Object-based SCM - An Efficient Interface for Storage Class Memories

Yangwook Kang, Jingpei Yang, Ethan L. Miller

UC Santa Cruz

NSF Center for Research in Intelligent Storage

Storage Systems Research Center

Center for Information Technology Research in the Interest of Society

Baskin Engineering UC Santa Cruz

Storage Class Memories

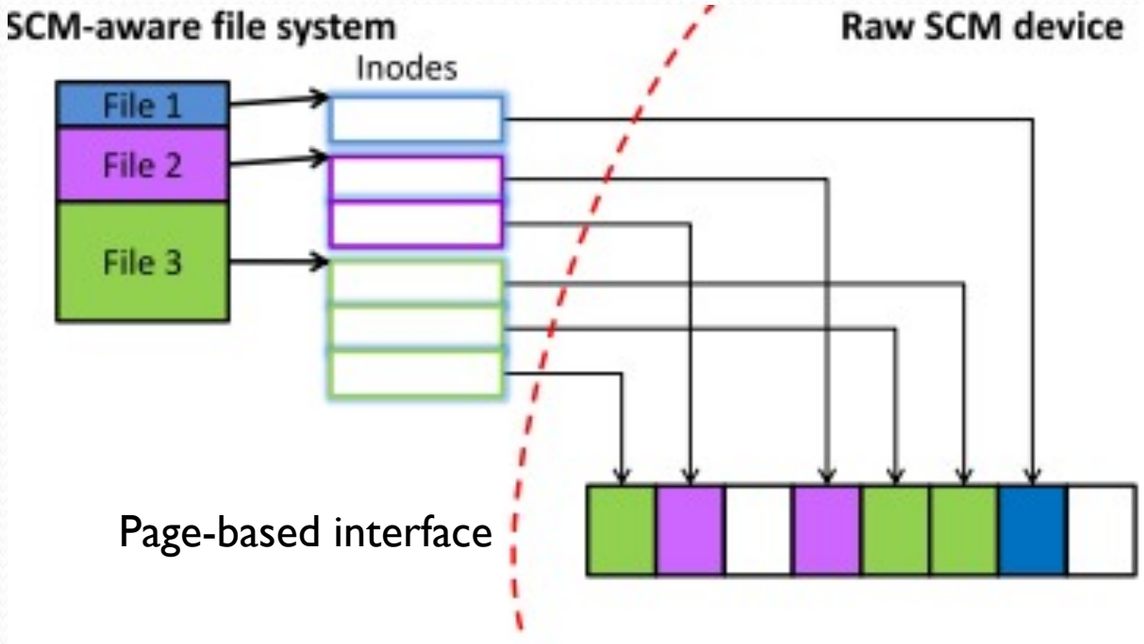
- A new class of data storage and memory device blurs the distinction between storage and memory
  - Fast read/write speed like DRAM, Non-volatility like disks
  - e.g. Memristor, Phase-change RAM, STT-RAM, Flash memory

- Differences in characteristics
  - Endurance
    - PRAM 10^8, RRAM 10^8 , STT-RAM 10^{15}, MEMRISTOR 10^8, FLASH 10^5
  - Asymmetric read/write performance
    - PRAM, FLASH
  - Destructive read
    - FRAM
  - Out-of-place update
    - FLASH

- The use of SCMs in current systems
  - Direct access: SCM-aware file system & custom designed device
  - FTL-based storage devices

