Multicore support in databases

Panu Silvasti

TKK

Outline

- Building blocks of a storage manager
- How do existing storage managers scale?
- Optimizing Shore database for multicore processors

Building blocks of a storage manager

Buffer pool manager

- Buffer pool is a set of frames, each of which can hold one page of data from disk
- When an application requests a database page not currently in memory the buffer pool manager must fetch it from disk while the application waits
- Applications can pin in-use pages in the buffer pool to prevent pages being evicted too soon
- The buffer pool manager (and log manager) are responsible that modified pages are flushed to the disk

Buffer pool manager (2)

- Buffer pools are typically implemented as hash tables
- Operations within the hash table must be protected from concurrent structural changes
- Hash collisions and hot pages can cause contention among threads for the hash buckets
- The buffer pool must flush dirty pages and identify suitable candicates for eviction without impacting requests from applications

Page latches

- Each page has a reader-writer lock called a latch
- Each operation acquires the latch in either read or write model before accessing the page
- Very hot data (such as metadata) are accessed so frequently that even compatible read operations can serialize attempting to acquire the latch
- Latch operations are typically very short, but when transaction blocks for I/O it can hold a latch for several milliseconds

Lock manager

- **Database locks enforce logical consistency at the transaction level**
- **Two-phase locking (2PL): transaction may not acquire any new locks once it has released any**
  - ensures that all transactions execute in some serial order
  - can restrict concurrency
- **Hierarchical locks balance the overhead of locking with concurrency**

Lock manager (2)

- **Lock manager maintains a pool of locks and lock requests (similar to buffer pool)**
- **The hash table is likely to have longer chains, since the number of active locks changes a lot**
  - more contention for buckets
- **Hierarchical locking results in extremely hot locks, which most or all transactions acquire**
  - this contention can serialize transactions

Log manager

- All database operations are logged to ensure that they are not lost if database crashes
- The log allows the database to roll back modifications in case of transaction abort
- Most storage managers follow ARIES scheme [3]
- The database log is a serial record of all modifications to the database and forms a potential performance bottleneck

Transaction management

- The storage manager must maintain information about all active transactions in order to coordinate services such as checkpointing and recovery
- Checkpointing allows the log manager to discard old log entries saving space and shortening recovery time
- No transaction may begin or end during checkpointing, producing a potential performance bottleneck

Free space and metadata management

- The storage manager must manage disk space efficiently
- Pages which are scanned frequently should be allocated sequentially in order to improve disk access times
- Table reorganizations may be necessary to improve data layout on disk
- Storage manager must ensure that changes to metadata and free space do not corrupt running transactions, while also servicing high volume of requests, especially for metadata

How do existing storage managers scale?

Experimental environment

- Sun T2000 (Niagara) server running Solaris 10
- Multi-core architecture with 8 cores at 1GHz
- Each core supports 4 thread contexts (total of 32 OS visible processors)
- The 8 cores share a common 3MB L2 cache
- 16GB of RAM
- Raid array of 11 15kRPM disks
- All database data resides on the RAID-0 array
- Database log files are sent to in-memory file system

Storage engines tested

- **PostgreSQL v8.1.4**
  - Sun distribution of PostgreSQL optimized for T2000
- **MySQL v5.1.22-rc**
  - InnoDB transactional storage engine
- **BerkeleyDB v4.6.21**
  - client drivers link against the database library and make calls directly into it through the C++ API avoiding the overhead of SQL front end
  - depends on the client application to provide multithreaded execution
  - only storage engine without row-level locking; its page-level locks can severely limit concurrency in transactional workloads

Storage engines tested (2)

- **Shore**
  - developed at the University of Wisconsin in the early 1990's
  - client driver code links directly to the storage manager and calls into it using the API provided for value-added servers
  - the client code must use the threading library that Shore provides
- **Shore-MT**
  - multi-threaded version of Shore database (explained later)
- **Database X**
  - commercial database manager
  - licensing restrictions prevent revealing the vendor name

Benchmarks

- Throughput (transactions per second)
- Scalability (how throughput varies with the number of active threads)
- Ideal engine would be both fast and scalable

