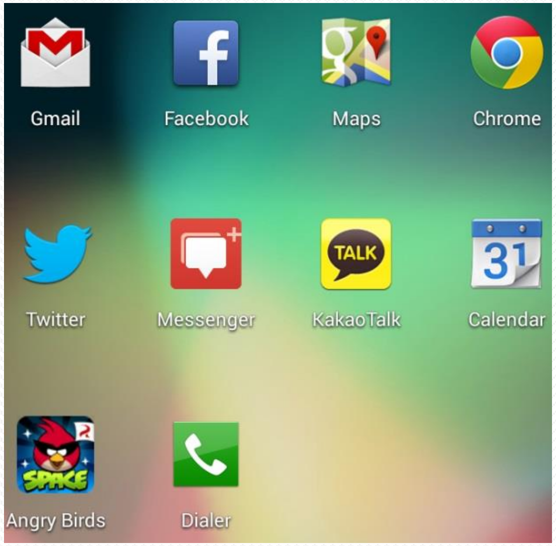
X-FTL: Transactional FTL for SQLite Databases

Sang-Won Lee
Info & Communication Engineering
Sungkyunkwan University

Bongki Moon
Computer Science & Engineering
Seoul National University

SQLite

- SQLite is the standard database for smartphones
  - Google Android, Apple iOS
  - Almost every apps uses **SQLite**
- Why SQLite?
  - Development productivity
  - Solid transactional support
  - Lightweight runtime footprint
- SQLite takes a simpler but costlier journaling approach to transactional atomicity



SQLite Journaling

- Two journaling modes in SQLite
  - Rollback journal mode (**RBJ**)
  - Write ahead logging (**WAL**) (**≠ Aries-style physiological WAL)**
- SQLite journaling mode is the main cause of slow performance in smartphone applications
  - Kim [USENIX FAST12], Lee [ACM EMSOFT 12]
  - **70%** of all write requests are from SQLite and mostly random
- eMMC flash card is the default storage in smartphones

- SQLite optimization is the **practical and critical problem**

- We propose a transactional FTL for SQLite, **X-FTL**

X-FTL: Overview

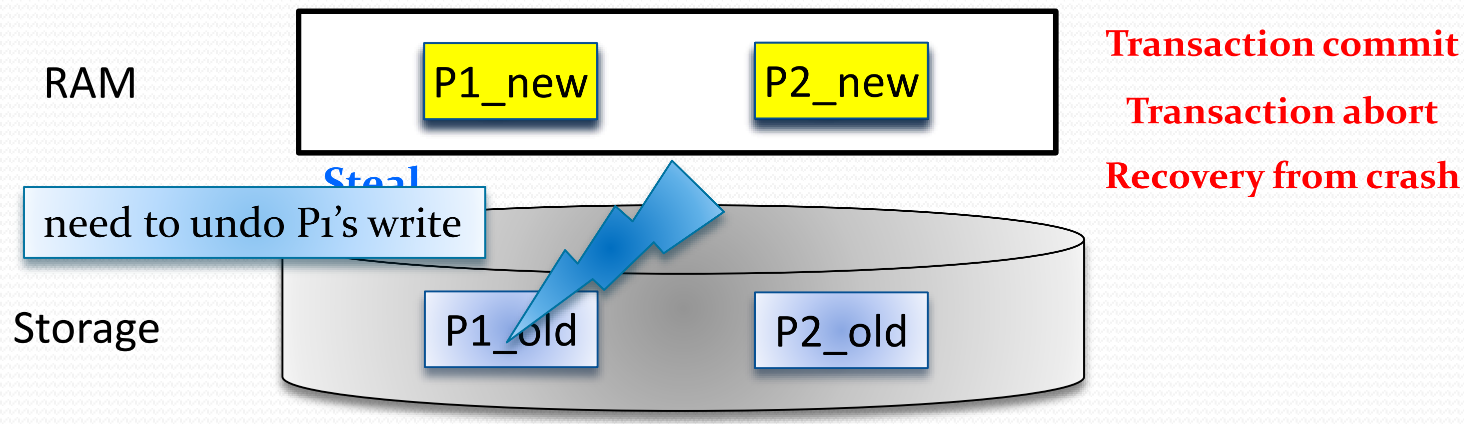
- Identify a performance problem in SQLite and its causes
- Develop new solution for flash-aware atomic propagation
  - Implement X-FTL using OpenSSD platform
  - Extend the storage interface for transactional atomicity
  - Demonstrate SQLite and ext4 file system can benefit from X-FTL with only minimal changes in their code
- Show that 2x speedup can be achieved in SQLite

Transactional Atomicity in SQLite

- A transaction updates one or more pages
  - {P1, …, Pn}
- **Steal** and **force** policies are taken in SQLite
  - Uncommitted changes can be propagated
  - Atomic write of multiple pages may not be enough
- Atomic propagation of updated page(s) by TXs is crucial for commit, abort, and recovery in SQLite

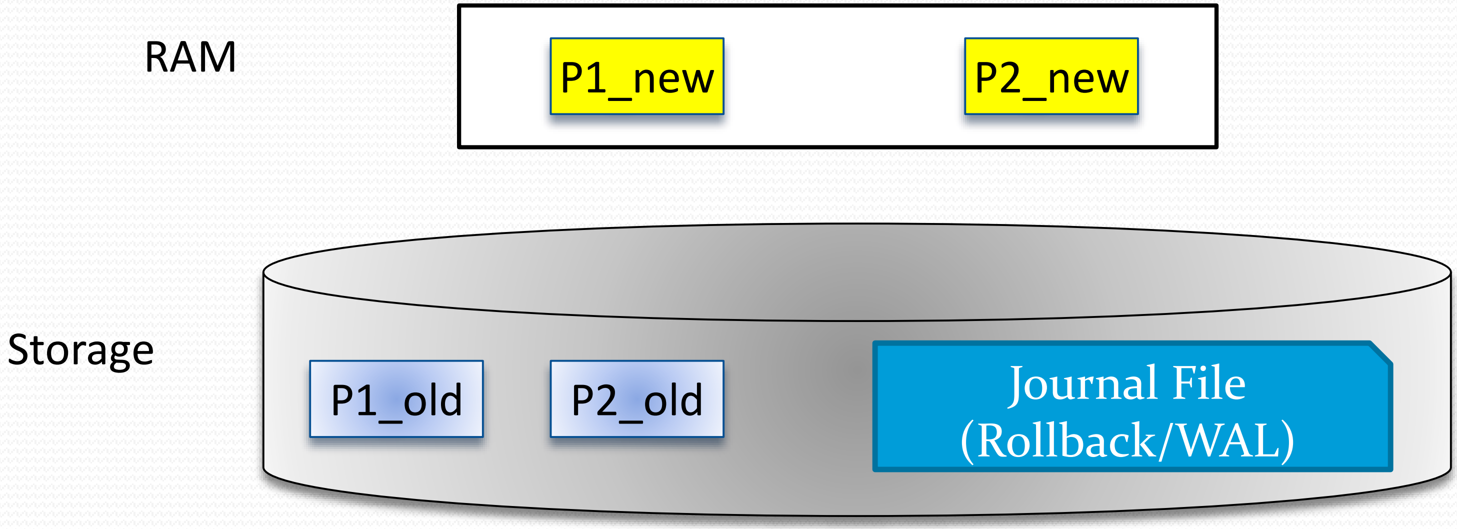
Ex) Two pages (P1,P2) are updated

- Transactional atomicity is all or nothing
  - Force policy need write both pages at commit (**ALL**)
  - Steal policy allows overwriting P1 prior to commit, so undoing P1's write may be necessary upon abort (**NOTHING**)
  - Recovery from crash checks whether both pages are successfully written, and if not, need to undo (**ALL or NOTHING**)