Current Interface for SCMs
Direct Access Model

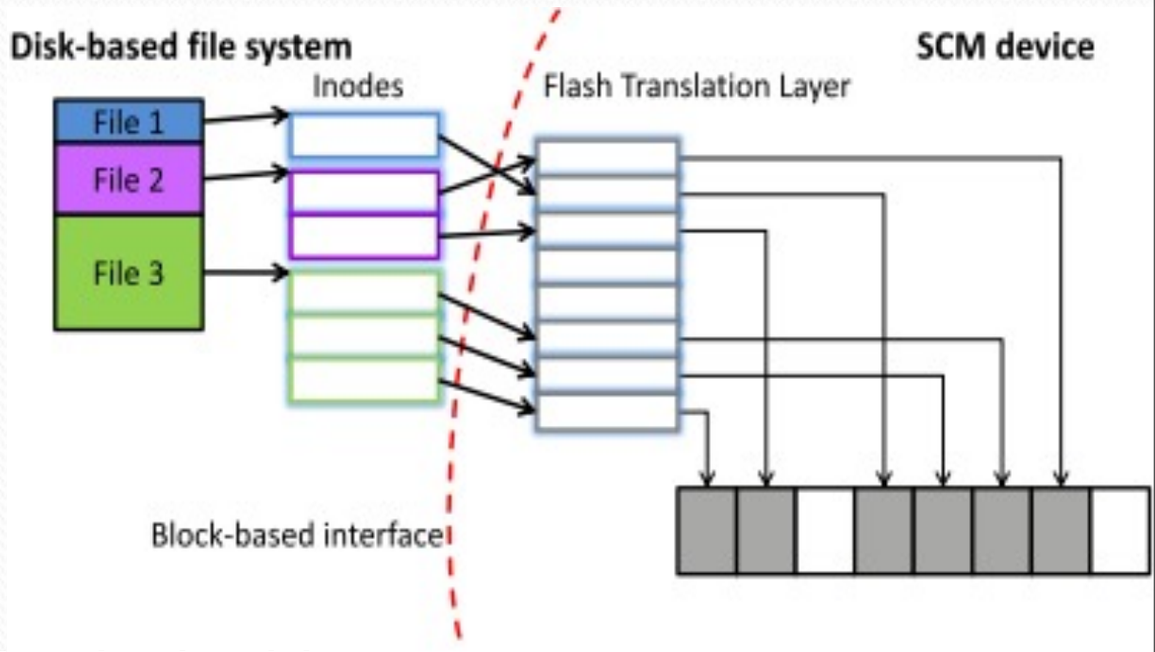
- Allows a file system to fully control the medium
  - A file system is designed to handle the characteristics of the target SCM
  - Raw medium can be accessed directly
    - memory bus, PCI-e slot



- ✚ Efficient
- ➖ Not easy to deploy
  - requires specific fs, hw, os
- ➖ Difficult to adopt new SCM

Current Interface for SCMs
FTL-based model

- Allows the legacy file systems to access SCMs as a block-based storage device
  - Device maintains a mapping table and uses block interface
  - SCM characteristics are handled by the device



- ✚ Compatibility
- ✚ Portability
- ➖ Sub-optimal performance
  - two translation layers
  - lacks of information about the requests (requires TRIM)
    - approaches to alleviate this problem:
      - Nameless writes: let inodes contain physical address
      - DFS: move FTL to the operating systems, not using block-interface anymore

