BP-Wrapper: A System Framework Making Any Replacement Alg. *(Almost)* Lock Contention Free

Xiaodong Zhang

The Ohio State University

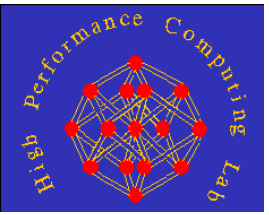
In collaboration with

Xiaoning Ding

The Ohio State University

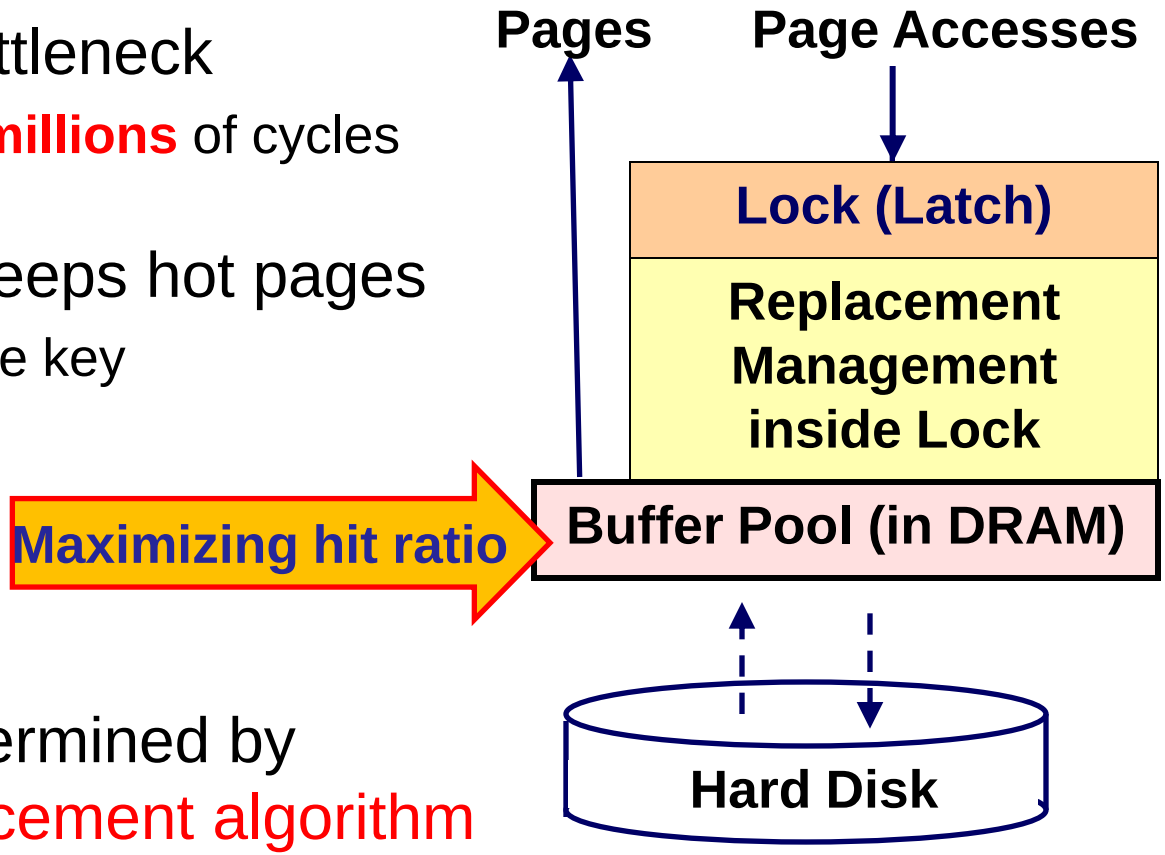
Song Jiang

Wayne State University



Buffer Cache (Pool) is Critical for Data Intensive Workloads

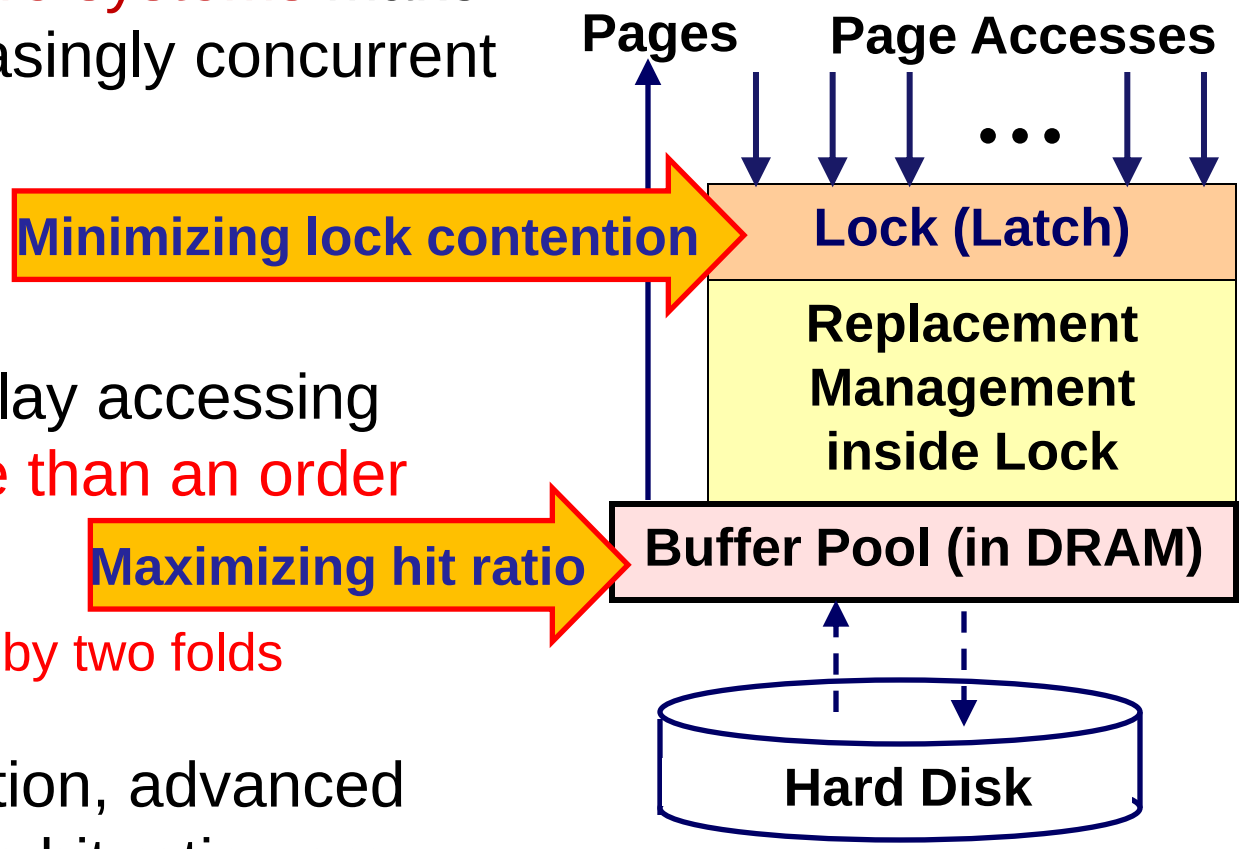
- Disk I/O is a major bottleneck
  - One disk access takes **millions** of cycles
- Buffer cache (pool) keeps hot pages
  - Maximizing hit ratio is the key



- Hit ratio is largely determined by effectiveness of replacement algorithm
  - It determines which pages to be kept and which to be evicted
  - LRU-k, 2Q, LIRS, ARC, …
  - Lock (latch) is required to serialize the update after each page request

