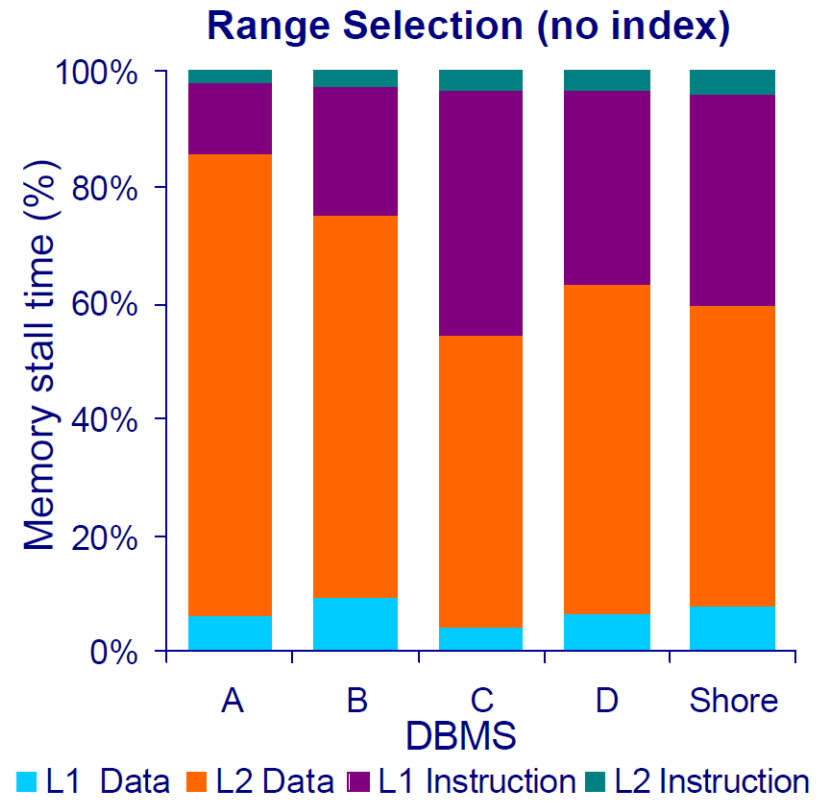
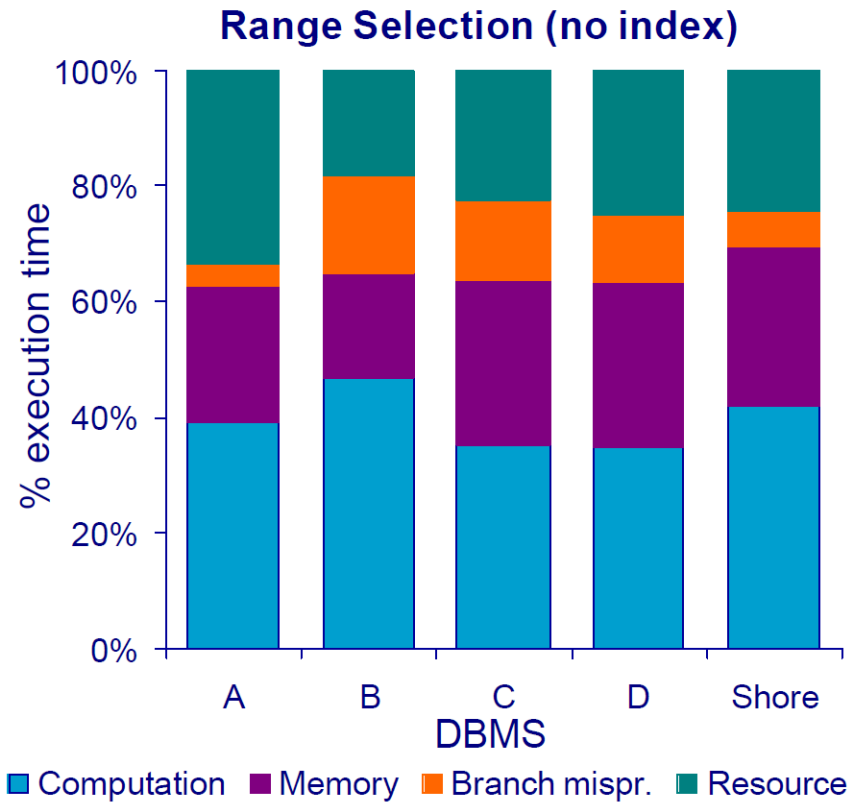


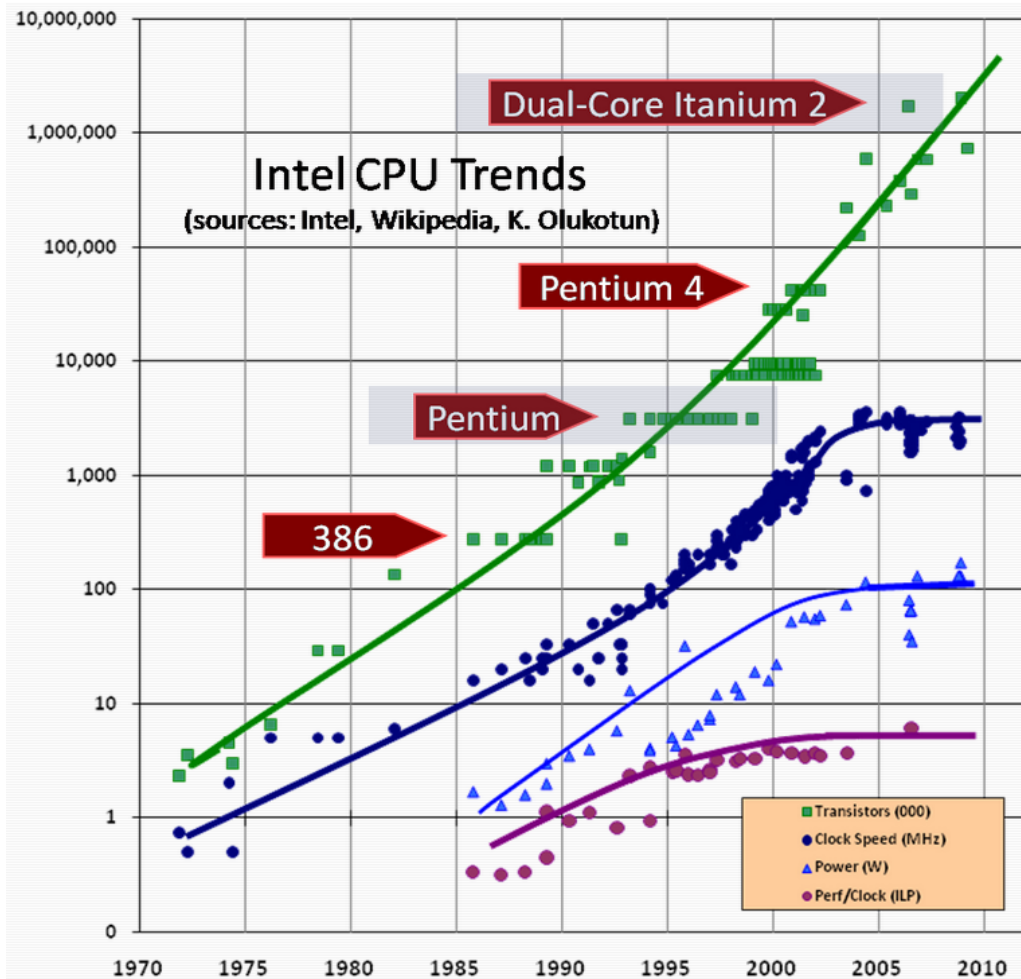
# Databases on Multicores

# The Past



**processor stalled >50% of the time**

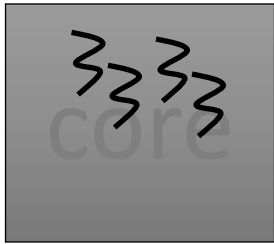
# Moore's Law



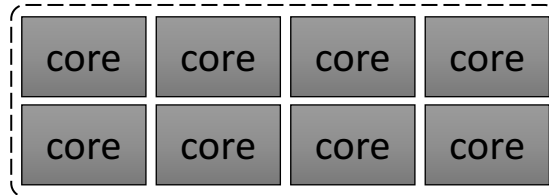
**doubling of transistor counts continues**  
**clock speeds and power hit the wall**

# Processor Trends

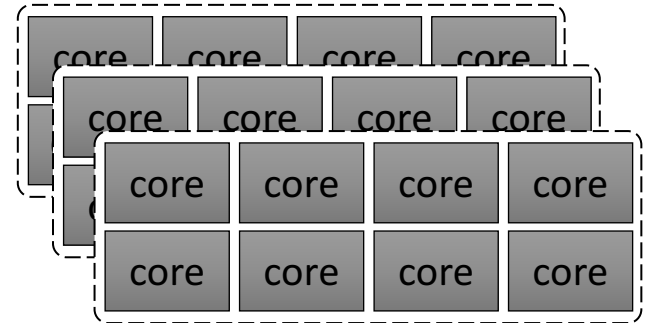
2005



pipelining  
ILP  
multithreading



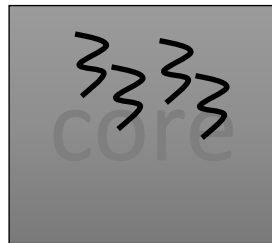
multicores  
(CMP)



