┐     │  lwb 2   │
    │  lwb 1   │     └──────────┘
    └──────────┘
```
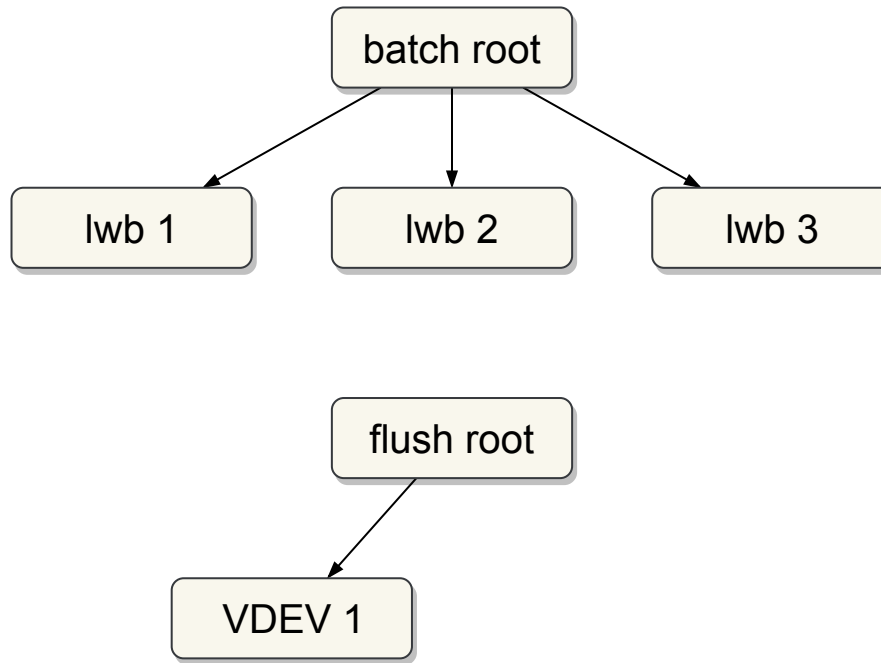
Example: Before
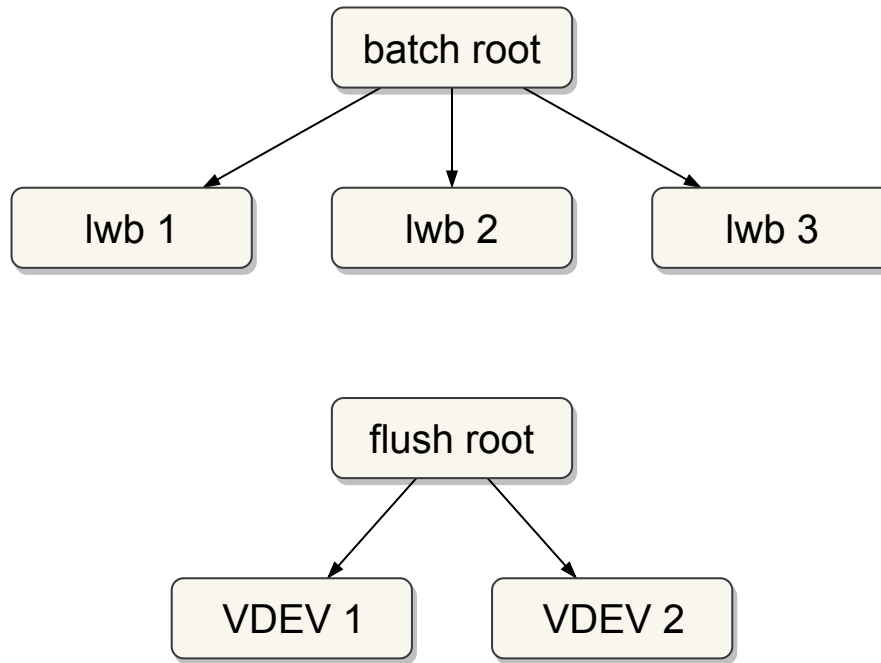
Example: Before
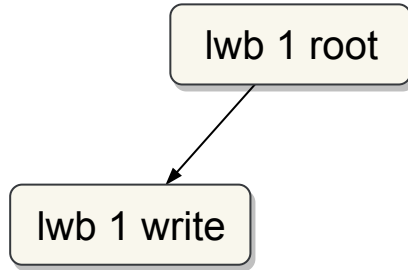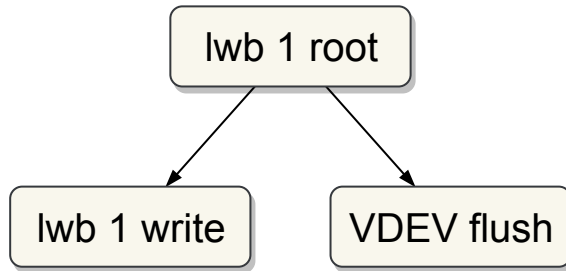
Example: Before

Example: Before

```
                    ┌─────────────┐
                    │ batch root  │
                    └─────────────┘
                   ↙       ↓        ↘
         ┌─────────┐  ┌─────────┐  ┌─────────┐
         │  lwb 1  │  │  lwb 2  │  │  lwb 3  │
         └─────────┘  └─────────┘  └─────────┘


                    ┌─────────────┐
                    │ flush root  │