Lock Contention is a Serious Bottleneck

- Multi-processor/core systems make transactions increasingly concurrent

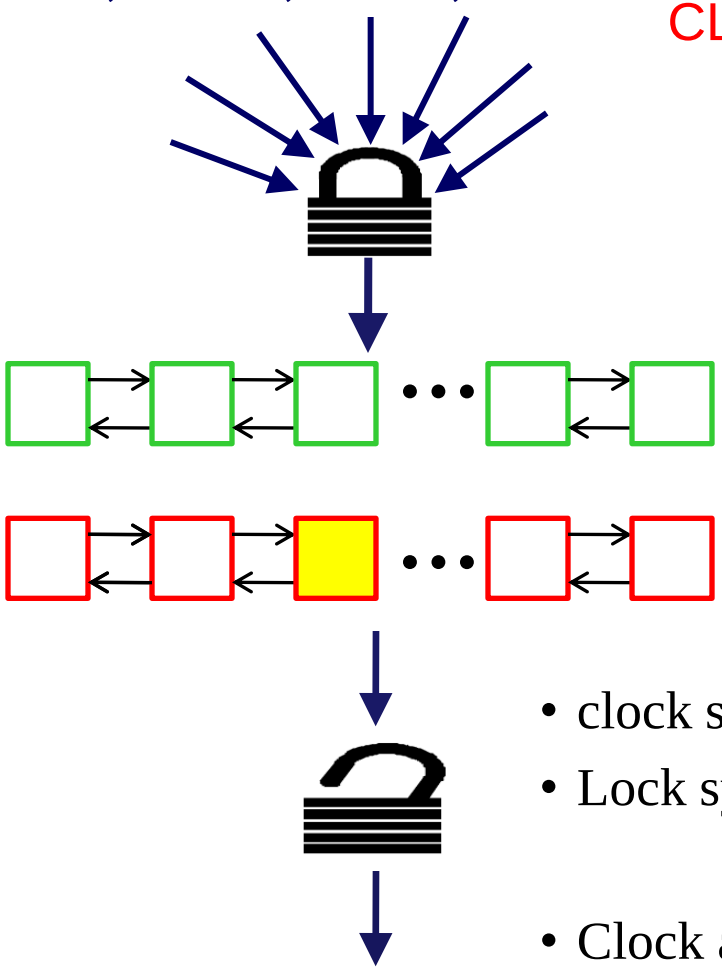


- Lock contention delay accessing buffer pool by more than an order of magnitude
  - Throughput reduced by two folds

- Due to lock contention, advanced replacement of high hit ratios are rarely adopted in practice

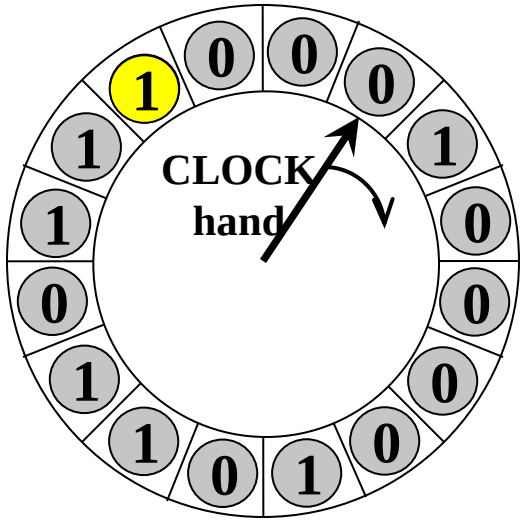
Accurate Algorithms and Their Approximations

LRU, LIRS, ARC, ….



Approximations

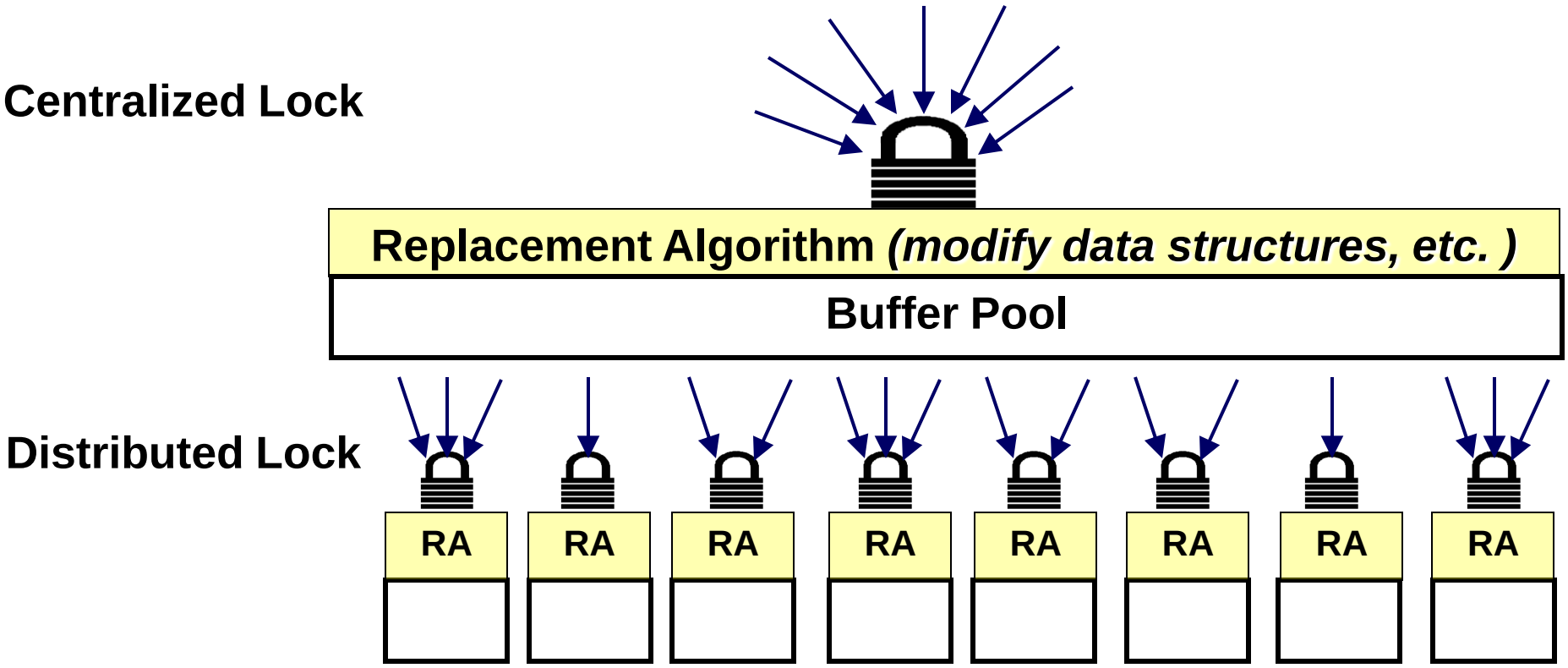
CLOCK (LRU), CLOCK-Pro (LIRS), CAR (ARC)



- clock sets bit to 1 without lock for a page hit.
- Lock synchronization is only used only for misses.

- Clock approximation reduces lock contention at the price of reducing hit ratios.

Distributed Locks are Not Applicable



- Another approximation due to partial access info.: lower hit ratios
- Some replacement algorithms require global information.
- The contention may not be evenly distributed among sub-locks;

Trade-offs between Hit Ratios and Low Lock Contention

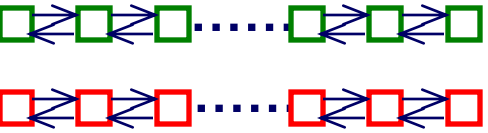
for high hit ratio

LRU-k, 2Q, LIRS, ARC, SEQ, ….

