Data Organization and Processing

Indexing Techniques for Solid State Drives

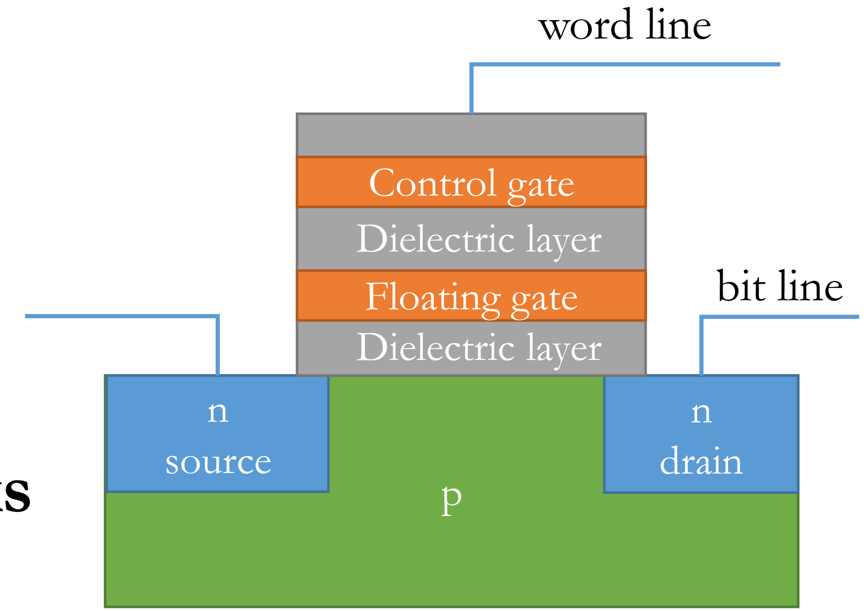
(NDBI007)

David Hoksza, <u>Petr Škoda</u>
http://siret.ms.mff.cuni.cz/hoksza

Outline

- SSD technology overview
- Motivation for standard algorithms modification
- New hierarchical algorithms
  - update minimization
  - parallelization

Solid State Drive (SSD)



- **No** moving **mechanical** components
- Contains **multiple NAND flash** memory **blocks**

- **Flash memory** is based on **floating gate transistors** (*tranzistory s plovoucími hradly*) supporting memory non volatility
  - **floating gate transistors form floating gates (cages)** capable of holding electrons and the **charge** they represent
  - if the **cell is uncharged** it represents a **1**, if it is **charged** it represents a **0**
    - uncharged gate conducts current
  - multiple cells can then store complex information

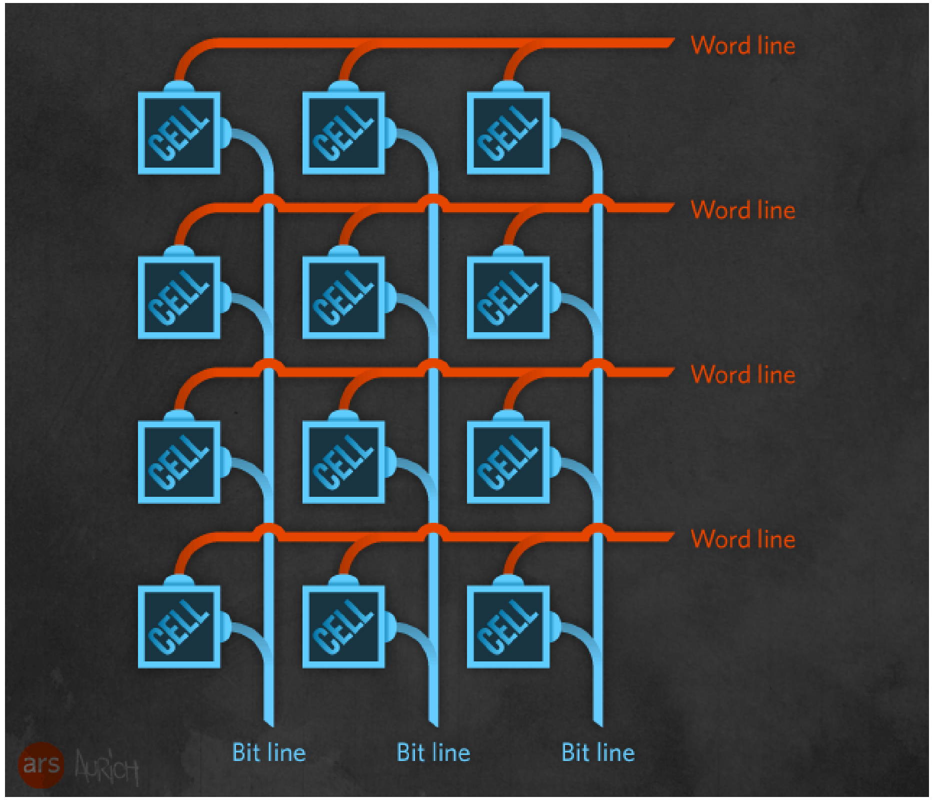
SSD Memory Types

- **Cells** are stored **in a grid** called **block** with **rows** called **pages**
  - **pages** consists of **main memory area** and **spare area** (error correction, management information)
  - an **SSD** consists of **multiple blocks**
- **Wiring** of the cells determines the **memory type**
  - **NOR** memory
  - **NAND** memory

NOR Memory

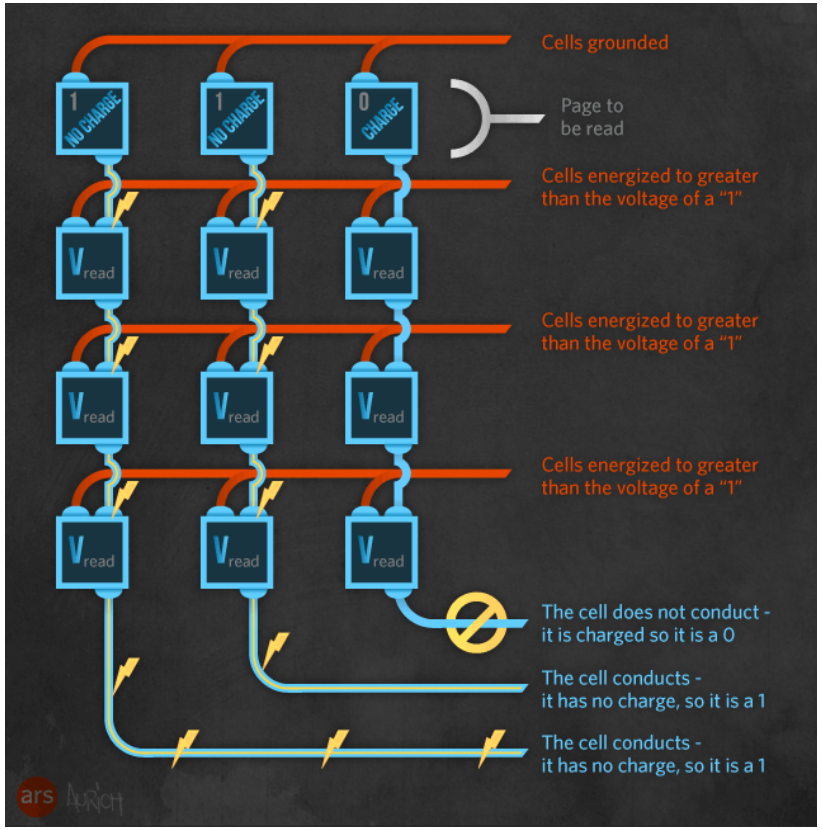
- Each **row and column is wired together**