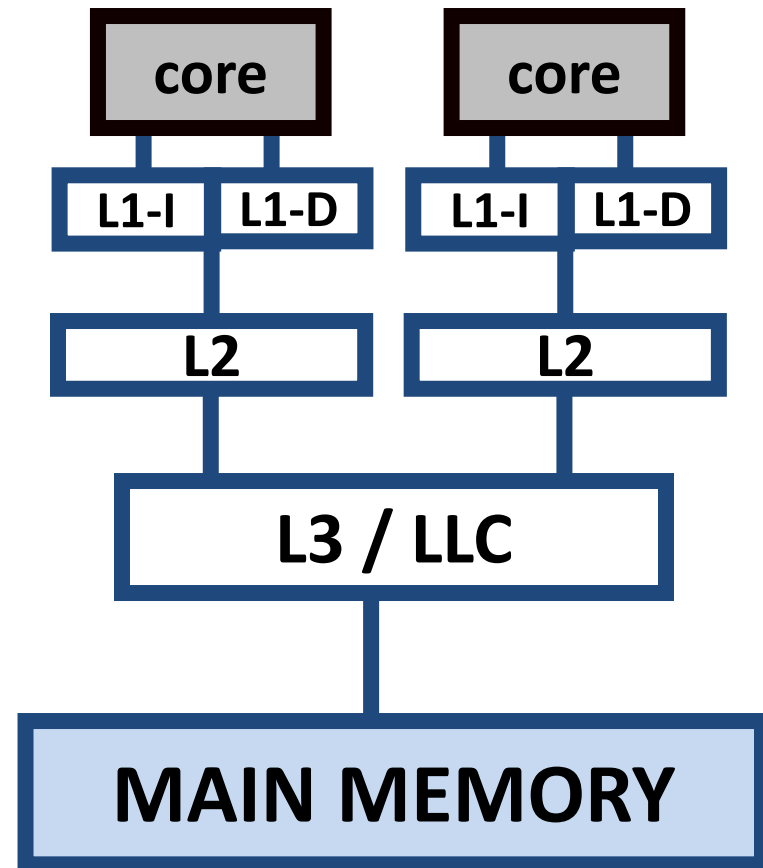
multisocket  
multicores

**goal: scalability**

# Vertical Dimension: Cores & Caches



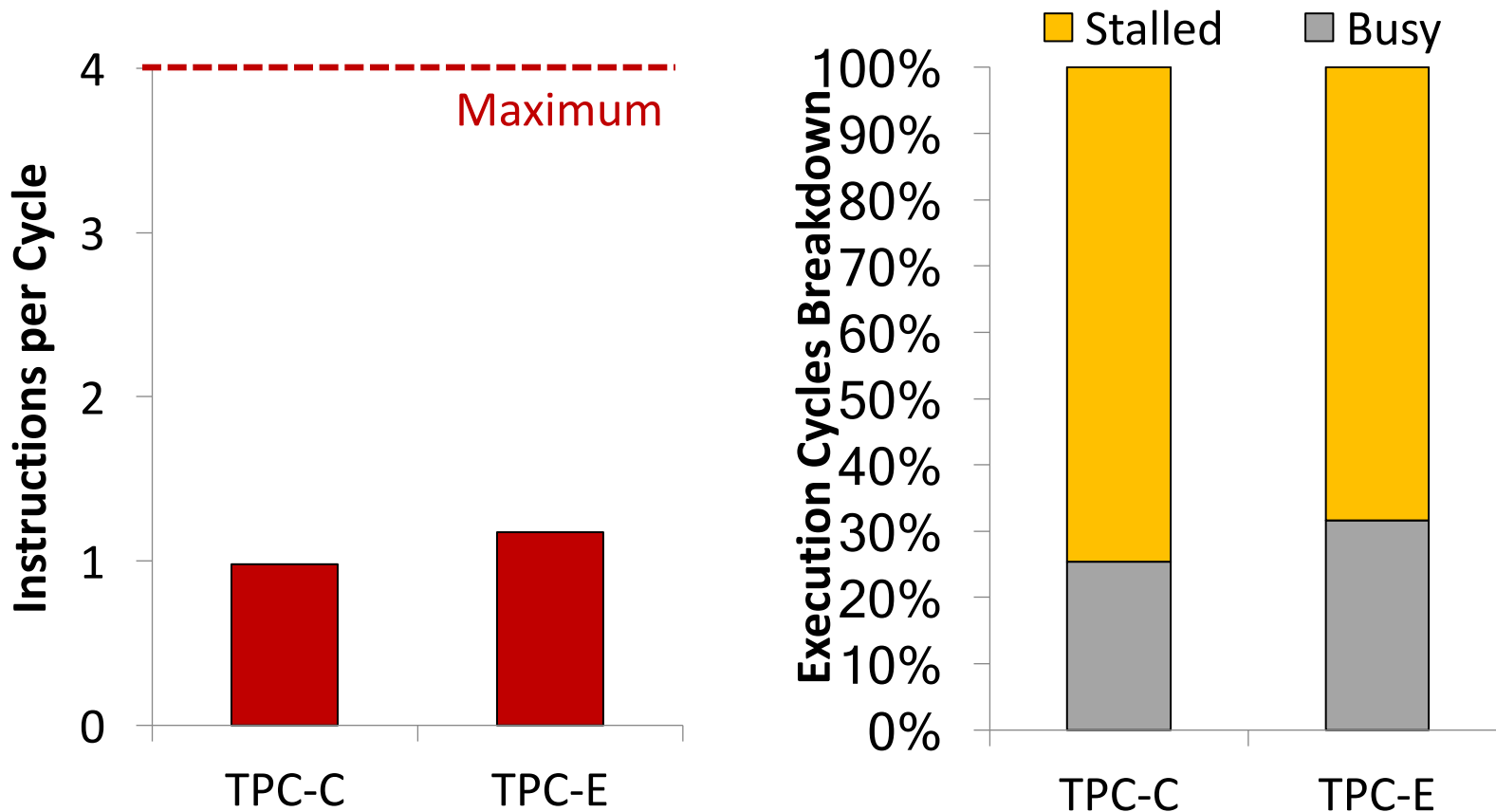
pipelining  
ILP  
multithreading



**implicit parallelism & memory matters**

# Now: Cores & Cache Utilization

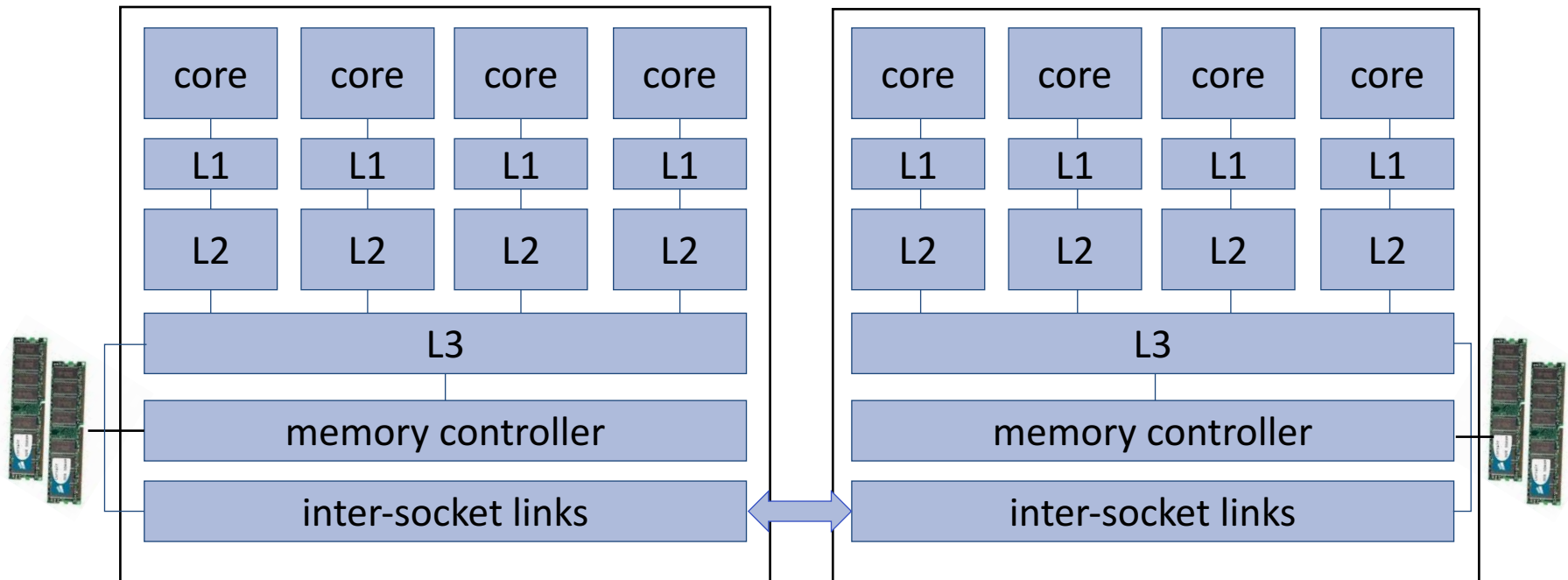
at peak throughput on Shore-MT, Intel Xeon X5660



**Instructions per Cycle < 1**

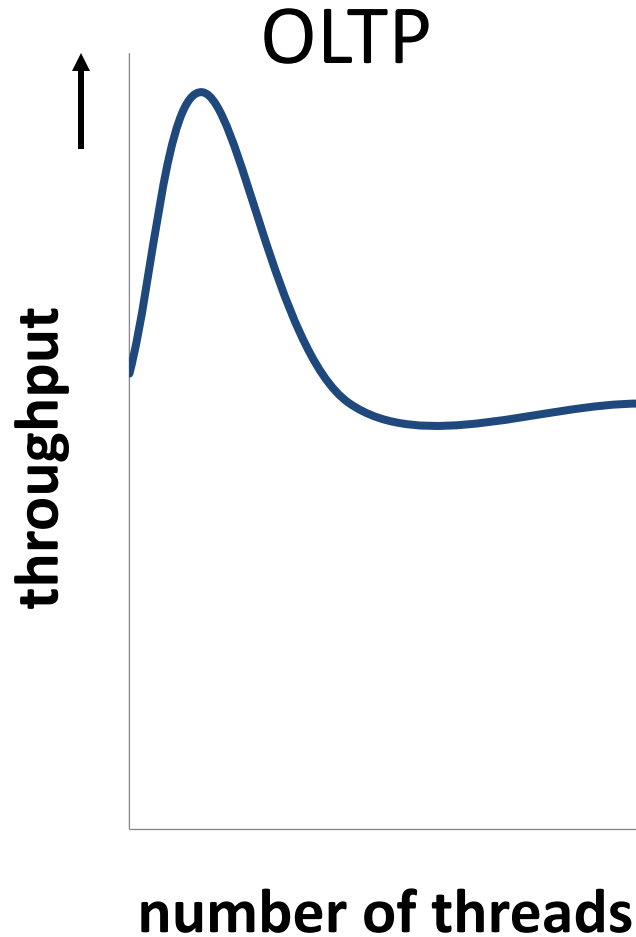
**70% of the execution time goes to stalls**