                    └─────────────┘
                     ↙           ↘
           ┌─────────┐         ┌─────────┐
           │ VDEV 1  │         │ VDEV 2  │
           └─────────┘         └─────────┘
```

Example: After

lwb 1 root

Example: After
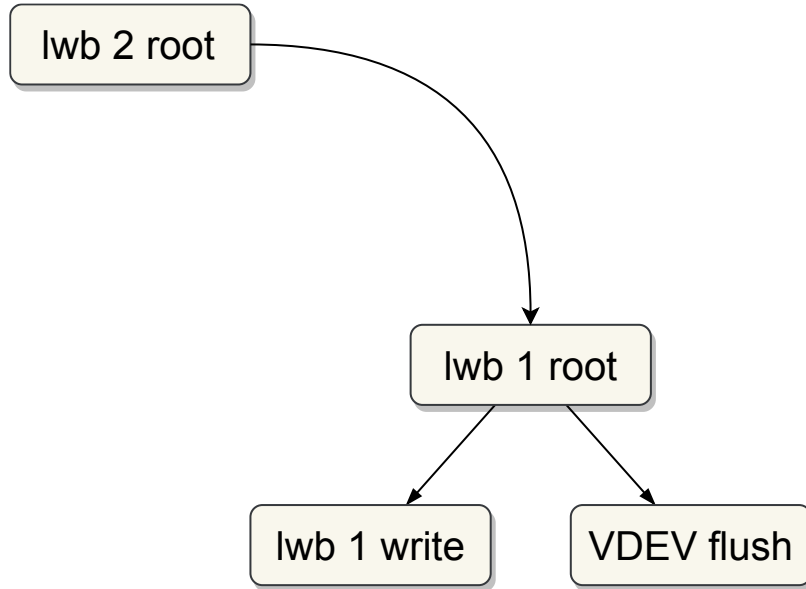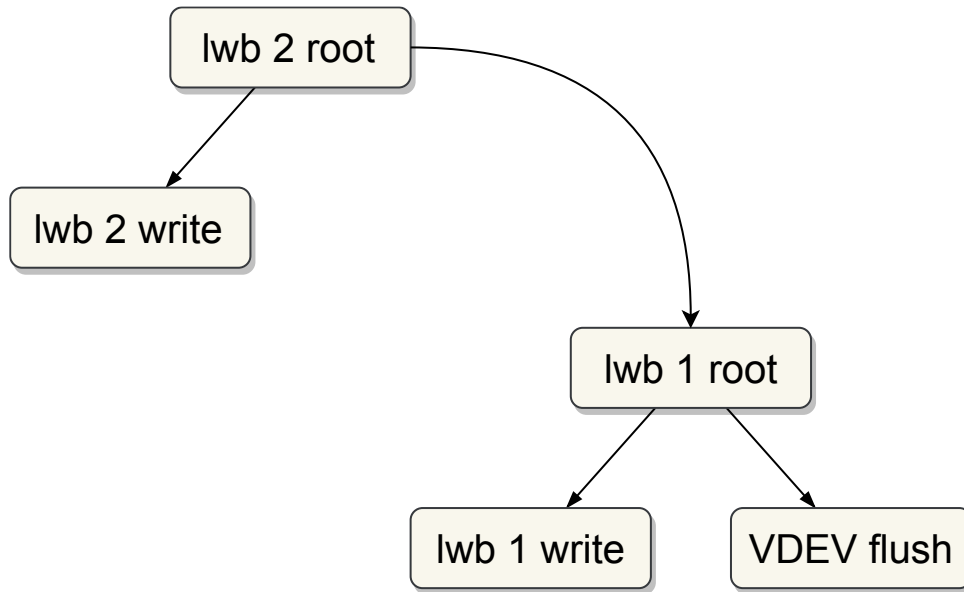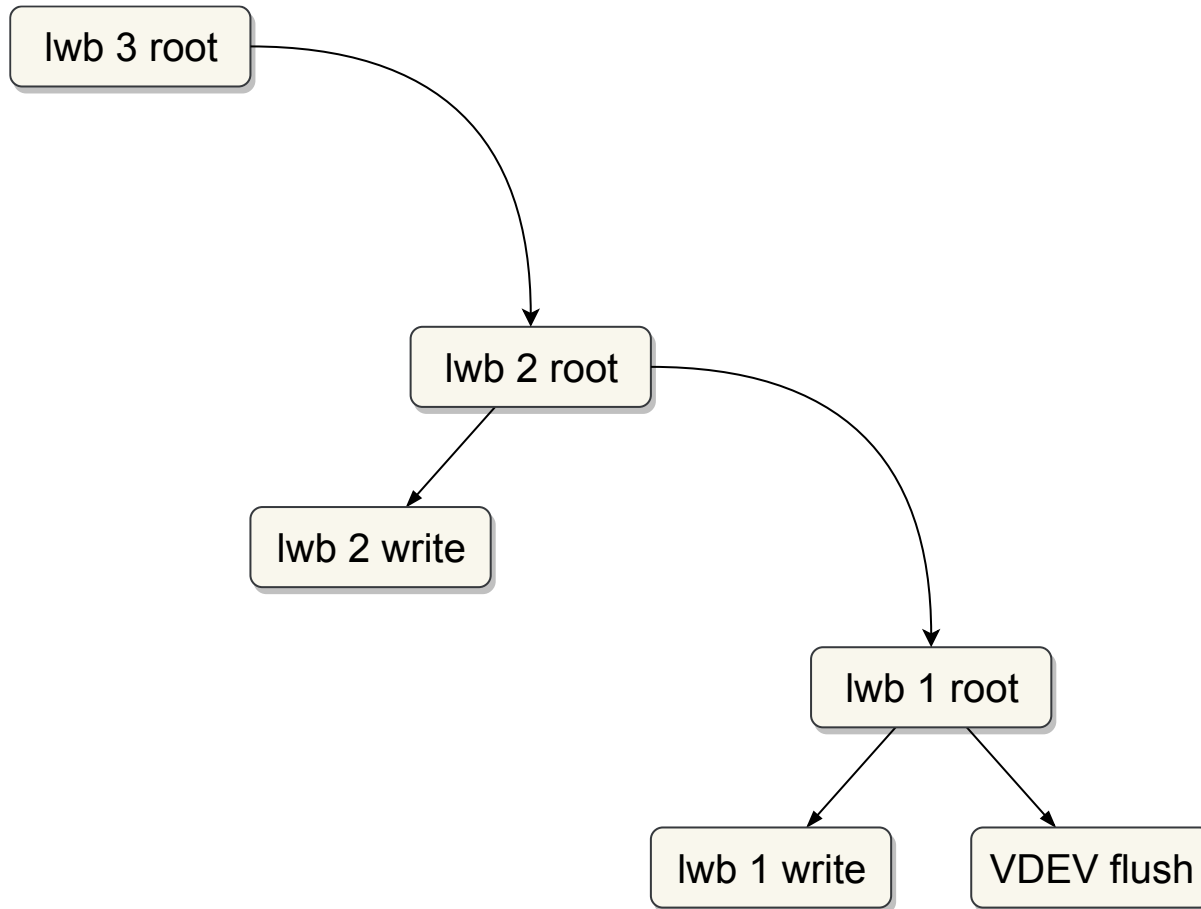
lwb 1 root

lwb 1 write

Example: After

```
          ┌──────────────┐
          │  lwb 1 root  │
          └──────┬───────┘
           ╱           ╲
          ╱             ╲
┌──────────────┐   ┌──────────────┐
│  lwb 1 write │   │  VDEV flush  │
└──────────────┘   └──────────────┘
```

Example: After

