

An Update-Aware Disk Access Interface for High-Throughput Database Indexes

Tzi-cker Chiueh

**Experimental Computer Systems Lab
Stony Brook University**

Random Index Update Workload

- **Block-level continuous data protection**
 - ◆ Logs every disk write operation to disk
 - ◆ Inserts entries into two indexes:
 - Timestamp+LogicalBlockAddress (with locality)
 - LogicalBlockAddress+Timestamp (no locality)
- **Very-large-scale data deduplication**
 - ◆ 1PByte backup server using 4KByte block as unit of deduplication → 250B entries of block fingerprint
 - ◆ Every disk block backed up needs to access this fingerprint index and update some reference count

Conventional Disk Access Interface

- Disk read: `read(target_disk_address, dest_buffer, len)`
Optimization: read ahead or prefetching
- Disk write: `write(source_disk_address, src_buffer, len)`
Optimization: logging and write behind
- Typical policy
 - ◆ Disk reads are serviced mostly synchronously
 - ◆ Disk reads are serviced at a higher priority than disk writes
- Disadvantage: reads in write-after-read **update** operations cannot be serviced in the same asynchronous way as write operations

Update-Aware Interface

- A new interface:

`update(target_disk_address, in_buffer, ptr_update_fn)`

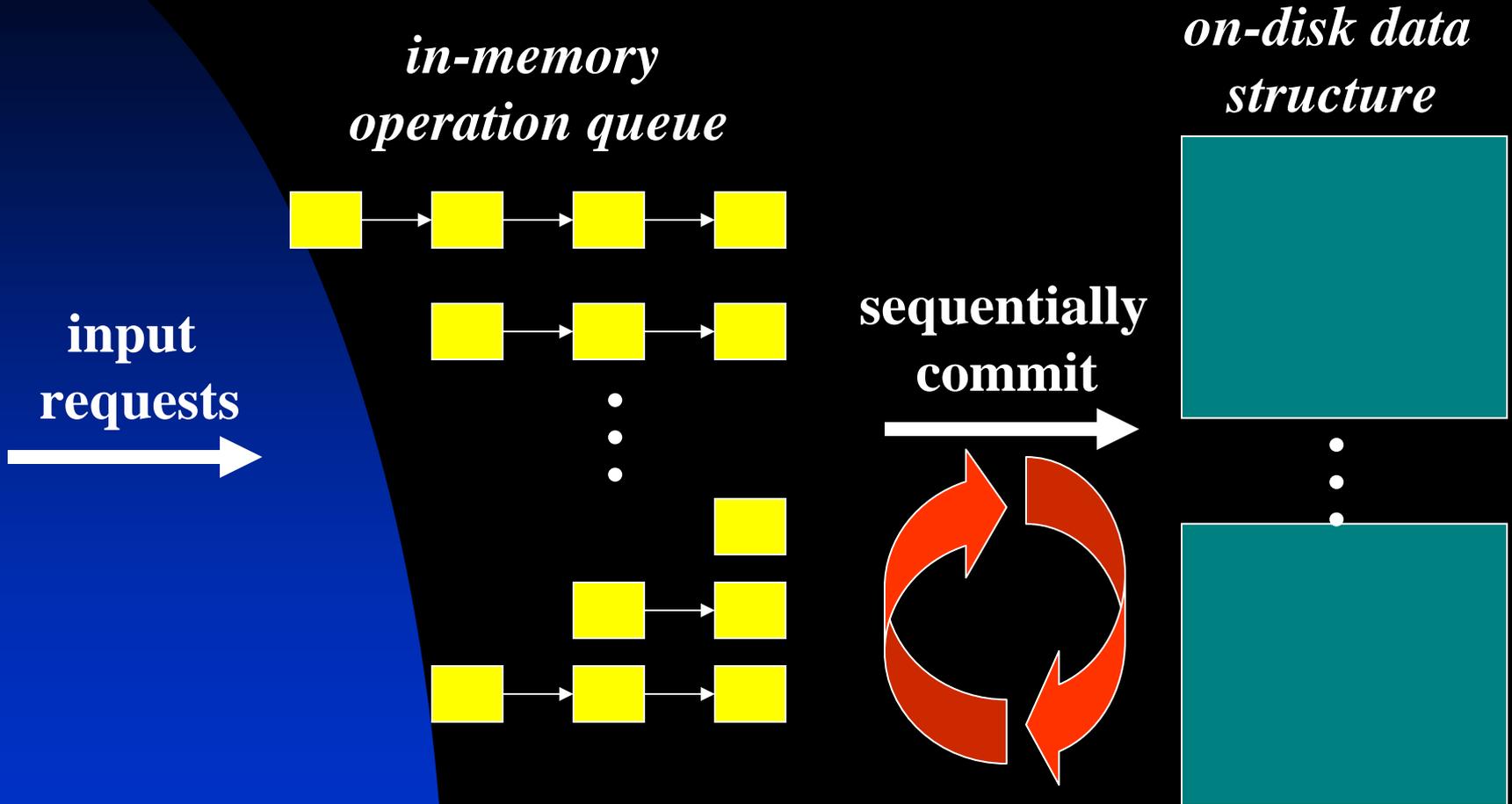
Modify a **disk block** by applying an **update function** (insert, delete, modify) using an **input buffer** as the argument