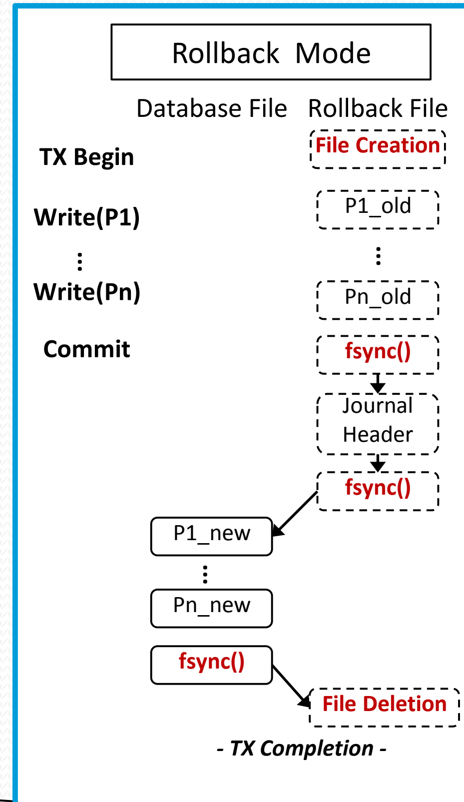
Journaling in SQLite

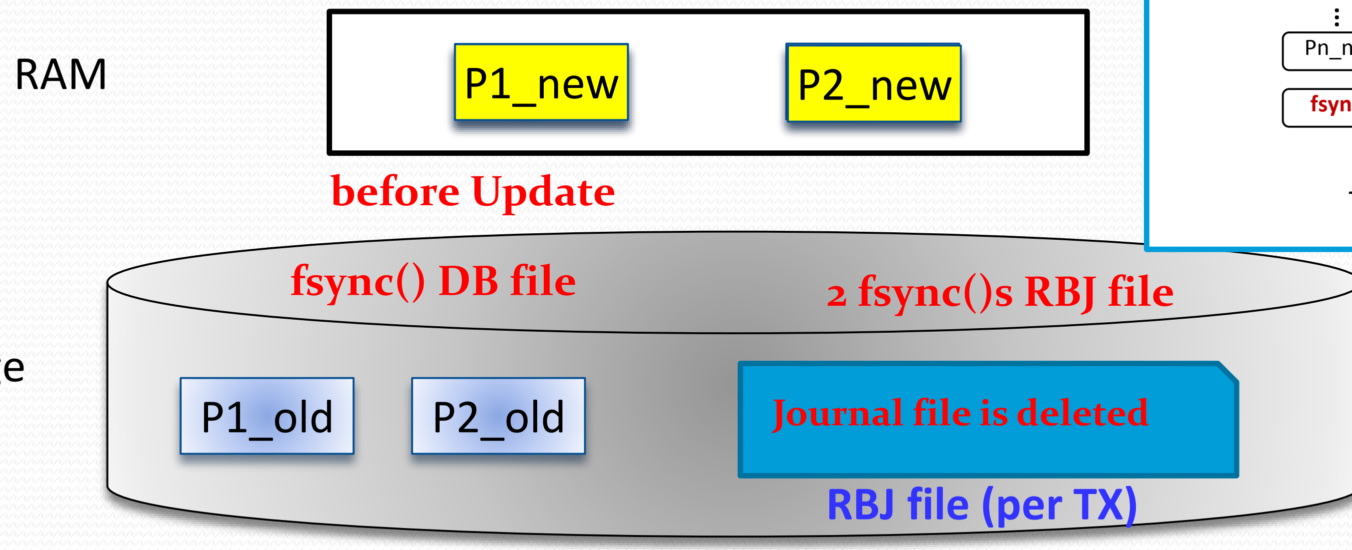
- Two journal modes
  - Rollback journal (RBJ, default) and Write Ahead Logging (WAL)
- Why SQLite's own journaling modes, instead of file system journaling?
  - Portability : every file system does not support journaling
  - Steal policy semantics



Rollback Journal

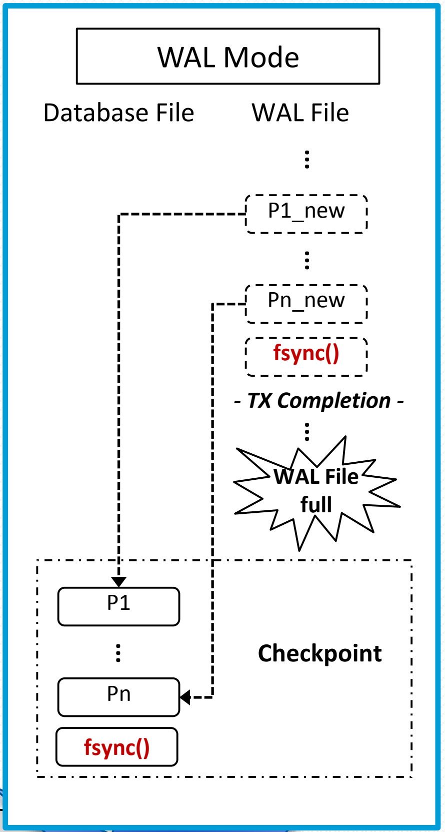
- Rollback Journal (RBJ)
  - **Old page images are backed up in RBJ file for undo**
  - RBJ file is deleted at commit time
    - Transaction commit is regarded as success only after RBJ file is deleted
- Run-time overhead
  - RBJ file creation/deletion
  - 3 fsync() operations per transaction
  - Two writes per each update page
  - A logical update can cause
    - 22 physical page writes [Lee and Won 12]