# Horizontal Dimension: Cores & Sockets

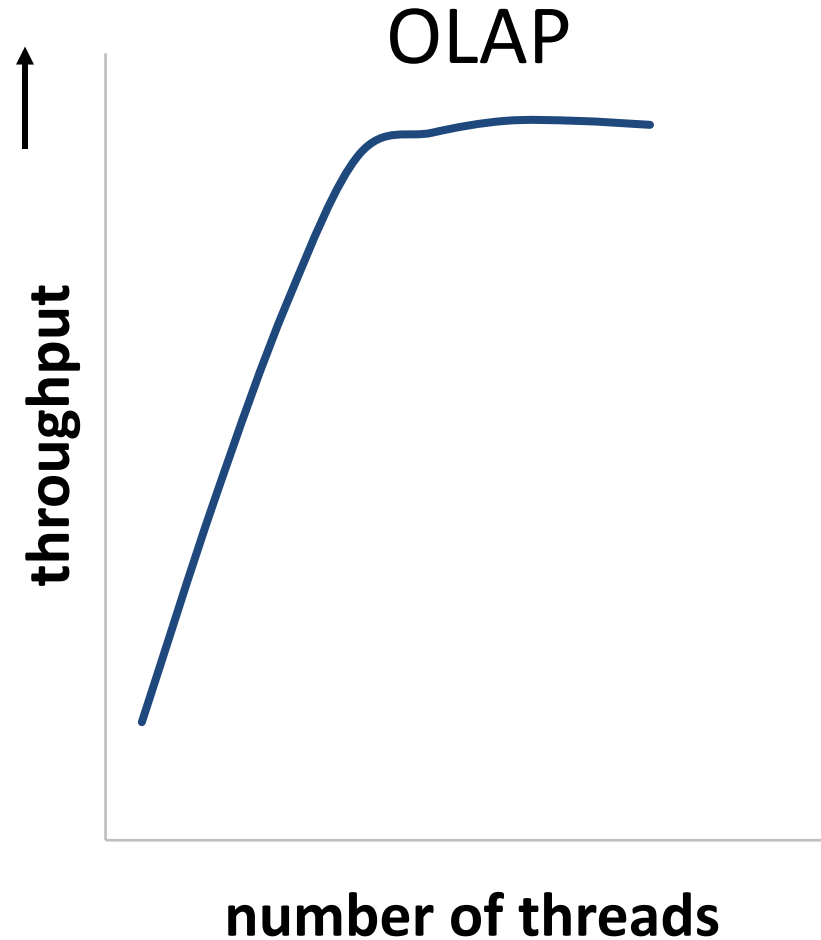


**exploit abundant parallelism**

# Workload Scalability on Multicores



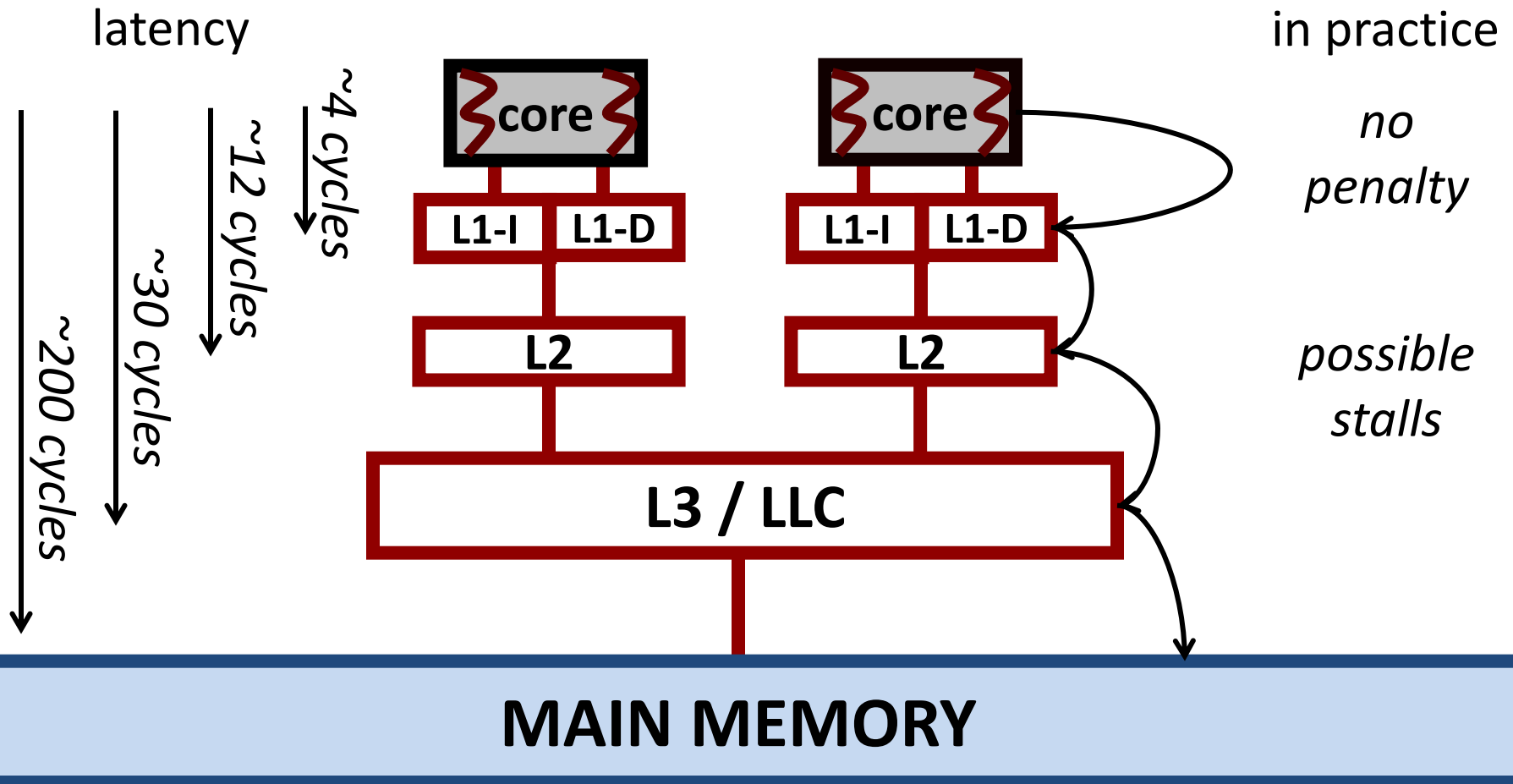
**access latency**



**memory bandwidth**

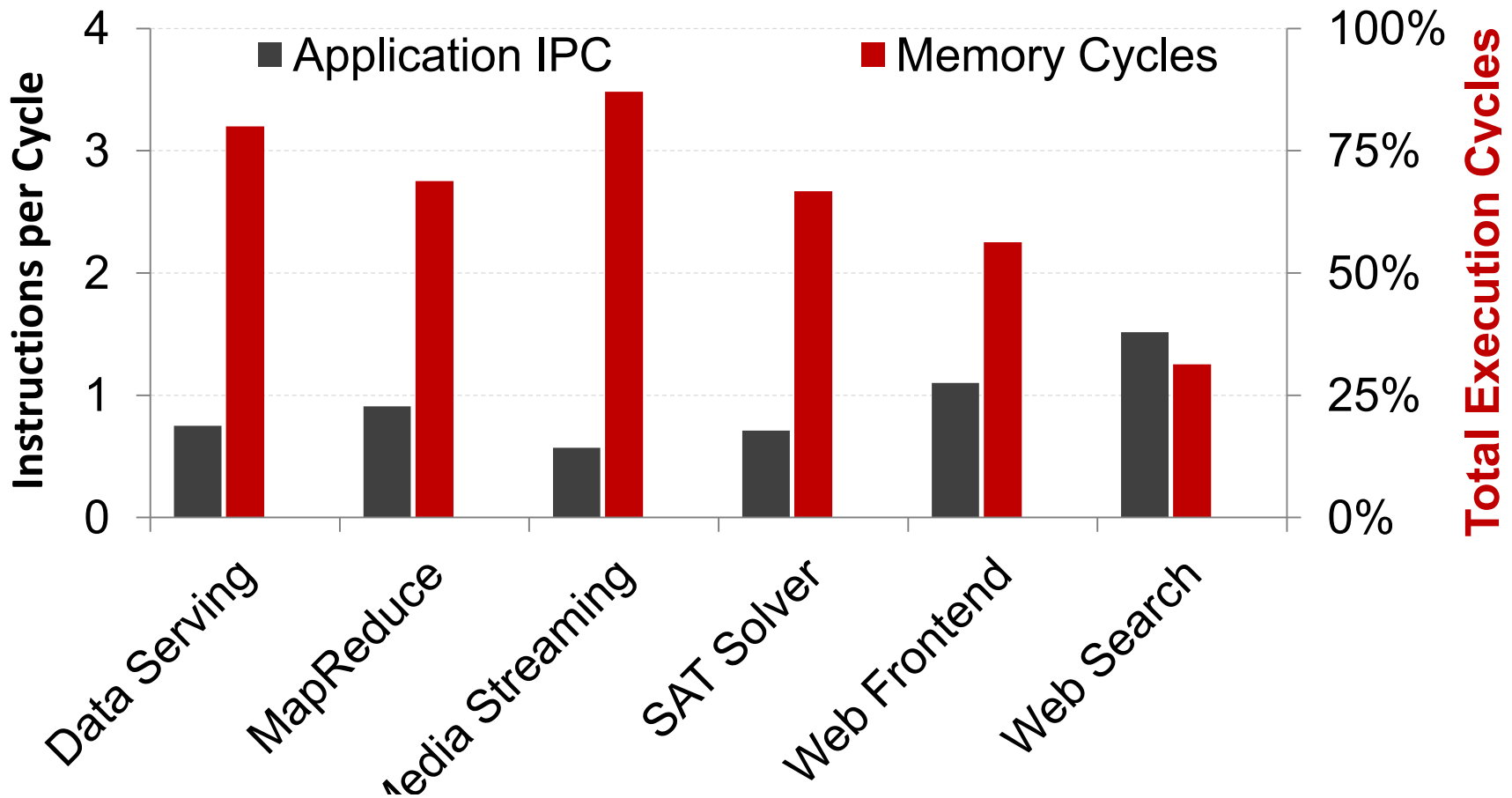


# Today's Memory Hierarchy



**stalls → wasted power & money**

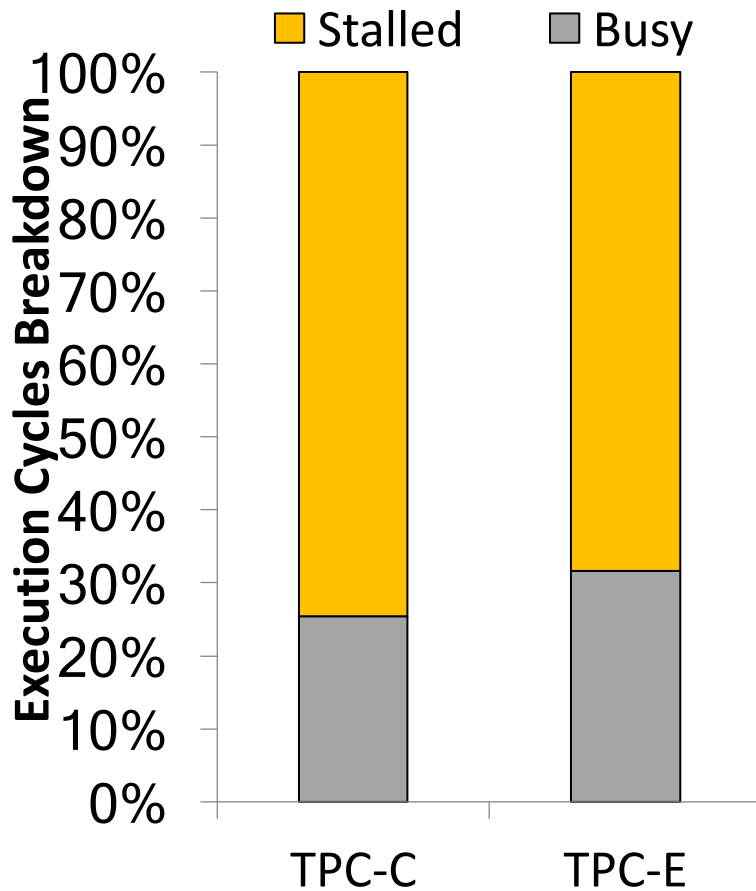
# Stalls in Cloud Workloads



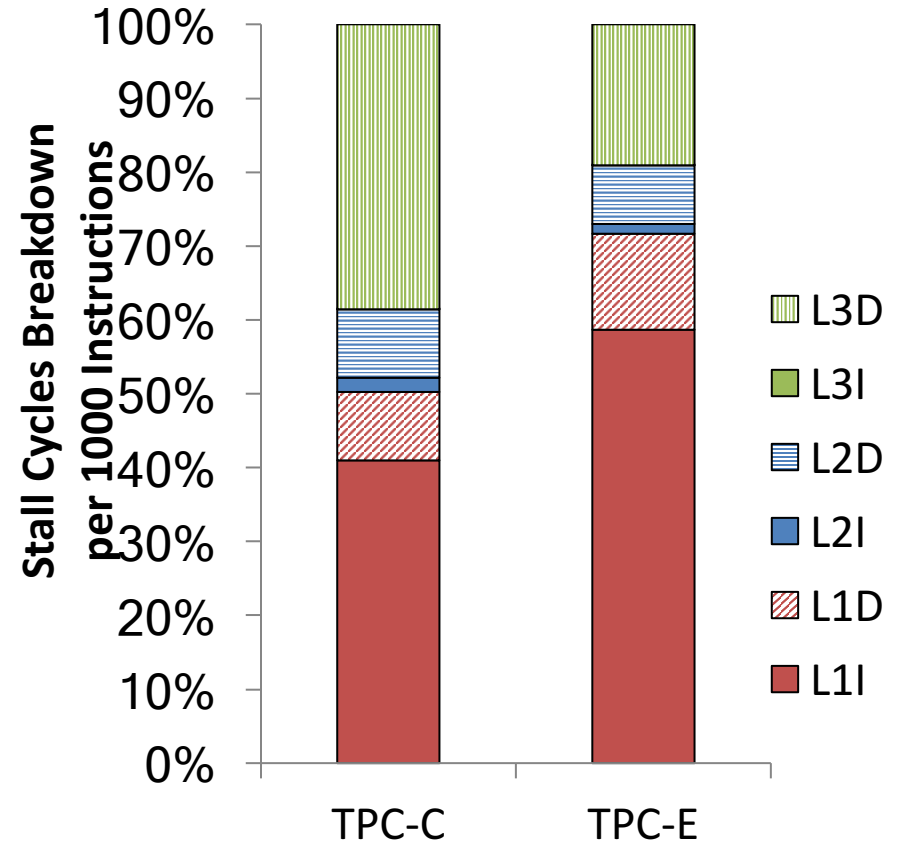
**~1 instructions per cycle**

**> 50% of the time goes to stalls on average**

# Sources of Memory Stalls



100GB data on Shore-MT, Intel Xeon E5-2660



**L1-I & LLC data misses dominate the stall time**

# For Data Intensive Applications ...

- 50%-80% of cycles are stalls