- Why is this interface useful
 - ◆ Provides more scheduling flexibility because reads in update operations can be serviced asynchronously
 - ◆ Enables higher batching efficiency for disk write operations: The disk scheduler can **directly** invoke application-specific modifications on disk blocks

Batching Operations Using Sequential Commit

- Given an update operation
 - ◆ Log the update operation
 - ◆ Buffer it in a queue for batching
 - ◆ **Sequentially** commit them to disk
- Advantages:
 - ◆ More efficient use of physical memory, which is used for batching operations rather than caching disk blocks
 - ◆ Disk accesses used in committing updates are sequential
 - ◆ No impact on read performance
 - ◆ Applicable to inserts, deletes and modifications

How It Works



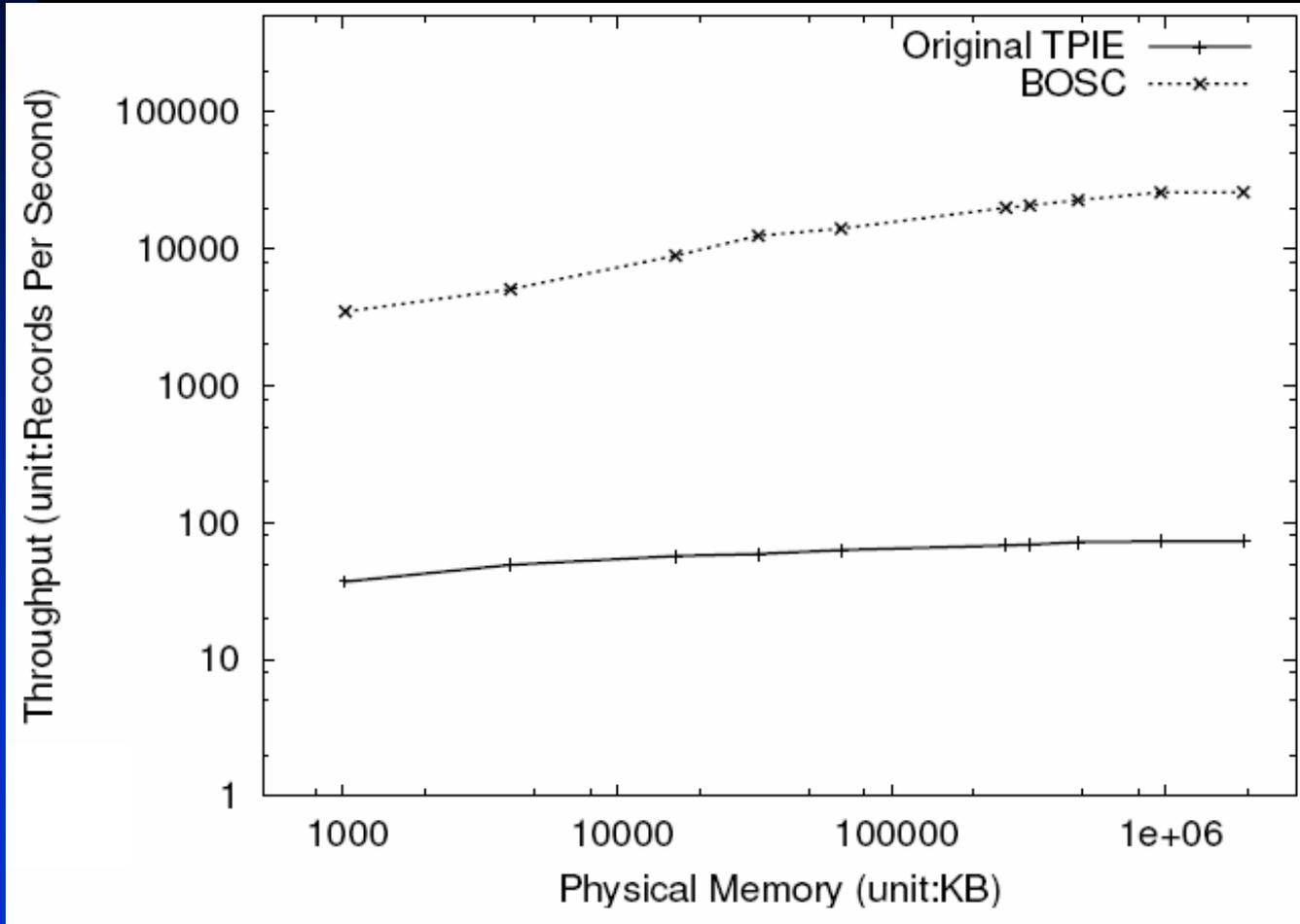
Low-Latency Space-Efficient Disk Logging

- Disk geometry-aware Disk Array Logging
 - ◆ Implements “write to where the disk head happens to be” semantics with logical disk write batching
 - ◆ Supports multiple physical writes per disk track
 - ◆ Leverages multiple disks to mask track-to-track seek delays
- Performance (five 7200RPM IDE disks connected through Promise Ultra100 TX2 IDE controller)
 - ◆ 12500 4KB logical disk writes with 1.8 msec average latency for each logical write
 - ◆ Space utilization: 70%