Record insertion test

- **Repeatedly inserts records into database table backed up by a B-tree index**
- **Each client uses a private table**
- **Transactions commit every 1000 records**
  - except MySQL every 10000 records, since there was a performance bottleneck
  - record insertion stresses primarily the free space manager, buffer pool and log manager
  - test is entirely free from logical contention

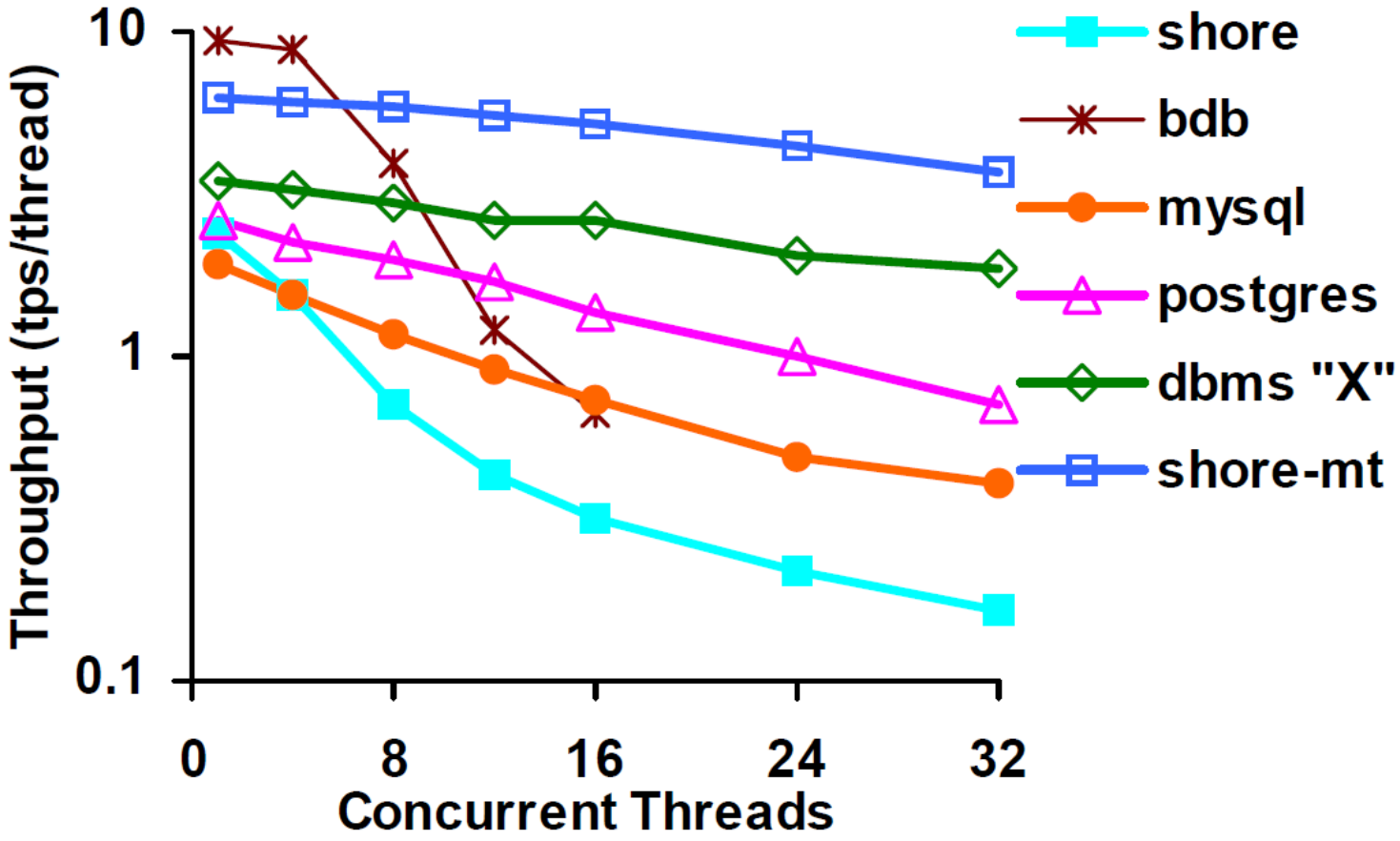
TPC-C Payment test

- Workload of short transactions arriving at high frequency
- Payment transaction updates the customer's balance and corresponding district and warehouse statistics
- It is the smallest transaction of the TPC-C transaction-mix, reading 1-3 rows and updating 4 others
- One of the updates made by Payment is to a contended table
- Payment stresses the lock manager, log manager and B-tree probes