  - *Problem:*  
instruction fetch & long-latency data misses
  - *Instructions* need more *capacity*
  - *Data misses* are *compulsory*
- Focus on maximizing:
  - *L1-I locality & cache line utilization for data*

# Minimizing Memory Stalls

## prefetching

- light
- temporal stream
- software-guided

## being cache conscious

- code optimizations
- alternative data structures/layout
- vectorized execution

## exploiting common instructions

- computation spreading

# Prefetching – Lite

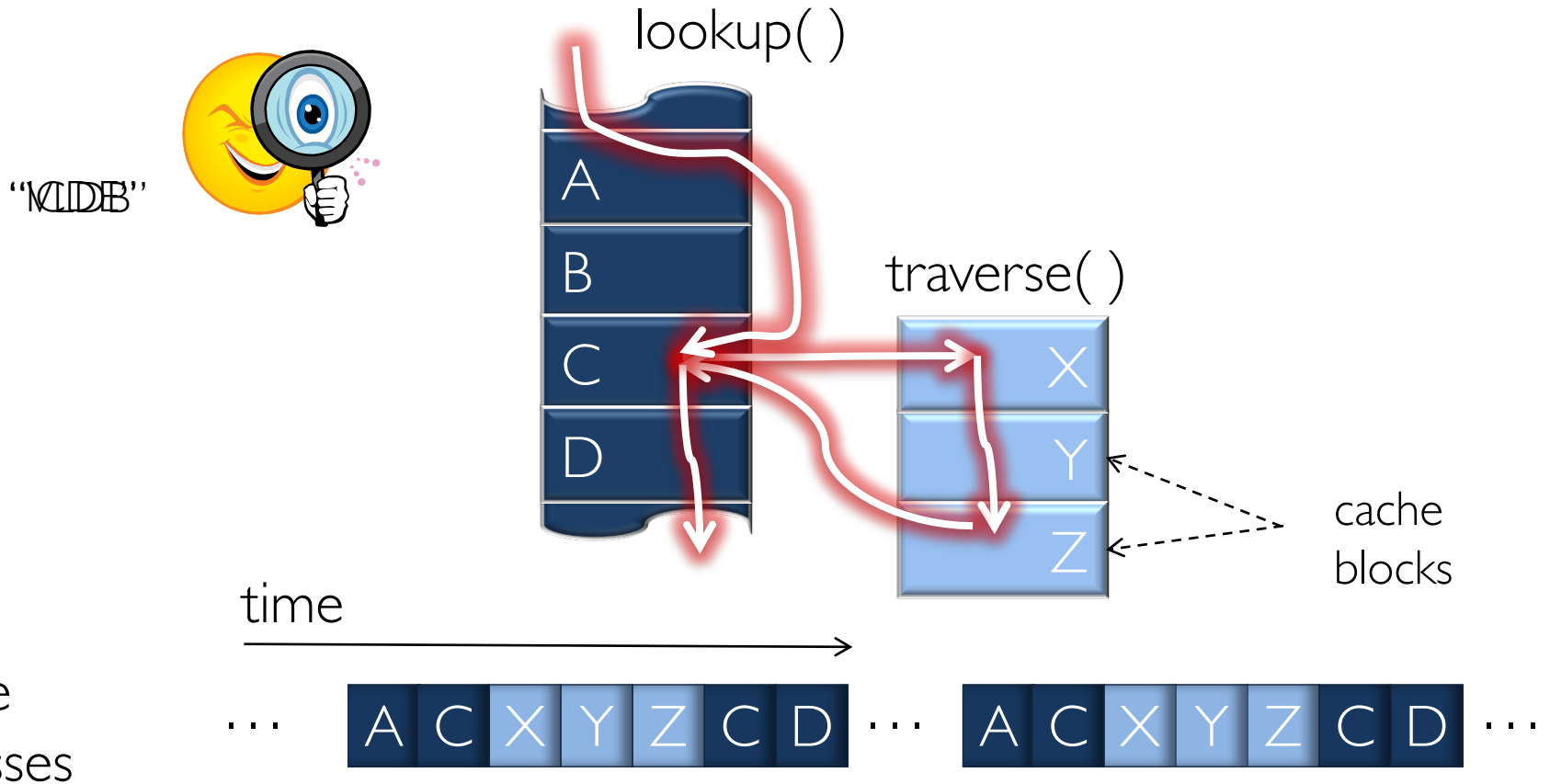
- next-line: miss  $A \rightarrow$  fetch  $A+1$
- stream: miss  $A, A+1 \rightarrow$  fetch  $A+2, A+3$