```
lwb 2 root  ──────┐
                  │
                  ▼
              lwb 1 root
             ╱        ╲
            ▼          ▼
      lwb 1 write   VDEV flush
```
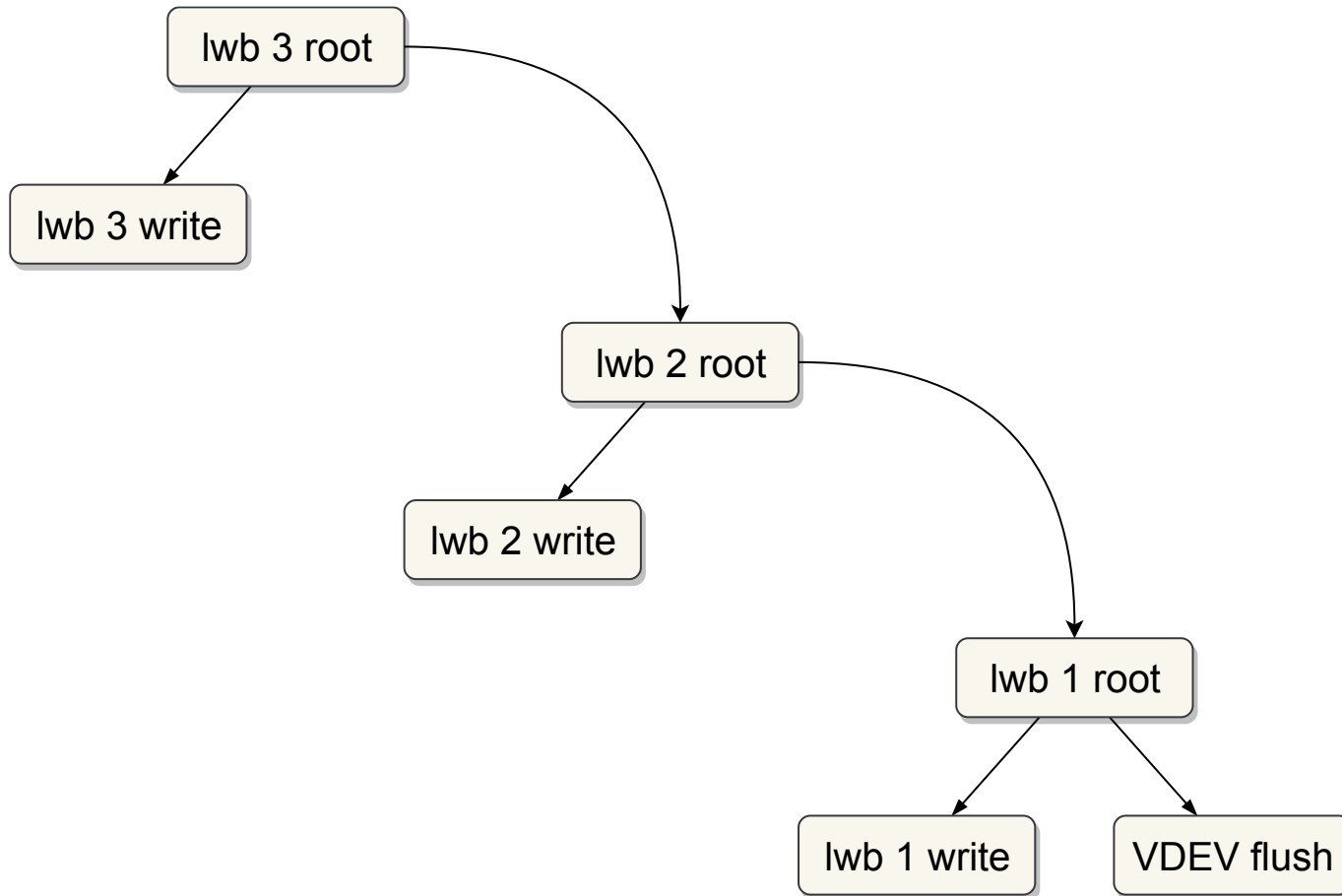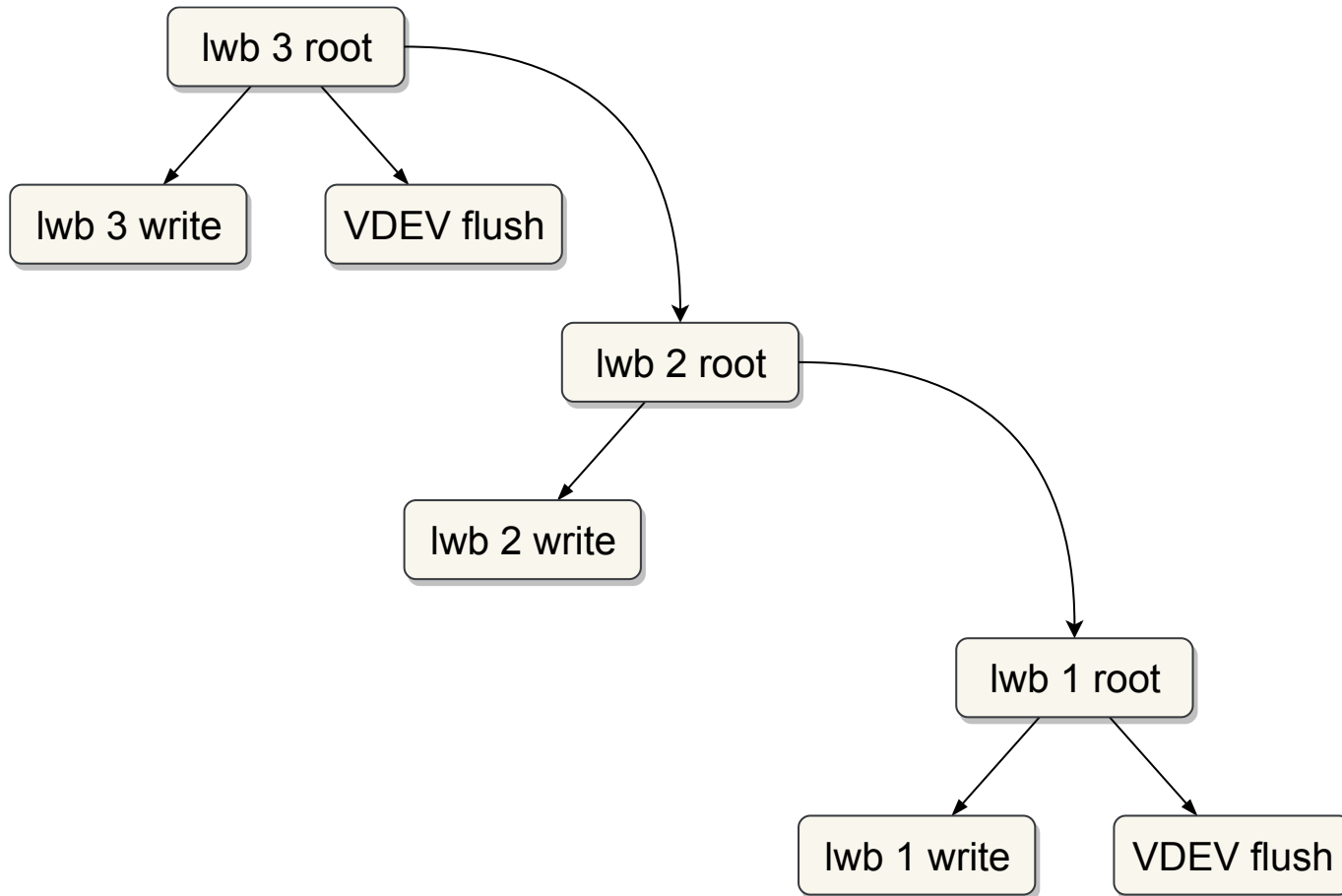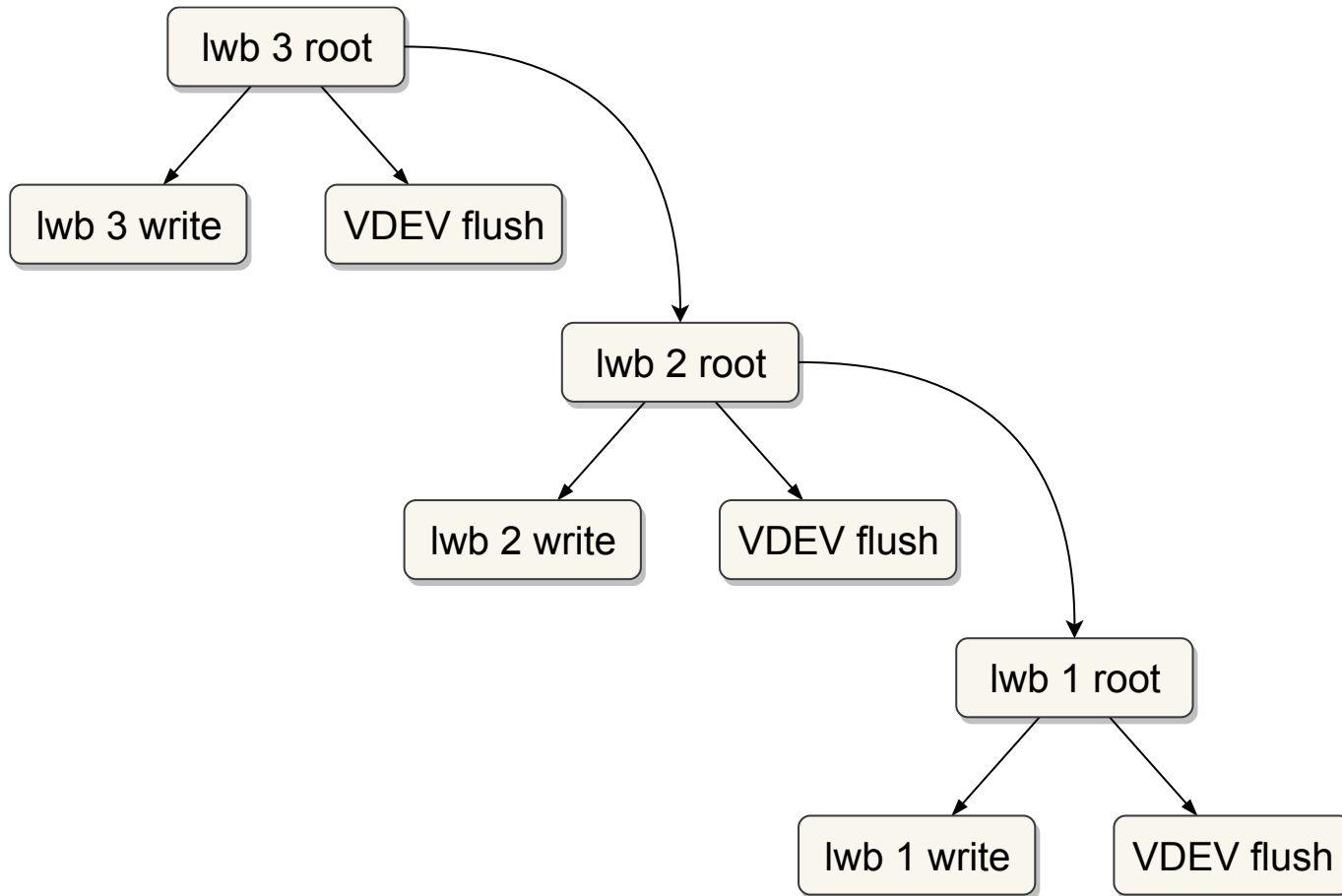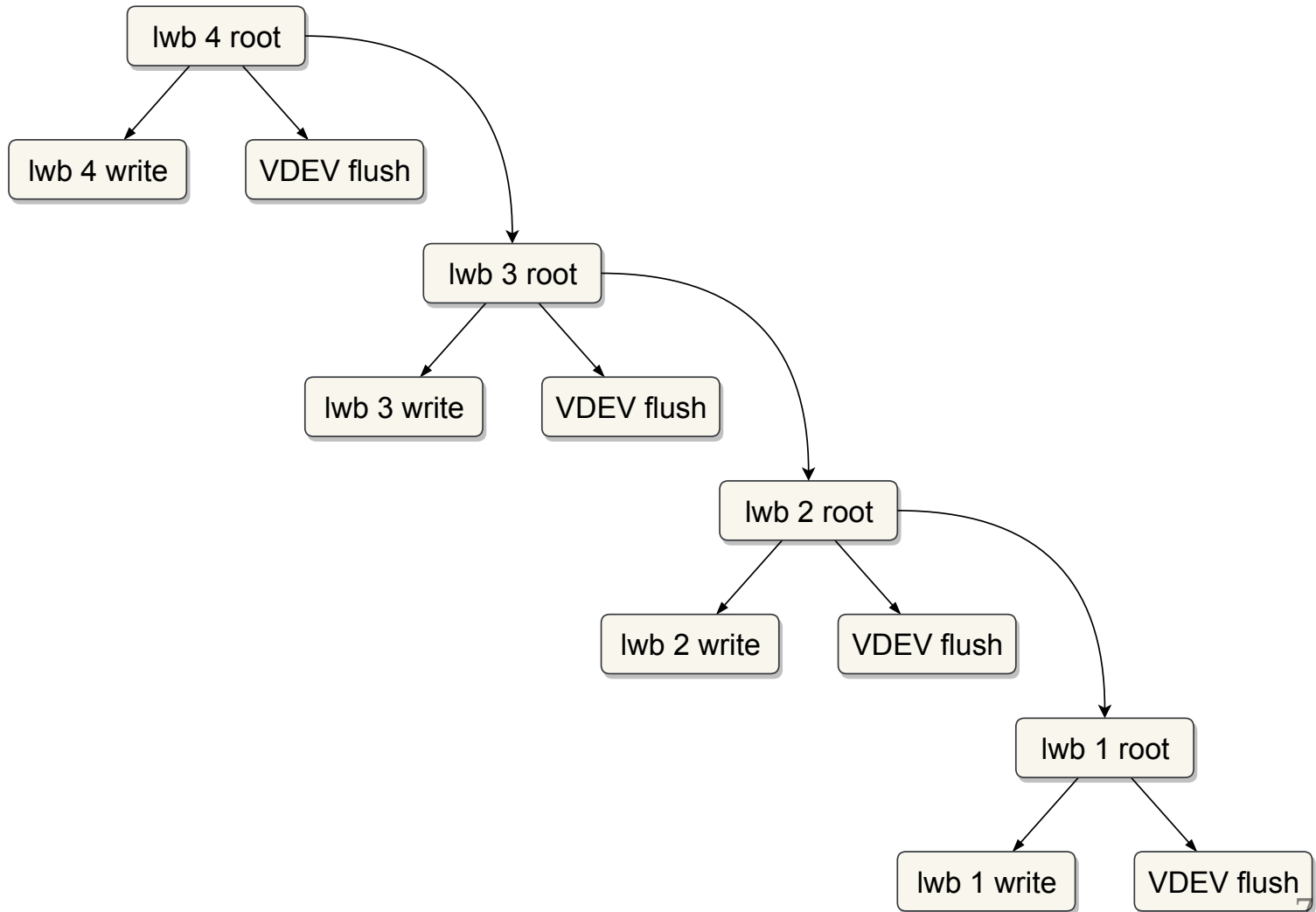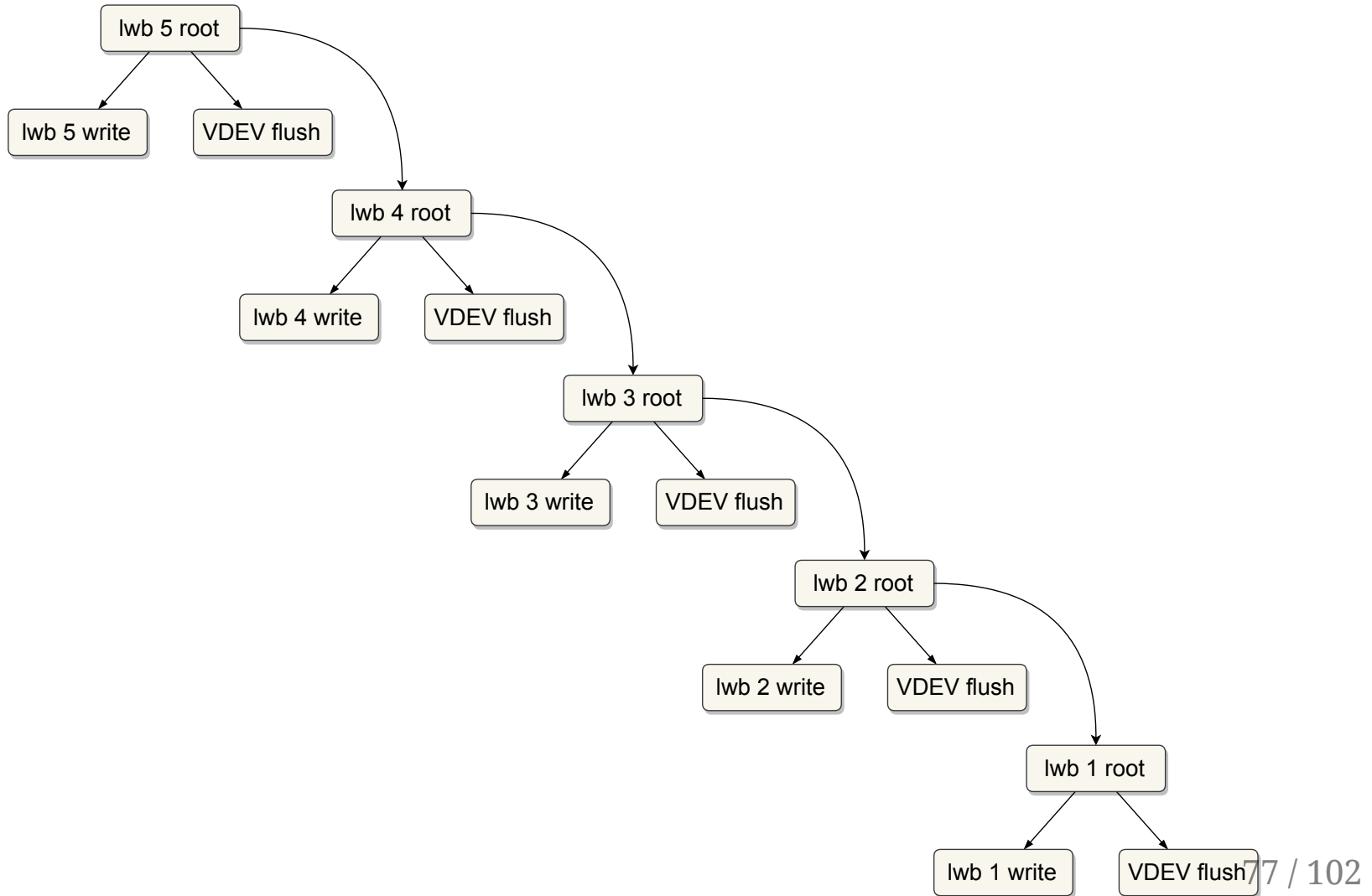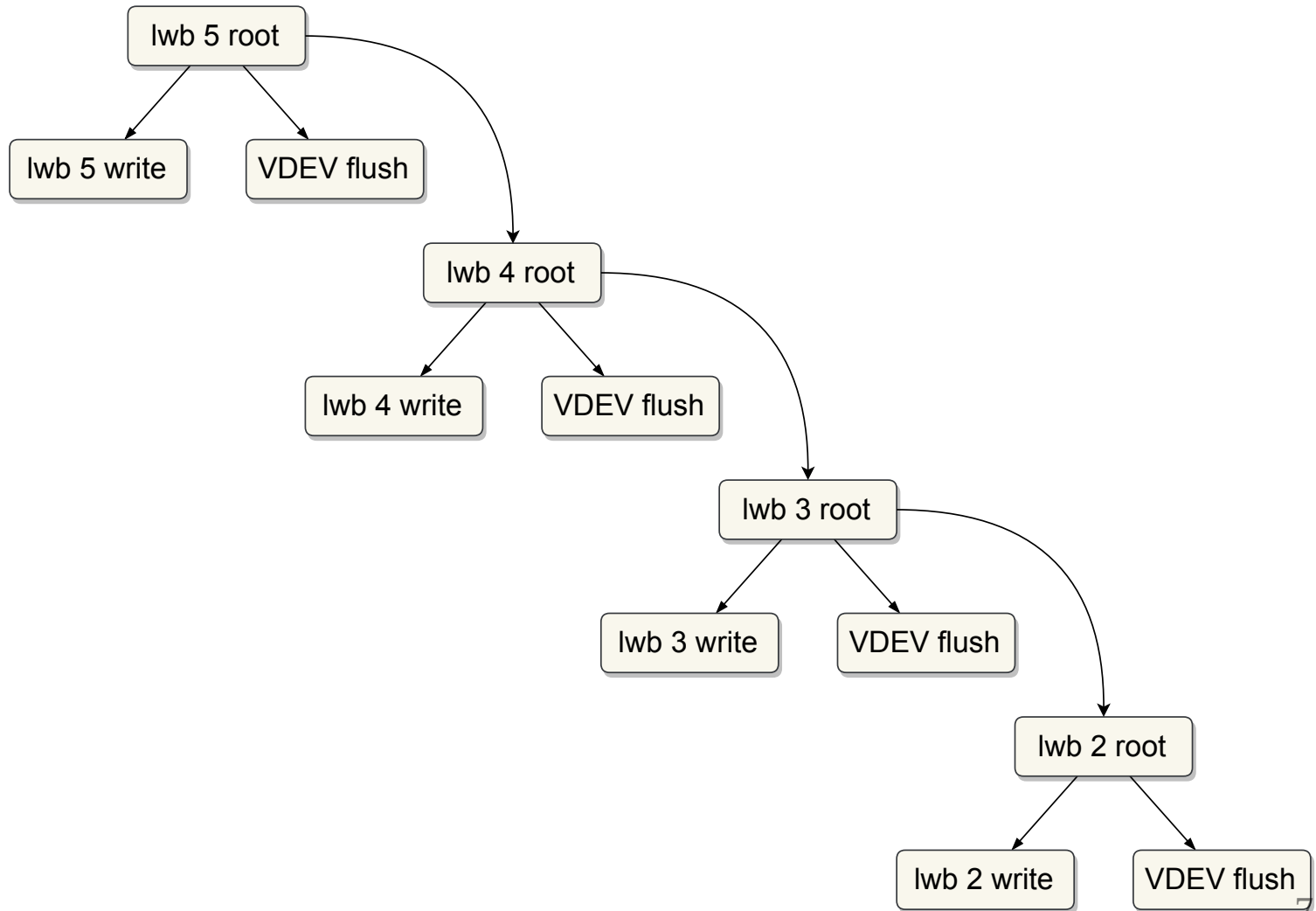
Example: After

Example: After

Example: After

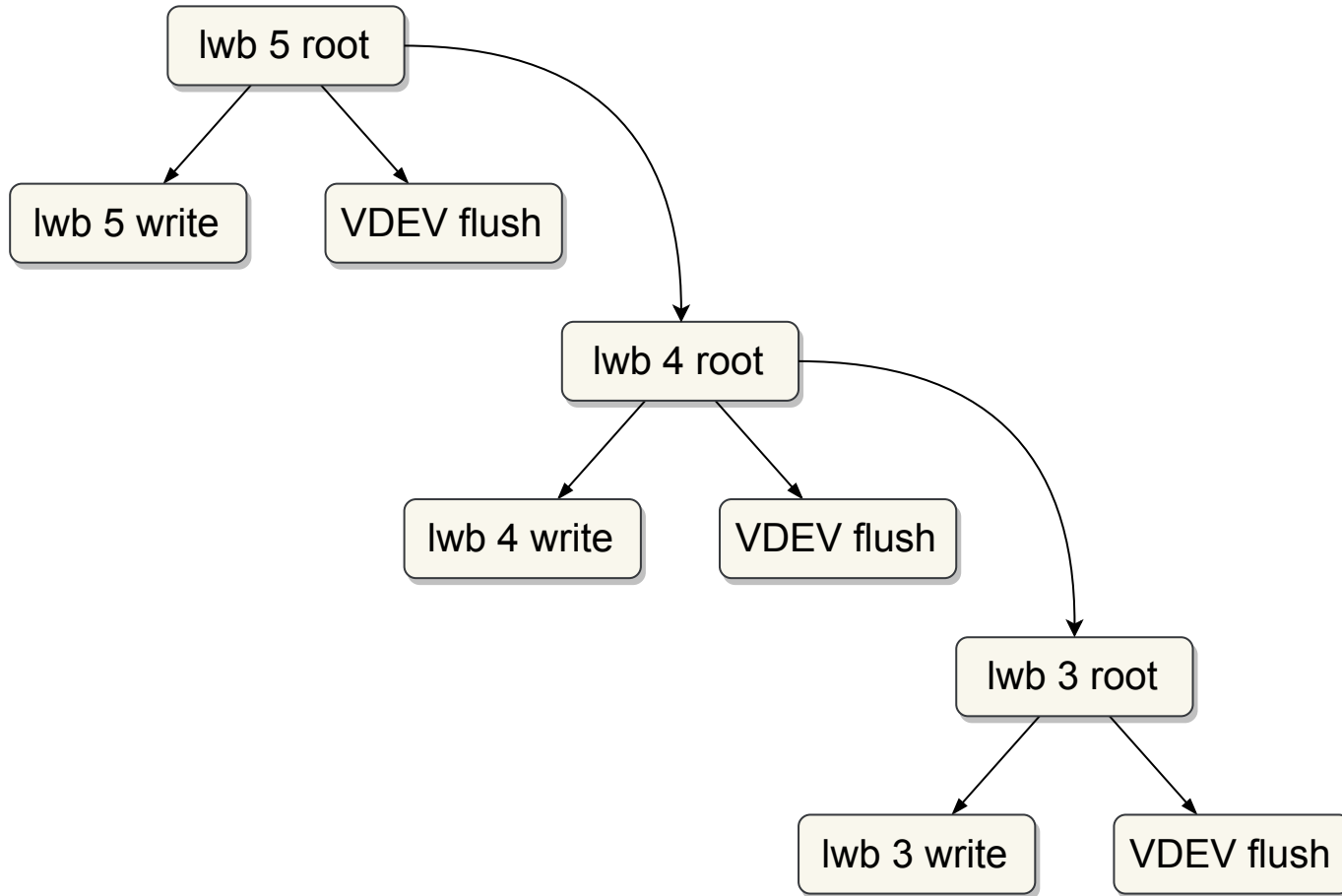# Example: After

Example: After
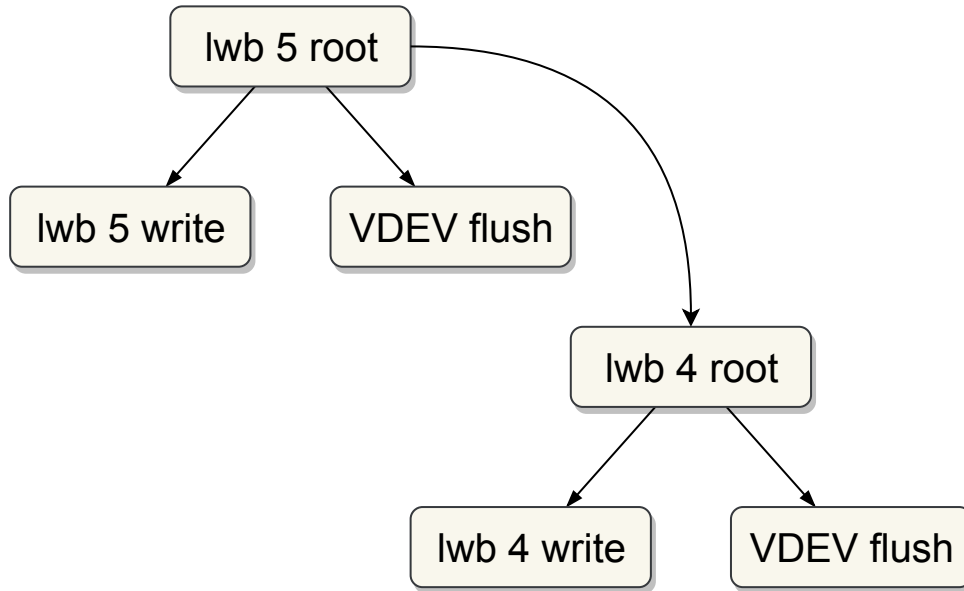
Example: After

Example: After

Example: After

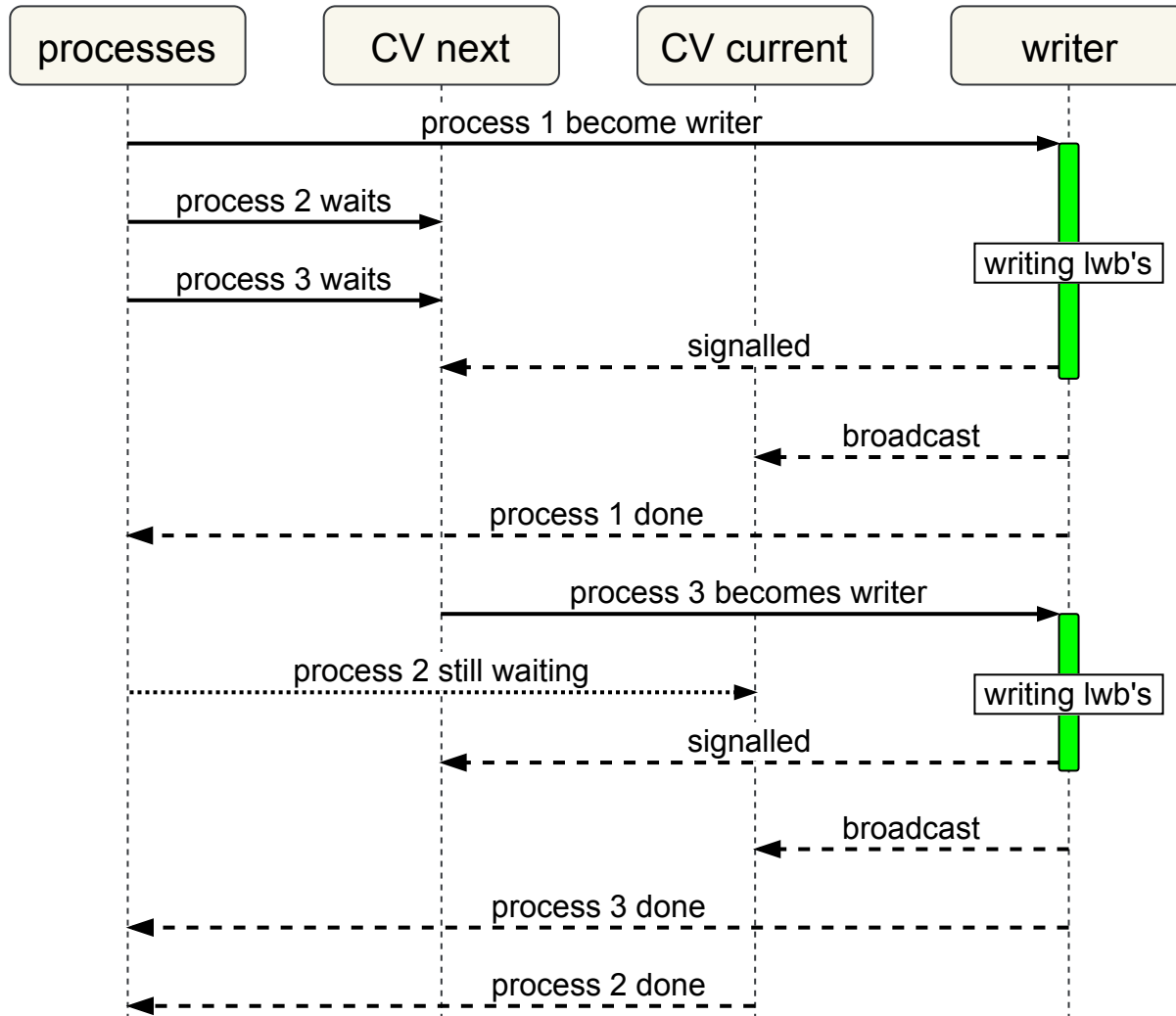# Example: After