TPC-C New Order test

- Medium-weight transaction which enters an order and its line items into the system
- Updates customer and stock information to reflect the change
- Inserts a dozen records in addition to reading and updating existing rows
- Stresses B-tree indexes (probes and insertions) and lock manager

Insert-only benchmark



TKK

Bottlenecks

- **PostgreSQL**
  - contention of log inserts causes threads to block
  - calls to malloc add more serialization during transaction creation and deletion
  - transactions block trying to lock index metadata
  - these bottlenecks only account for 10-15% of total thread time

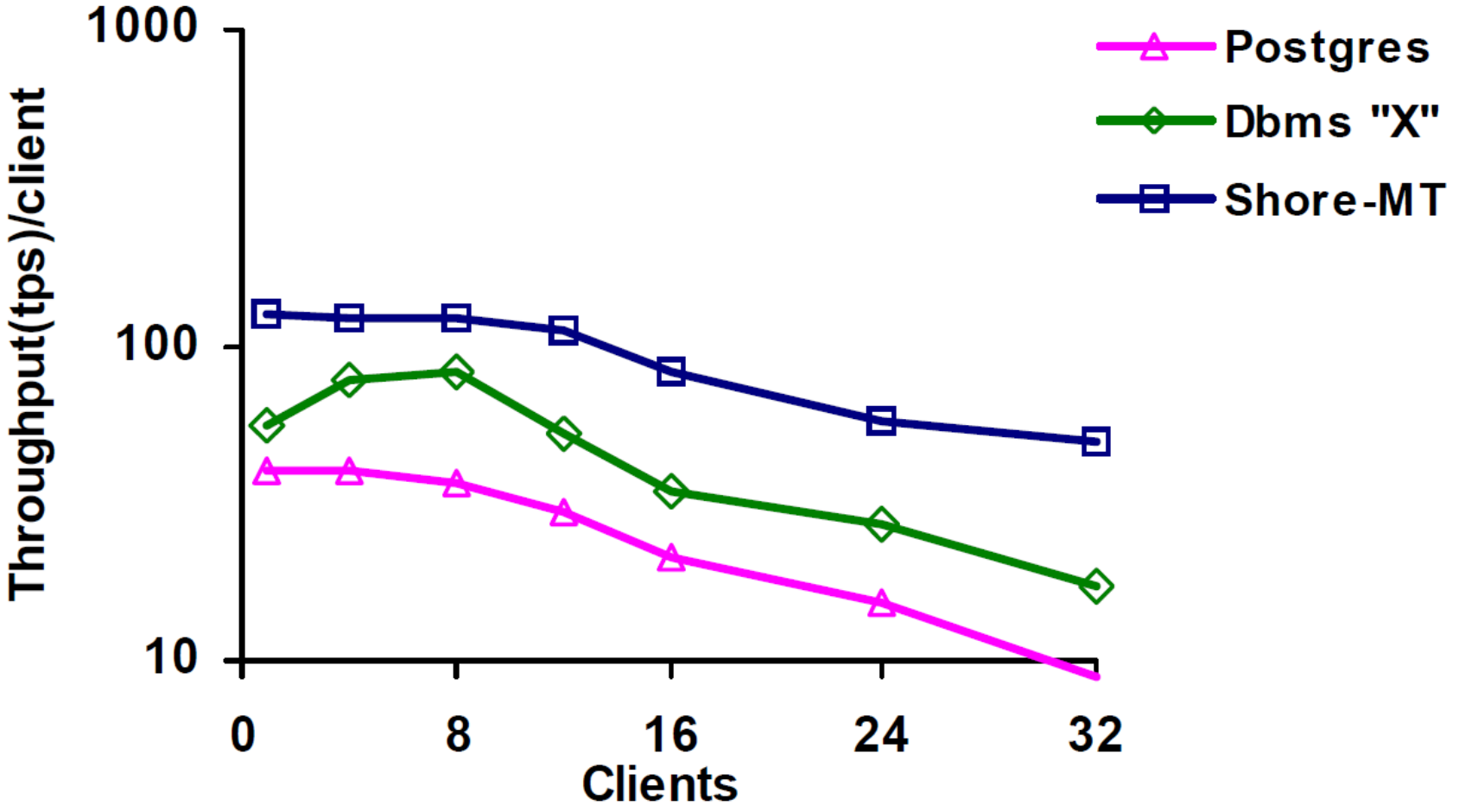
Bottlenecks

- **MySQL**
  - in interface to InnoDB (srv_conc_enter_innodb function) threads remain blocked 39% of total execution time