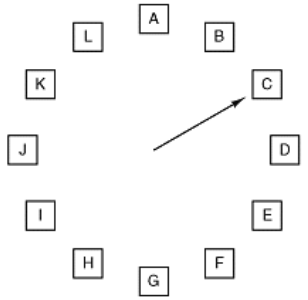
Our Goal: to have both!

for high scalability

CLOCK, CLOCK-Pro, and CAR



Lock Synchronization	Low Lock Synchronization
modify data structures	*Update page metadata*



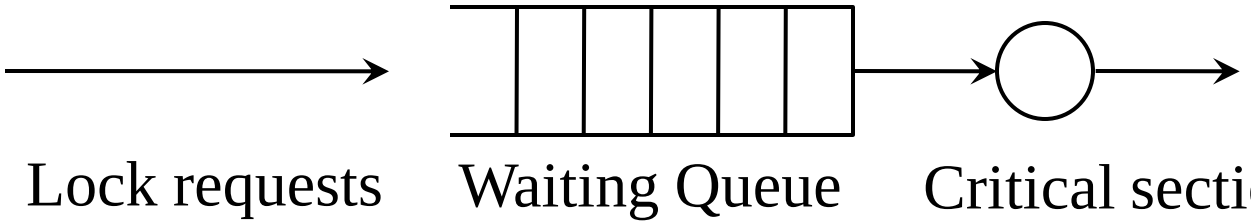
- Clock-based approximations lower hit ratios (compared to original ones).
- The transformation can be difficult and demand great efforts;
- Some algorithms do not have clock-based approximations.

Outline

- Identifying critical issues in buffer pool management
  - Where do we lose the accuracy?
  - What are the sources of overhead?
- BP-Wrapper: an implementation framework for any replacement algorithms in DBMS.
  - Retain the accuracy of original algorithms
  - Minimize the lock contention
  - Design and analysis
- Performance evaluation
- Conclusion

What are the Overhead Sources in Buffer Pool Management

- Buffer pool management with locks can be roughly modeled by a M/M/1 model



- The probability of a contention event is Prob(q>=2)

Reduce the lock frequency by batching

$$N_{CPU}^2 \left(\frac{T_{lock\_acq\_rel}}{T_{total}} + \frac{T_{op}}{T_{total}} \right)^2$$

Minimize the latency of atomic operations by prefetching

N_{CPU}:number of processors or computing cores

$T_{lock\_acq\_rel}$: the time to acquire a lock plus the time to release it.

T_{op}: the time of atomic operations inside the lock.

T_{total}: $T_{lock\_acq\_rel}$ + T_{op} + average request interval time

BP-Wrapper: A Framework Making Any Replacements *(Almost)* Lock Contention Free

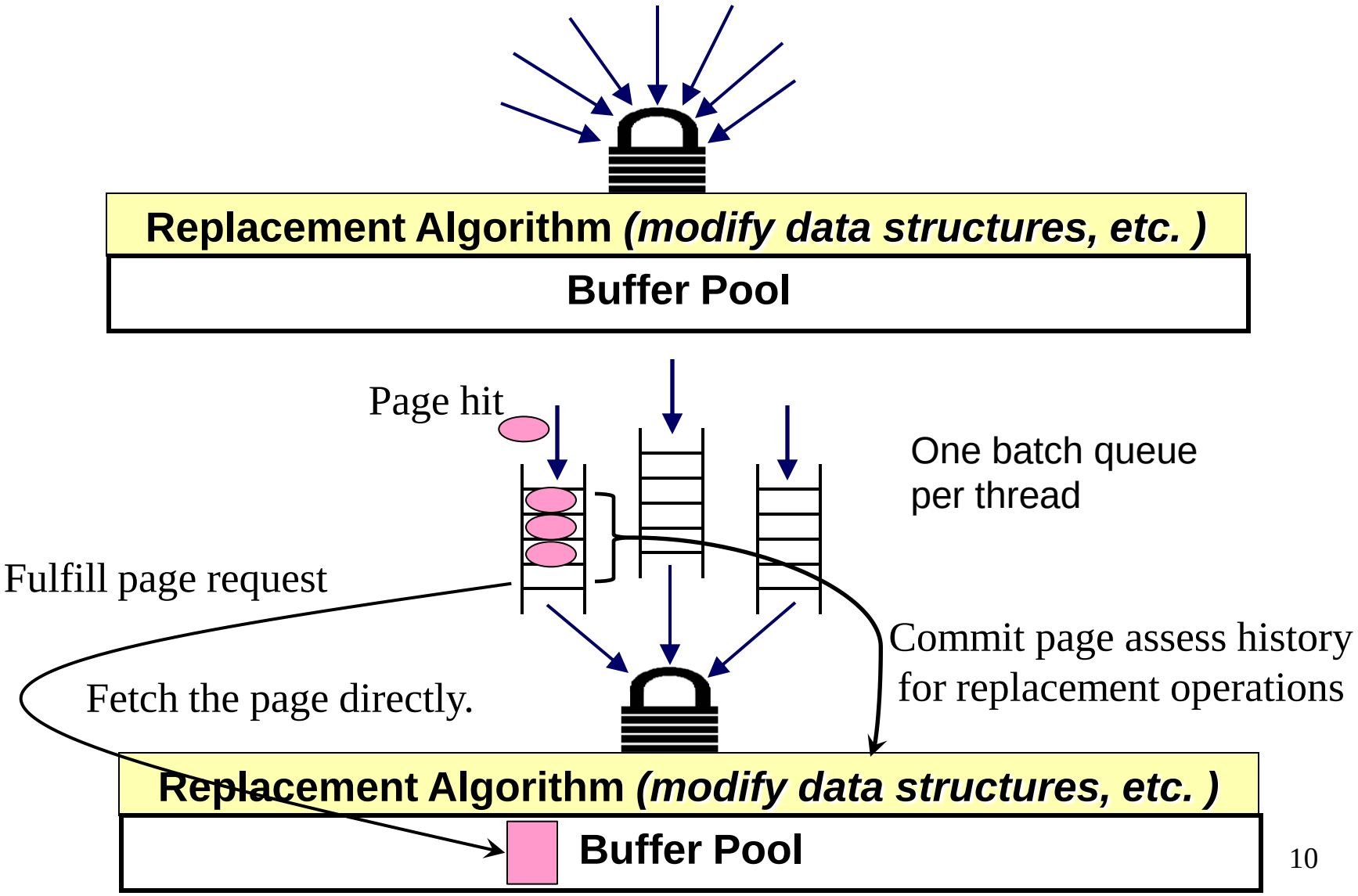
Objectives:

- Do not require changes of existing replacement algorithms
- Retain the advantages of high hit ratios
- Replacement algorithms are as scalable as the clock-based approximations.

Our solutions

- Updating hit request information in batch mode
  - Each mutual exclusive operation **correctly** works for a group of hits
  - **Amortizing lock acquisition costs** among a batch of page accesses
- prefetching
  - Pre-loading the **to-be-used data** inside locks to reduce holding time

Reducing Lock Contention by Batching Requests



Reducing Lock Contention by Batching Requests

What operations before acquiring the lock for hits?

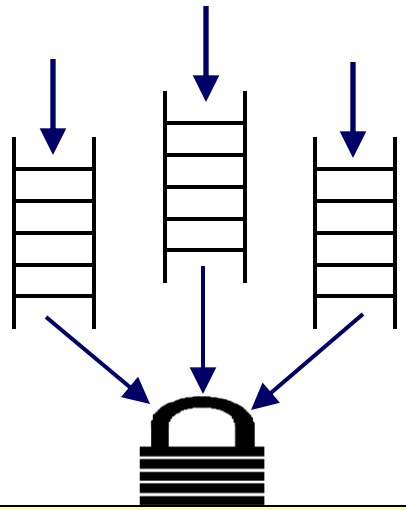
(1) Each access is recorded without a synchronization

(2) As the "hit batch queue" is full, Lock() is acquired

What do we gain?

(1) The clock approximation is eliminated: retain the hit ratio;

(2) Batching process of hits reduces lock contention significantly.



Replacement Algorithm *(modify data structures, etc.)*

Buffer Pool

What operations inside the lock?