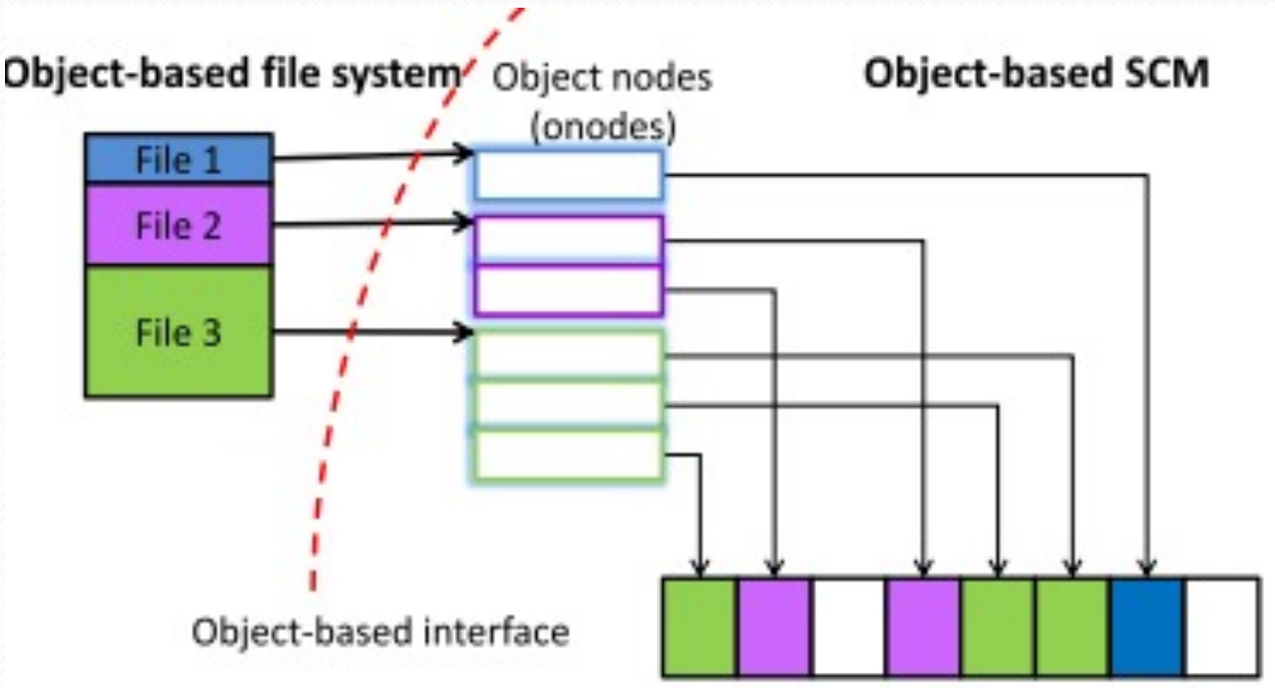
Object-based model for SCMs

- The object-based model
  - Offloads the storage management layer to the device
  - Encapsulates the VFS-level requests in an object with their metadata information

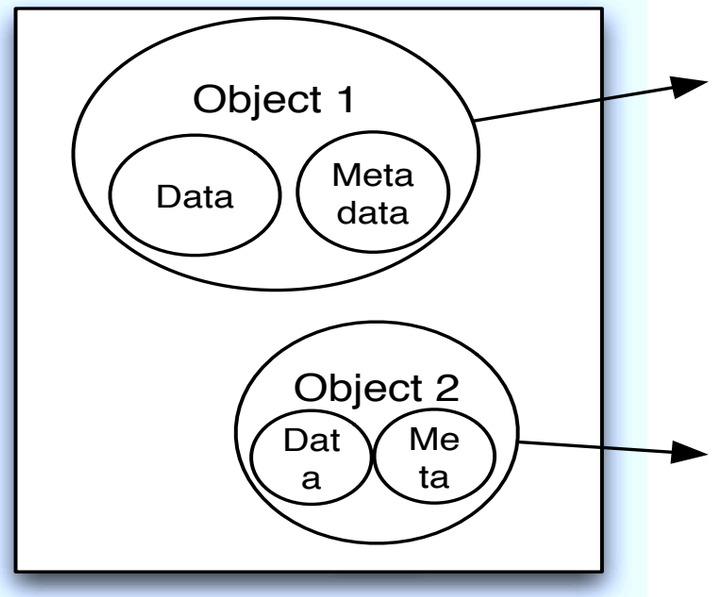
✚ Drop-in replacement for new type of SCM
  - object-based FS need not know about the underlying medium

✚ Better H/W optimizations
  - Rich information about the requests
  - The block management layer is inside
    e.g. splitting a large requests into concurrent writes to multiple SCM chips

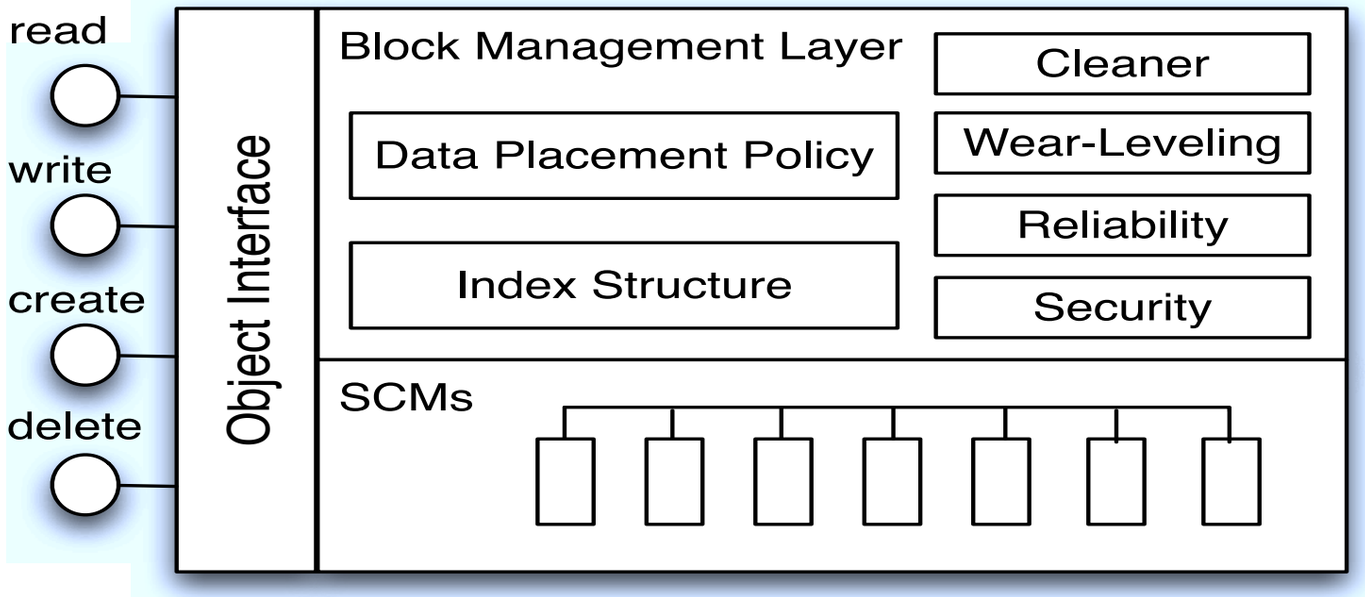
✚ The existence of objects
  - better space efficiency, reliability