  - log flushes: 20% of total execution time
  - spends also some time on two malloc-releted functions
    - take_deferred_signal and mutel_lock_internal

Bottlenecks

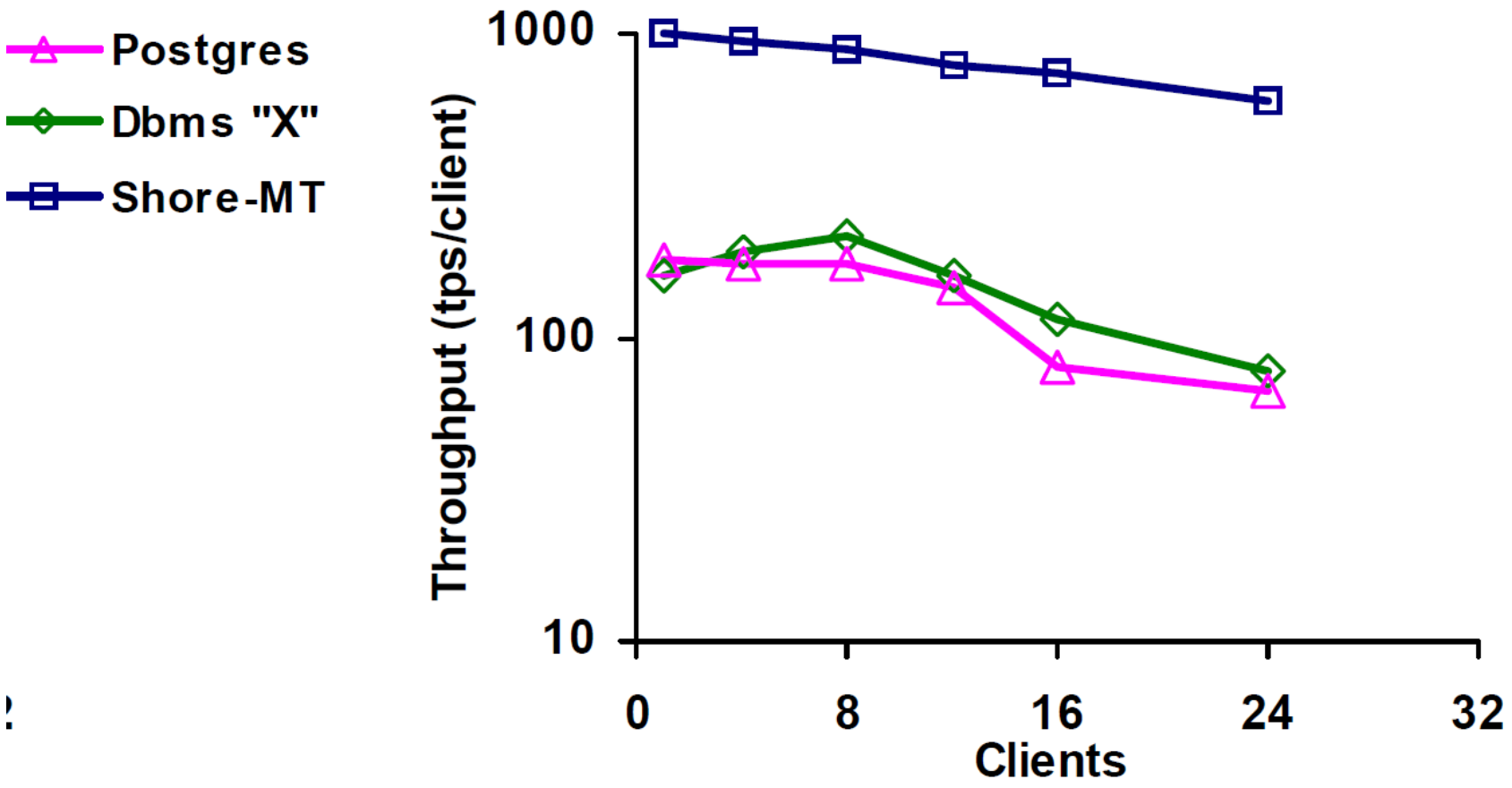
- **Berkeley DB**
  - spends the majority of its time (80%) on testing the availability of mutex or trying to acquire a mutex
    - db_tas_lock and _lock_try functions
  - these form test-and-test-and-set mutex primitive [3] (read from cache and then try test-and-set), which is efficient on low contention but fails miserably on high contention
  - also page-level locking imposes scalability problems
    - in high contention BDB spends most of its time trying to acquire the latches for tree probes