(1) Update the data structure for accumulated page accesses

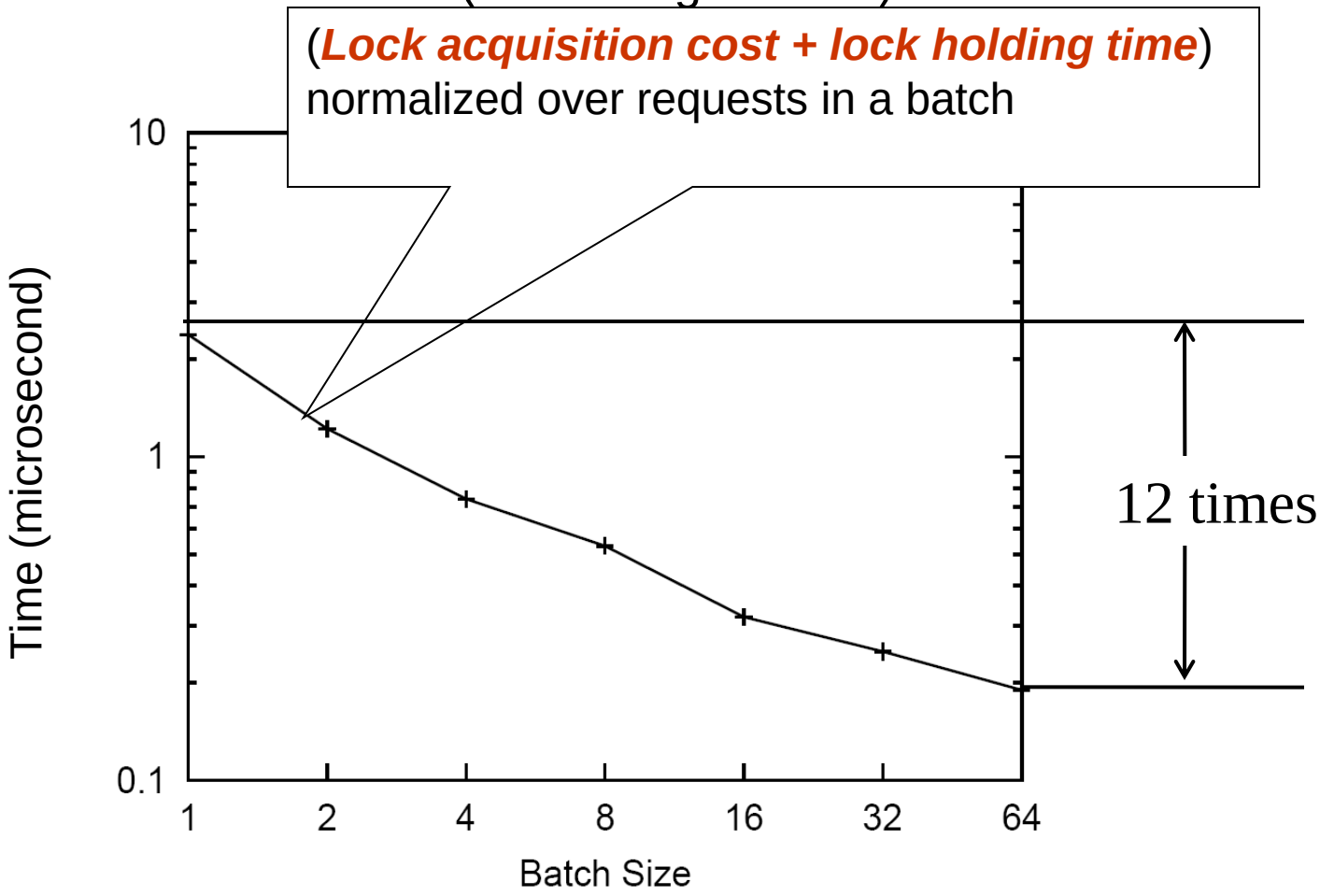
(2) Release lock and empty the queue

Rationale behind Batching in DBMS

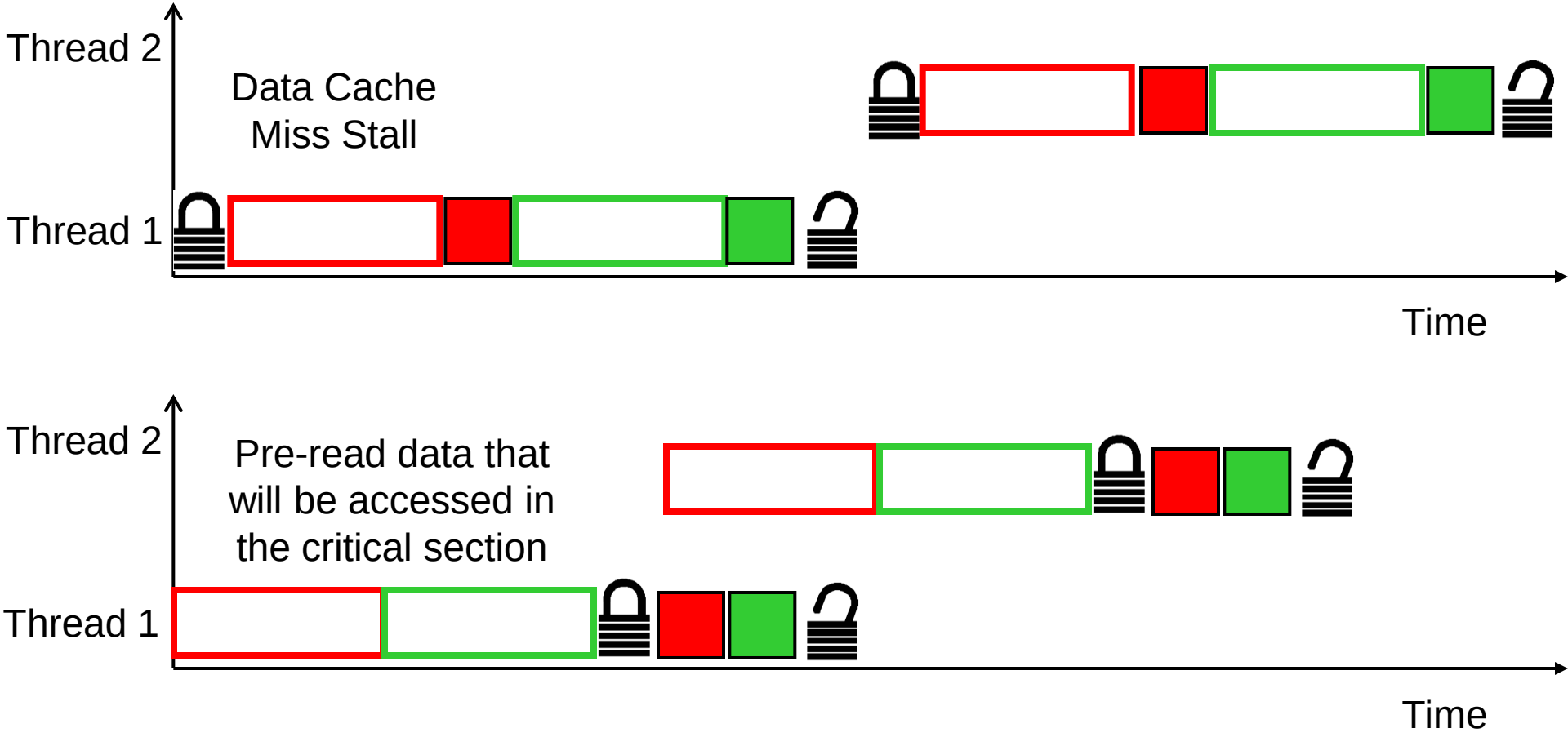
- Batching and fetching pages from buffer are done in parallel
- A small batching makes a huge impact on lock behavior
  - Batch size (64 pages) vs. Buffer (1 million pages in 8GB memory)
  - Lock frequency is reduced exponentially.
- The batching is independent of replacement algorithms
  - No need to change original replacement algorithms
  - Buffer pool is not partitioned, thus global information is preserved
  - Access sequences are preserved
- Batching is applicable in buffer cache/pool management
  - Page request is too fast to batch in virtual memory (VM)
  - Management of VM and buffer cache are separated in Window
  - The management of the two are connected in Linux