- ✓ favors sequential access & spatial locality
- ✗ instructions: branches, function calls
  - branch prediction
- ✗ data: pointer chasing
  - stride: miss  $A, A+20 \rightarrow$  fetch  $A+40, A+60$

**preferred on real hardware due to simplicity  
but memory stalls are still too high**

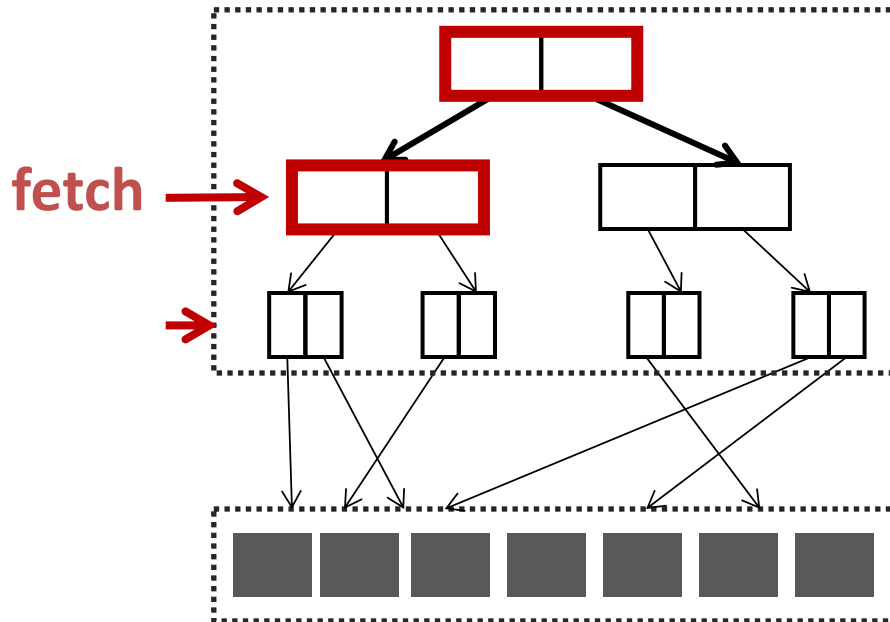
# Temporal Streaming



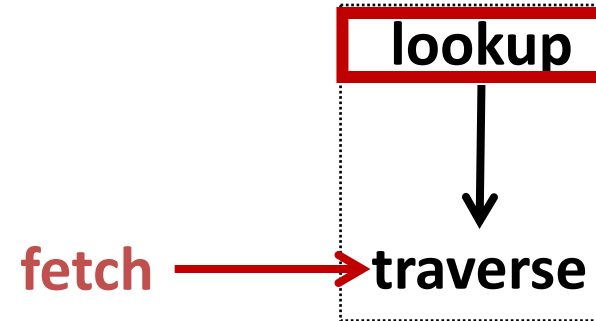
**exploits recurring control flow**  
**more accurate → higher space cost**

# Software-guided Prefetching

data



instructions





# Minimizing Memory Stalls

## prefetching

light  
temporal stream  
software-guided

## being cache conscious

code optimizations  
alternative data structures/layout  
vectorized execution

## exploiting common instructions

computation spreading

# Code Optimizations

- simplified code
  - in-memory databases have smaller instruction footprint
- better code layout
  - minimize jumps → exploit next line prefetcher
  - profile-guided optimizations (static)
  - just-in-time (dynamic)
- query compilation into machine/naïve code
  - e.g., HyPer, Hekaton, MemSQL

# Cache Conscious Data Layouts

erietta	blue
pinar	black
danica	green
iraklis	orange

**16 bytes columns**

**goal:**

*maximize cache line utilization & exploit next-line prefetcher*

**row stores: good for OLTP**

*accessing many columns*

**column stores: good for OLAP**

*accessing a few columns*

cache lines (64bytes)