- **Reading**
  - application of current to gates and seeing whether the current flows
  - **energize the word line** to a low voltage and the **bit line will show charge only if the floating gate contains no charge**, otherwise if the gate contains a charge the low voltage will not go through
    - → charged gate represents a 0



NAND Memory

- **NOR** chips are complex and take **a lot of space** → **NAND used in SSDs**
- In **NAND** wiring, **transistors** are connected **in series** with respect to the bit lines
- **Reading**
  - a **cell always conducts** the current when it is **energized to a higher than threshold current C_T**
  - all the word lines which are **not read** are energized to **voltage >= C_T**
  - the word **line to be read** is energized with **lower current**, the current is let in and respective bit lines are checked
  - the bit lines are read simultaneously

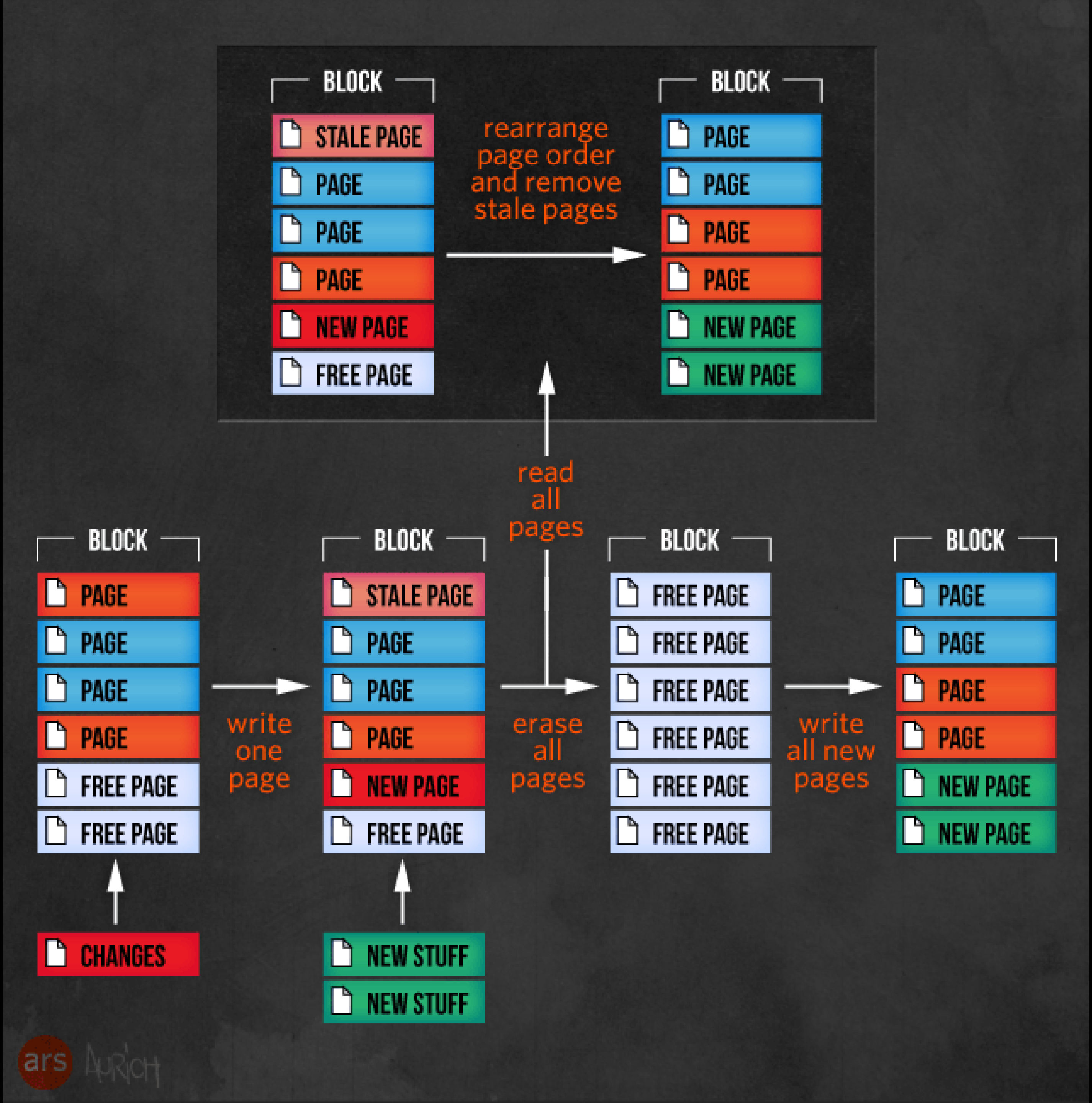


Page Modification (1)

- **Smallest addressable unit** in SSD is a **page**
  - the size in modern SSDs uses to be 8,192 bytes
- **Freshly erased** page stores **all 1s** (no charge in the gates) and the cells can be written to on the page level
- Erasing a page is done by application of high voltages
- **Turning individual cell or page back** into 1s is **not possible** due to the effect on the adjacent cells (high voltage) → erase operation is possible on block level only
- SSDs therefore **do not allow in-place update** of data

Page Modification (2)

- Three **types of pages**
  - **free page**
  - **live/used page**
  - **dead/stale page**
- **Updating a page** differs based on whether a free page is available
  - **free page available**
    - the updated content is written into the new free page
    - the old pages is marked as dead and can not be used until the whole block is erased
  - **no free page available** (but some dead/stale page is)
    - the block is read to cache
    - the block is erased
    - the modified content is written back


BLOCK
STALE PAGE
PAGE
PAGE
PAGE
NEW PAGE
FREE PAGE
rearrange page order and remove stale pages
BLOCK
PAGE
PAGE
PAGE
PAGE
NEW PAGE
NEW PAGE
read all pages
BLOCK
PAGE
PAGE
PAGE
PAGE
FREE PAGE
FREE PAGE
CHANGES
write one page
BLOCK
STALE PAGE
PAGE
PAGE
PAGE
NEW PAGE
FREE PAGE
NEW STUFF
NEW STUFF
erase all pages
BLOCK
FREE PAGE
FREE PAGE
FREE PAGE
FREE PAGE
FREE PAGE
FREE PAGE
write all new pages
BLOCK
PAGE
PAGE
PAGE
PAGE
NEW PAGE
NEW PAGE
ars
AURICH