Example: After

4 – Changes to Waiter Notification

Details: Before

- 2 condition variables (CV), for "current" and "next" batch

- Threads that called `zil_commit`:

  - Assigned to "next batch", wait on next batch's CV

- When "current" batch completes

  1. All threads waiting on "current" signalled, they'd return

  2. One thread waiting on "next" signalled, becomes "writer"

  3. "next" and "current" CV swapped

- Ultimately, these two CVs are the source of original problem

Example: Before

Example: Before
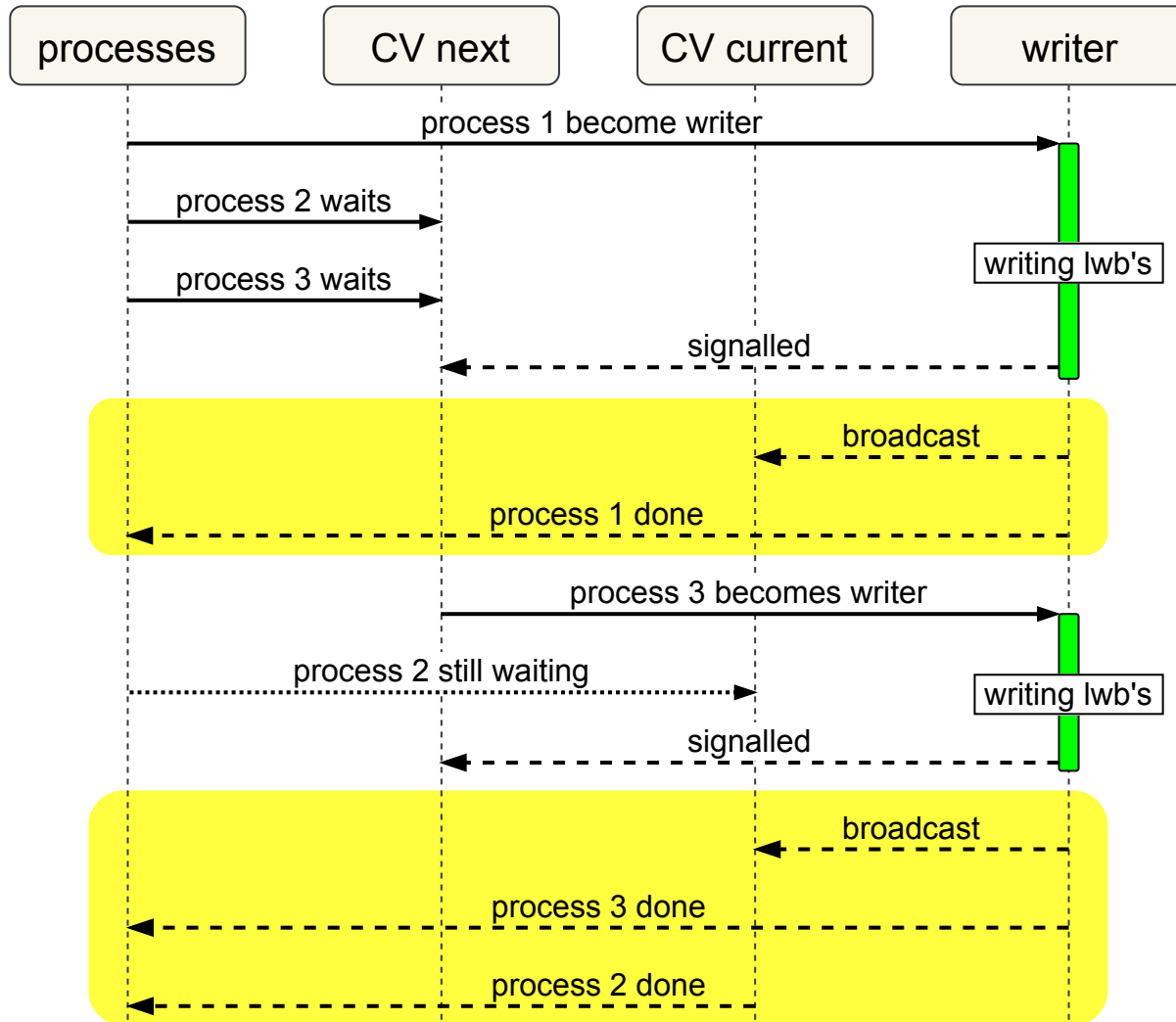
Details: After

- Each time a process calls `zil_commit`:

 - A new CV is allocated for this specific process to wait on