Amortized Lock Acquisition Cost

- Hardware: SGI Altix 350 SMP with 16 Itanium 2 processors;
- Software: postgreSQL 8.2.3
- Workload: DBT-1 test kit (simulating TPC-W)



Reducing Lock Holding Time by Prefetching



Rationale behind Prefetching Technique

- Why not prepare to-be-used data outside critical section?
  - The data is unique for replacement , and would not be used outside lock
  - Prefetched data will be invalidated when it is changed by other threads
  - Prefetching is independent of replacement algorithm

- What to be prefetched?
  - Data touched by each replacement operation
    - Easy to determine because the operations are well structured

- Prefetching can shorten lock holding time by over 70%

History of Buffer Pool's Caching Management in PostgreSQL

- 1996-2000: LRU (suffer lock contention moderately due to low concurrency)
- 2000-2003: LRU-k (hit ratio outperforms LRU, but lock contention became more serious)
- 2004: ARC/CAR are implemented, but quickly removed due to an IBM patent protection.
- 2005: 2Q was implemented (hit ratios were further improved, but lock contention was high)
- 2006 to now: CLOCK (approximation of LRU, lock contention is reduced, but hit ratio is the lowest compared with all the previous ones)

Performance Evaluation

- Hardware architecture:
  - SGI Altix 350 SMP with 16 Itanium II processors
  - Dell PowerEdge 1900 Server (two quad-core Xeon X5355 processors)
- DBMS and system environment:
  - PostgreSQL 8.2.3 + Linux Red Hat Enterprise Linux AS;
- Workloads
  - DBT-1 (simulating TPC-W) from OSDL database test suite;
  - DBT-2 (simulating TPC-C) from OSDL database test suite;
  - TableScan: each transaction sequentially scan a table of 800,000 rows (128-byte)

Experiment Setup in PostgreSQL

- Tested policies

Queue size: 64

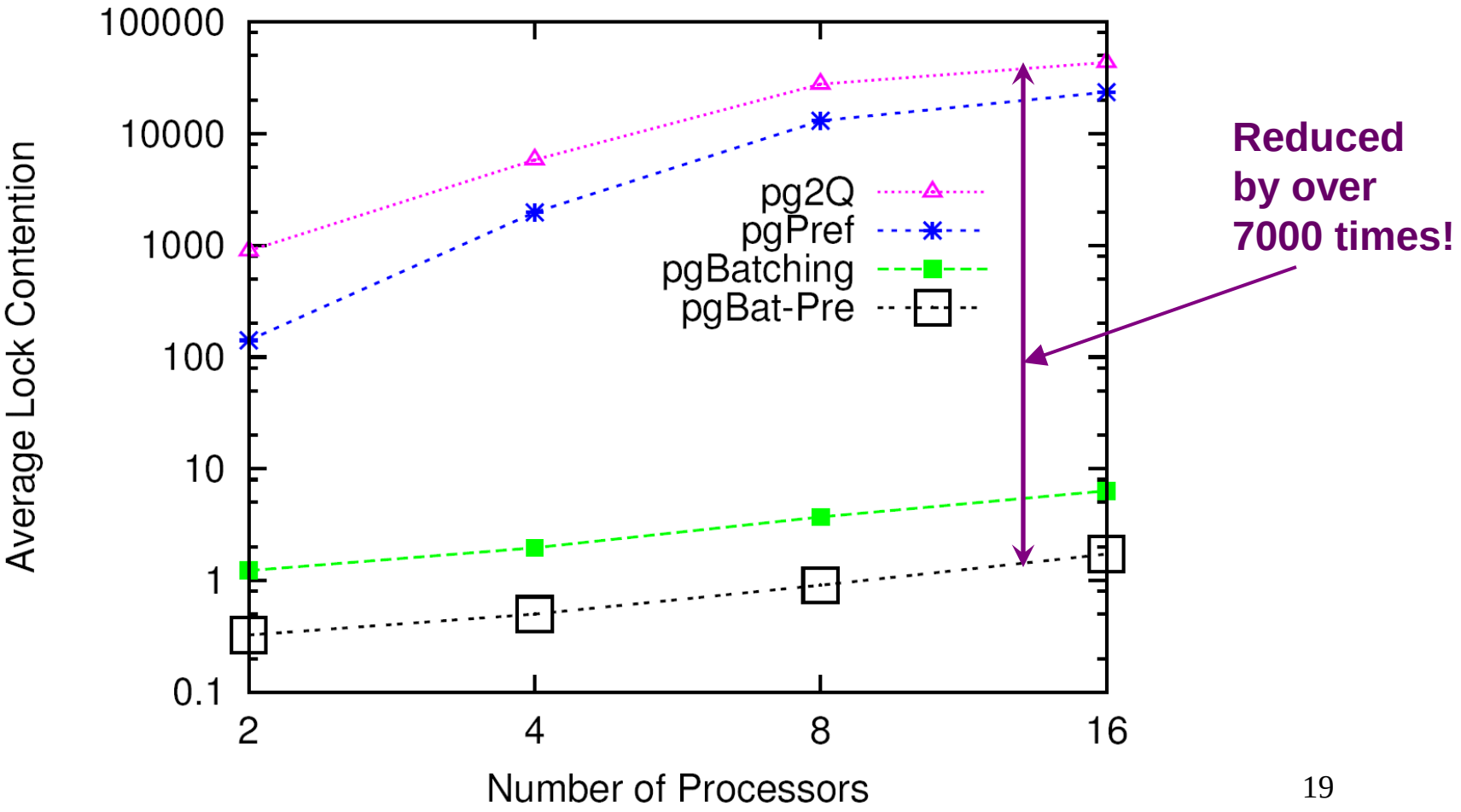
Name	Replacement	Enhancement
pgClock	Clock	None
pg2Q	2Q	None
pgBatching	2Q	Batching
pgPref	2Q	Prefetching
pgBat-Pre	2Q	Batching and Prefetching

- Experiments
  - Experiments on database scalability
    - Increasing the number of CPU/Cores used, and measuring throughput
    - Use large buffer to hold whole working sets (hits only)
    - With BP-Wrapper, 2Q can be as scalable as CLOCK (same throughout)
    - **Lock contention is minimized**
  - Experiments on overall performance
    - Increasing the buffer pool size with fixed system scale (8 cores)
    - Buffer is not as large to hold whole working set
      - Causing misses
    - With BP-Wrapper, 2Q outperforms CLOCK
    - **The advantage of high hit ratio of 2Q is retained.**

Reduction of Lock Contention (SGI Altix 350, DBT-1, no misses)

Lock contention: a lock cannot be obtained without blocking;