erietta	blue	pinar	black
---------	------	-------	-------

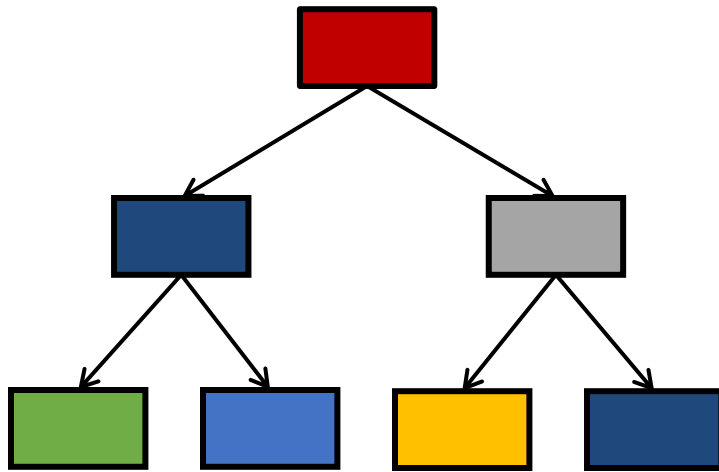
row store

erietta	pinar	danica	iraklis
---------	-------	--------	---------

column store

# Cache Conscious Data Structures

index tree



*+ align nodes to cache lines*

in memory

*lookup-heavy workload*



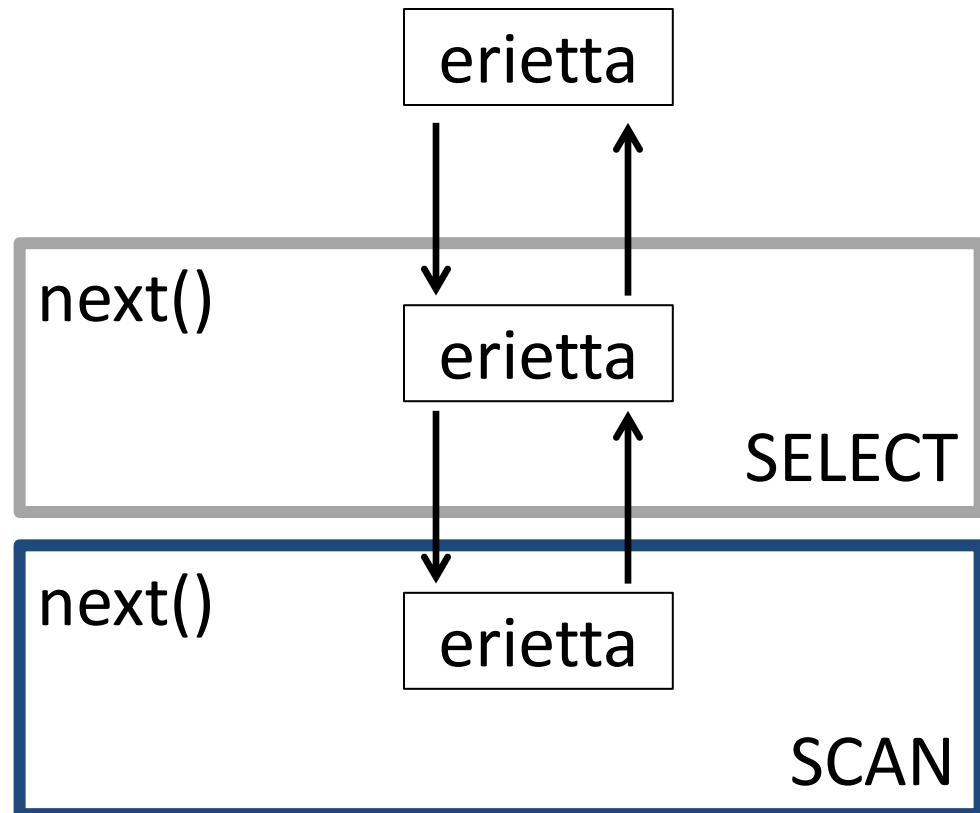
*scan-heavy workload*



**goal: maximize cache line utilization & exploit next-line prefetcher in tree probe**

# Volcano Iterator Model

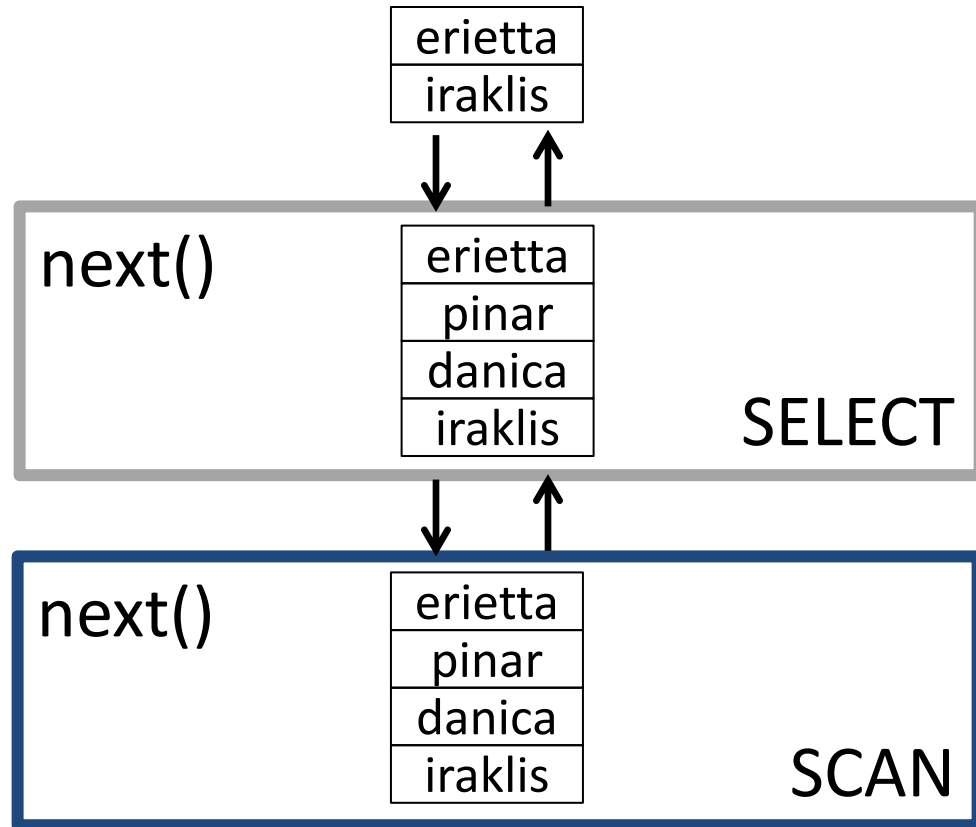
erietta	blue
pinar	black
danica	green
iraklis	orange
...	...



**✗ poor data & instruction cache locality**

# Vectorized Execution

erietta	blue
pinar	black
danica	green
iraklis	orange
...	...



- ✓ good data & instruction cache locality
- ✓ allows exploiting SIMD

# Minimizing Memory Stalls

## prefetching

- light
- temporal stream
- software-guided

## being cache conscious

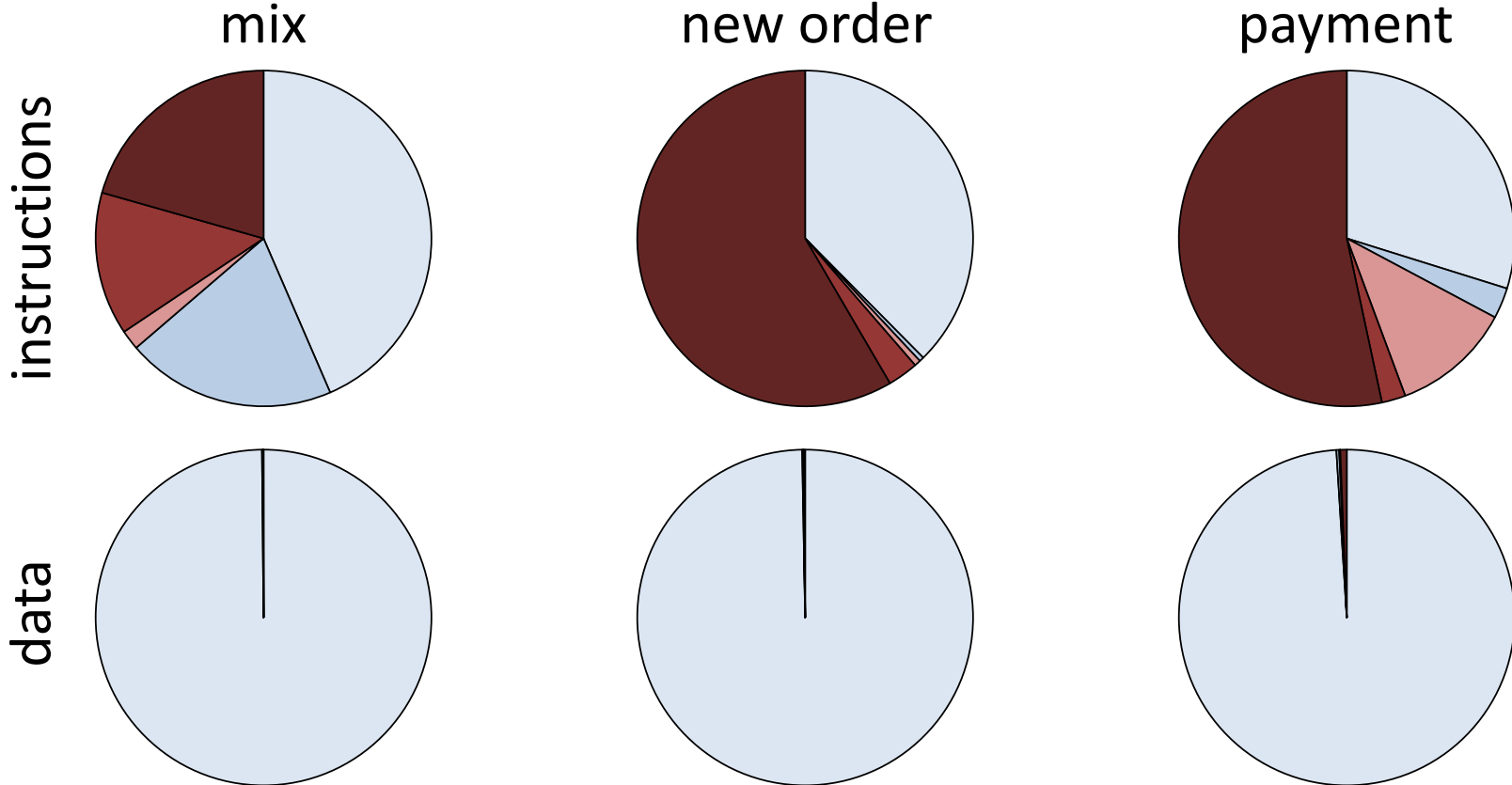
- code optimizations
- alternative data structures/layout
- vectorized execution