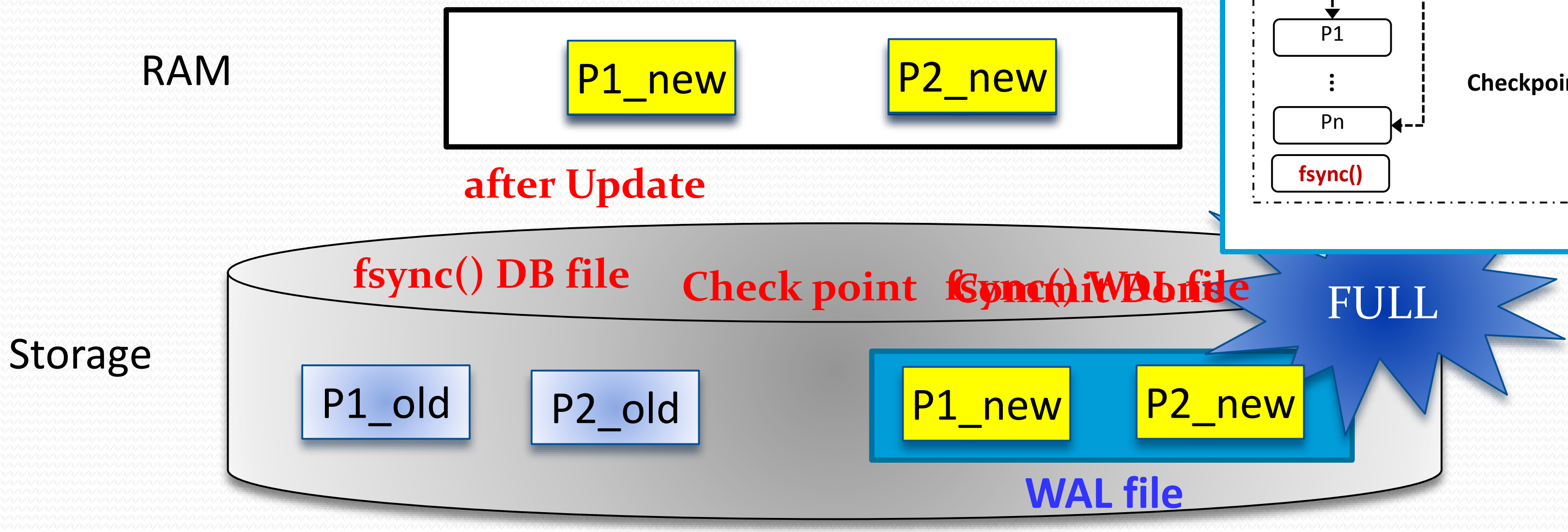




Write Ahead Logging

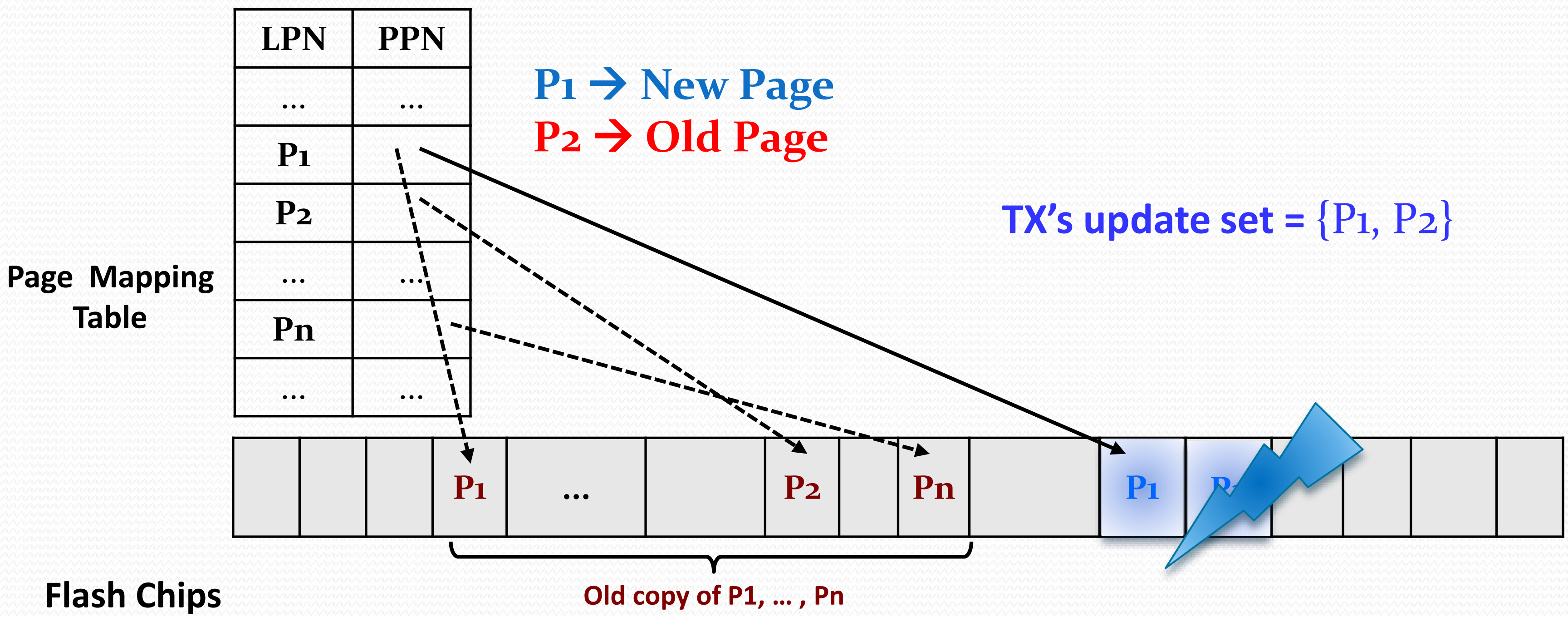
- Write Ahead Logging (WAL)
  - Recently introduced, and better performance than RBJ
  - WAL file is reused and shared by many transactions
  - **New page images are appended to WAL file for redo**
  - Check-point when it becomes full
  - No file creation/deletion overhead
  - less frequent fsync()
  - But, 2X writes per each updated page





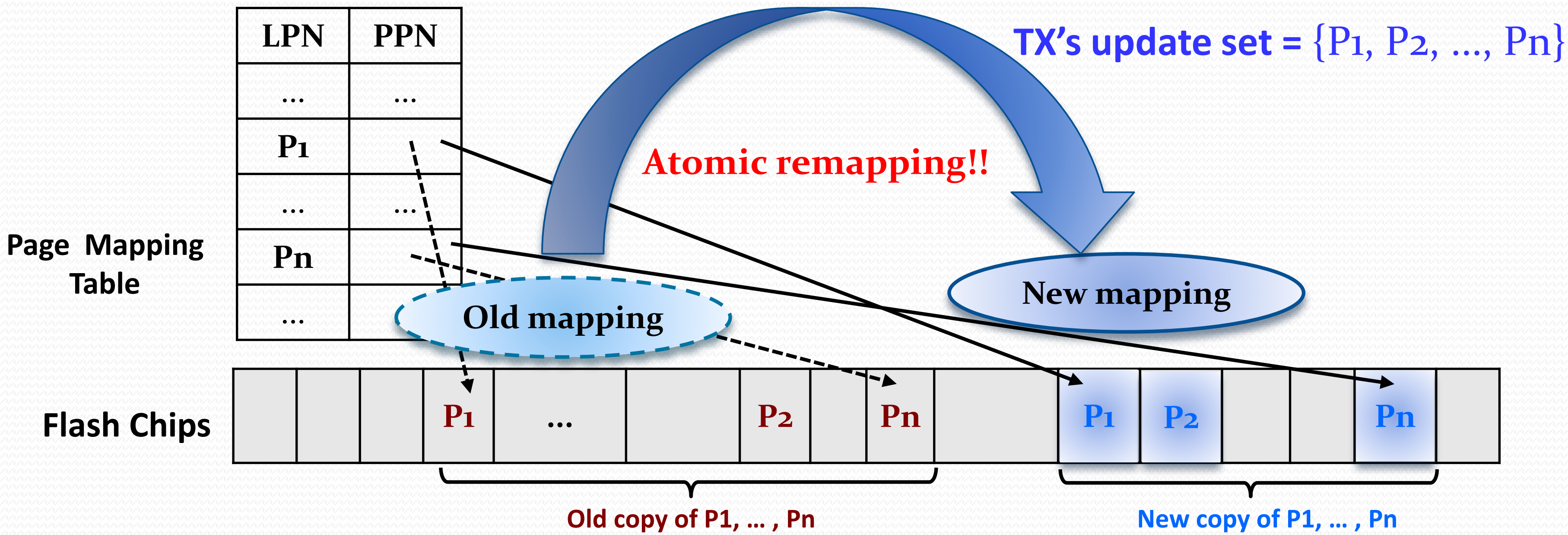
Flash Copy-on-Write

- In-place update is not allowed in flash memory
- FTLs take Copy-on-Write (CoW) strategy
  - **Both old and new copy** of a page co-exist
- But, current FTLs change L2P address mapping at the granularity of page, not a set of pages
  - Can not support atomic propagation of multiple pages



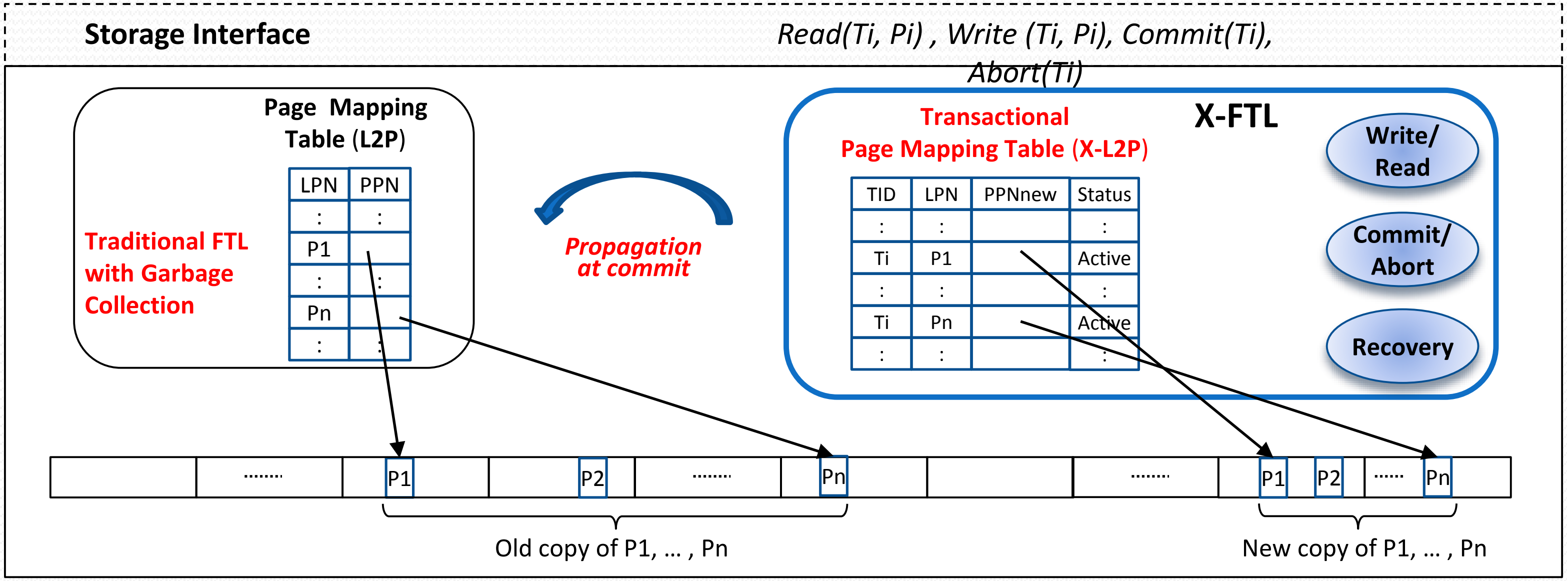
CoW and Shadow Paging

- CoW strategy provides an opportunity for transactional atomicity
- What if FTL can support atomic remapping of multiple page updates by a transaction?
  - FTL need to provide ***transactional interface*** to the upper layer
  - For undo, old pages should be ***exempt from GC*** until TX commit