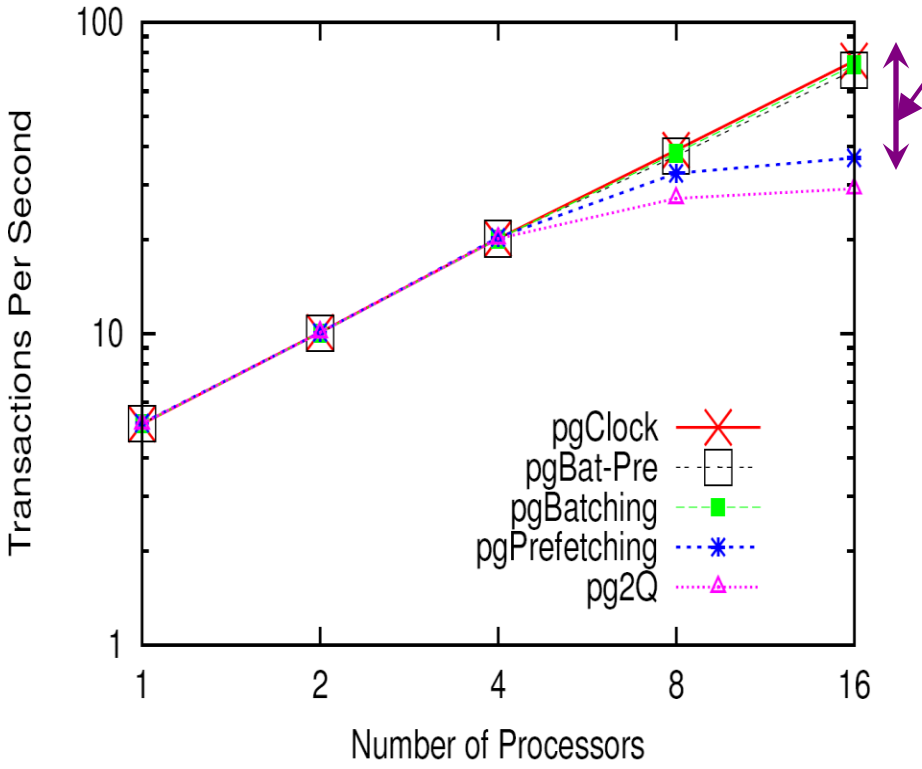
Number of lock acquisitions (contention) per million page accesses.

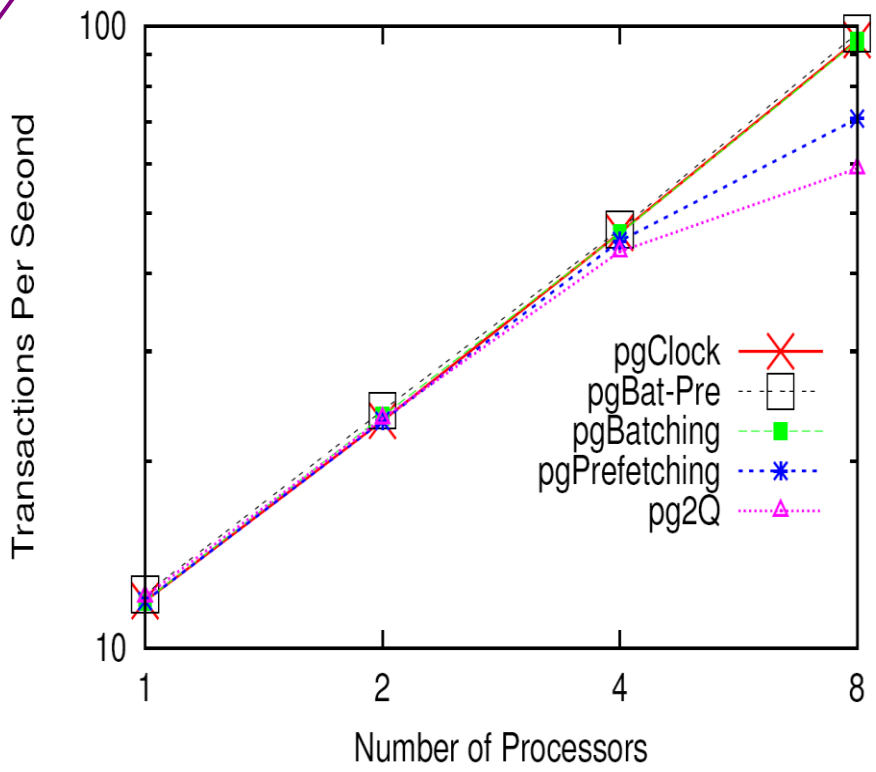


Improving Throughput by lock contention reduction

(DBT-1, no misses)

1.5 Times Improve in Throughput



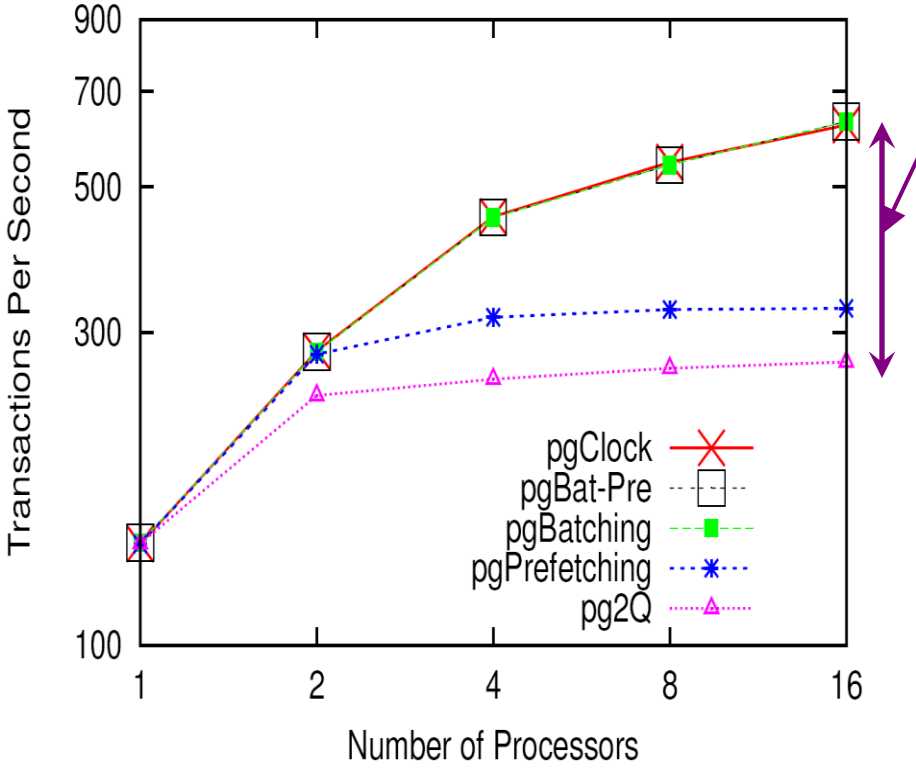


SGI Altix 350
(16 Itaniums)

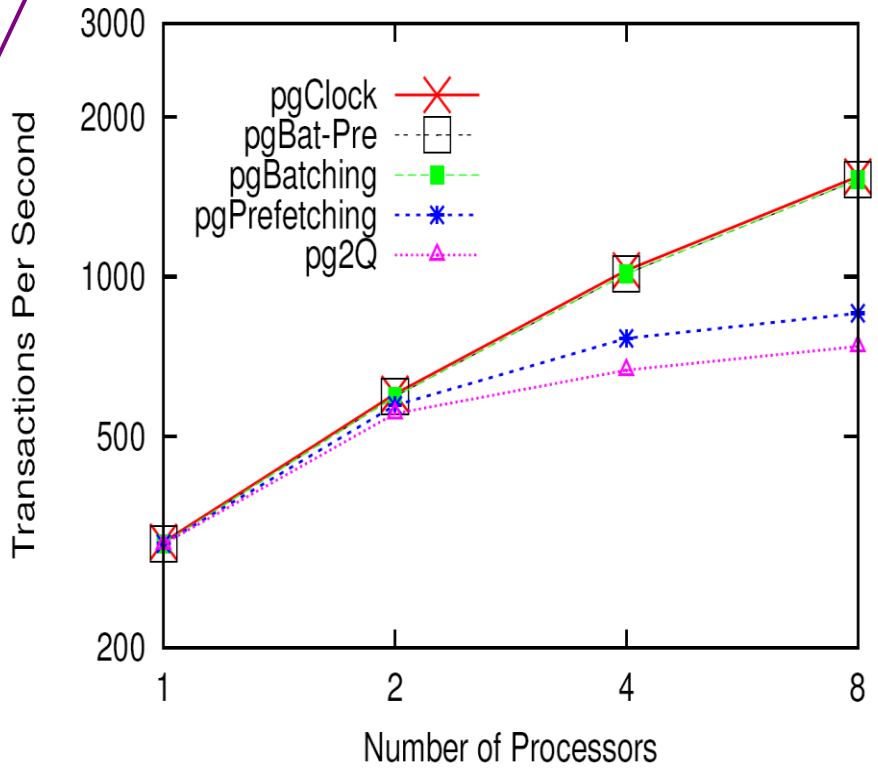
Poweredge 1900
(two quad-core Xeon)