New Order transaction (three best DBMS)



TKK

Payment transaction (three best DBMS)



TKK

Optimizing Shore database for multicore processors

TKK

Optimizing Shore

- Completely ignored single-thread performance and focused only on removing bottlenecks
- However, Shore-MT achieved nearly 3 times better single-thread performance than Shore
- While Shore-MT and database X scale up well, profiling indicates that both face looming bottlenecks in log inserts
- However, Shore-MT proves that even the largest bottlenecks can be addressed using the principles detailed in upcoming slides

Optimizing Shore

- **Shore was chosen for optimization because it supports all the major features of modern database engines**
  - full transaction isolation
  - hierarchical locking
  - a CLOCK buffer pool [5] with replacement and pre-fetch hints
  - B-tree indexes
  - ARIES-style logging and recovery
- **Optimization process by profiling dominant performance bottlenecks and addressing them**
  - after addressing a dominant bottleneck, scalability would either increase or a previously minor bottleneck would take its place
  - this step was repeated
- **In the absence of external bottlenecks such as I/O, virtually all bottlenecks centered on the various critical sections**

First optimization attempt

- **Profiling identified a contended pthread mutex in the free space manager as the primary bottleneck**
- **Replaced pthread-mutex with a test-and-test-and-set mutex**
  - improved single-thread performance
  - scalability dropped
- **Replaced mutex with a scalable MCS mutex [2]**
  - scalability improved noticeably, but the critical section remained contended

First optimization attempt (2)

- **Noticed that free space manager acquired a page latch while holding the mutex**
- **Refactored the code and moved the latch acquire outside critical section**
- **Reduced single-thread performance by 30%, but increased performance by 200% with 32 threads**
- **This shows how focusing on performance for few threads can be misleading**
  - one should focus first on scalability, with performance as a second goal

Principles for Scalable Storage Managers

- **Efficient synchronization primitives are critical**
- **Every component must be explicitly designed for scalability or it will become a bottleneck. Common operations must be able to proceed in parallel unless they truly conflict**
- **Hotspots must be eliminated, even when the read data is read mostly**
- **Abstraction and encapsulation do not mix well with critical sections**

Use the right synchronization primitives

- **Every page search that hits the buffer pool requires at least three critical sections**
  - one to lock the hash bucket
  - one to pin the page (preventing evictions while the thread is using the page)
  - request a latch on the page (preventing concurrent modifications on the page's data)
- **Observation**
  - pinned pages cannot be evicted

Use the right synchronization primitives (2)

- **Solution**
  - when a buffer pool search finds a promising page it applies an atomic pin-if-pinned operation to the page, which increments the page's pin-count only if it is non-zero
  - if the page is not already pinned the thread must lock the bucket in order to pin it
  - but if conditional pin succeeds, the page cannot have been evicted because the pin-count was non-zero
  - the thread then verifies that it pinned the correct page and returns it
  - in the common case this eliminates the critical section on the bucket

Shorten or remove critical sections (problem 1)

- Shore uses logical logging for many low-level operations such as page allocations
- Shore must verify that a given page belongs to the correct database table before inserting new records into it
- The original page allocation code entered a global critical section for every records insertion in order to search page allocation tables