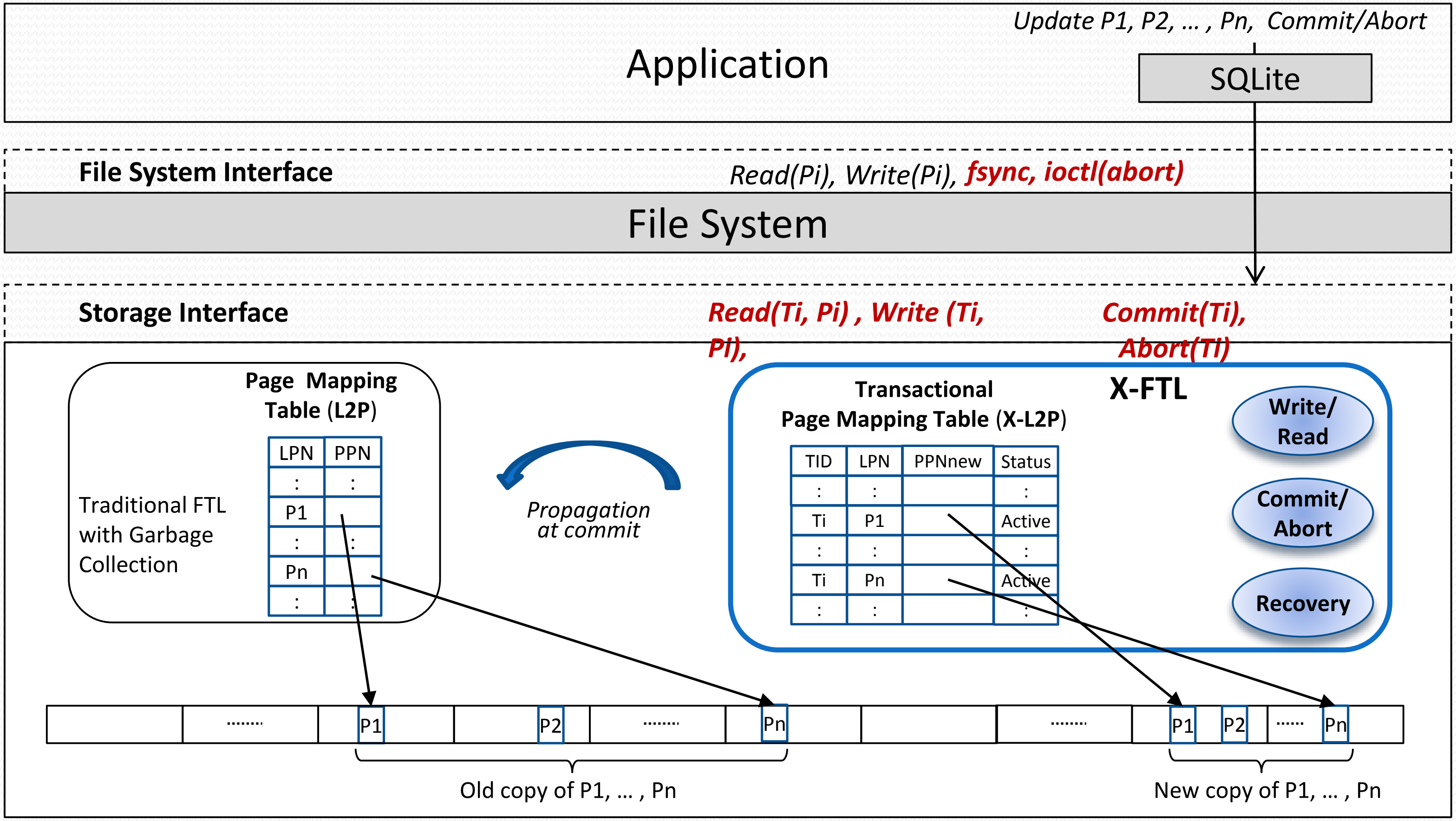
X-FTL: Architecture

- Transactional mapping table : **X-L2P** table
  - Page mapping table : L2P table (original FTL)
  - Transaction ID, Logical Page No, Physical Page No(new), Status
- Garbage collection
  - Prevent **active transaction** pages from GC
  - Only pages invalidated by committed transactions
- Atomic propagation of mapping information at commit
  - Atomic remapping of committed entries in **X-L2P table to L2P table**

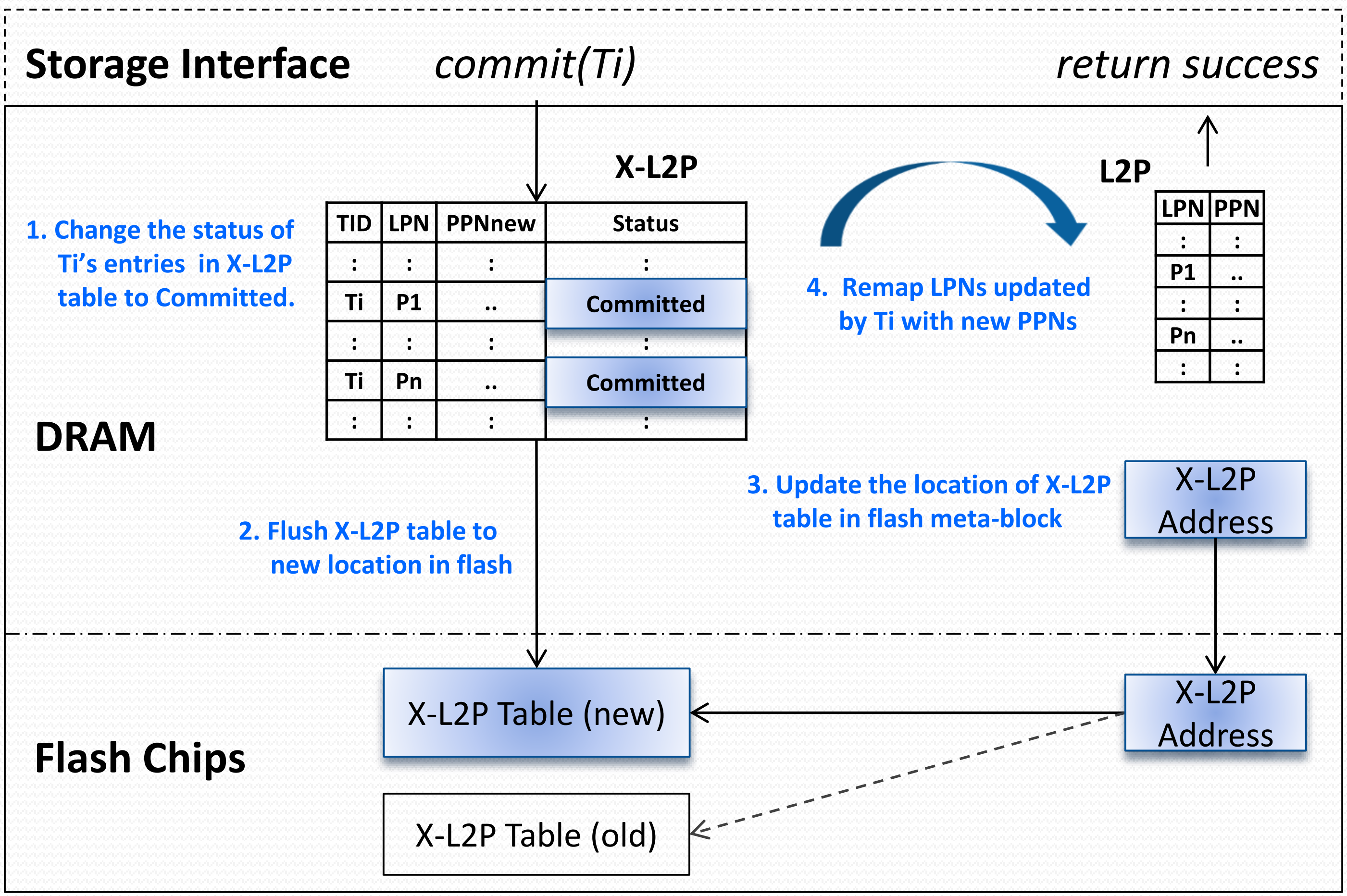


Extended API (SATA Interface)