Memory Degradation

- When writing to a page the **word line is charged** with **high voltage** and the **bit lines** which are **to be set to 0 are grounded** which causes the electrons to migrate into the respective cells
- Erasing a page is provided **by releasing the negative charge** from the gate
- **Each cycle** causes some **residual charge to remain** in the cells (damages the dielectric oxide layer) which changes the resistance of the gate → **flipping** a gate needs **higher current and takes longer**
- Data can be still read but can't be written into the worn-out cells any more
- In a standard use, the SSD disks should not reach the maximum amount of the program/erase (P/E) cycles sooner than in 5 years (magnetic HDDs lifetime)
  - depends also on the level of the NAND memory

Multi-Level Cell (MLC)

- With **one charge level**, the cell can contain **one bit → single-level cell (SLC)**
- With **four levels of charge**, each cell could contain **2 bits** **→ multi-level cell (MLC)**
  - each charge level corresponds to a value (e.g., highest charge = 11 … lowest charge = 00)
  - **increases storage density** and **complexity** of reading and writing
  - **decreases lifetime**
- SLCs are more reliable and less complex but much more expensive → only **enterprise solutions contain SLCs**
- **2 (MLC) and 3-level (TLC) cells** are **standard** in todays solid state drives

Update Issue Solutions

- Since **updating a page** needs considerable effort, which is moreover related to the **memory degradation**, measures need to be taken to decrease the impact of such behavior

- **“Hardware” oriented approach**

  - controller design

- **Application oriented approach**

  - minimization of the number of update operations

SSD Controller (1)

- **Processor mediating the communication** between **SSD** memory blocks and the **computer**
- Deals with all the logic regarding page management within the NAND chips (including reading, writing, erasing)
- **Tasks**
  - **parallelization**
    - SSD **has multiple channels** and can thus **address multiple NAND chips** at the same time
    - the controller **stripes data** in a similar way as the controller in a RAID array does and also provides error correction
  - **caching**
    - when striping is not enough, **controller can use SDRAM** to hold data until they can be written to the disk → further decrease in latency
    - requires **additional power supply** for the volatile SDRAM

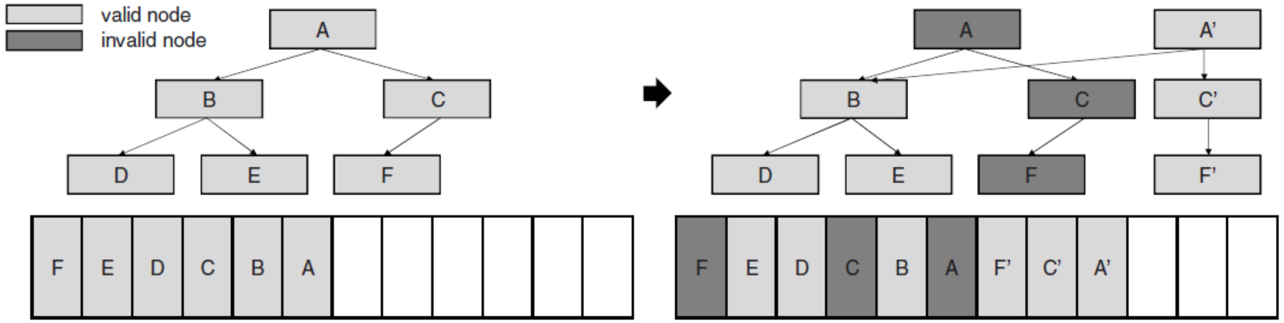
SSD Controller (2)

- **wear leveling**
  - keeping track of **highly used pages**
  - once a time **highly and sparsely used pages** can be **swapped** to ensure roughly the same lifespan for all the cells
- **garbage collection**
  - **keeping track** of blocks which contain **dead/stale pages**
  - **once a time, blocks** with sufficiently **enough dead pages** are **rewritten** into newly erased blocks and the old blocks are erased
- **TRIM**
  - existing operating systems do not physically delete files but only remove pointers to them → no way for garbage collector to find out that a given page is dead → TRIM command
  - without TRIM
    - instead of erasing the blocks as soon as they are known to be holding stale data, the erasing is being delayed → performance degradation
    - garbage collection mechanism will continue move them around to ensure wear leveling

Application Approach

- **Modification of the algorithms** dealing with the data access can lead to multiple advantages, e.g.:
  - **decreased number** of **update** operations
    - decrease of memory degradation
  - utilization of controller **parallelization** capabilities

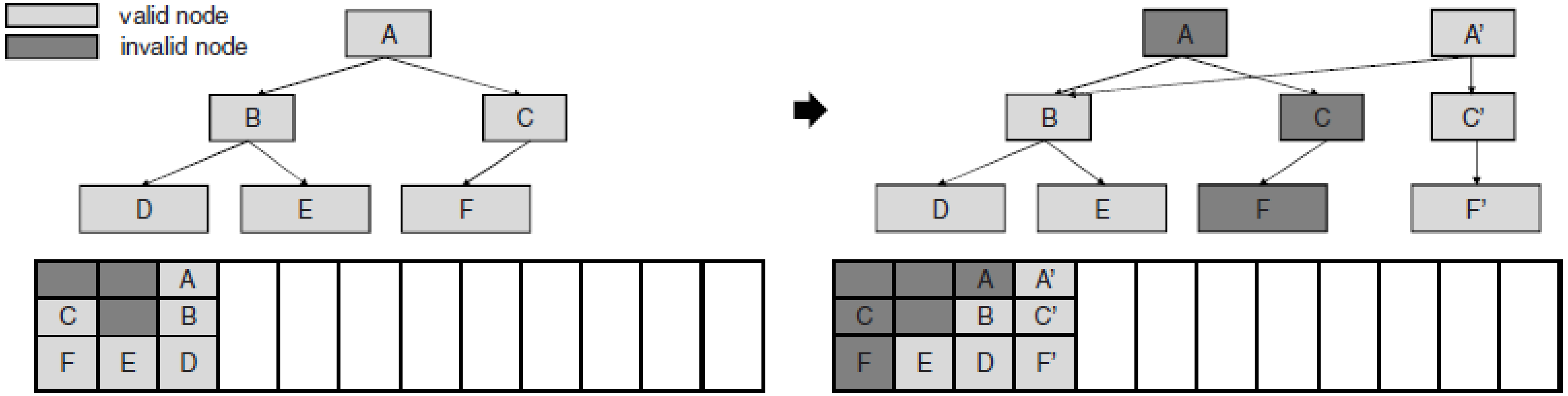
B+-Tree Issue



- B+-tree efficiently makes use of block-oriented storage by keeping related records together on the same page
- **Update** operations in B+-**tree affect only one or few pages** in mechanic **HDDs**
- Issue with naïve implementation of B+-tree on SSDs → **wandering (*putující*) tree**
  - inner nodes of a B+-tree store only keys and pointers
  - when a **record update** happens, the **leaf node** needs to be **modified**
  - since in-place update is not supported by the SSDs, the whole **page needs to be moved into a new location**
  - page move requires **modification of the pointer in the parent node** → **iterative** process ending only in the root
  - **relevant for B-tree implementations over raw flash**