  - A new `TX_COMMIT` itx is inserted into the ZIL itx tree

    - The "commit itx" has a pointer to the process's CV

- When a commit itx is copied to an lwb:

  - No data copied into the lwb's buffer

  - Instead, itx's CV added to lwb's list of CVs

- When lwb's ZIO completes, list of CVs iterated and signalled

  - This is how we map which lwb a process is waiting for

Example: After
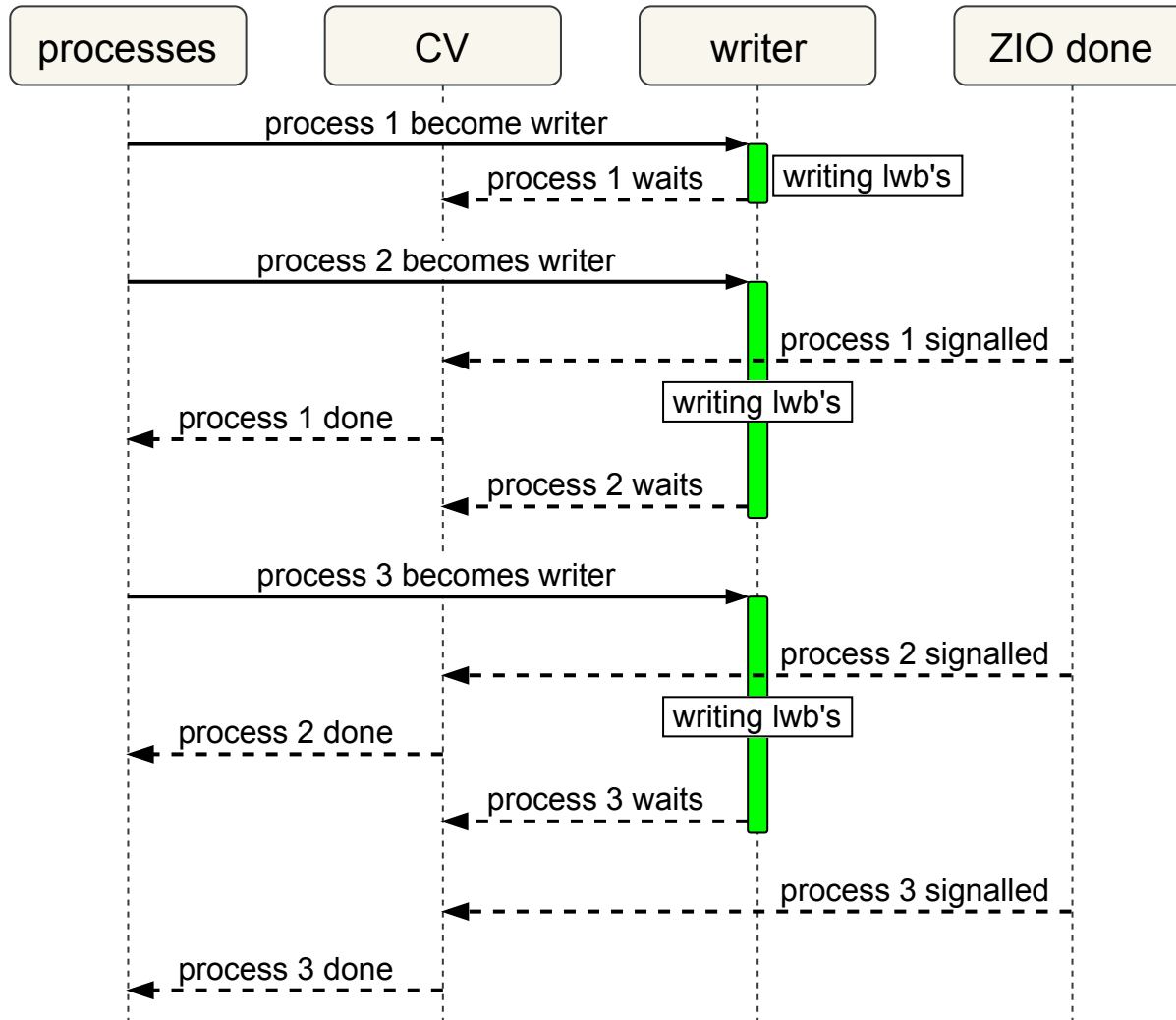
Example: After
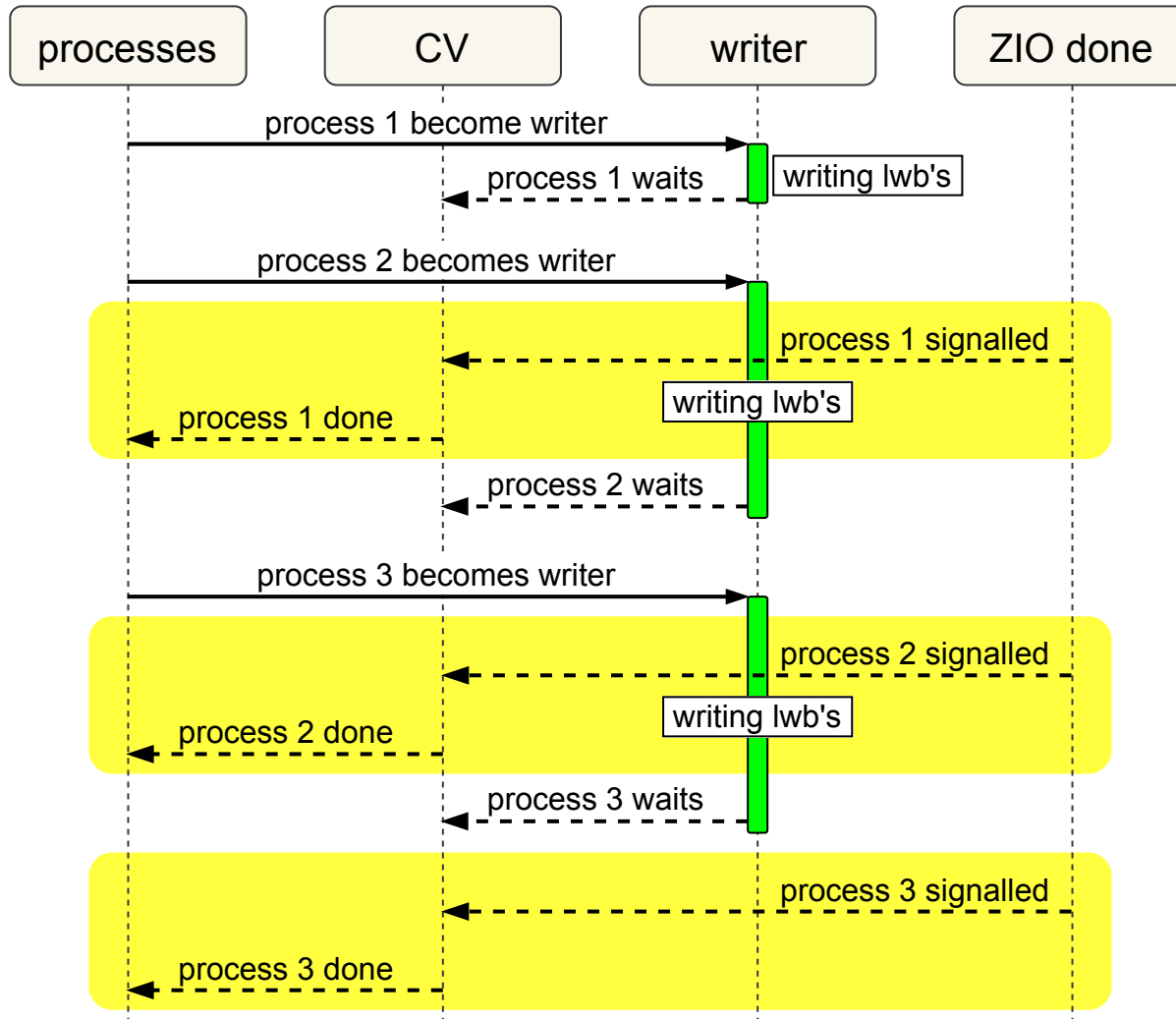
5 – Performance testing and results.

Details

- Two `fio` workloads used to drive a sync write workload

    1. `fio` was trying to perform sync writes as fast as it could