Overview
the object-based model for SCM



(a) Object-based file system.



(b) Object-based device.

Design Issues of Object-based SCM

- Core Subsystems
  - Data placement policy
  - Index structure
  - Wear-leveling and Cleaning

- Advanced Features
  - Object-level reliability
  - Object-level compression and encryption
  - Client library for object-based SCMs
  - Object-level transactions

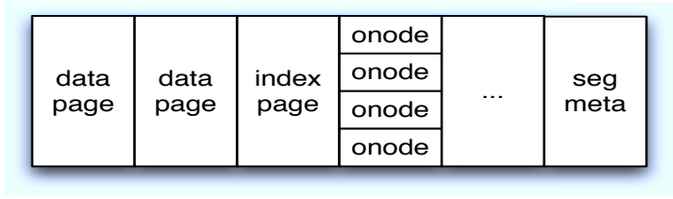
Object-based Flash
Data placement policy

- Goal: maximize the performance and the lifetime of Flash

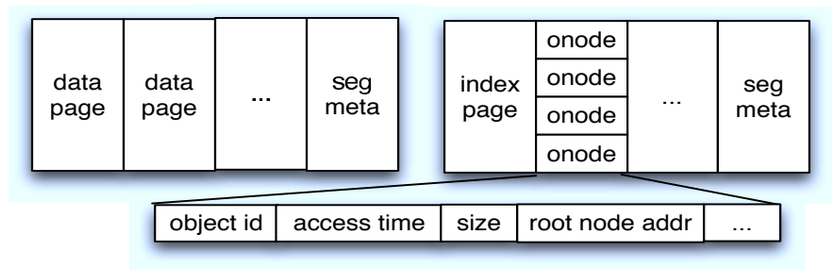
- A log-structure is typically used for Flash
  - wear-leveling
  - out-of-place update

- Separate frequently accessed data
  - metadata: object metadata, indices
  - access time

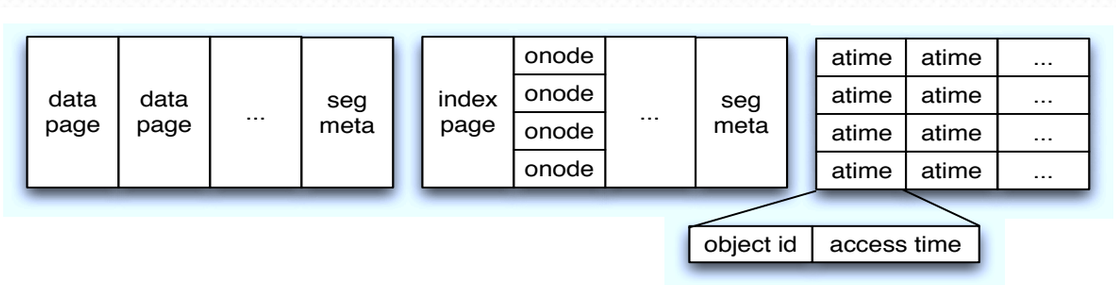
- Cleaner threshold
  - lower the priority of metadata segments



(a) Typical log-structured layout.