Shorten or remove critical sections (problem 1)

- **Observation**
  - page allocations do not change often
  - the information immediately becomes stale upon exit from the critical section
  - once the page has been brought into the buffer pool, Shore must check the page itself anyway
- **Solution**
  - added a small thread-local cache to the record allocation routine, which remembers the result of the most recent lookup
  - this optimization cut the number of page checks by 95%

Shorten or remove critical sections (problem 2)

- The Shore log manager used a non-circular buffer and synchronous flushes
- Log inserts would fill the buffer until a thread requested flush or it became full
- The flush operation would drain the buffer to file before allowing inserts to continue

Shorten or remove critical sections (problem 2)

- **Observation**
  - log inserts have nothing to do with flushes as long as the buffer is not full
  - flushing should almost never interfere with inserts
  - using a circular buffer means that the buffer never fills as long as flushes can keep up with inserts
- **Solution**
  - converted Shore to use a circular buffer
  - protected each operation (insert, compensate, flush) with a different mutex

Shorten or remove critical sections (problem 2)

- **Solution (cont.)**
  - insert and compensate use a light-weight queuing mutex
  - slower flush uses a blocking mutex
  - inserts own the buffer head, flushes own the tail
  - compensations own a marker somewhere between (everything between the tail and marker is currently being flushed)
  - inserts also maintain a cached copy of the tail pointer
  - if an insert would pass the cached tail, the thread must update the tail to a more recent value and potentially block until the buffer drains
  - flush operations acquire the compensation mutex (while holding the flush mutex) just long enough to update the flush marker

Shorten or remove critical sections (problem 2)

- **Solution (cont.)**
  - log compensations acquire only the compensation marker knowing that anything between the flush marker and insertion point is safe from flushing
  - distributing critical sections over the different operations allows unrelated operations to proceed in parallel and prevents fast inserts and compensations from waiting on slow flushes

Eliminate hotspots

- Shore's buffer pool was implemented as a open chaining hash table protected by a single global mutex
- In-transit pages (those being flushed to or read from disk) resided in a single linked list
- Virtually every database operation involves multiple accesses to the buffer pool, and the global mutex became a bottleneck

Eliminate hotspots

- **Observation**
  - every thread pins one buffer pool page per critical section
  - most buffer pool searches (80-90%) hit
  - most in-transit pages are reads thanks to page cleaning (page cleaning algorithm is used to write changed pages to disks before they are selected for replacement)
- **Solution**
  - we distributed locks in the buffer pool in three stages
  - the first stage added per-bucket locks to protect the different hash buckets
  - global lock protecting in-transit and free lists and the clock hand
  - however, each page search had to lock a bucket in order to traverse the linked list
    - -> this remained a serious bottleneck for hot pages

Eliminate hotspots

- **Solution (cont.)**
  - second state replaced the open chaining hash table with a 3-ary cockoo hash table [6]
  - cockoo hashing has two desirable properties:
    - deletions are straight-forward
    - updates and searches only interfere with each other when they actually touch the same value at the same time
  - one drawback: operations cost more because they must evaluate multiple high-quality hashes
  - however, the improvement in concurrency more than offsets the increased cost of hashing by eliminating hotspots on bucket lists

Eliminate hotspots

- **New problems arise**
  - increasing the number of threads still further also increased the number of (page) misses per unit time leading to contention for in-transit lists
    - on every miss the buffer pool must ensure that the desired page not currently in-transit
    - an in-transit page cannot be read back in until data has been written out
    - requests for an in-transit page must block until the read completes
    - the large number of transits in progress at any moment led to long linked list traversals and slowed down miss handling
- **Solution**
  - the third state broke in-transit lists into several smaller ones (128) to cut the length of each list
  - in addition, the buffer pool immediately places in-transit-in pages in a hash table, making it visible to searches and bypassing the transit lists
  - as a result, only true misses search the in-transit-list, which contains only dirty pages currently being evicted

Eliminate hotspots

- **Solution (cont.)**
  - finally, page misses release the clock hand (of the clock buffer pool) before evicting the selected replacement frame or attempting to read in the new data
  - this allows the thread to release the clock hand mutex before attempting expensive I/O
  - as a result, the buffer pool can service many page misses simultaneously, even though an individual page miss is very slow