    2. `fio` was trying to perform 64 sync writes per second

- IOPs and latency measured with and without my changes

    - Other metrics also observed (`iostat`, flamegraphs, lwb info, etc.)

- 1, 2, 4, and 8 disk zpools; tested both SSD and HDD

- Full details can be found here

5 – Max Rate Workload – HDDs

% Change IOPs – Max Rate – HDDs

fio -- % change in write iops vs. number of fio threads

% Change Latency – Max Rate – HDDs

fio -- % change in average write latency vs. number of fio threads

Legend:
- project - 1 hdd
- project - 2 hdd
- project - 4 hdd
- project - 8 hdd

x-axis: number of fio threads issuing writes

y-axis: % change in average write latency reported by fio
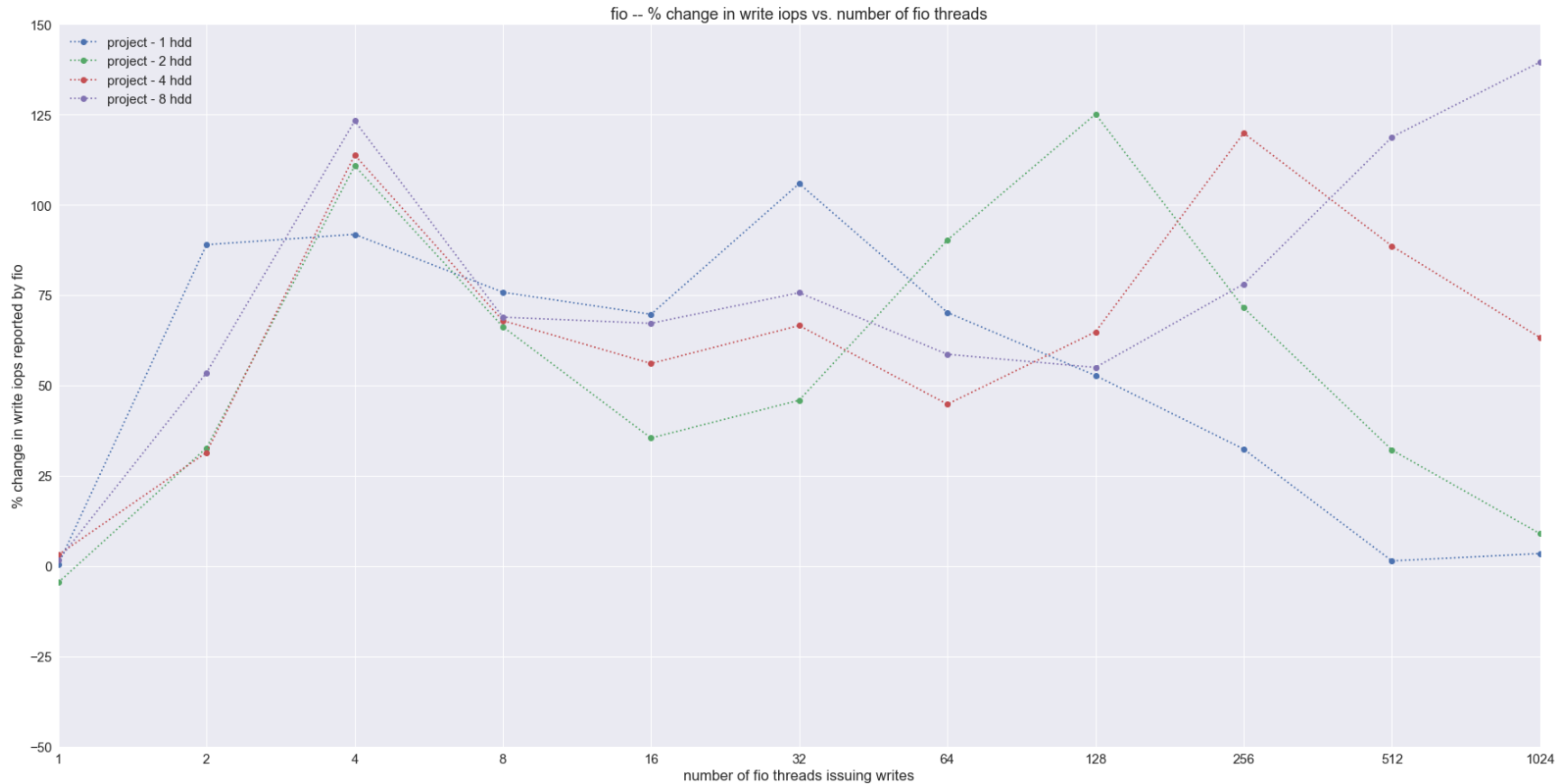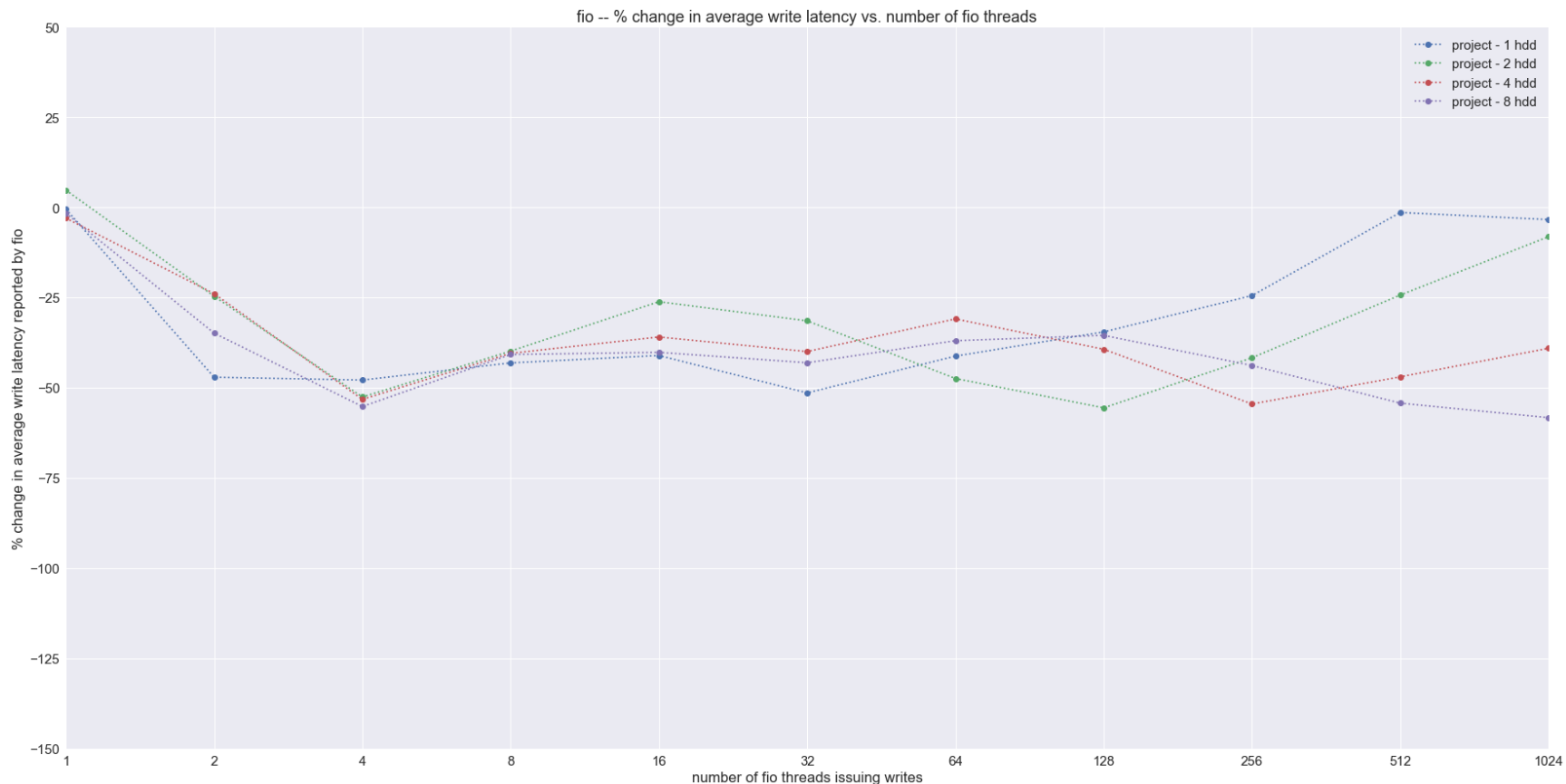
5 – Max Rate Workload – SSDs

% Change IOPs – Max Rate – SSDs

fio -- % change in write iops vs. number of fio threads

% Change Latency – Max Rate – SSDs

fio -- % change in average write latency vs. number of fio threads

Legend:
- project - 1 ssd
- project - 2 ssd
- project - 4 ssd
- project - 8 ssd

y-axis: % change in average write latency reported by fio

x-axis: number of fio threads issuing writes
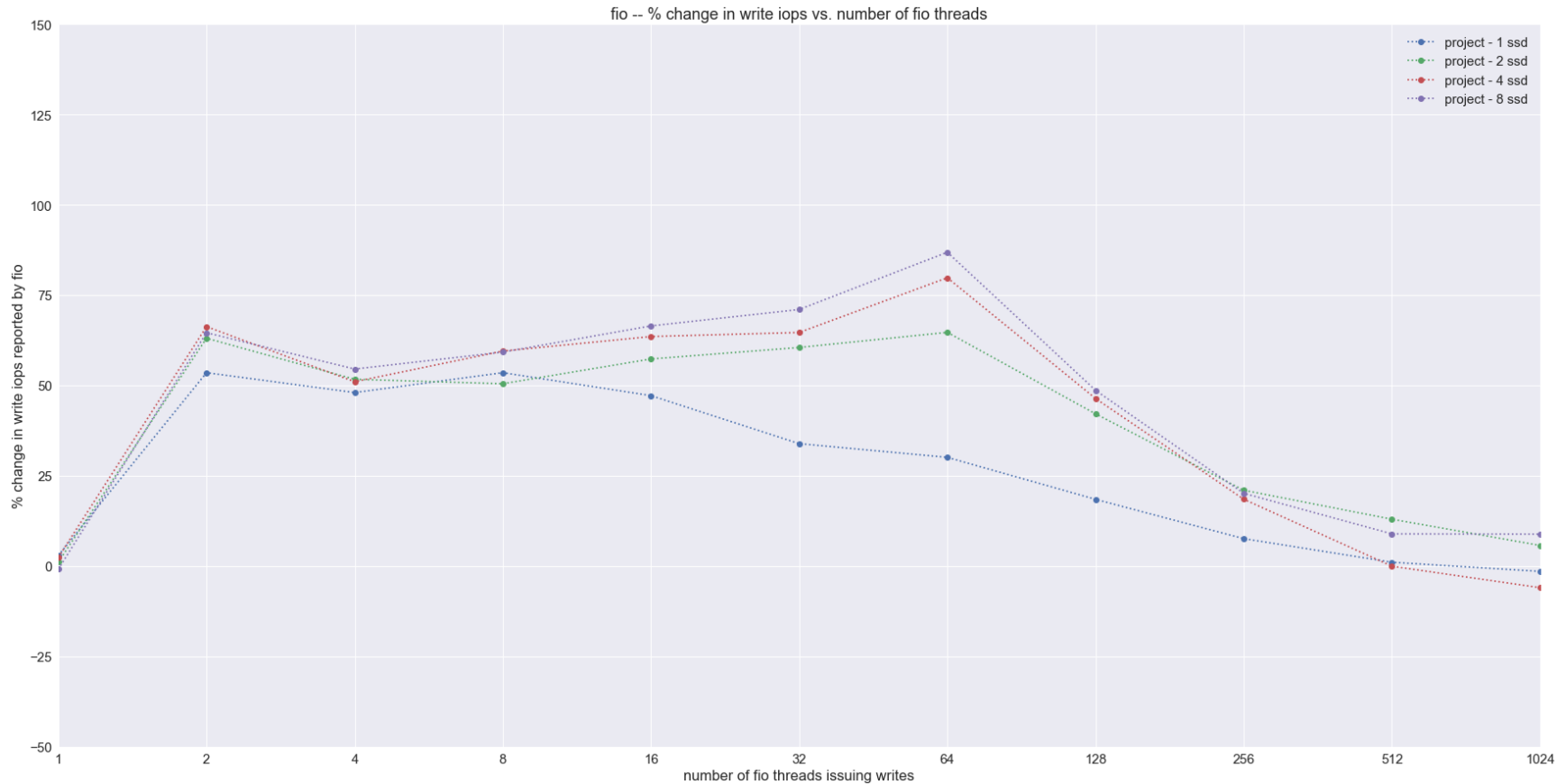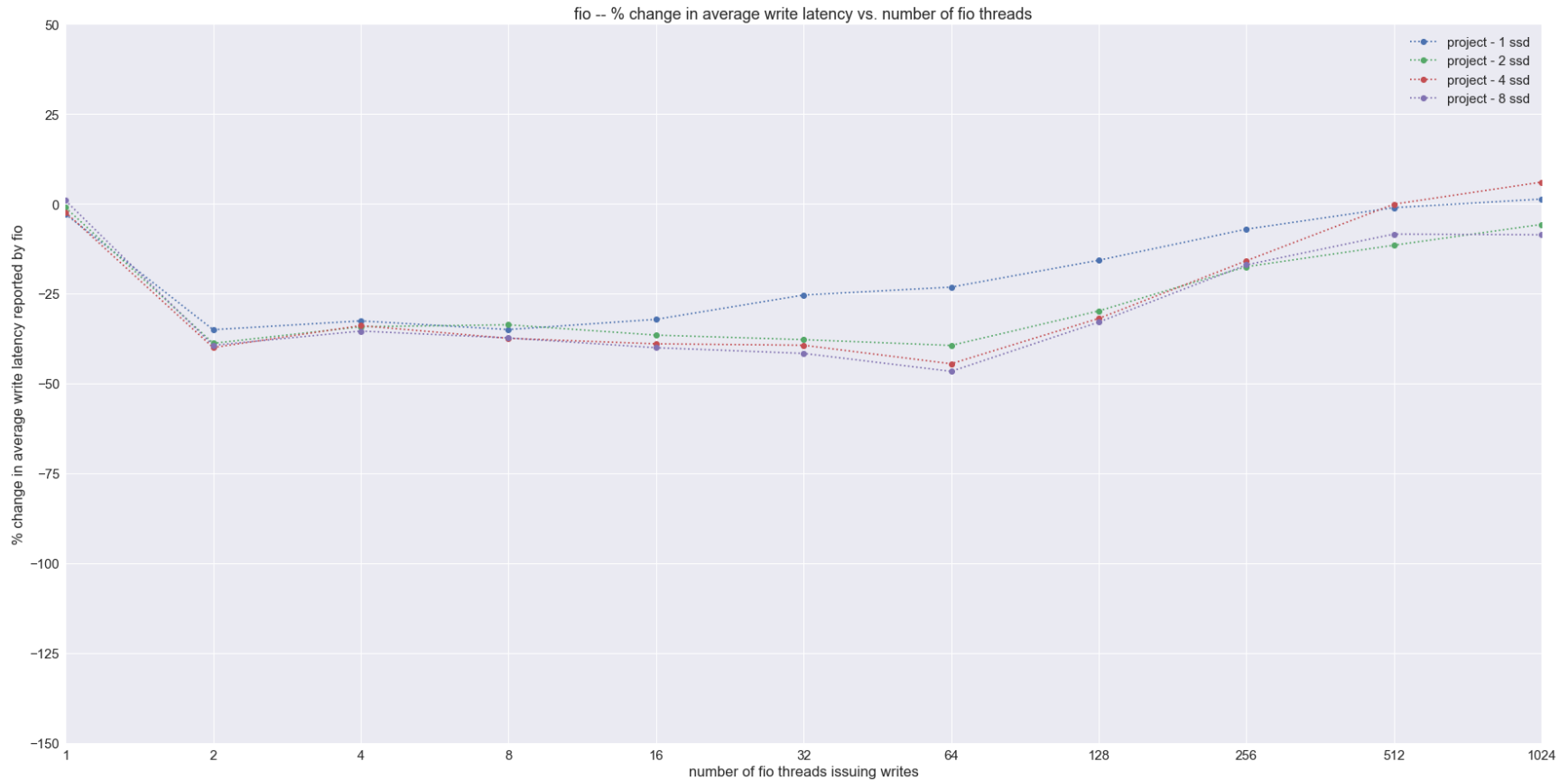
5 – Fixed Rate Workload – HDDs

% Change IOPs – Fixed Rate – HDDs