(b) Separation of data and metadata.



(c) Separation of data, metadata and access time.

Object-based Flash Index Structure

- Goal: improve space efficiency
  - reduces the number of I/Os
  - increase the lifetime of Flash

- An index structure designed for a Flash-aware file system can be used
  - OSD and Flash-aware file system have the same level of information
  - SSD uses heuristics to extract those information from the request patterns
    e.g. detecting sequential writes in FAST

- Wandering Tree
- Two optimizations
  - Extent-based allocation for large objects
  - Write buffer for small objects

Object-based Flash Wear-leveling and Cleaning

- Wear-leveling is to maintain the wear-level of each erase unit and use them evenly
  - use a log-structure and a cleaner

- Selecting target segments to be cleaned
  - Erase count
  - Age of the segments (current time - erased time)
  - The number of live data pages in each segment
  - Type of segments: data, metadata, access time
    - e.g. metadata segments tend to be updated more frequently than data segments -> skip selecting metadata segments until they contains very small # of live pages

Object-based Flash Advanced Features

- Object-level Reliability
  - Per-page error correction code can only detect and correct a certain number of bits
    - cannot protect against misdirected writes, whole page failures
  - Detection
    - Use algebraic signature to generate a fingerprint of each page (stored in the spare area)
  - Correction
    - Per-object parities
      - parities for all data pages
      - parities for all index pages
      - a copy of an onode
    - Users can adjust the number of parities and replicas to meet their requirements
- Consistency check
  - misdirected writes:
    - read all data pages belonging to an object, and compare with parities
  - page failures
    - recover from temporal/permanent page failures regardless the number of bit-flips within a page

Object-based Flash Advanced Features

- Object-level compression and encryption
  - Selective-compression/encryption
    - infer the type of an object, and determine if a compression or encryption is required
  - Right order: compress followed by encrypt
- Client library
  - Allows users to customize the OSDs by providing hints to the devices e.g. bypassing VFS for small objects
  - File system can also use hints: embedding inodes in onodes