- Transaction ID is passed to storage with Read/Write command
- Add Commit/Abort command

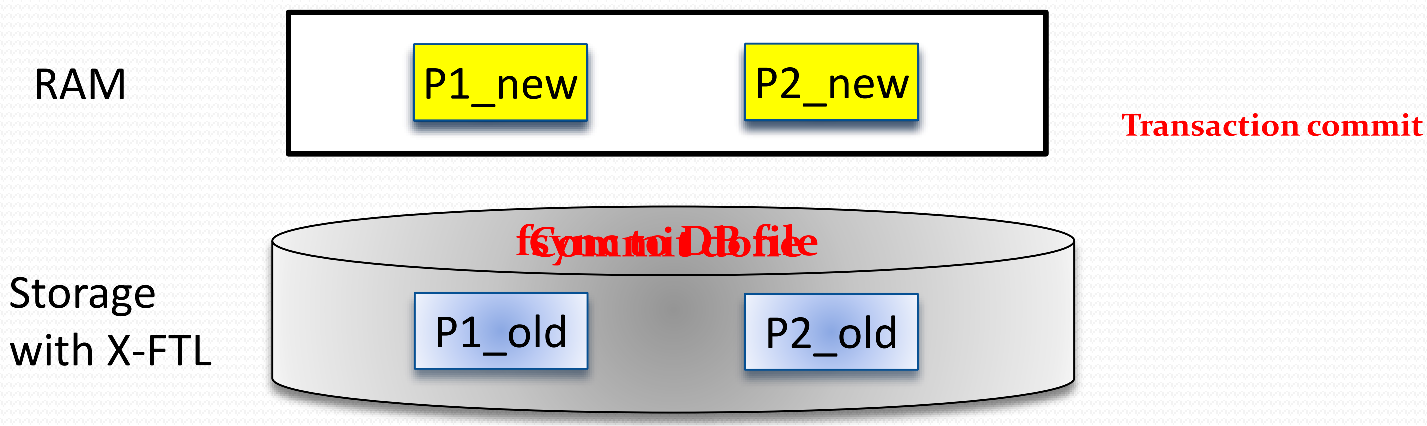


X-FTL: Commit Procedure



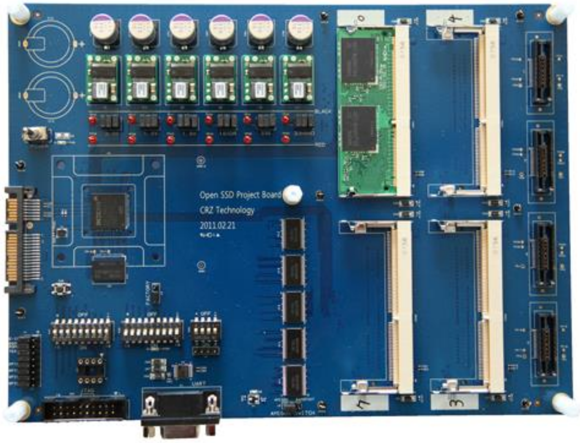
Transactional Atomicity in X-FTL

	fsync count	Write count
RBJ	**3 fsyncs per tx** - 2 syncs for journal - 1 sync for db	1 page write → 2 page writes
WAL	**1 per tx and 1 per checkpoint** - 1 sync for journal - 1 sync for db when checkpoint	1 page write → 2 page writes
X-FTL	**1 per tx** **- 1 sync for db**	**1 page write** **→ 1 page write**



Performance Evaluation

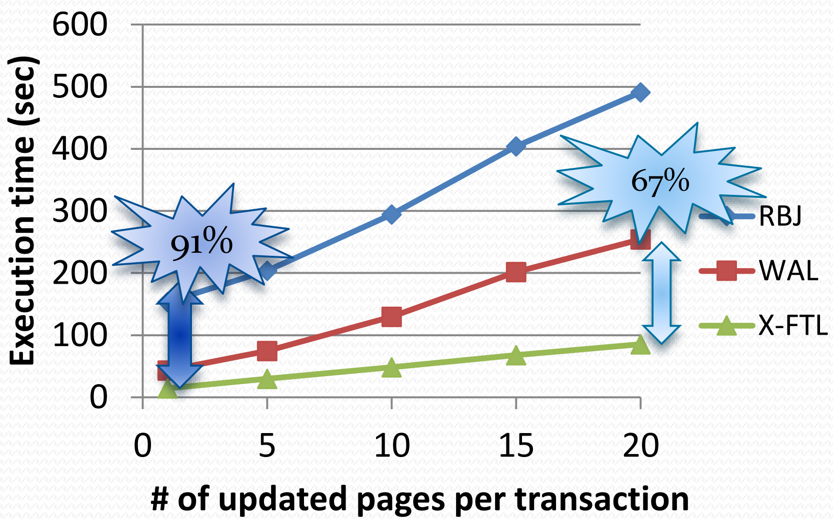
- Evaluation setup
  - OpenSSD development platform :
    - MLC NAND : Samsung K9LCG08U1M
      - Page size : 8KB, Block : 128 pages
    - 87.5 MHz ARM, 96KB SRAM, 64MB DRAM
  - Linux ext4 file system (kernel 3.5.2)
  - Intel core i7-860 2.8GHz and 2GB DDR3
  - SQLite 3.7.10
- Workloads
  - **Synthetic**
    - **TPC-H** partsupply table, random update, adjust transaction length
  - Android smartphone
    - SQL trace using Android emulator, **RL bench, Gmail, Facebook, web browser**
  - Database
    - **TPC-C** (DBT2), read intensive, TPC-C original
  - **File system benchmark**
    - **Flexible I/O(FIO)**, random write, adjust fsync frequency

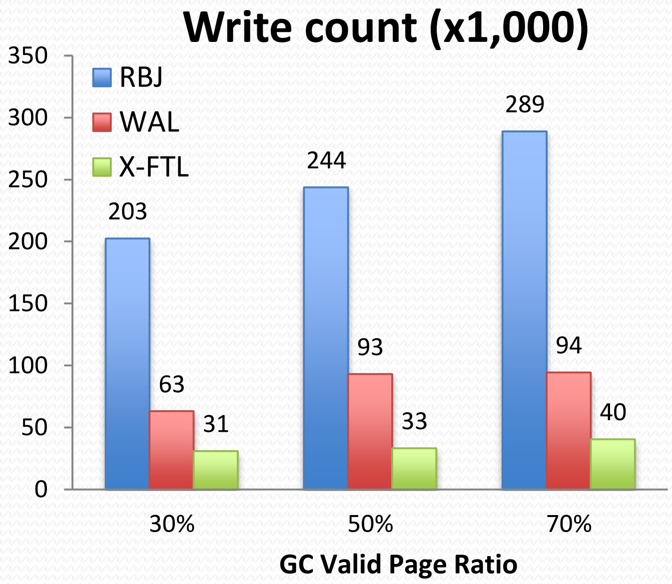


http://www.openssd-project.org

Synthetic Workload

- **TPC-H** partsupply table (60,000 tuples, 220 bytes tuple)
- Random update, 1-20 page updated by a transaction