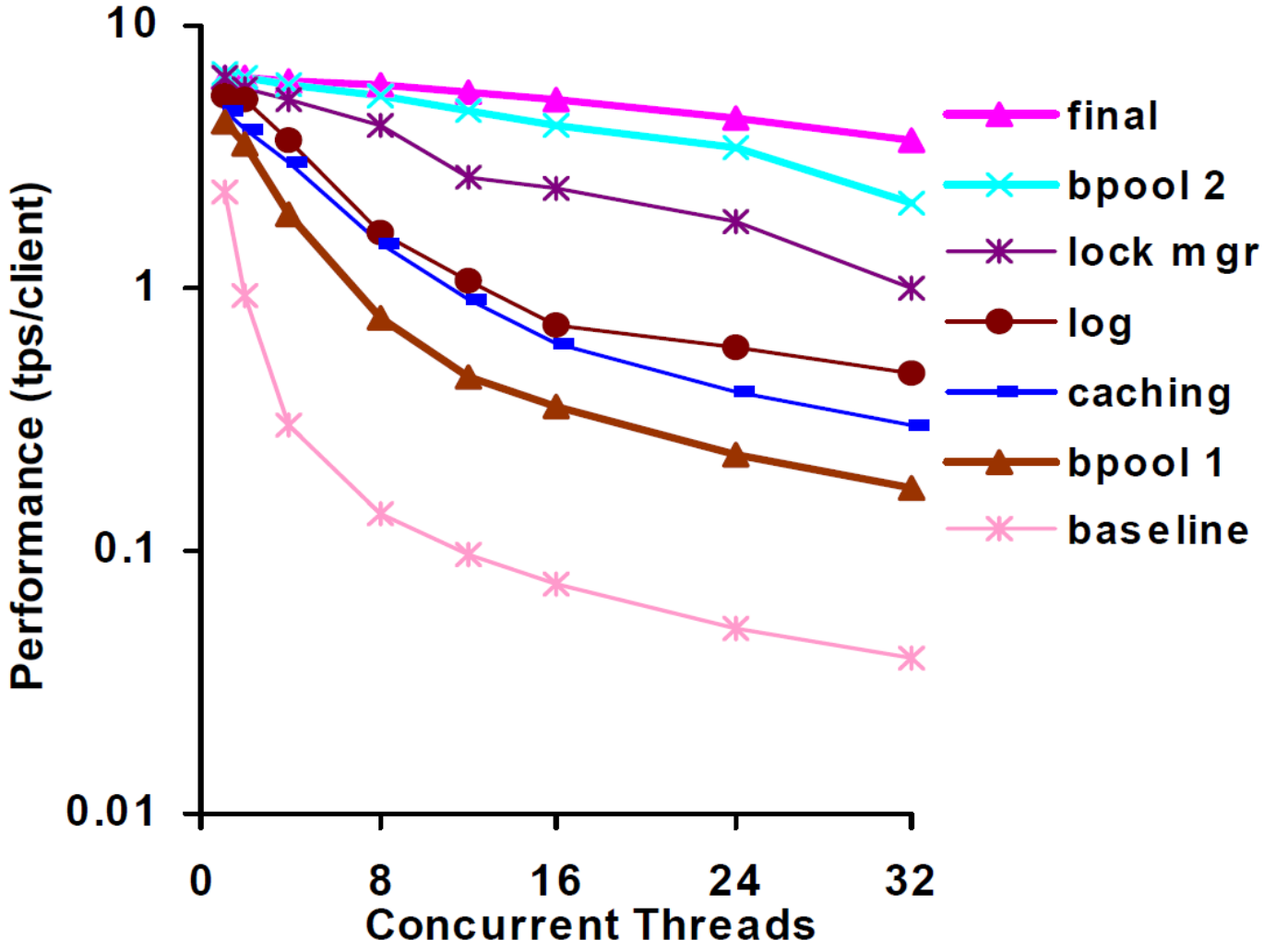
Beware of over-abstraction and encapsulation