fio -- % change in write iops vs. number of fio threads

% Change Latency – Fixed Rate – HDDs

fio -- % change in average write latency vs. number of fio threads

Legend:
- project - 1 hdd
- project - 2 hdd
- project - 4 hdd
- project - 8 hdd

y-axis: % change in average write latency reported by fio
x-axis: number of fio threads issuing writes
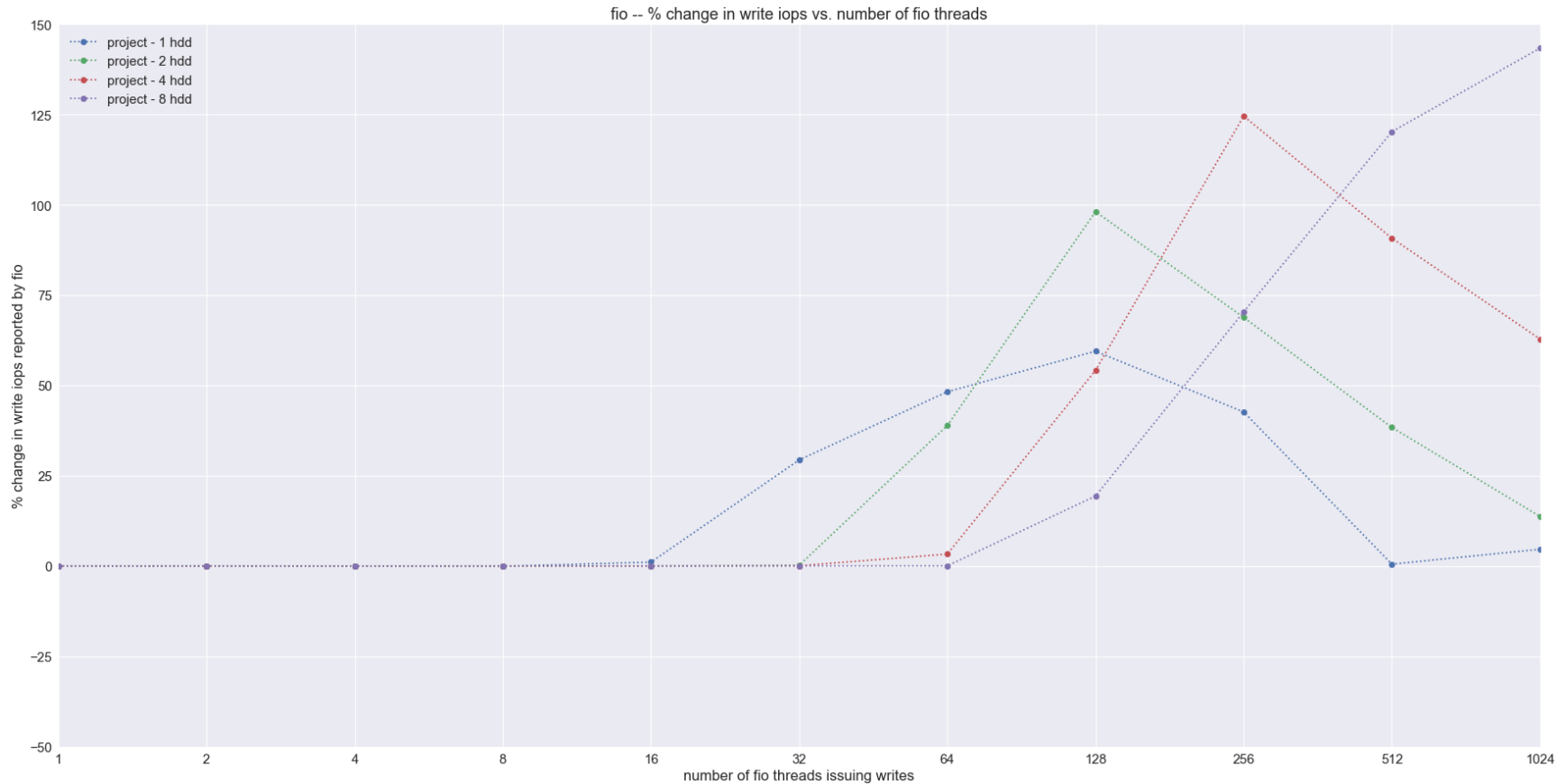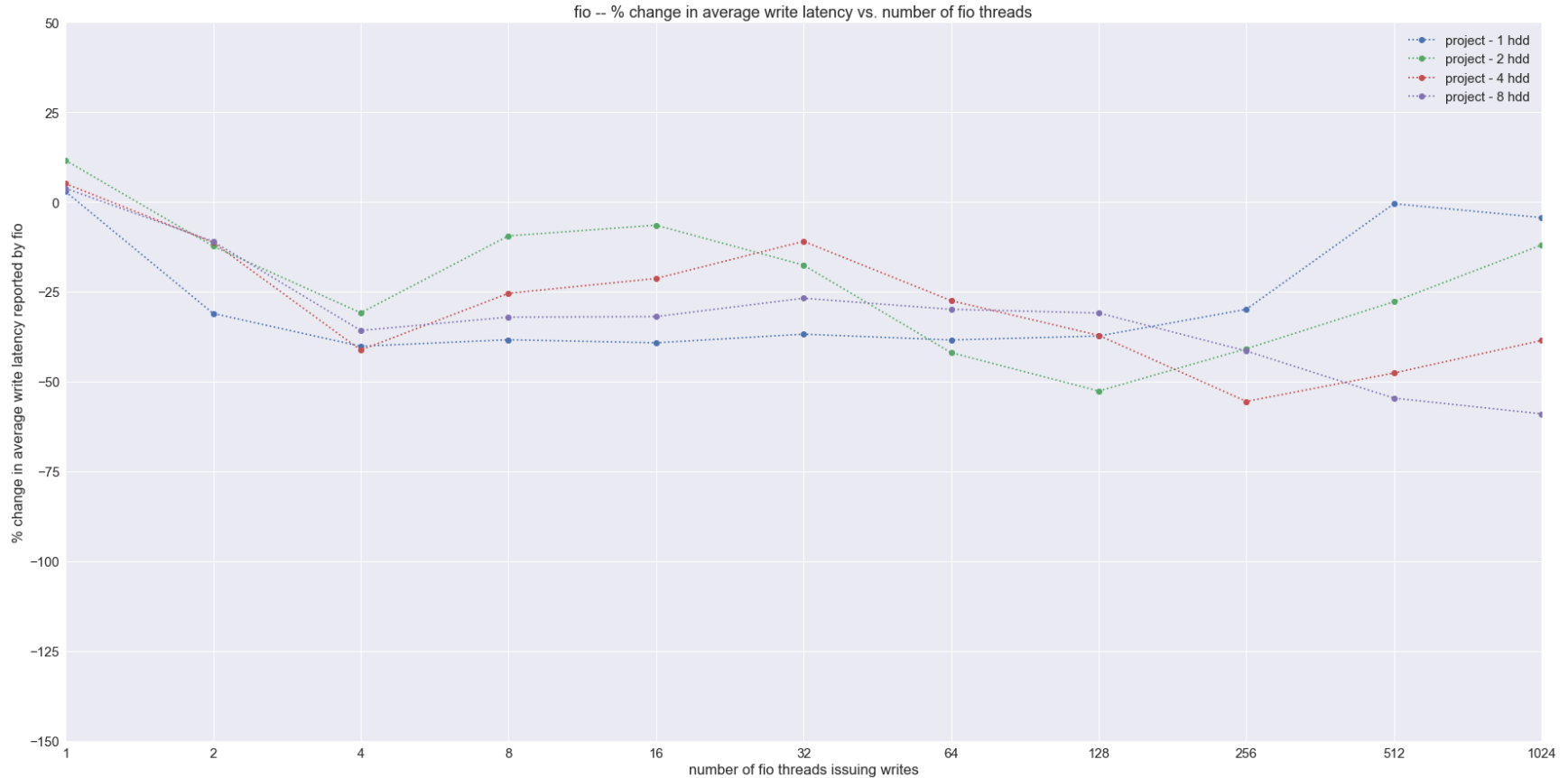
5 – Fixed Rate Workload – SSDs

% Change IOPs – Fixed Rate – SSDs

fio -- % change in write iops vs. number of fio threads

% Change Latency – Fixed Rate – SSDs

fio -- % change in average write latency vs. number of fio threads

End