Android Workloads

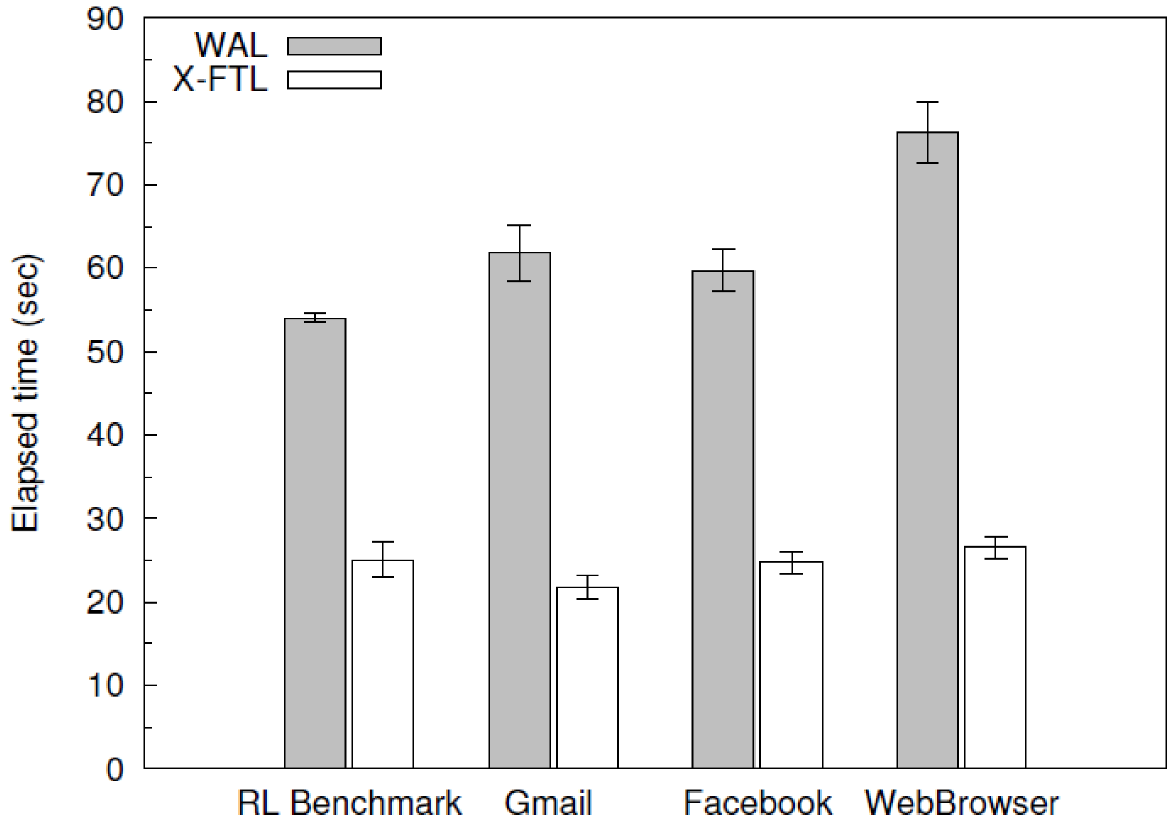


Figure 7: Smartphone Workload Performance

Conclusion

- X-FTL: Transactional FTL for SQLite databases
  - Offload the transactional atomicity semantics from SQLite to flash storage by leveraging the copy-on-write strategy of modern SSDs.
  - Achieve the transactional atomicity almost for free eliminating redundant writes by 50%.

Extent Mapping Scheme for Flash Memory

Bongki Moon
Computer Science & Engineering
Seoul National University

Alon Efrat
Computer Science
University of Arizona

Motivation: Traditional Mapping Schemes

- Page mapping
  - Highly flexible due to the size of granularity (page)
  - As the capacity of flash memory grows, the mapping table requires more space
- Block mapping
  - Smaller mapping table size
  - Less flexible and impractical

Both page and block mappings have limitations

Extent-Based Mapping (1)

- I/O request consists of a logical start address and the number of sectors to read or write
- Treat a given I/O request as an **extent** which serves as the basic mapping unit
  - Store extents in the mapping table as a whole unit
  - The degree of granularity changes determined by each individual write request

Extent-Based Mapping (2)

- Upon a write request:
  - Create new mapping information if the request writes into a clean logical area
  - Update existing mapping information if the request overwrites any valid data
- Upon a read request:
  - Treat the request as an **inquiry** extent
  - Search for all existing extents that overlap the inquiry extent

Virtual Extent Trie Design

- VET is a logical (virtual) trie of binary strings
- Each binary string is composed of 0's, 1's, and *'s

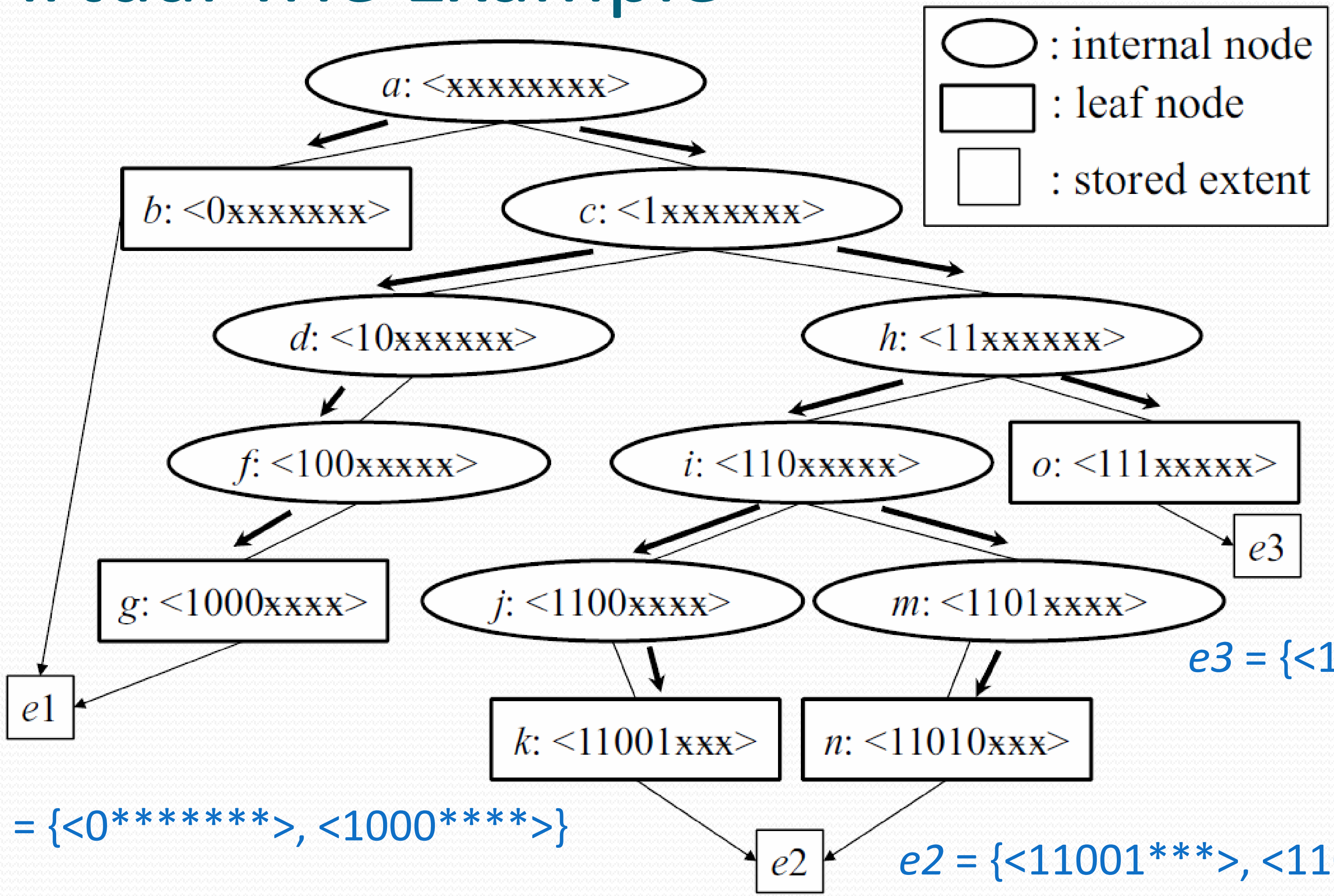
- **Don't care bits (*'s)**
  - Only appear at the end of a string
  - A string with don't care bits represents an extent whose length is a power of two
  - E.g. 0010**** can be used to represent
    - Logical start address: 00100000
    - Length: 16