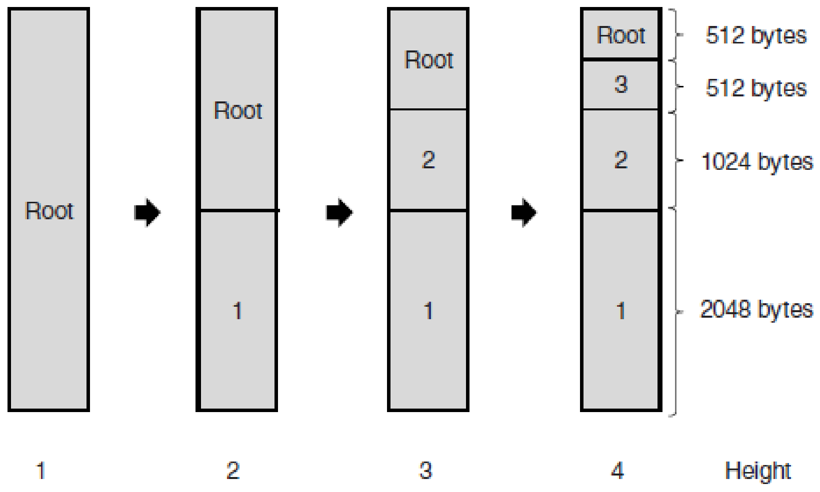
μ-Tree

- **M**inimally **U**pdated-**Tree** [Kang et. al.; 2007]
- Main idea
  - in μ-Tree all the **nodes along the path from the root to the leaf** are put together into a **single flash memory page**
- μ-Tree outperforms B+-tree by up to 28% and by up to 90% with a 8KB in-memory cache



μ-Tree - Overview

- Unlike B-tree, μ-Tree's **nodes are of variable size**
  - **leaf node** always occupies **half of the page**
  - as the **level is increased**, the node size is **reduced by half**
  - only the root node has the same size as its children nodes



- no significant difference between B+-Tree and μ-Tree except that the **size of a node in a μ-Tree is determined by its level** and the height of the tree

μ-Tree - Retrieval

Leaves reside in level 1

Algorithm 2 Retrieval

Input: *key* K (search predicate)
Output: *page_address* O (which points to the record corresponding to K)

$C \Leftarrow$ GetNodeFromPage(root page address, H)
$L \Leftarrow H$
while $C.type \neq LEAF$ **do**
    $K_i \Leftarrow$ smallest search-key greater than K
    $L \Leftarrow L - 1$
    if K_i exists **then**
        $C \Leftarrow$ GetNodeFromPage(P_i, L)
    else
        $C \Leftarrow$ GetNodeFromPage(P_m, L), where m is the number of pointers in C
    end if
end while
if K_i exists in C, such that $K_i = K$ **then**
    return P_i
else
    return $NULL$
end if

Tree height

μ-Tree - Retrieval

Size of the block/node to read

Offset of the block/node to read

Algorithm 1 GetNodeFromPage

Input: *page_address* P, *level* L

Output: *node* N

1: $S \Leftarrow Q/2^L$, where Q is the size of a page

2: $O \Leftarrow S$

3: **if** $L = H$ **then**

4: $\quad S \Leftarrow S * 2$

5: $\quad O \Leftarrow 0$

6: **end if**

7: $N \Leftarrow$ read at page P from offset O with size S

8: **return** N

μ-Tree - Insertion

Algorithm 3 Insertion

Input: *key* K, *page_address* P (which points to the record corresponding to K)

allocate a new page N
$(R, K', P') \Leftarrow$ InsertEntry$(K, P, N,$ root page address, $H)$
if $R = FULL$ **then**
    allocate a new page N'
    $C \Leftarrow$ GetNodeFromPage$(N,\ H)$
    $H \Leftarrow H + 1$
    $(C_l, C_r) =$ Split(C)
    $C' \Leftarrow$ GetNodeFromPage$(N,\ H)$
    insert $(C_l.K_1, N)$ and $(C_r.K_1, N')$ into C'
    write node C_l on page N
    write node C_r on page N'
    write node C' on page N'
end if

The root node becomes full as a result of the current insertion

μ-Tree – Insertion (cont.)

Algorithm 4 InsertEntry

Input: *key* K, *page_address* P, N, B, *level* L
Output: *return_value* R, *key* K', *page_address* P'

$C \Leftarrow$ GetNodeFromPage(B, L)
if $C.type \neq LEAF$ **then**
  find $C.P_i$, such that $C.K_i \leq K < C.K_{i+1}$
  if $C.P_i$ doesn't exist **then**
    $i \Leftarrow m$, where m is the number of pointers in C
  end if
  $(R, K', P') \Leftarrow$ InsertEntry$(K, P, N, C.P_i, L-1)$
  $C.P_i \Leftarrow N$
  if $R = SPLIT$ **then**
    $K \Leftarrow K', P \Leftarrow P', N \Leftarrow P'$
  else
    write node C on page N
    return $R \Leftarrow NULL$
  end if
end if
if C has space for (K, P) **then**
  insert (K, P) into C
  write node C on page N
  if C is full **then**
    return $R \Leftarrow FULL$
  else
    return $R \Leftarrow NULL$
  end if
else
  allocate a new page N'
  $(C_l, C_r) \Leftarrow$ Split(C)
  insert (K, P) into $(C_r.K_1 > K)?\ C_r : C_l$
  if $C_l.type \neq LEAF$ & $\exists C_r.P_i = N$ **then**
    swap $C_l \Leftrightarrow C_r$
  end if
  write node C_l on page N
  write node C_r on page N'
  return $R \Leftarrow SPLIT, K' \Leftarrow C_r.K_1, P' \Leftarrow N'$
end if

Flash Memory Addressing

Usually, the raw flash is hidden from the user

Block mapping techniques

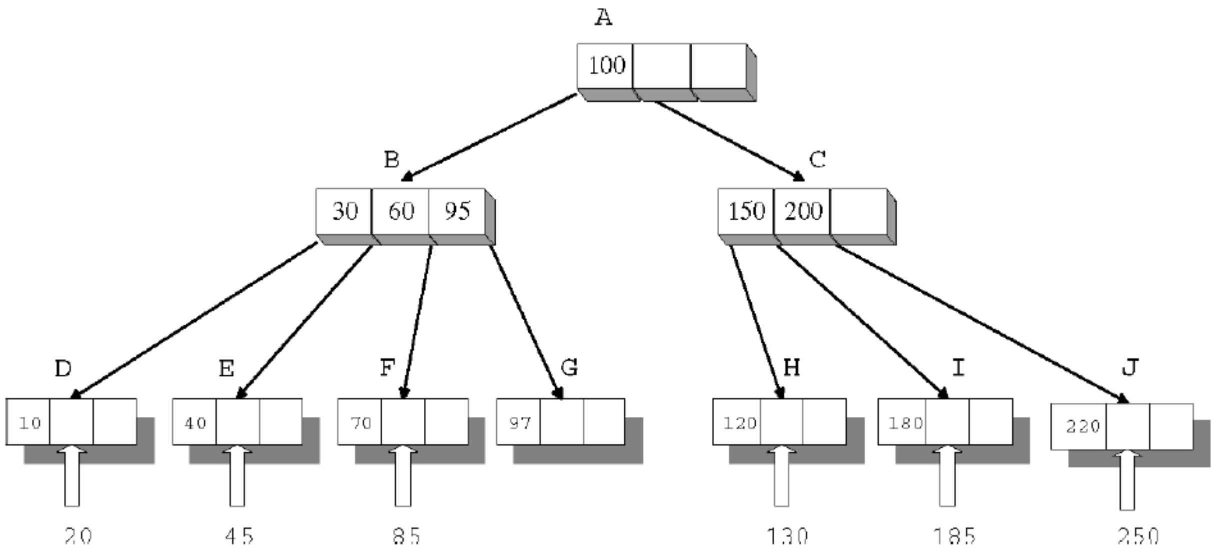
- **interface to the flash chip - Flash Translation Layer (FTL)**
  - implemented using a micro-controller within the flash package
  - **disk-like interface**
  - **mapping the physical page addresses** to **logical block addresses** and only the logical address is visible outside the package
  - **simulation of in-place updates** by mapping re-writes of a page to an empty page
  - **wear leveling** by distributing writes uniformly across the media