Implementation Object-based Flash

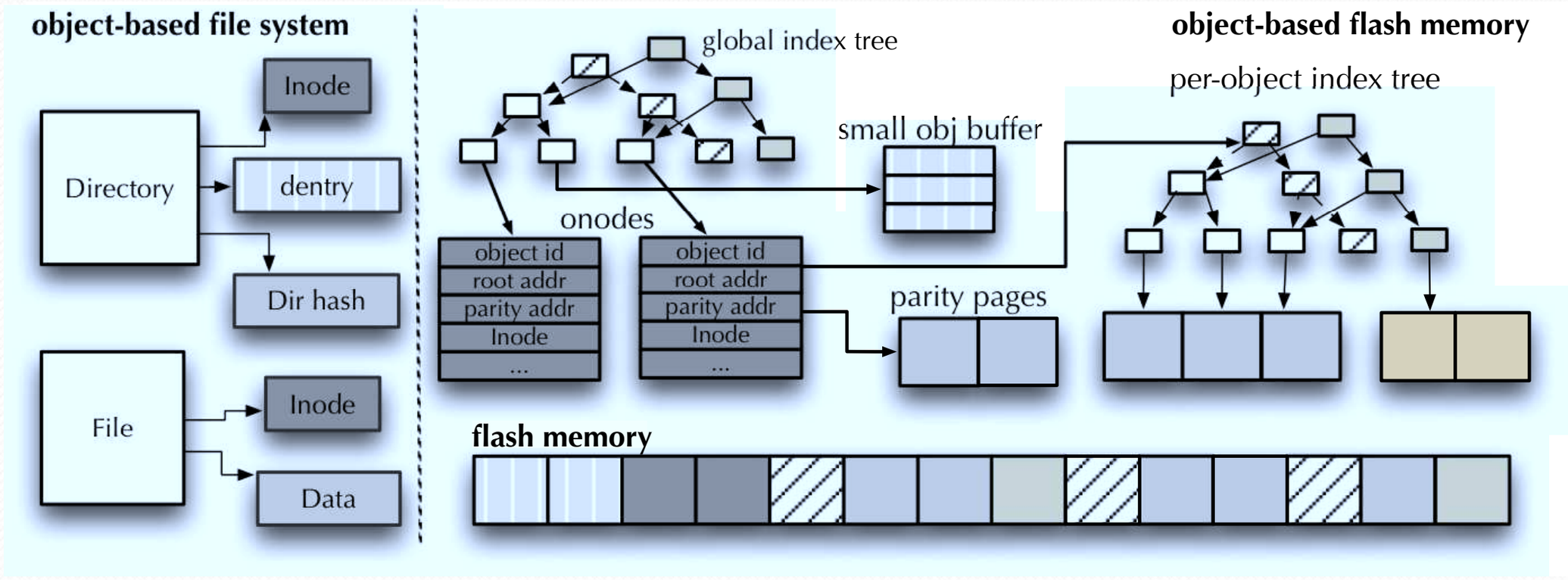
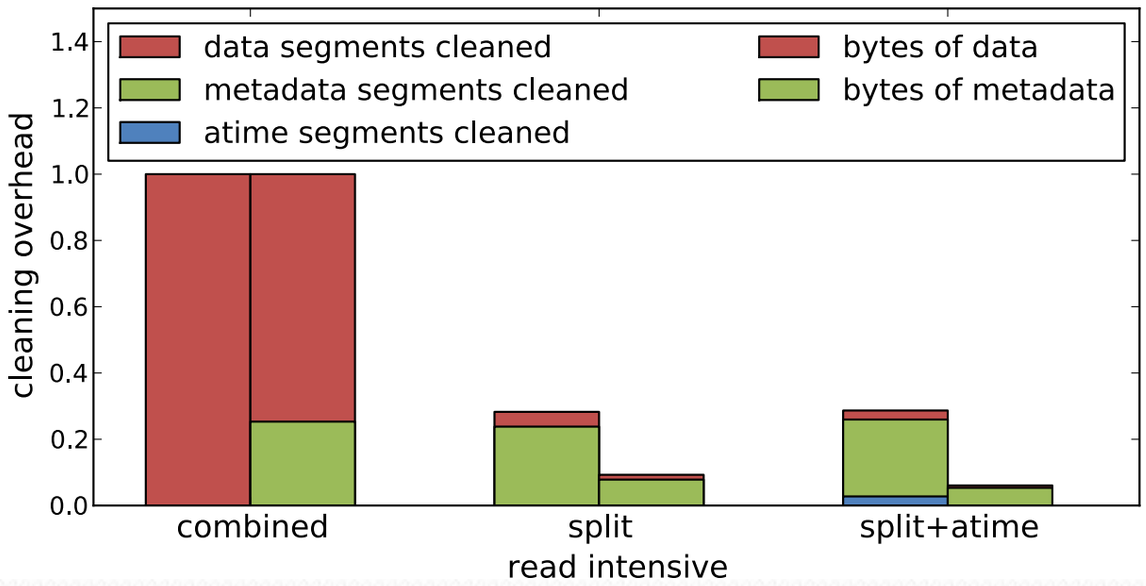


Figure 4. The implementation of Object-based flash memory

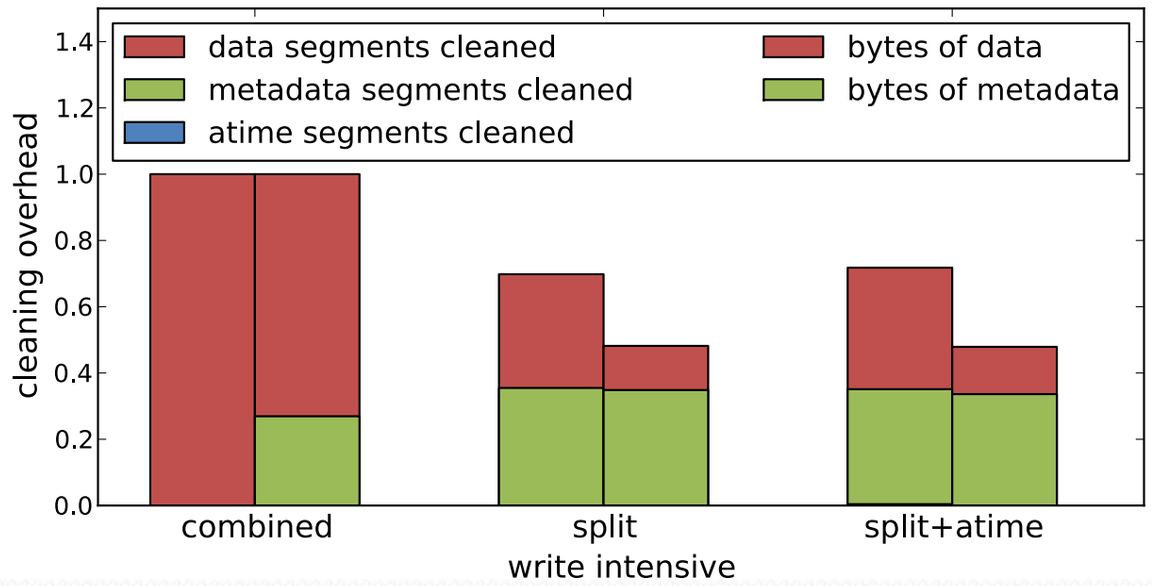
- Hash-directory
- Small object buffer
- Three data placement policies
- Next-K algorithm
- Wandering tree with extents
- Embedding inodes in onodes
- Object-level Reliability