Canonical Extent

An extent <s, l> is said to be ***canonical*** *if the length l is a power of two and the start address s is a multiple of l*

- A canonical extent <*s*, *l*> can always be represented by a single binary string
  - Obtained by replacing the least significant zeros of *s* with $log_2\ l$ many '*' bits
  - E.g. canonical extent <8, 4> → <000010**>
- Not all extent has a power-of-two length
  - Partition an extent to one or more canonical extents
- Serves as a key to identify each node

Virtual Trie Example



e3 = {<111*****>}

e1 = {<0*******>, <1000****>}

e2 = {<11001***>, <11010***>}

Virtual Trie Design

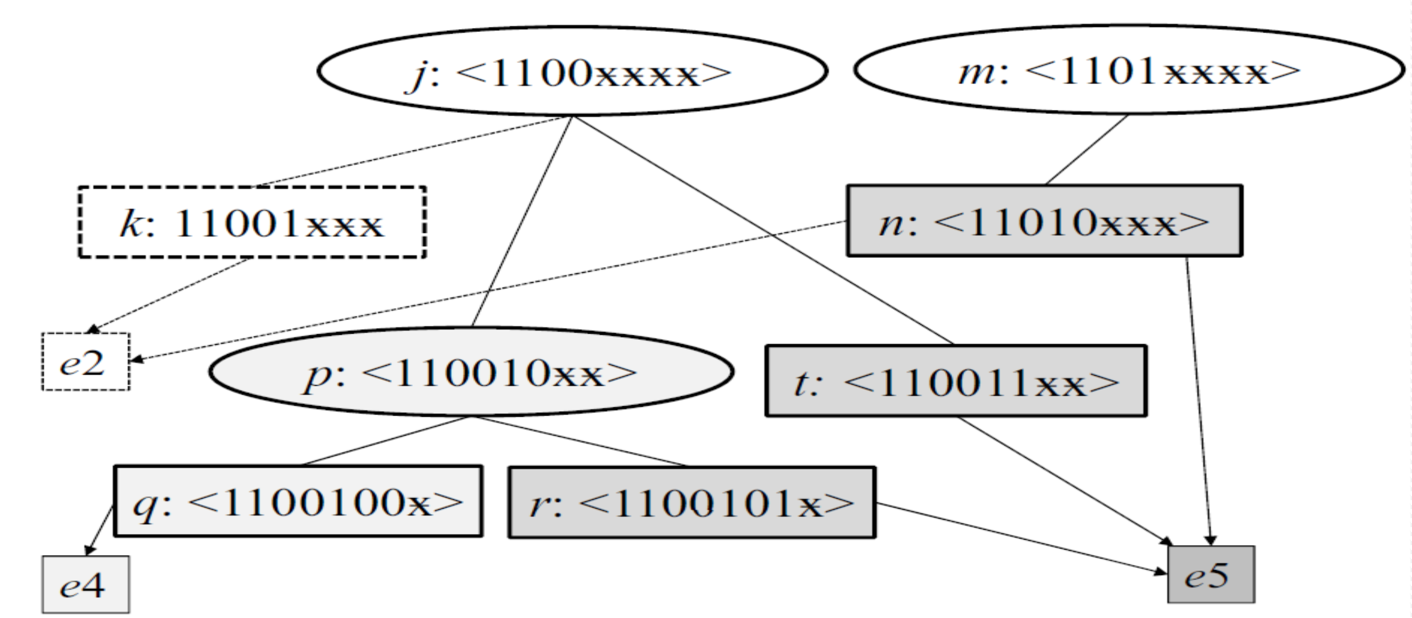
- Only a leaf node can have an extent
  - Internal node just serves as a helper
- VET is a virtual trie, but it physically stores canonical extents in a hash table

Algorithm: Update for Write Request (1)

- Inserting an extent:
  - Locate any existing extents overlapping the given extent
  - If overlaps are found, reinsert the existing extents updated by the overlap and delete outdated extents
  - Add the given extent
- LIS (Linear Insertion Scheme):
  - VET creates all of a given extent's ancestor nodes and adds the canonical extent itself to the virtual trie

Algorithm: Update for Write Request (2)

- Deleting an extent example (partial invalidation):
  - Extent e4 = <1100100*> arrives at the trie
  - Since e4 overlaps e2 = {<11001***>, <11010***>}, e2 is decomposed into e4 and e5

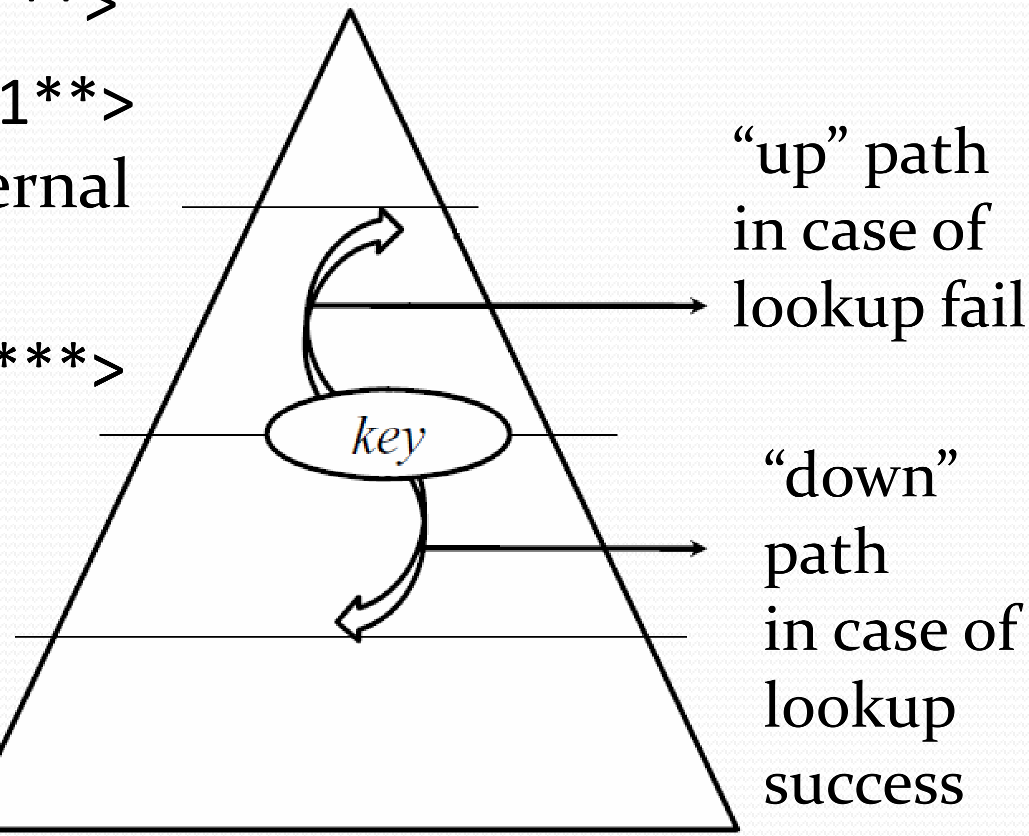


Algorithm: Search for Read Requests

- Perform a **binary search** against the nodes
- Search starts at the mid point of the root-to-bottom path (replace the second half of the string with '*' bits)
- Lookup succeeds:
  - Match found in a leaf node: terminate the search
  - Match found in an internal node: continue on the lower half (less * bits)
- Lookup fails:
  - Continue by searching upwards (more * bits)

