An Efficient Backup and Replication of Storage

Takashi HOSHINO
Cybozu Labs, Inc.
2013-05-29@LinuxConJapan
Self-introduction

• Takashi HOSHINO
  – Works at Cybozu Labs, Inc.

• Technical interests
  – Database, storage, distributed algorithms

• Current work
  – WalB: Today’s talk!
Contents

• Motivation
• Architecture
• Alternative solutions
• Algorithm
• Performance evaluation
• Summary
Contents

• Motivation
• Architecture
• Alternative solutions
• Algorithm
• Performance evaluation
• Summary
Motivation

• **Backup and replication are vital**
  – We have our own cloud infrastructure
  – Requiring high availability of customers’ data
  – With cost-effective commodity hardware and software
Requirements

- **Functionality**
  - Getting consistent diff data for backup/replication achieving small RPO

- **Performance**
  - Without usual full-scans
  - Small overhead

- **Various kinds of data support**
  - Databases
  - Blob data
  - Full-text-search indexes
  - ...
A solution: WalB

• A Linux kernel device driver to provide wrapper block devices and related userland tools
• Provides consistent diff extraction functionality for efficient backup and replication
• “WalB” means “Block-level WAL”
WAL (Write-Ahead Logging)

- **Write at 0**
  - Ordinary Storage: [Blue bars]
  - WAL Storage: [Blue bar]

- **Write at 2**
  - Ordinary Storage: [Blue and purple bars]
  - WAL Storage: [Blue and two purple bars]

- **Read at 2**
  - Ordinary Storage: [Blue and purple bars]
  - WAL Storage: [Blue and two purple bars]

- **Write at 2**
  - Ordinary Storage: [Blue and green bars]
  - WAL Storage: [Blue and two purple bars]
How to get diffs

• Full-scan and compare

Data at t0: a b c d e
Data at t1: a b' c d e'
Diffs from t0 to t1: 1 b' 4 e'

• Partial-scan with bitmaps (or indexes)

Data at t0: a b c d e
          00000
Data at t1: a b' c d e'
          01001
Diffs from t0 to t1: 1 b' 4 e'

• Scan logs directly from WAL storages

Data at t0: a b c d e
Data at t1: a b' c d e'
Diffs from t0 to t1: 1 b' 4 e'
Contents

• Motivation
• Architecture
• Alternative solutions
• Algorithm
• Performance evaluation
• Summary
WalB architecture

Walb dev controller

A walb device as a wrapper

Any application (File system, DBMS, etc)

Control

Read

Write

Walb log Extractor

A block device for data (Data device)

Not special format

A block device for logs (Log device)

An original format
Log device format

Log address = (log sequence id) % (ring buffer size) + (ring buffer start address)
Ring buffer inside

The latest logpack

The oldest logpack

Ring buffer

Log pack

Logpack header block

1st written data

2nd written data

...

Log pack header block

Checksum

Logpack lsid

Num of records

Total IO size

1st log record

IO address

IO size

...

2nd log record

IO address

IO size

...

...
Redo/undo logs

- **Redo logs**
  - to go forward
- **Undo logs**
  - to go backward
How to create redo/undo logs

- Undo logs requires additional read IOs
- Walb devices do not generate undo-logs
Consistent full backup

- The ring buffer must have enough capacity to store the logs generated from t0 to t1
Consistent incremental backup

- To manage multiple backup generations, defer application or generate undo-logs during application
Backup and replication

Primary host
- Walb device (online)
- Log device

Backup host
- Full archive
- Logs
  - Apply (A')

Remote host
- Full archive
- Logs
  - Apply (B')

Get logs from t0 to t1
- Backuped at t1
- Replicated at t3

Replication delay
- Time
  - t0
  - t1
  - t2
  - t3
Contents

• Motivation
• Architecture
• Alternative solutions
• Algorithm
• Performance evaluation
• Summary
Alternative solutions

• DRBD
  – Specialized for replication
  – DRBD proxy is required for long-distance replication

• Dm-snap
  – Snapshot management using COW (copy-on-write)
  – Full-scans are required to get diffs

• Dm-thin
  – Snapshot management using COW and reference counters
  – Fragmentation is inevitable
## Alternatives comparison