Flash-specific file systems

- **based on log-structured file systems**
- file systems often **implement B-trees** themselves in order **to manage the storage structure**
- JFFS2, JFFS3 (wandering tree problem), YAFFS, …

BFTL

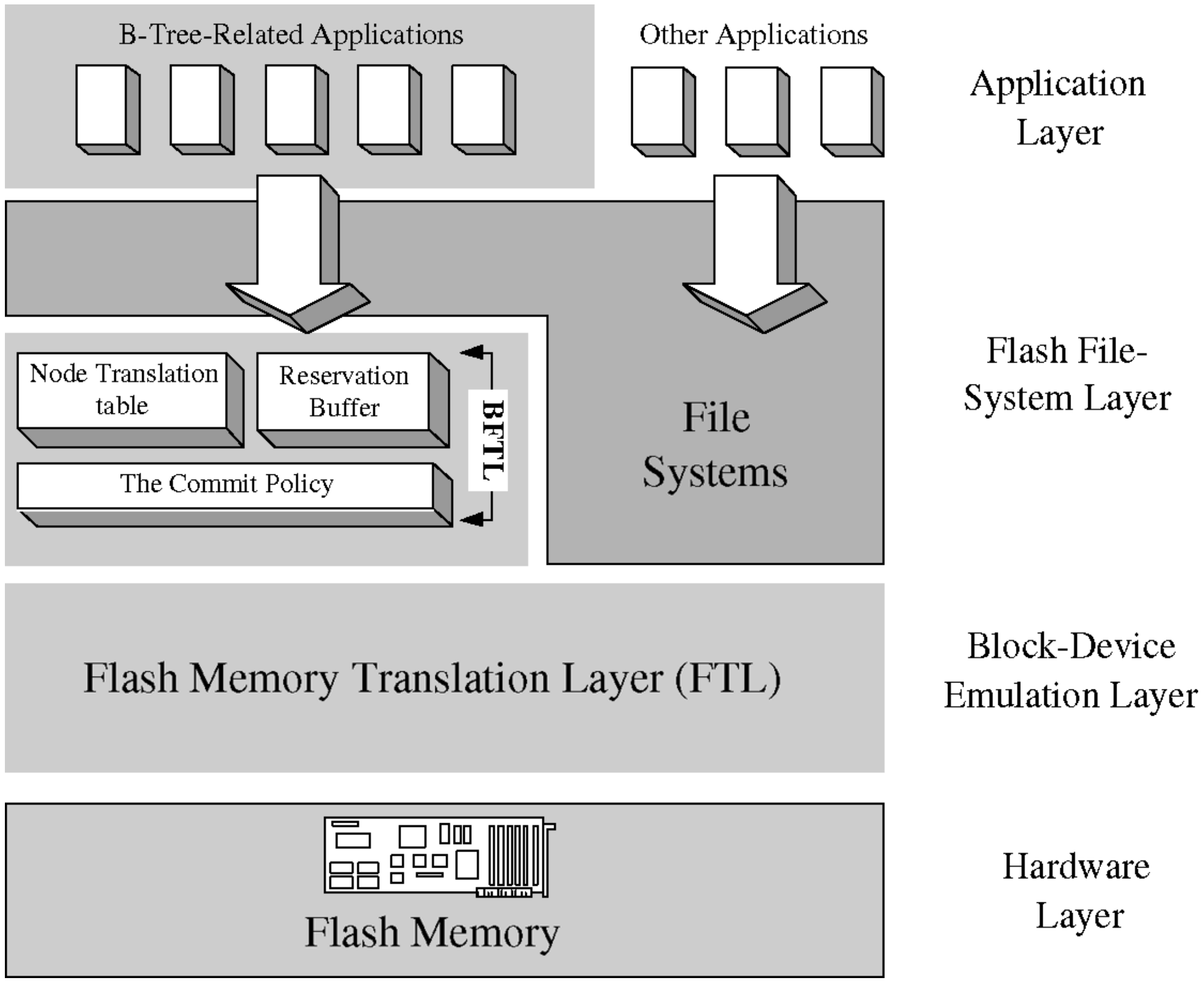
- B-tree index management over flash-memory storage systems [Wu and Kuo; 2007]
- Motivation
  - built **over FTL**
  - **inserting a single record causes whole page copy** (possibly more when rebalancing is needed) → free space consumption → garbage collection
  - let us **bu** … **hunks**