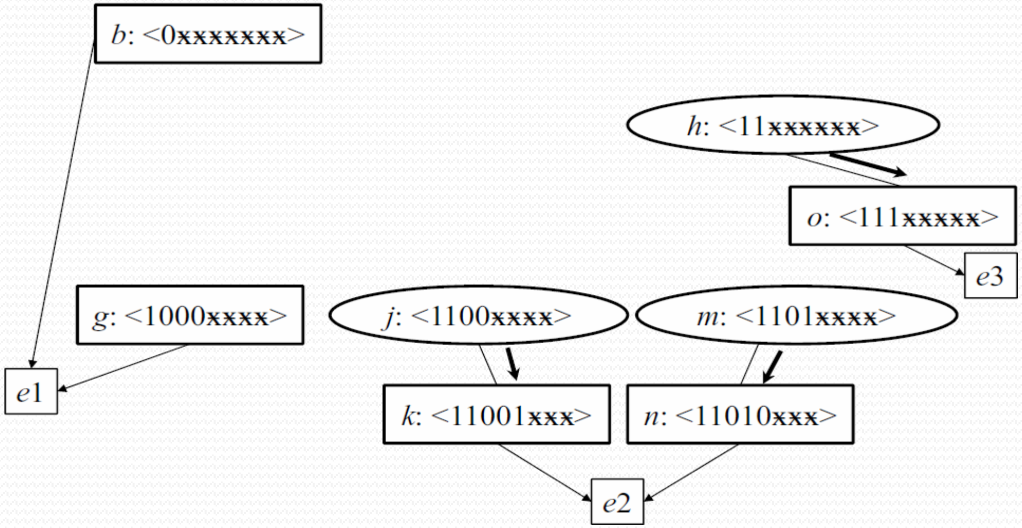
Search for Read Request Example

- Read request = 11001100
- 1st search key = <1100****>
- 2nd search key = <110011**> (match found in an internal node)
- 3rd search key = <11001***> (lookup failed)

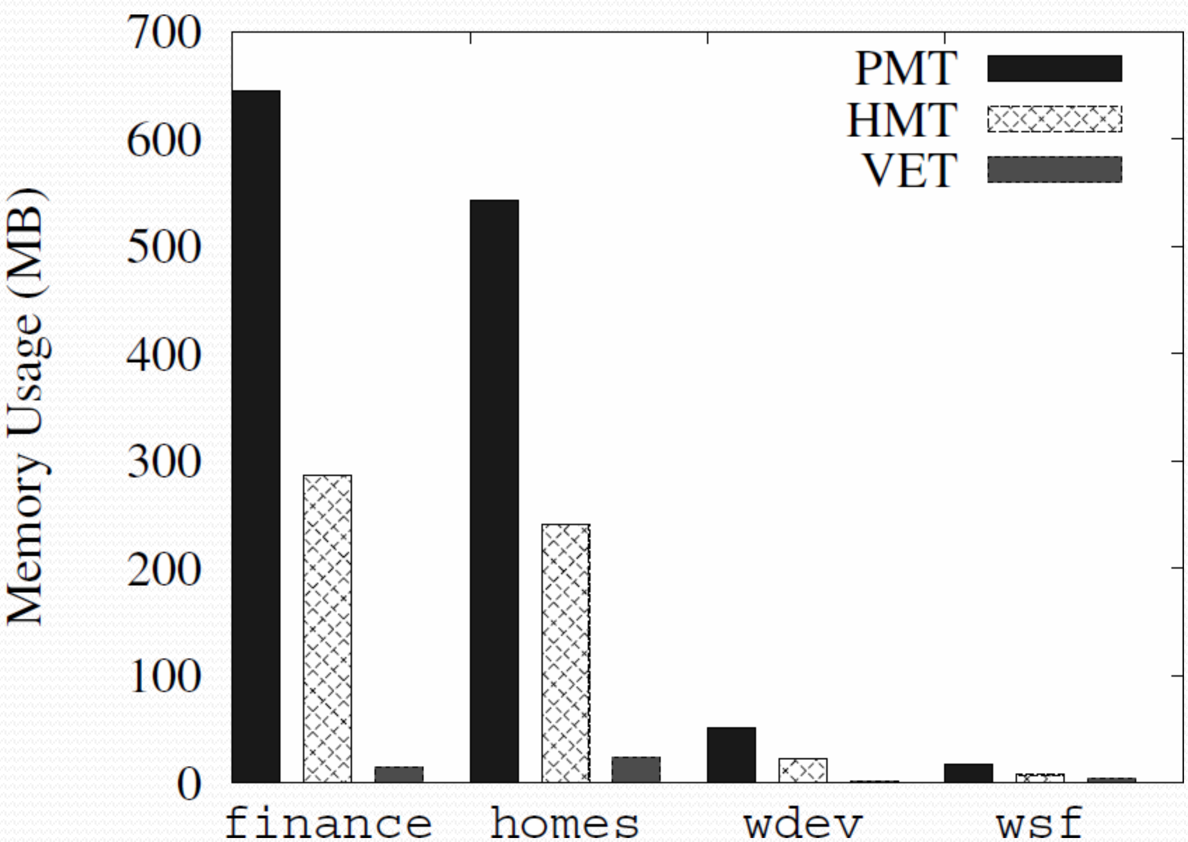


Optimization - Binary Insertion Scheme

- Some ancestors for a canonical extent are not used for the binary search
- Add only an indispensable internal node(s)
  - Less time and memory for inserting an extent
- Improvements on LIS in terms of memory usage and processing time



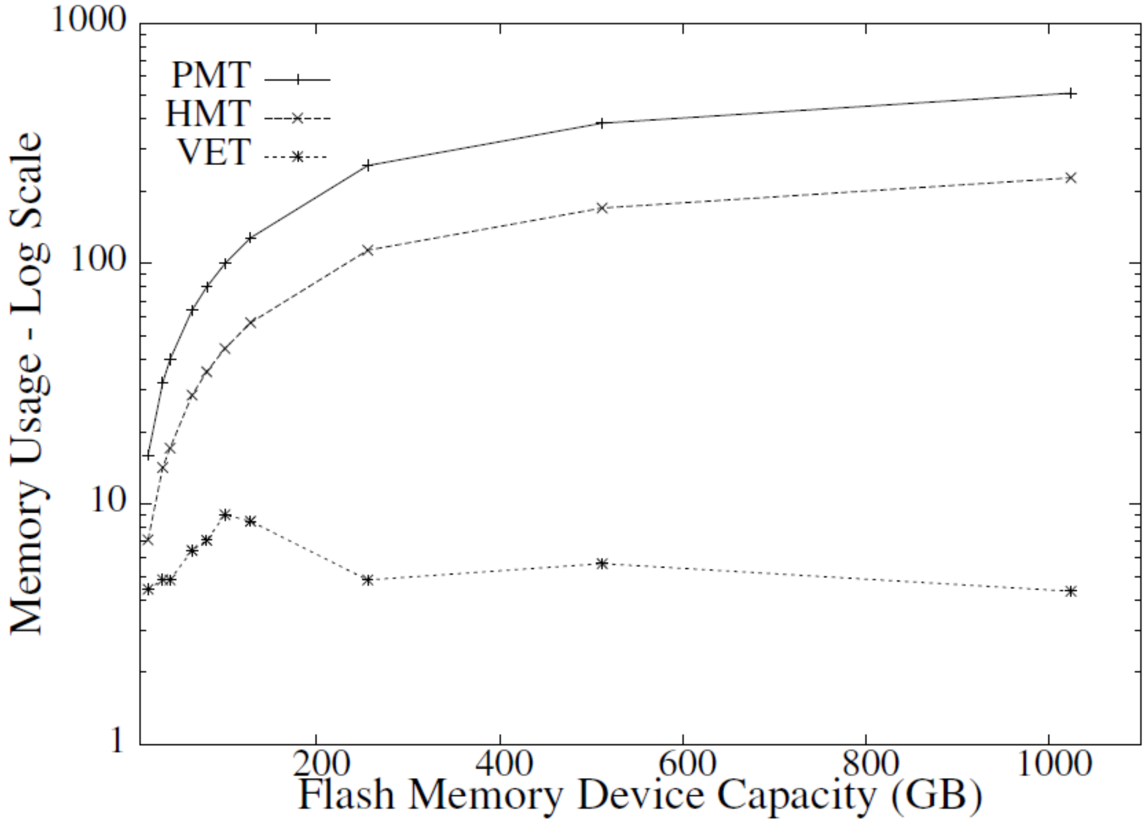
Memory Overhead Comparison



VET used much less memory than page mapping table (PMT) or hybrid mapping table (HMT)

- finance: OLTP
- homes, wdev: MS exchange servers
- wsf: web surfing

Scalability Test



As the space got larger, the traditional schemes suffered from enormous memory overhead while VET remained flat

However, …

- Updating and retrieving mapping information takes more time than PMT.
- Need further optimization for the overhead.