<table>
<thead>
<tr>
<th></th>
<th>Capability</th>
<th>Performance</th>
<th>Fragmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Incr. backup</td>
<td>Sync replication</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Async replication</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read response overhead</td>
<td>Write response overhead</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WALB</td>
<td>✓</td>
<td>✓</td>
<td>Negligible</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Write log</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>instead data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Never</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRBD</td>
<td>✓</td>
<td>✓</td>
<td>Negligible</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Send IOs to</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>slaves (async</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>repl.)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Never</td>
</tr>
<tr>
<td>dm-snap</td>
<td>✓</td>
<td>✓</td>
<td>Search idx</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Modify idx</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(+COW)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Never</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(original lv)</td>
</tr>
<tr>
<td>dm-thin</td>
<td>✓</td>
<td>✓</td>
<td>Search idx</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Modify idx</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(+COW)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Inevitable</td>
</tr>
</tbody>
</table>
WalB pros and cons

• Pros
  – Small response overhead
  – Fragmentation never occur
  – Logs can be retrieved with sequential scans

• Cons
  – 2x bandwidth is required for writes
WalB source code statistics

SLOC of WalB and Linux components in kernel 3.9

- Block device wrappers
- File systems

- WalB
- device-mapper
- DRBD
- ext3
- ext4
- btrfs
Contents

• Motivation
• Architecture
• Alternative solutions
• Algorithm
• Performance evaluation
• Summary
Requirements to be a block device

• Read/write consistency
  – It must read the latest written data

• Storage state uniqueness
  – It must replay logs not changing the history

• Durability of flushed data
  – It must make flushed write IOs be persistent

• Crash recovery without undo
  – It must recover the data using redo-logs only
WalB algorithm

• **Two IO processing methods**
  – Easy: very simple, large overhead
  – Fast: a bit complex, small overhead

• **Overlapped IO serialization**

• **Flush/fua for durability**

• **Crash recovery and checkpointing**
IO processing flow (easy algorithm)

**Write**
- Submitted
- Packed
- Log submitted
- Log completed
- Data submitted
- Data completed
- WalB write IO response
- Wait for log flushed and overlapped IOs done

**Read**
- Submitted
- Completed
- Data submitted
- Data completed
- Data IO response
IO processing flow (fast algorithm)

Write

Submitted

Packed

WalB write IO response

Log submitted

Log IO response

Log completed

Completed

Wait for log flushed and overlapped IOs done

Data submitted

Data IO response

Data completed

Pdata inserted

Read

Submitted

Pdata copied

(Data submitted)

(Data completed)

Completed
Pending data

- A red-black tree
  - provided as a kernel library
- Sorted by IO address
  - to find overlapped IOs quickly
- A spinlock
  - for exclusive accesses
Overlapped IO serialization

- Required for storage state uniqueness
- Oldata (overlapped data)
  - similar to pdata
  - counter for each IO
  - FIFO constraint
### Flush/fua for durability

<table>
<thead>
<tr>
<th></th>
<th>Property to guarantee</th>
<th>What WalB need to do</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>REQ_FLUSH</strong></td>
<td>All write IOs submitted before the flush IO are persistent</td>
<td>Set FLUSH flag of the corresponding log IOs</td>
</tr>
<tr>
<td><strong>REQ_FUA</strong></td>
<td>The fua IO is persistent</td>
<td>Set FLUSH and FUA flags to the corresponding log IOs</td>
</tr>
</tbody>
</table>

- The condition for a log to be persistent
  - All logs before the log and itself are persistent
- Neither FLUSH nor FUA is required for data device IOs
  - Data device persistence will be guaranteed by checkpointing
Crash recovery and checkpointing

• Crash recovery
  – Crash will make recent write IOs not be persistent in the data device
  – Generate write IOs from recent logs and execute them in the data device

• Checkpointing
  – Sync the data device and update the superblock periodically
  – The superblock contains recent logs information
Contents

• Motivation
• Architecture
• Alternative solutions
• Algorithm
• Performance evaluation
• Summary
Experimental environment

- **Host**
  - CPU: Intel core i7 3930K
  - Memory: DDR3-10600 32GB (8GBx4)
  - OS: Ubuntu 12.04 x86_64
  - Kernel: Custom build 3.2.24

- **Storage HBA**
  - Intel X79 Internal SATA controller
  - Using SATA2 interfaces for the experiment
**Benchmark software**