- **Problem**
  - log inserts remained a major bottleneck in Shore, even after decoupling inserts from flushes
  - log manager was highly encapsulated and abstracted in ways that cut through critical sections
- **Solution**
  - rewrote the log manager to make encapsulation boundaries surround rather than cut through critical sections
  - threads perform only the minimum work required to allow them to safely insert a log record later
  - copying data and notifying daemon threads is done after releasing the mutex

Beware of over-abstraction and encapsulation

- **Solution (cont.)**
  - redesigned log buffer as an extended queuing lock (MCS queuing lock [2]), combining the functionality of the critical section with the mechanism for protecting it
    - as each thread enters the log buffer's queue it computes which portion of the log buffer it will eventually write to
    - it then hands off this information to its successors in the queue at the same time it exists the critical section

Performance and scalability of optimizations



TKK

Conclusions

- In the multicore era also database storage managers must provide scalability
- Modern open source storage managers are not very scalable
- However, the process of converting Shore to Shore-MT shows that it can be done

Rererences

- [1] Johnson, R., Pandis, I., Hardavellas, N., Ailamaki, A., and Falsafi, B. 2009. Shore-MT: a scalable storage manager for the multicore era. In Proc. EDBT, 2009.
- [2] J. Mellor-Crummey, and M. Scot. “Algorithms for scalable synchronization on shared-memory multiprocessors.” ACM TOCS, 9(1), 1991.
- [3] C. Mohan, “ARIES/KVL: a key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes.” In Proc. VLDB, 1990.
- [4] Wikipedia: http://en.wikipedia.org/wiki/Test_and_Test-and-set
- [5] Wikipedia: http://en.wikipedia.org/wiki/Page_replacement_algorithm
- [6] Wikipedia: http://en.wikipedia.org/wiki/Cuckoo_hashing

TKK

Additional material

Test and Test-and-set locking [4]

- **The main idea is not to spin in test-and-set but increase the likelihood of successful test-and-set by using following entry protocol to the lock:**

```
boolean locked := false // shared lock variable
procedure EnterCritical() {
  do {
    while (locked == true) skip // spin until lock seems free
  } while TestAndSet(locked) // actual atomic locking
}
```

- **Exit critical:**

```
procedure ExitCritical() {
  locked := false
}
```

Clock page replacement algorithm [5]

- **Keeps a circular list of pages in memory, with the "hand" (iterator) pointing to the oldest page in the list**
- **When a page is referenced, a referenced bit is set for that page, marking it as referenced**
- **When a page fault occurs and no empty frames exist, then the R (referenced) bit is inspected at the hand's location**
- **If R is 0, the new page is put in place of the page the "hand" points to**
- **Otherwise the R bit is cleared**
  - Then, the clock hand is incremented and the process is repeated until a page is replaced

Cuckoo hashing [6]

- **N hash functions instead of only one (N is e.g. 2 or 3)**
  - N possible locations in the hash table for each key
- **Insertion**
  - new key is inserted in one of its N possible locations, "kicking out", that is, displacing any key that might already reside in this location
  - displaced key is then inserted in its alternative location (N-1), again kicking out any key that might reside there, until a vacant position is found
- **Lookup**
  - requires inspection of just N locations in the hash table, which takes constant time in the worst case
- **Deletion is straightforward**

MCS queuing mutex [2]

```
type qnode = record
   next :  *qnode                    // ptr to successor in queue
   locked :  Boolean                 // busy-waiting necessary
type lock = *qnode                   // ptr to tail of queue

// I points to a queue link record allocated (in an enclosing scope)
// in shared memory locally accessible to the invoking processor

procedure acquire_lock( L: *lock, I: *qnode )
   var pred:  *qnode
   I->next := nil                    // Initially, no successor
   pred := swap(L, I)                // Queue for lock
   if pred ≠ nil                     // Lock was not free
      i->locked := true              // Prepare to spin
      pred->next := I                // Link behind predecessor
      repeat while I->locked         // Busy wait for lock

procedure release_lock( L: *lock, I: *qnode )
   if I->next = nil                  // No known successor
      if compare_and_swap(L, I, nil)
         return // No successor, lock free
      repeat while I->next = nil     // Wait for successor
   I->next->locked := false          // Pass lock
```


TKK