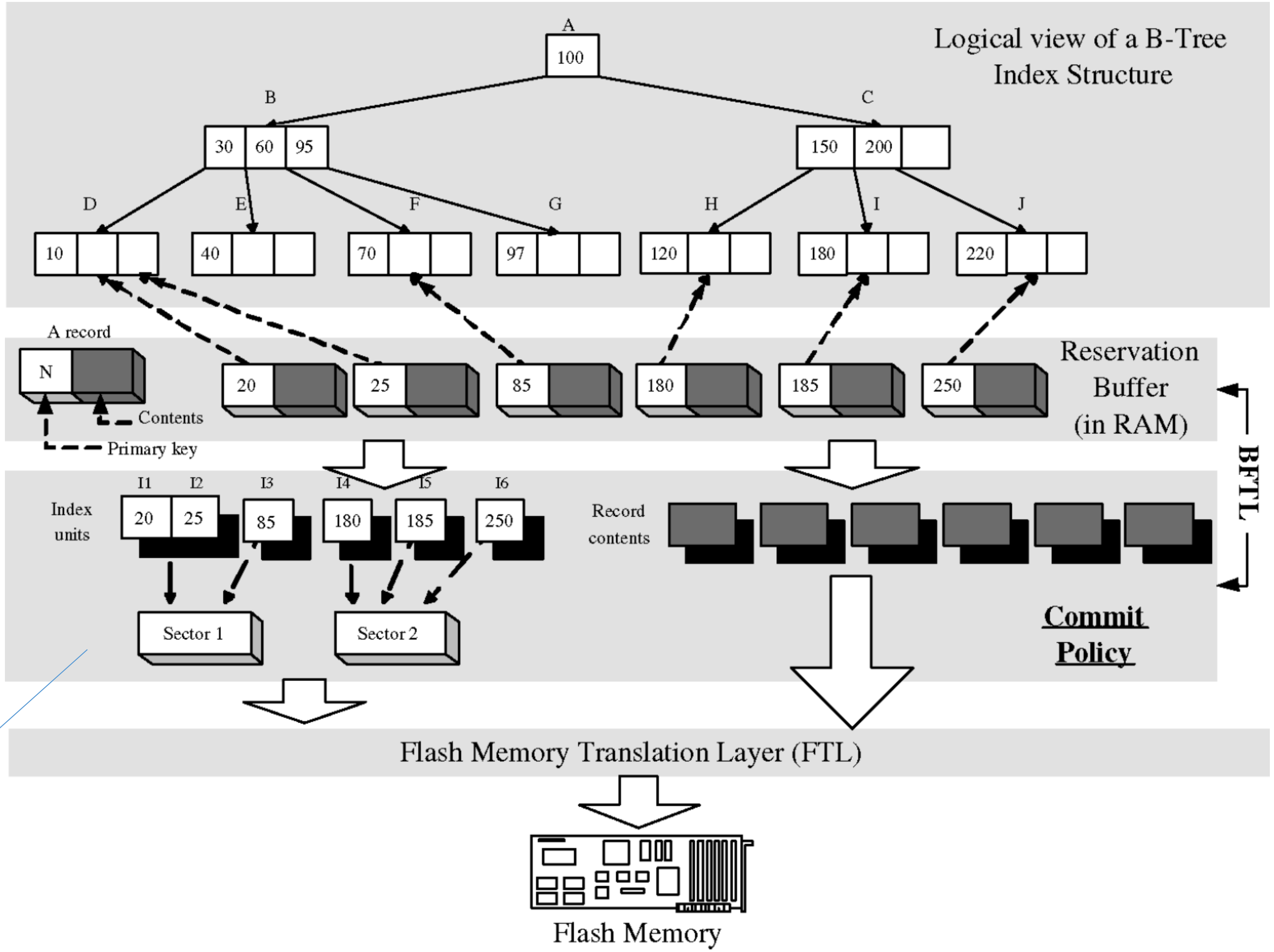
BFTL Architecture

- **B-tree index services** requested by the upper-level applications are **handled** and translated **from file systems to BFTL** and then **block-device requests** are sent **from BFTL to FTL**

- BFTL was meant to be a part of the operating system

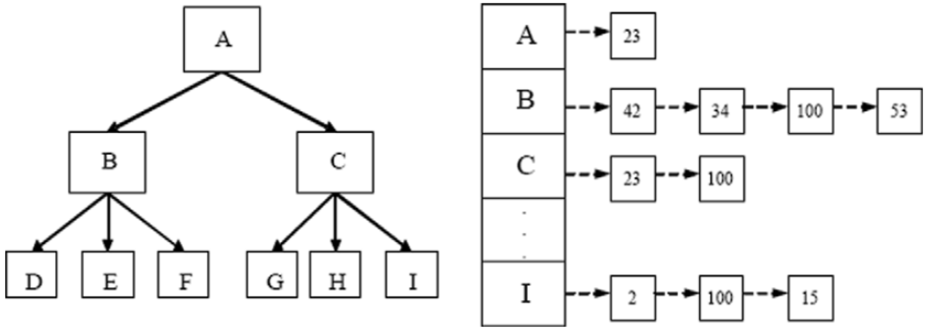


BFTL (cont.)



Two sector writes instead of up to six.

BFTL (cont.)



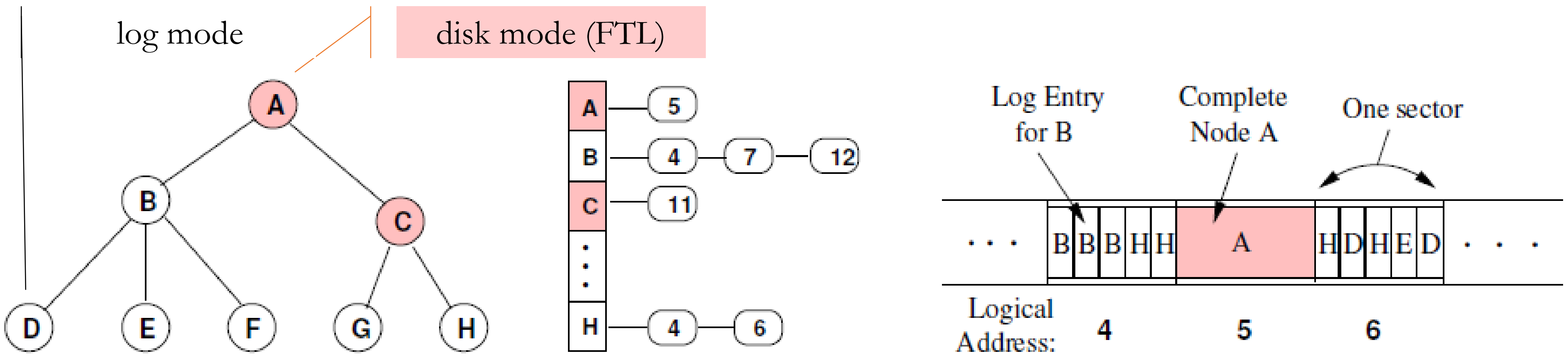
- **BFTL** places a **new layer** on the flash memory file system composed of **Reservation Buffer (RB)** and **Node transition Table (NT)**
  - **modification operations** are **gathered in the RB** and once a time **flushed to as little segments as possible**
    - insert into a node can cause the **content of the node** to exist in **multiple segments/pages** over flash memory
  - **NT** logically **binds node entries** being distributed through different pages (linked list of pages where the node actually resides)
    - **to construct a logical node requires multiple read operations**
    - to prevent uncontrollable growth of the chains a threshold is set which, when reached, causes given list to compact

FlashDB

- Self-tuning database optimized for sensor networks using NAND flash storage [Nath and Kansal; 2007]
- Contains a **self-tuning index** that dynamically **adapts its storage structure with respect to the workload** and the underlying storage device
- Considers **two index designs**
  - **B+-tree(Disk)** - **built upon a disk-like abstraction (FTL) over flash**
    - when modifying a small part of a node, the whole node needs to be read and written elsewhere → **not suitable for write-intensive workload**
  - **B+-tree(Log)** - **inspired by log-structured file systems**
    - write operation on a B+-tree node is encoded as a log entry and is placed in an in-memory buffer and when the buffer contains enough data to fill a page, it is written to flash
    - reading a node is expensive because many log entries → **inappropriate for read-intensive workloads**

FlashDB (cont.)