- **Self-developed IO benchmark for block devices**
  - [https://github.com/starpos/ioreth](https://github.com/starpos/ioreth)
  - uses DIRECT_IO to eliminate buffer cache effects

- **Parameters**
  - Pattern: Random/sequential
  - Mode: Read/write
  - Block size: 512B, 4KB, 32KB, 256KB
  - Concurrency: 1-32
Target storage devices

- **MEM**
  - Memory block devices (self-implemented)

- **HDD**
  - Seagate Barracuda 500GB (ST500DM002)
  - Up to 140MB/s

- **SSD**
  - Intel 330 60GB (SSDSC2CT060A3)
  - Up to 250MB/s with SATA2
Storage settings

• **Baseline**
  – A raw storage device

• **Wrap**
  – Self-developed simple wrapper
  – Request/bio interfaces

• **WalB**
  – Easy/easy-ol/fast/fast-ol
  – Request/bio interfaces
WalB parameters

- **Log-flush-disabled experiments**
  - Target storage: MEMs/HDDs/SSDs
  - Pdata size: 4-128MiB

- **Log-flush-enabled experiments**
  - Target storage: HDDs/SSDs
  - Pdata size: 4-64MiB
  - Flush interval size: 4-32MiB
  - Flush interval period: 10ms, 100ms, 1s
MEM 512B random: response

• Smaller overhead with bio interface than with request interface

• Serializing write IOs of WalB seems to enlarge response time as number of threads increases
**MEM 512B random: IOPS**

**Read**

- Large overhead with request interface
- Pdata search seems overhead of walb-bio

**Write**

- IOPS of walb decreases as num of threads increases due to decrease of cache-hit ratio or increase of spinlock wait time
Pdata and oldata overhead

MEM 512B random write

IOPS with pattern random, mode write, blocksize 512

Overhead of pdata

Overhead of oldata

Queue length

IOPS

Kernel 3.2.24, walb req prototype
HDD 4KB random: IOPS

- Negligible overhead
- IO scheduling effect was observed especially with walb-req
HDD 256KB sequential: Bps

- Request interface is better
- IO size is limit to 32KiB with bio interface

- Additional log header blocks decrease throughput
SSD 4KB random: IOPS

Read

- WalB performance is almost the same as that of wrap req

Write

- Fast algo. is better than easy algo.
- IOPS overhead is large with smaller number of threads
SSD 4KB random: IOPS (two partitions in a SSD)

- Almost the same result as with two SSD drives
- A half throughput was observed
- Bandwidth of a SSD is the bottleneck
SSD 256KB sequential: Bps

- Non-negligible overhead was observed with queue length 1
- Fast is better than easy
- Larger overhead of walb with smaller queue length
Log-flush effects with HDDs

IOPS (4KB random write)

- IO sorting for the data device is effective for random writes

Bps (256KB sequential write)

- Enough memory for pdata is required to minimize the log-flush overhead
Log-flush effects with SSDs

- Small flush interval is better
- Log flush increases IOPS with single-thread

- Log-flush effect is negligible
Evaluation summary

• WalB overhead
  – Non-negligible for writes with small concurrency
  – Log-flush overhead is large with HDDs, negligible with SSDs

• Request vs bio interface
  – bio is better except for workloads with large IO size

• Easy vs fast algorithm
  – Fast is better
Contents

• Motivation
• Architecture
• Alternative solutions
• Algorithm
• Performance evaluation
• Summary
WalB summary

• A wrapper block device driver for
  – incremental backup
  – asynchronous replication

• Small performance overhead with
  – No persistent indexes
  – No undo-logs
  – No fragmentation
Current status

• **Version 1.0**
  – For Linux kernel 3.2+ and x86_64 architecture
  – Userland tools are minimal

• **Improve userland tools**
  – Faster extraction/application of logs
  – Logical/physical compression
  – Backup/replication managers

• **Submit kernel patches**
Future work

• Add all-zero flag to the log record format
  – to avoid all-zero blocks storing to the log device

• Add bitmap management
  – to avoid full-scans in ring buffer overflow

• (Support snapshot access)
  – by implementing pluggable persistent address indexes

• (Support thin provisioning)
  – if a clever defragmentation algorithm was available
Thank you for your attention!

• GitHub repository:
  – https://github.com/starpos/walb/

• Contact to me:
  – Email: hoshino AT labs.cybozu.co.jp
  – Twitter: @starpoz (hashtag: #walbdev)