ZIL Performance: How I Doubled Sync Write Speed

Prakash Surya | October 24 2017
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 operations get logged?
    - `zfs_create`, `zfs_remove`, `zfs_write`, etc.
    - Doesn't include non-modifying ZPL operations:
      - `zfs_read`, `zfs_seek`, etc.
  - What gets logged?
    - The fact that a logical operation is occurring is logged
      - `zfs_remove` → directory object ID + name only
      - Not logging which blocks will change due to logical operation
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**

What is the SLOG?

• SLOG: Acronym for (S)eperate (LOG) Device
• Conceptually, SLOG is different than the ZIL
  ◦ ZIL is mechanism for writing, SLOG is device written to
• An SLOG is not necessary
  ◦ By default (no SLOG), ZIL will write to main pool VDEVs
• An SLOG can be used to improve latency of ZIL writes
  ◦ When attached, ZIL writes to SLOG instead of main pool*

*For some operations; see code for details.
Why does the ZIL exist?

- Writes in ZFS are "write-back"
  - Data is first written and stored in-memory, in DMU layer
  - Later, data for whole pool written to disk via spa_sync()
- Without the ZIL, sync operations could wait for spa_sync()
  - spa_sync() can take tens of seconds (or more) to complete
- Further, with the ZIL, write amplification can be mitigated
  - A single ZPL operation can cause many writes to occur
  - ZIL allows operation to "complete" with minimal data written
- ZIL needed to provide "fast" synchronous semantics to applications
  - Correctness could be achieved without it, but would be "too slow"
ZIL On-Disk Format

- Each dataset has its 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`
     - Tells the ZIL an operation is occurring
  2. Commit the operation(s) — `zil_commit`
     - Causes the ZIL to write log record of operation to disk
Example: zfs_write

- zfs_write $\rightarrow$ zfs_log_write
- zfs_log_write
  $\rightarrow$ zil_itx_create
  $\rightarrow$ zil_itx_assign
- zfs_write $\rightarrow$ zil_commit
Example: zfs_fsync

- fsync → zil_commit
  - fsync doesn't create any new modifications
  - only writes previous itx's to disk
    - thus, no zfs_log_fsync function
Contract between ZIL and ZPL.

- Parameters to \texttt{zil\_commit}: ZIL pointer, object number
  - These uniquely identify an object whose data is to be committed
- When \texttt{zil\_commit} returns:
  - Operations \textit{relevant} to the object specified, will be \textit{persistent} on disk
    - relevant – all operations that would modify that object
    - persistent – Log block(s) written (completed) \rightarrow disk flushed
- Interface of \texttt{zil\_commit} doesn't specify \textit{which} operation(s) to commit
2 – How does the ZIL work?
How does the ZIL work?

- In memory ZIL contains an \textit{itxg\_t} structure*  
- Each \textit{itxg\_t} contains:
  - A single list of sync operations (for all objects)
  - Object specific lists of async operations

*Actually multiple \textit{itxg\_t} structures, one per-\textit{txg}.
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 relevant itx's, move them to the "commit list"
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: \texttt{zil\_commit} Object B

\begin{itemize}
  \item sync list \rightarrow itx S1 \rightarrow itx S2
  \item object A list \rightarrow itx A1 \rightarrow itx A2
  \item object B list \rightarrow itx B1
  \item commit list
\end{itemize}
Example: zil_commit Object B

- sync list
  - itx S1
  - itx S2
  - itx B1

- object A list
  - itx A1
  - itx A2

- object B list

- commit list
Example: zil_commit Object B

- sync list
- object A list
- object B list
- commit list

- itx S1
- itx S2
- itx B1
- itx A1
- itx A2
Example: zil_commit Object B

- sync list
- object A list → itx A1 → itx A2
- object B list
- commit list → itx S1 → itx S2 → itx B1
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 committed, 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