- **Implements a self-tuning mechanism** deciding whether **at given time** use **disk mode** or **log mode** – **self-tuning B+-tree (B+-tree(ST))**
  - **disk mode** – operating as the original B-tree does
  - **log mode** – operating as BFTL
  - **switch between disk and log mode** secured by a **variable** counting number of read vs. update operations
  - at any point in time a node can be either in disk or log mode
    - consolidation (log → disk) can be done offline when read-intensive workload is expected



(a) A logical B^{+}-tree(ST) (b) Node Translation Table (c) Flash storage

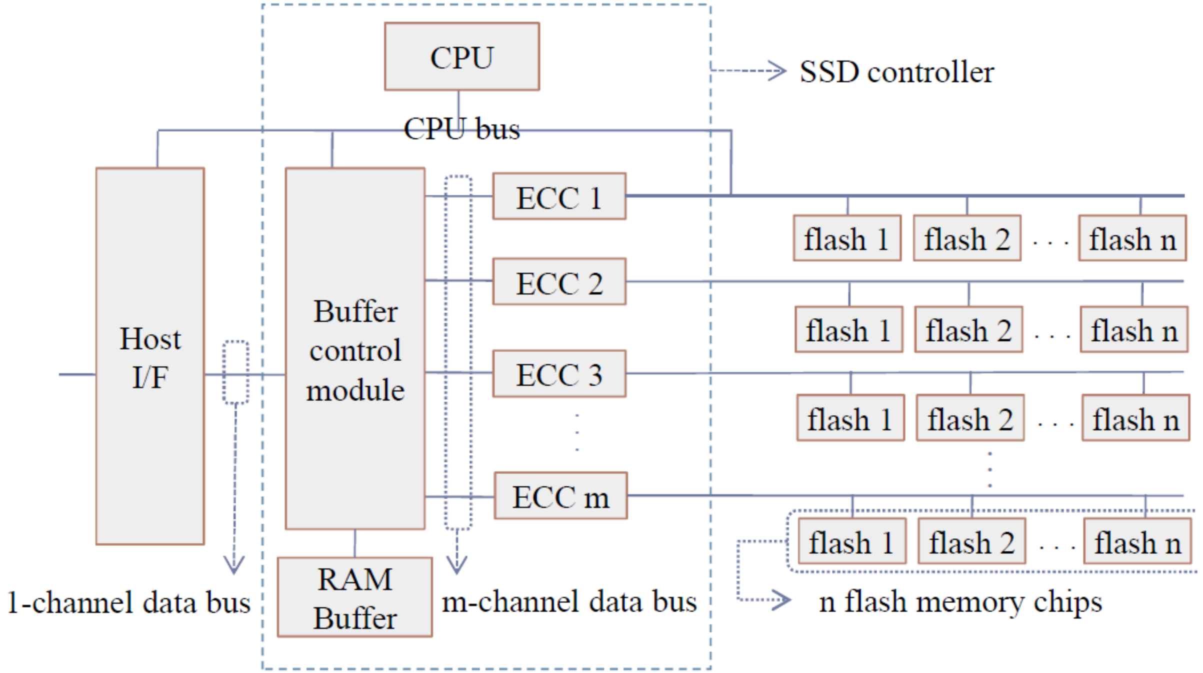
PIO B-Tree

- Optimized B-tree by **exploiting internal parallelism** of SSDs [Roh et. al.; 2012]

- **B-tree optimizations with respect to parallel capabilities of SSDs → PIO B-Tree**
    - multi path search algorithm + parallel range search
    - batch update

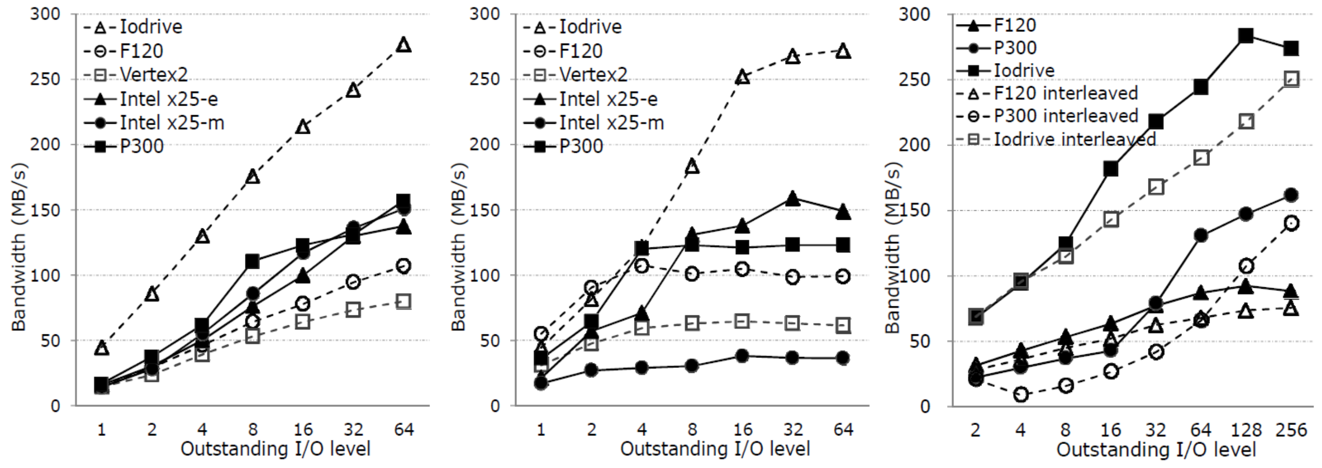
SSD Parallelism

- **Channel-level parallelism**
  - if the host I/F (host interface) **requests I/Os designated to different flash memory packages** spanning several channels, the channel-level parallelism is achieved by transferring the associated data through the multiple channels at the same time
- **Package-level parallelism**
  - **striping flash memory packages**
  - analogous to striping a disk array in the RAID approach



Utilizing SSDs Internal Parallelism

- **Principles**
  - **large granularity of I/Os**
    - package-level parallelism
  - **high outstanding I/O level**
    - channel-level parallelism
  - **no mingled read/writes**
    - avoid creating I/Os in a mingled read/write pattern



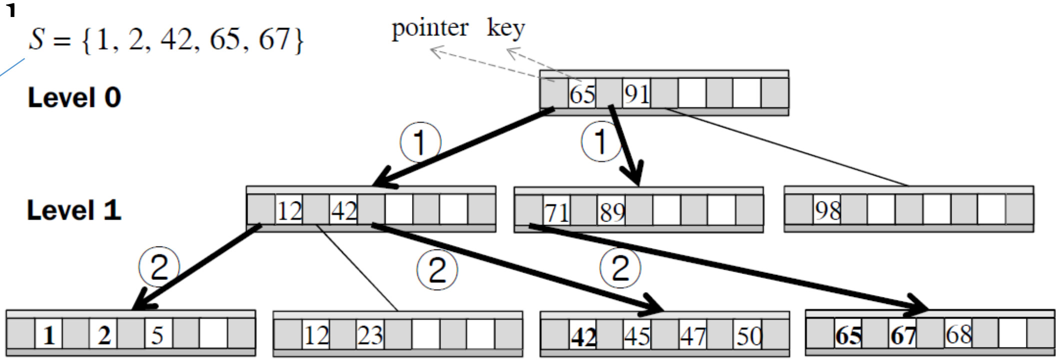
(a) Read **(b) Write** **(c) Read/write mixed**

- **Parallel synchronous I/O (psync I/O)**
  - currently not supported by OS kernels
  - operates as **standard synchronous I/O** except that the **unit of operation** is an **array of I/O requests**
  - implemented as a **wrapper using asynchronous I/O**