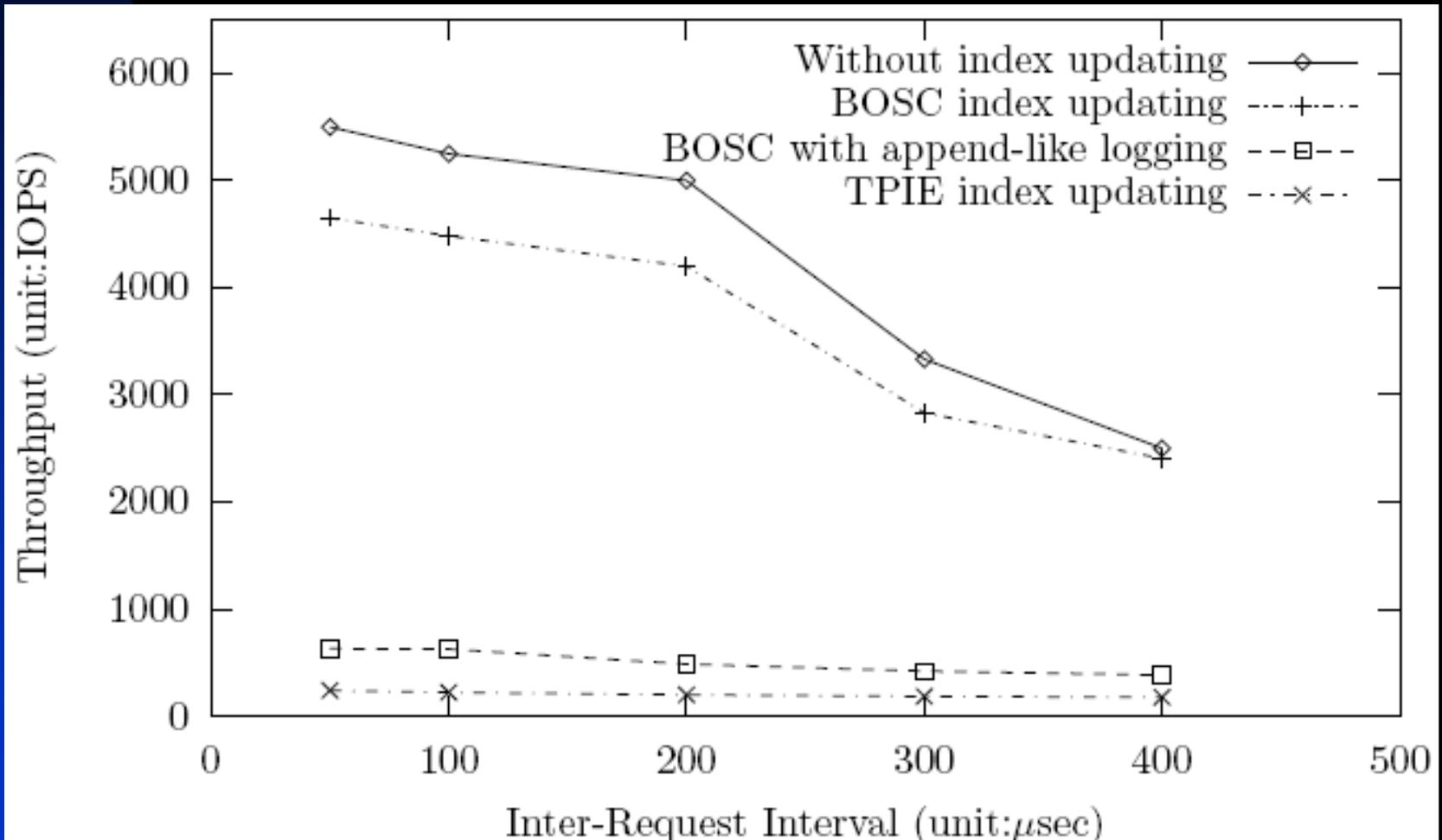
Example

- A B-Tree index facing an input workload with random record insertions and updates
 - ◆ Each B-tree index update request is implemented as a disk update operation of a leaf index page
 - ◆ Disk update operations against leaf index pages are batched and committed sequentially
 - ◆ Latency is not compromised because of low-latency logging
 - ◆ Throughput is optimized because of batching and sequential commit
- Same principle can be applied to other database index structures such as Hash Table, R Tree, Kd tree, etc.

BOSC vs. TPIE



BOSC-based Block-Level CDP



Summary

- Insight: Exposing an entire disk block update operation to the disk scheduler provides more scheduling flexibility (asynchronous read) and enables higher disk access efficiency (sequential commit)
- Research questions:
 - ◆ How far can this approach go?
 - ★ Extension to network storage system and other higher-level operations
 - ★ Interactions with concurrency control and failure recovery
 - ◆ How to integrate BOSC with data-intensive computing infrastructure such as Apache Hadoop and Column-based DBMS

Questions?

Thank You!

chiueh@cs.sunysb.edu

<http://www.ecsl.cs.sunysb.edu>