Evaluation
Data placement policies

Postmark Benchmark



(a) Read-intensive workload

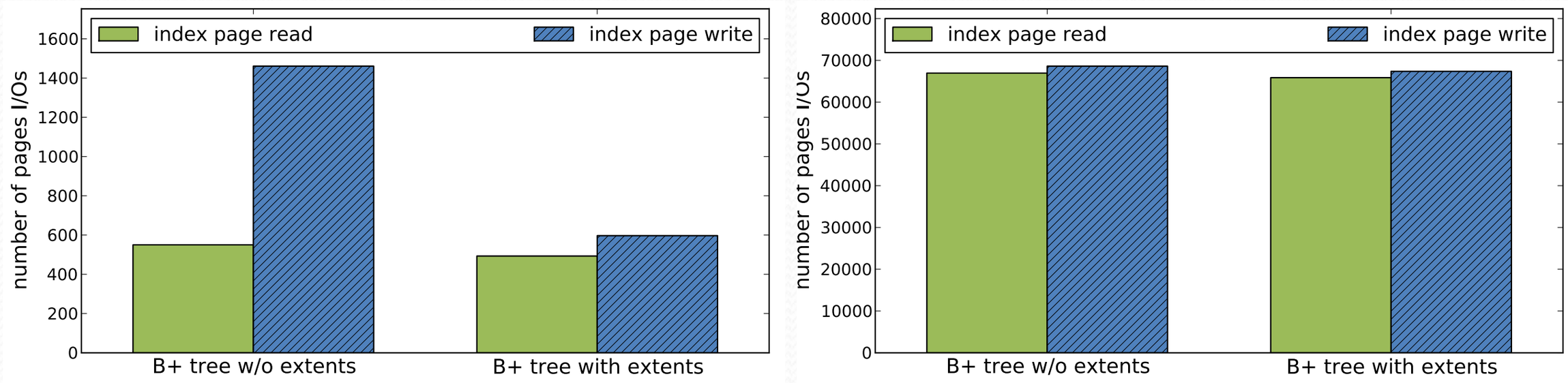


(b) Write-intensive workload

- object inode and index nodes are frequently written to update access time
- access times and modified fields are written together

- Left bar: the total number of segment cleaned
- Right bar: the number of bytes copied during cleaning