## exploiting common instructions

- computation spreading

# Instruction & Data Overlap

TPC-C (100GB data) on Shore-MT  
overlapping cache blocks

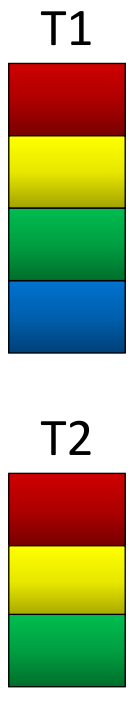


**overlap: significant for instructions & low for data**  
**higher overlap in same-type transactions**



# Computation Spreading

Threads



Conventional

#Cache

Miss

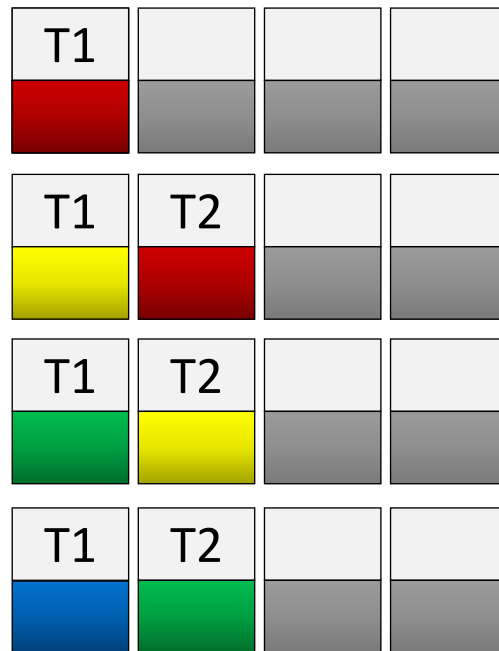
1

3

5

7

CORES



SLICC

CORES

#Cache

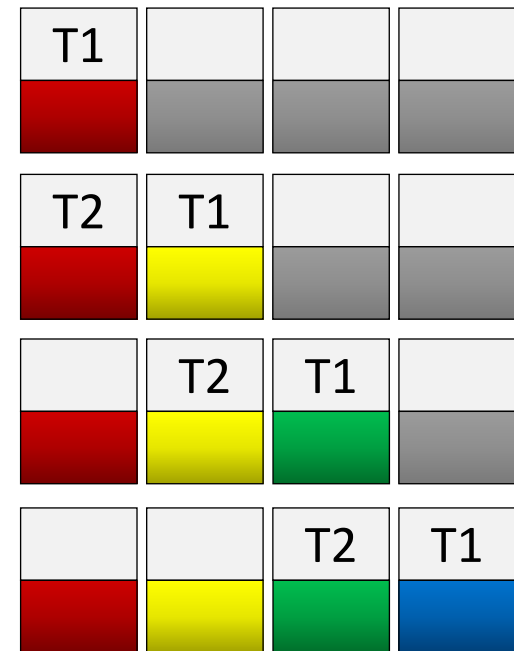
Miss

1

2

3

4

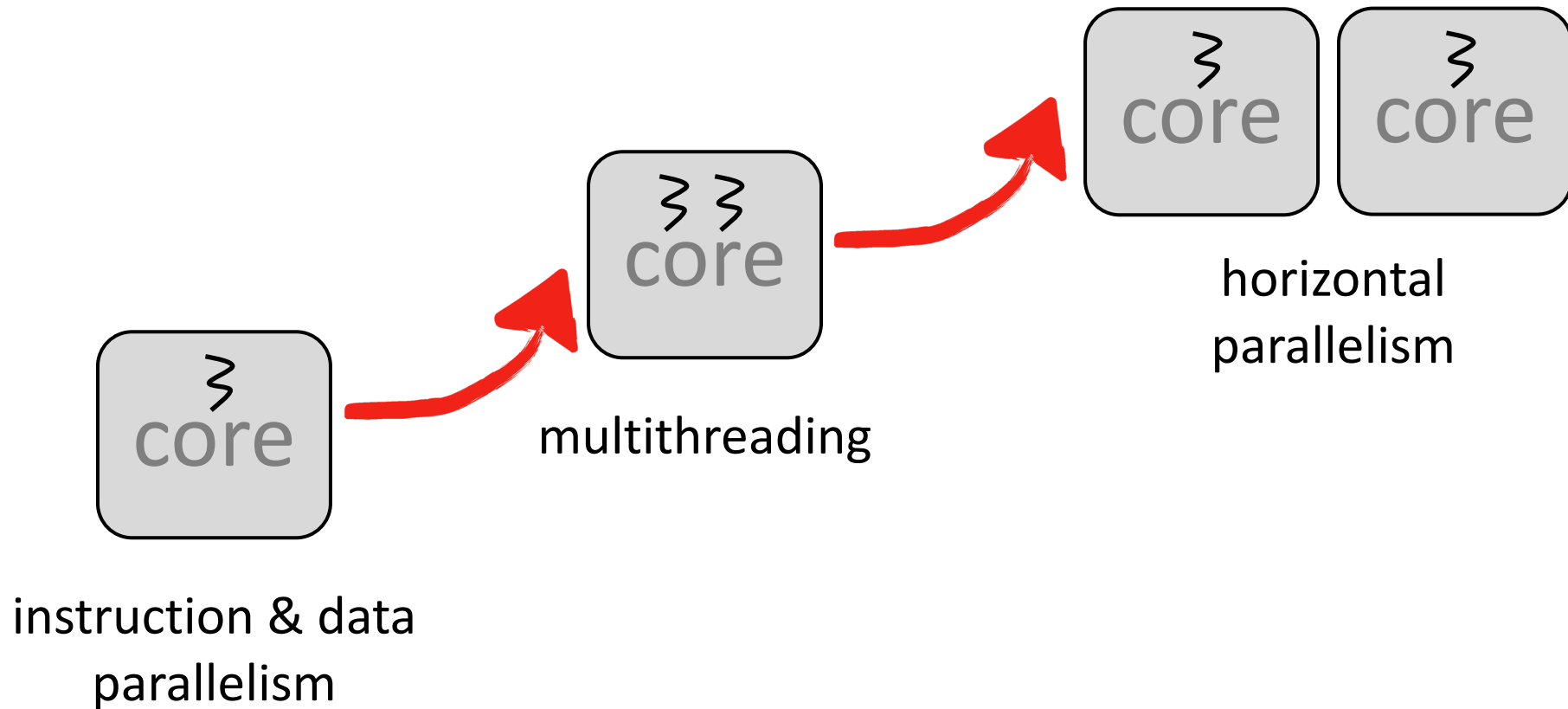


**exploits aggregate L1-I & instruction overlap**  
**need to track recent misses and cache contents**

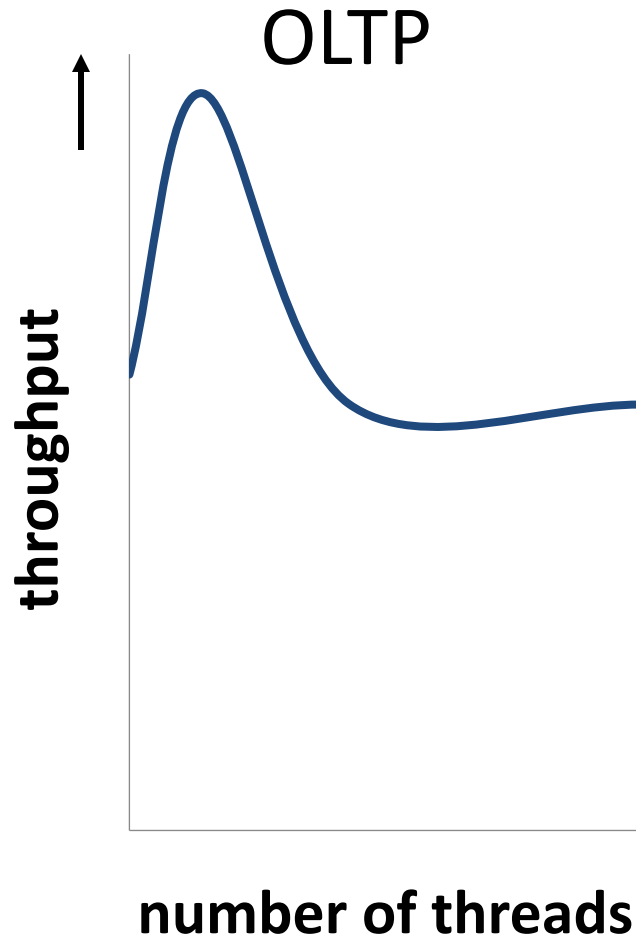
# Summary

- DBMSs underutilize a core's resources
- Problem 1: L1-I misses
  - due to capacity
  - minimized footprint & illusion of a larger cache by maximizing re-use
- Problem 2: LLC data misses
  - compulsory
  - maximize cache-line utilization through cache-conscious algorithms and layout