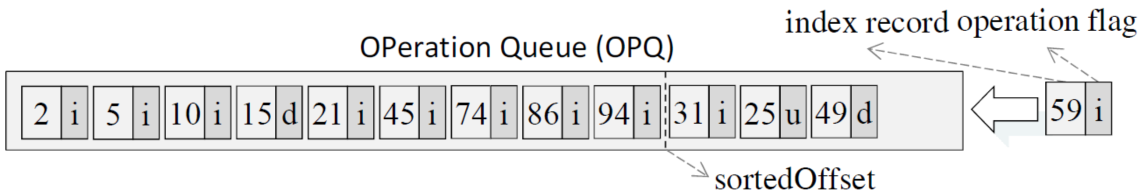
Multi Path Search (MPSearch)

- To search the i-th level, the results from $(i-1)$-st level need to be known → the only way to obtain parallelism is through **combining multiple searches**
- MPSearch processes a set of requests at once while searching multiple nodes level by level
- **Algorithm** (applies to point search - range search with minor modifications)
  - **examine** the **root node** for the relevant **pointers to pages at level 1**
  - **collect** all the pointers and **fetch respective pages using psync I/O**
  - **entries** of the read internal nodes are **examined node by node**, and the pointer set for each valid node is extracted, and the sets are joine
  - **repeat** until the leaf node is reached

Search requests



Update (1)



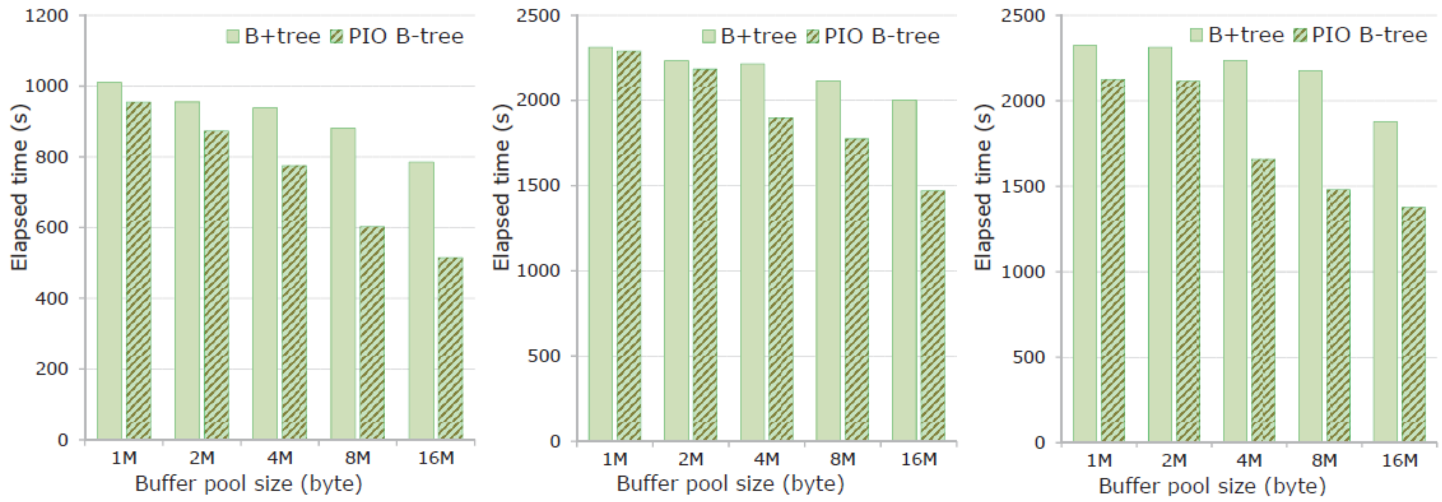
- **Optimization of update operations** such as insert, delete, and update
- **Index operations inserted into in-memory Operation Queue (OPQ)**
  - array-like structure consisting of **pairs [index record(key + pointer to the data page); operation type]**
  - divided into **sorted and unsorted part**
    - **new entry appended** and after every k inserts sorting occurs

Update (2)

- **Batch update (bupdate)**
  - **update** is triggered **when OPQ** is full for all the OPQ

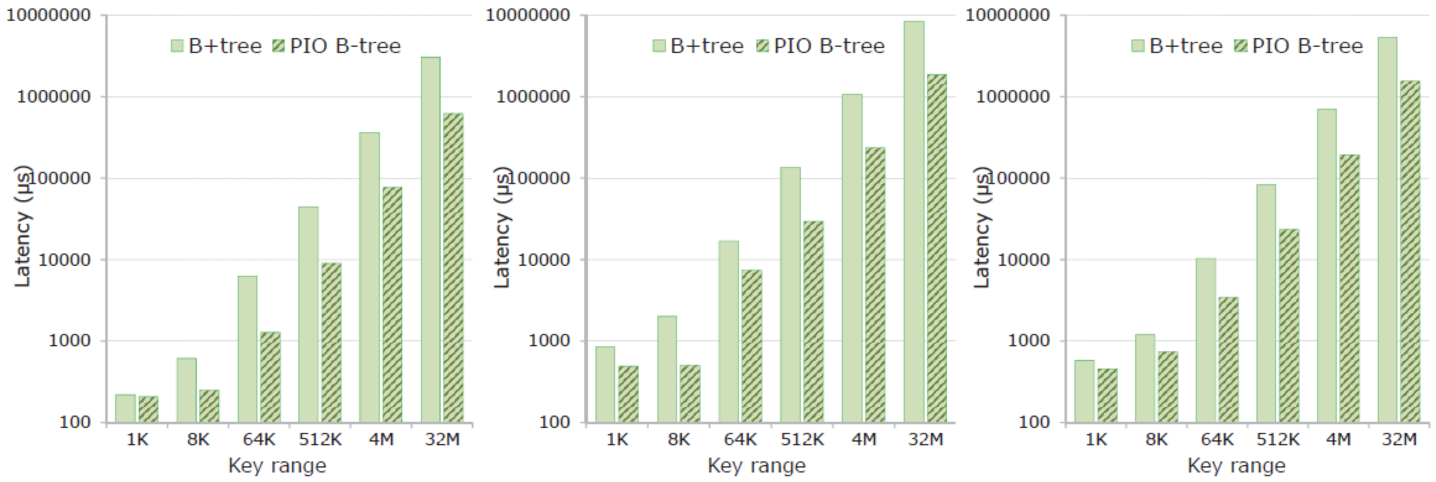
1. relevant **leaves** are **reached using psync I/O**
2. **update operations** of OPQ are performed to the leaf nodes
3. the **updated leaf nodes** are **written** to SSD at once **via psync I/O**
4. multiple fence keys can be generated by multiple node-splits, causing propagation of the fence keys to a parent node (similarly for merge operation)

- most of the update operations return instantly (only append to the OPQ occurs) but some update operations may take longer since the *bupdate* takes place



(a) Iodrive **(b) P300** **(c) F120**

Figure 9. Search time with different buffer sizes



(a) Iodrive **(b) P300** **(c) F120**

Figure 10. Range search time with different ranges in log scale