Improvement of Throughput (DBT-2, no misses)

1.3 Times Improve in Throughput



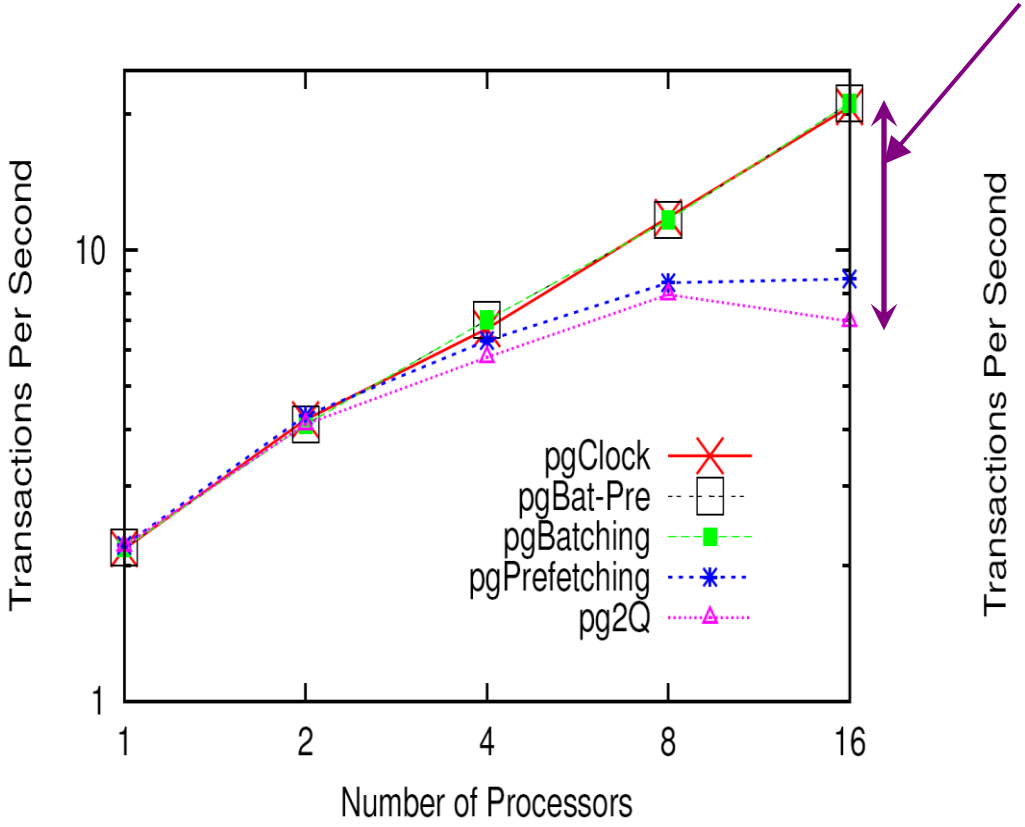
SGI Altix 350
(16 Itaniums)



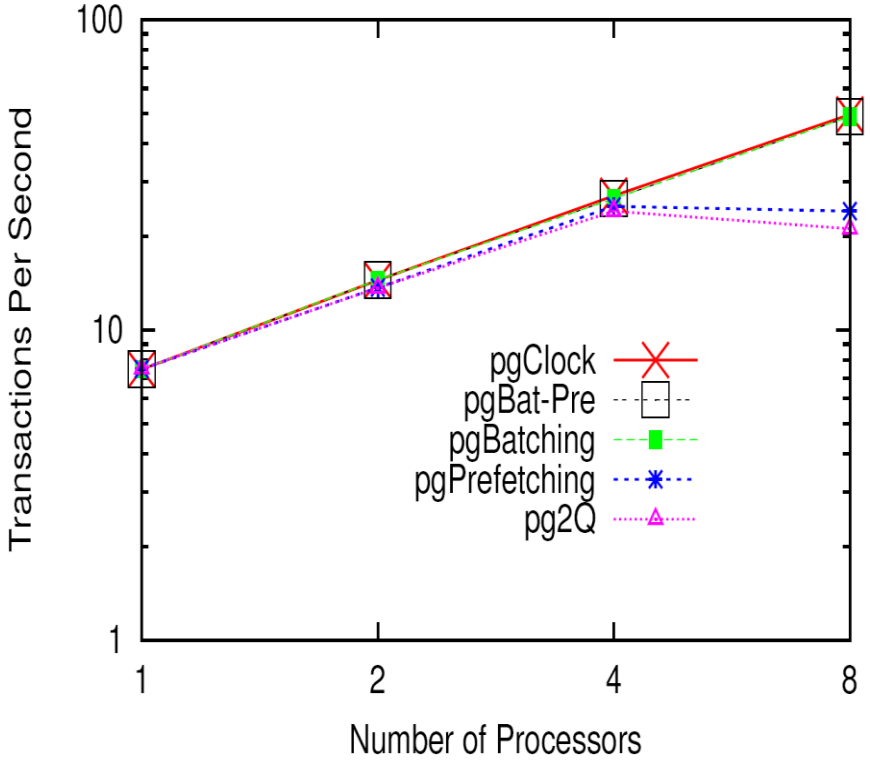
Poweredge 1900
(two quad-core Xeon)

Improvement of Throughput (TableScan, no misses)

2.0 Times Improve in Throughput



SGI Altix 350
(16 Itaniums)



Poweredge 1900
(two quad-core Xeon)

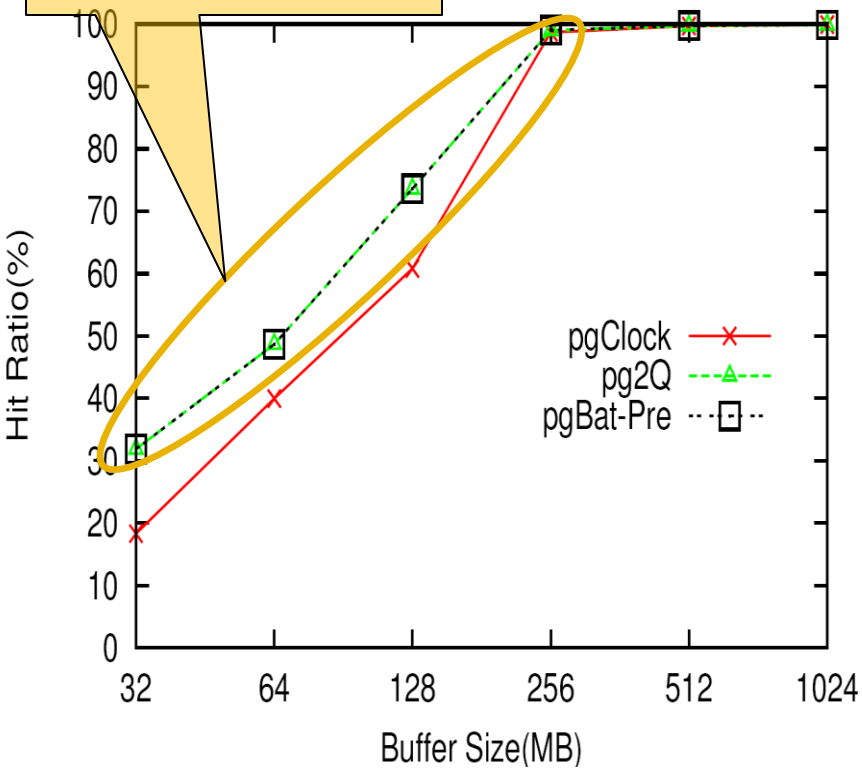
Benefits from both High Hit Ratio and Low Lock Contention