Evaluation
Extent-based Allocation



(a) Largefile benchmark

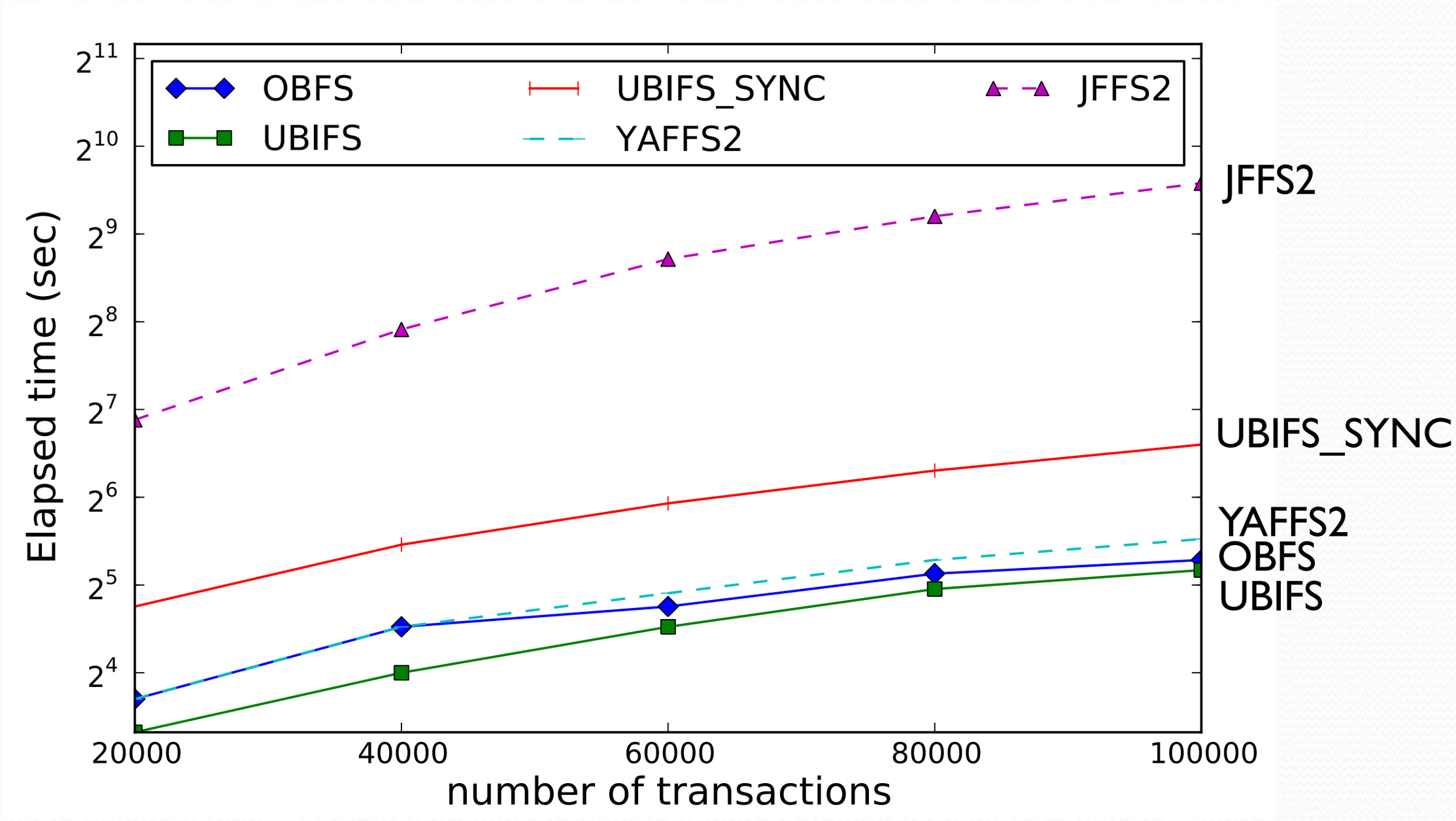
(b) Postmark benchmark

Figure 6. Effects of an extent-based allocation

Evaluation
Effects of Informed placement

	w/o inode_embed	w/o small obj buf	with all
pages read	378101	389165	364142
pages write	92006	91606	66615
seg write	718	714	288
seg clean	240	235	65

Evaluation
Overall Performance



Conclusion

- The object-based model
  - enables SCM devices to overcome the disadvantages of the current interfaces
  - provides new features such as object-level reliability and compression

- Object-based SCM prototype
  - Implemented in the 2.6 Linux kernel to explore the design issues of the object-based Flash

  - Separating frequently accessed data can significantly affect the cleaning overhead
  - Extent-based allocation works well for large objects
  - Achieve compatible performance with other flash-aware file systems

Fast-SSDs

- Fast-SSDs
  - Use PCI-E to avoid the limitation of block-interface
  - Move some parts of a FTL to the host
    - Obtain more information from the running FS, and use them to optimize the FTL
    - Translation Layer
    - Cache

- Advantage
  - High Performance

- Disadvantage
  - Platform, FS dependent(need for a module per each FS)
  - Two translation tables
  - Memory footprint
  - Less portability

Comparison of the three models

	Direct Access Model	FTL-based Model	Object-based Model
Performance			
Portability			
Design Flexibility			
Support for new types of SCM			
System change			
Legacy FS support			

Evaluation
Effects of cleaning threshold