ZIL header

lwb 1
```
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
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 committed, 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
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 committed, 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
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 committed, 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
  5. Notify waiting threads
3 – Problem
Problem

1. i tx'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
Problem

- Time spent servicing lwb's for each disk
- Color indicates order waiting threads notified
3 – Solution
Solution

- Remove concept of "batches":
  1. Allow `zil_commit` to issue new ZIL block writes immediately
    - In contrast to waiting for the current batch to complete
  2. Notify threads immediately when `dependent` `lwb's on disk`
    - In contrast to waiting for all `lwb's on disk`
Problem

- Time spent servicing lwb's for each disk
- Color indicates order waiting threads notified
Solution

- Time spent servicing lwb's for each disk
- Color indicates order waiting threads notified
4 – Details on the Changes I Made
Before

Step 1

batch root

lwb 1

lwb 2

lwb 3

Step 2

flush root

VDEV 1

VDEV 2

Step 3

CV

48 / 73
Before

Step 1
- lwb 1
- lwb 2
- lwb 3

Step 2
- flush root
- VDEV 1
- VDEV 2

Step 3
- CV

Before

49 / 73
Before

Step 1

Step 2

Step 3
Before

Step 1

batch root

lwb 1

lwb 2

lwb 3

Step 2

flush root

VDEV 1

VDEV 2

Step 3

CV
Before

Step 1
- batch root
  - lwb 1
  - lwb 2
  - lwb 3

Step 2
- flush root
  - VDEV 1
  - VDEV 2

Step 3
- CV
After

lwb 3

VDEV 1 flush

lwb 2

VDEV 2 flush

lwb 1

VDEV 1 flush

lwb 3 write

lwb 2 write

CV

...
After

```plaintext
lwb 3
lwb 2
lwb 1

VDEV 1 flush
VDEV 2 flush

lwb 3 write
lwb 2 write
lwb 1 write

CV
... -> CV
... -> CV
... -> CV

CV
```
After

- CV
  - lwb 3
    - lwb 3 write
    - VDEV 1 flush
  - CV
- CV
  - lwb 2
    - lwb 2 write
    - VDEV 2 flush
  - CV
- CV
  - lwb 1
    - lwb 1 write
    - VDEV 1 flush
After

lwb 3

lwb 3 write  VDEV 1 flush

lwb 2

lwb 2 write  VDEV 2 flush

lwb 1

lwb 1 write  VDEV 1 flush
After
After

lwb 3 write  VDEV 1 flush

lwb 2

lwb 2 write  VDEV 2 flush

lwb 1

lwb 1 write  VDEV 1 flush
After

lwb 3

VDEV 1 flush

lwb 3 write

VDEV 1 flush

lwb 2 write

VDEV 2 flush

lwb 1 write

VDEV 1 flush

CV
After

lwb 3

lwb 3 write  VDEV 1 flush

lwb 2

lwb 2 write  VDEV 2 flush

lwb 1

lwb 1 write  VDEV 1 flush

CV

CV

CV

CV
After

lwb 3

VDEV 1 flush

lwb 2

VDEV 2 flush

CV

lwb 1

lwb 1 write

VDEV 1 flush

lwb 2 write

CV
After

After
New Tunable: lwb Timeout
5 – Performance testing and results
~83% Increase in IOPs on Average – Max Rate – 8 HDDs

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

% change in write iops reported by fio

number of fio threads issuing writes

8 hdd
~48% Increase in IOPs on Average – Max Rate – 8 SSDs
~27% Decrease in Latency on Average – Fixed Rate – 8 HDDs

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

*IOPs increased with new code, and >64 threads; those data points omitted.
~16% Decrease in Latency on Average – Fixed Rate – 8 SSDs

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

% change in average write latency reported by fio

number of fio threads issuing writes
More Details

- Two fio workloads were used:
  1. each thread submitting sync writes as fast as it could
  2. each thread submitting 64 sync writes per second
- 1, 2, 4, and 8 disk zpools; both SSD and HDD
- fio threads ranging from 1 to 1024; increasing in powers of 2
- Full details can be found [here](#)
End