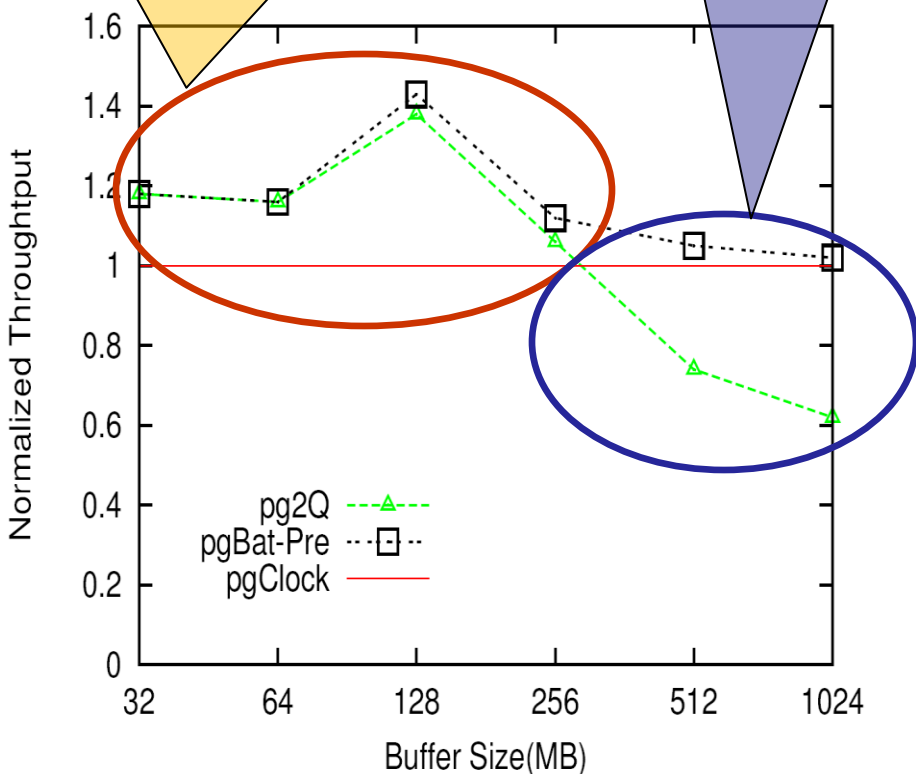
(PowerEdge 1900, DBT-1, #processors = 8)

High hit ratio is preserved

Higher hit ratio ➔ higher throughput

Less contention ➔ higher throughput



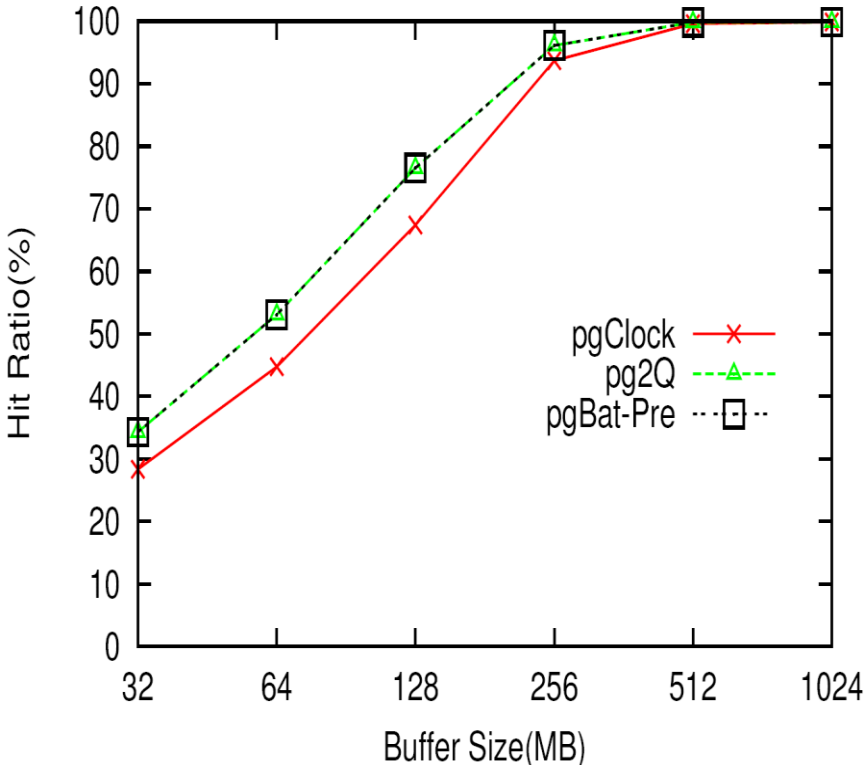


Hit Ratio

Normalized Throughputs

Benefits from both High Hit Ratio and Low Lock Contention

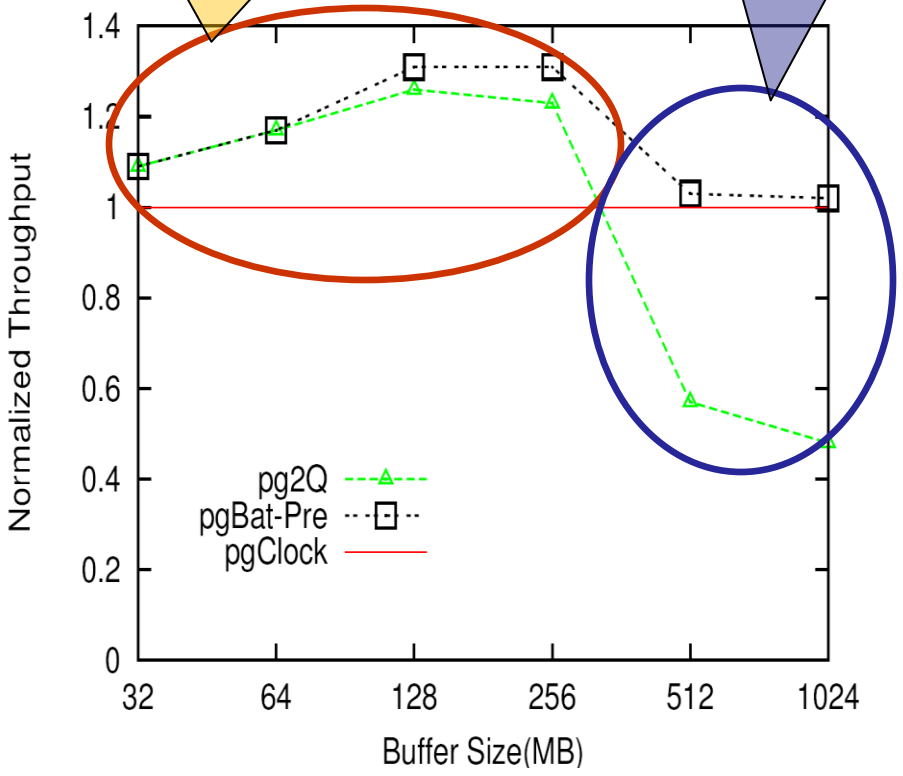
(PowerEdge 1900, DBT-2, #processors = 8)



Hit Ratio

Higher hit ratio ➔ higher throughput

Less contention ➔ higher throughput



Normalized Throughputs

Summary

- A 40+ year classical problem to address trade-off between
  - high hit ratios in buffer pool (by advanced algorithms) and
  - Low lock contention (by implementation approximations)
- Current system architecture demands both
  - high hit ratio to reduce increasingly high disk access latency
  - Low lock contention to accommodate increasingly more concurrent transactions served by multiprocessors/cores
- BP-Wrapper can achieve both performance goals by
  - Batching the hit requests (retain the accuracy of original algorithms)
  - Prefetching needed data (minimize the latency in critical sections)

- We hope BP-Wrapper turns a new chapter in PostgreSQL's history of buffer pool replacement.