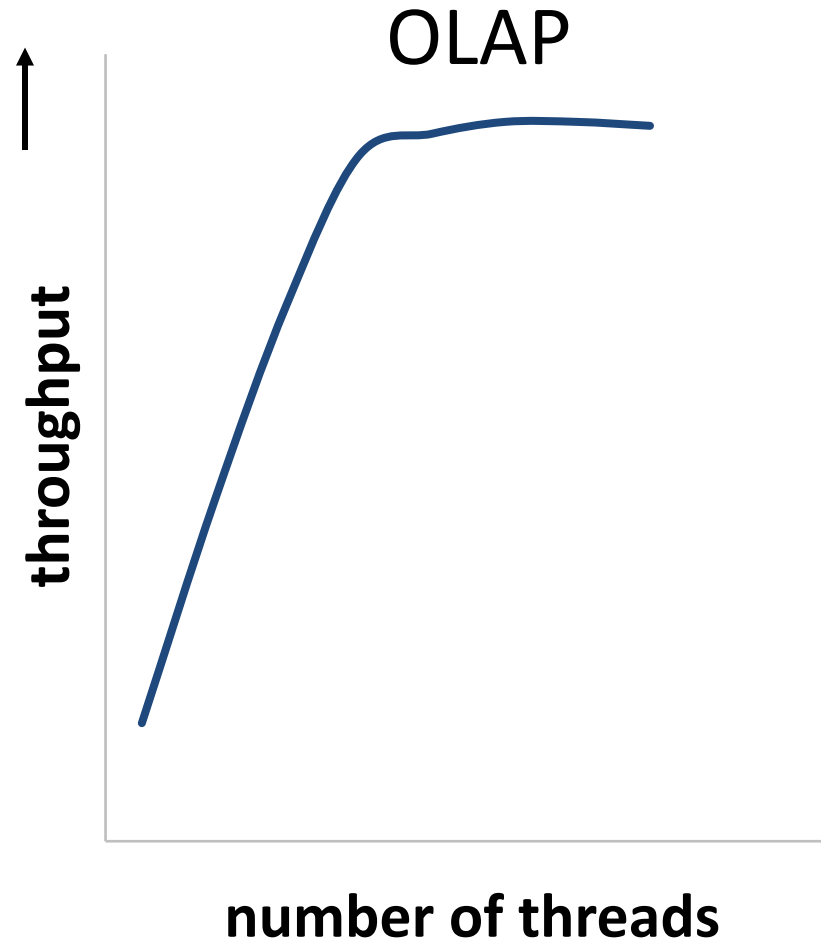
# Modern Parallelism



# Challenges when Scaling Up

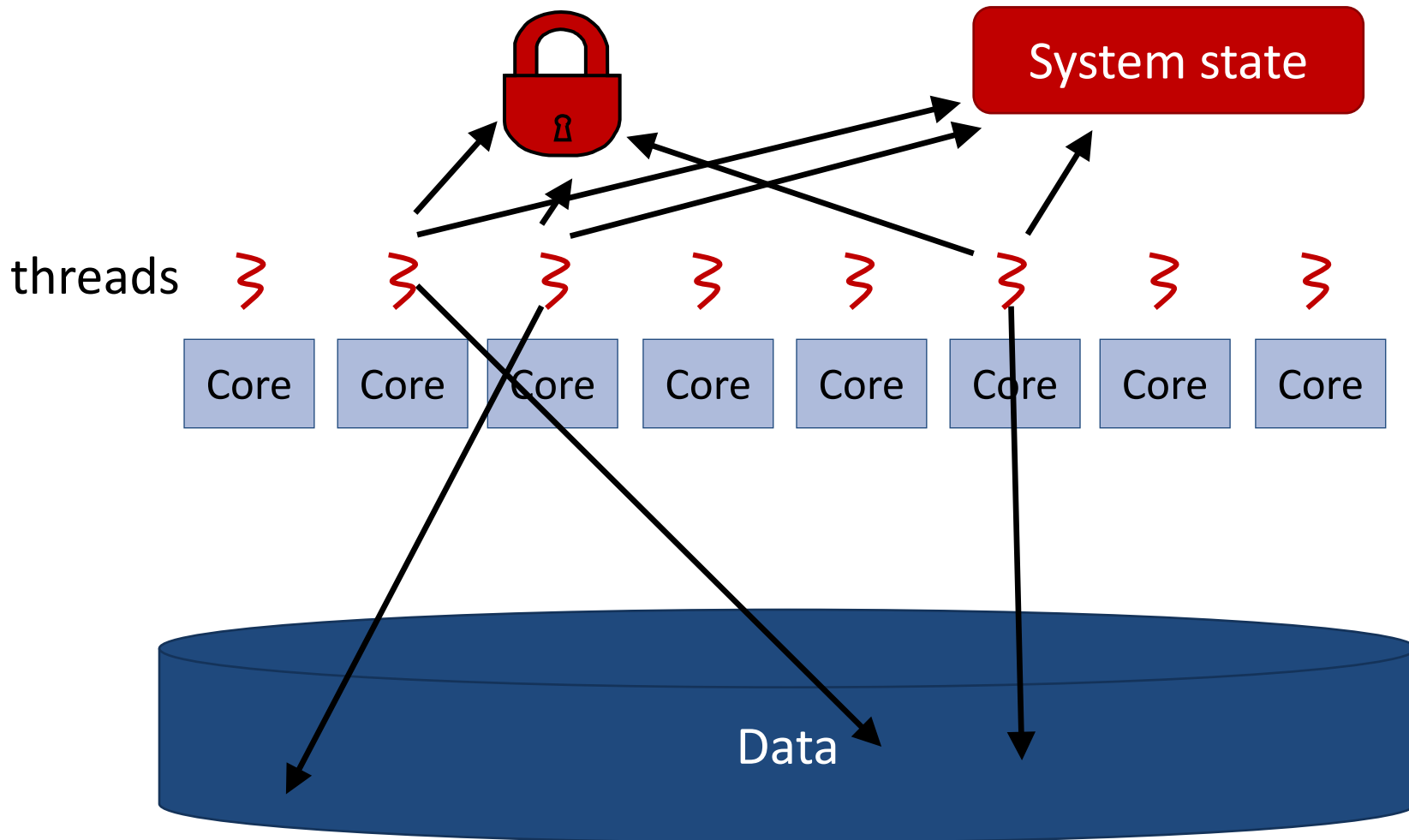


**access latency**



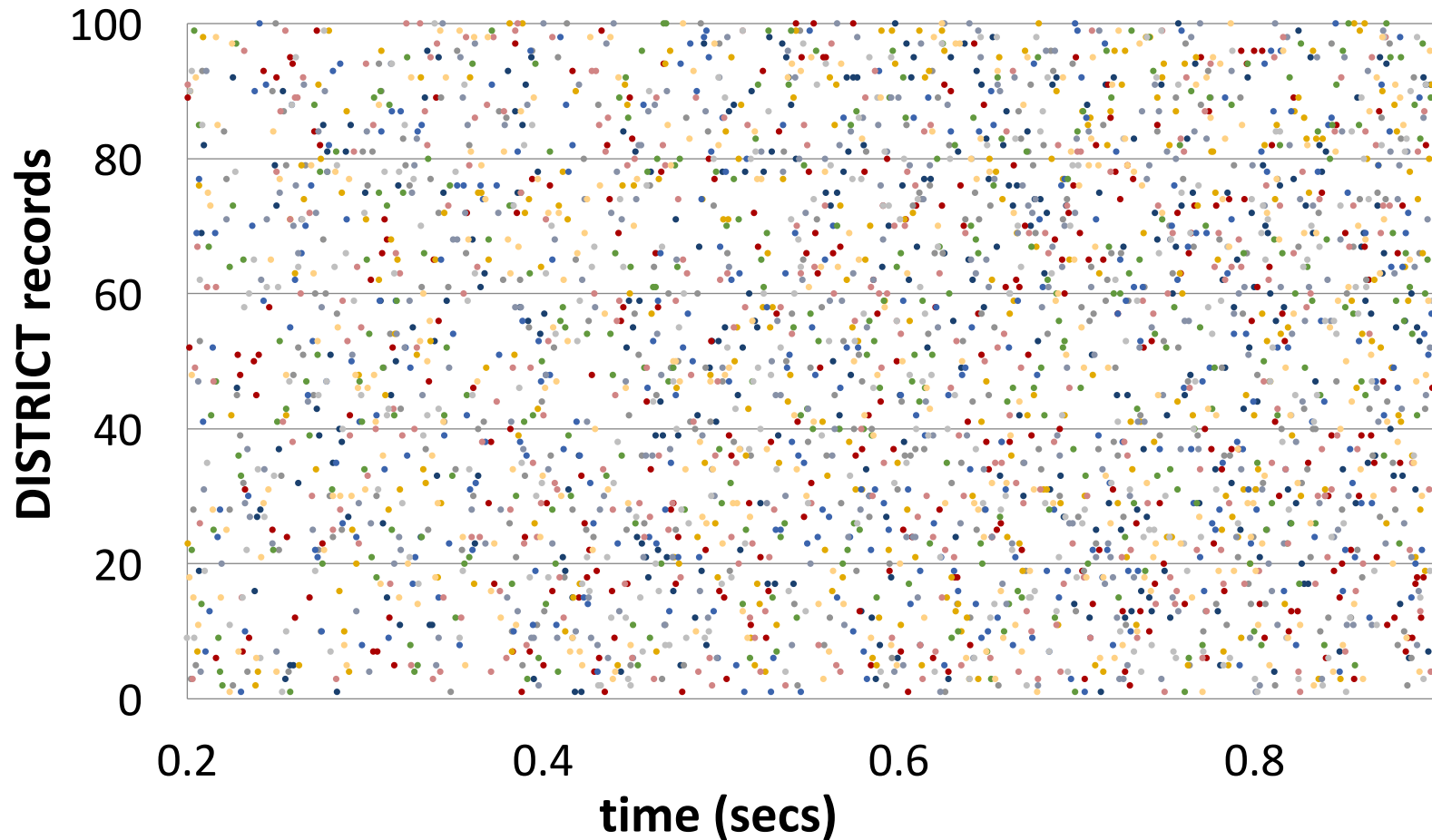
**memory bandwidth**

# Critical Path of Transaction Execution



**many accesses to shared data structures**

# Data Access Pattern

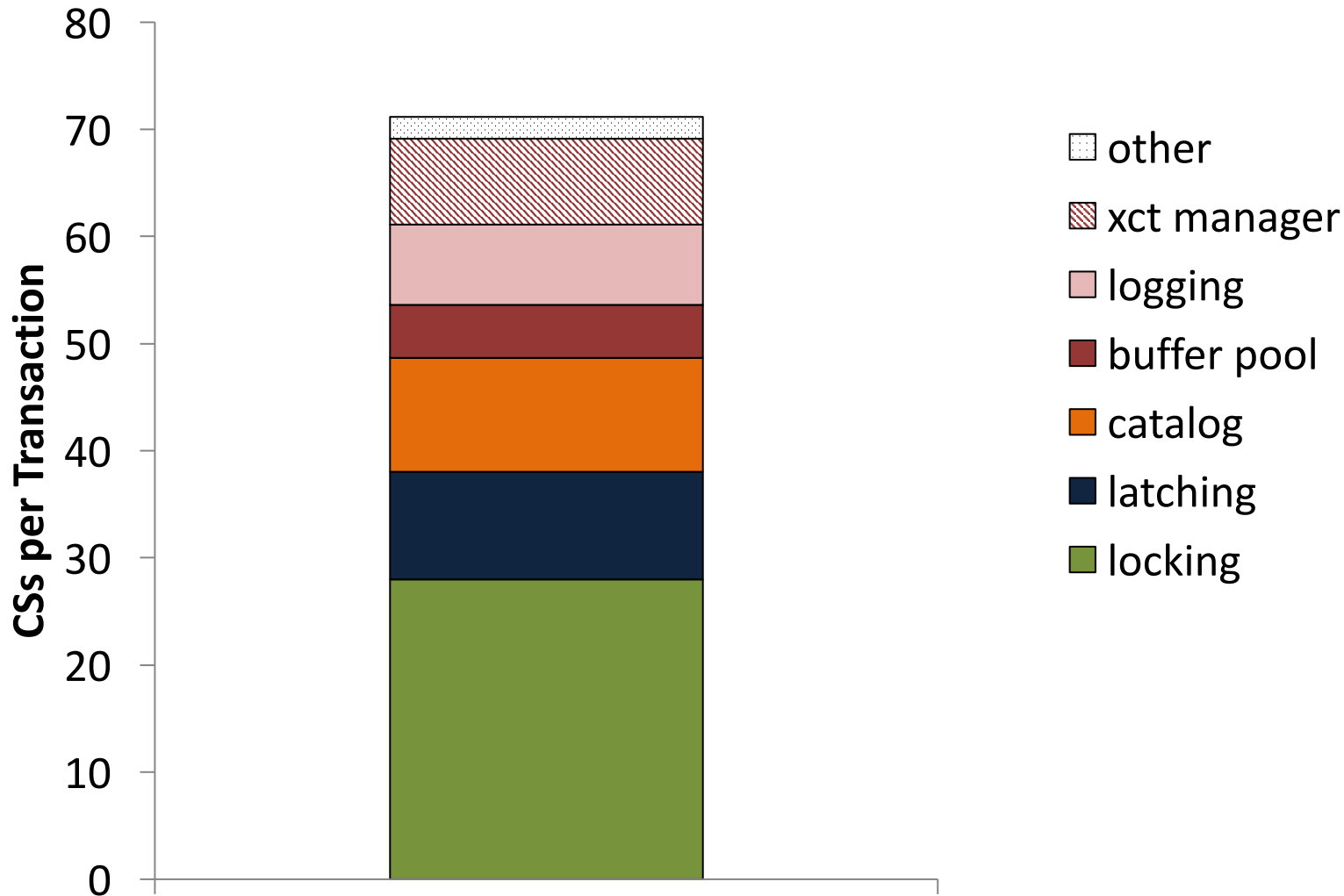


**unpredictable data accesses**

**clutter code with critical sections -> contention**

# Critical Sections

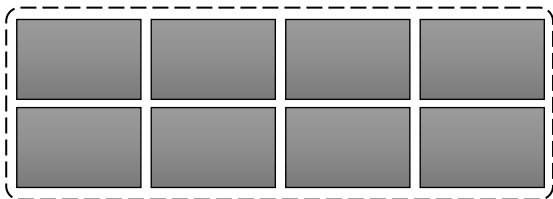
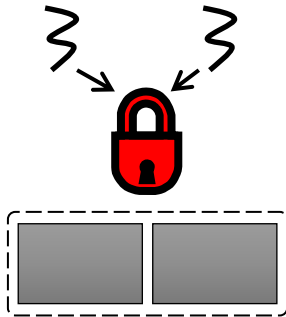
Updating 1 row



**many critical sections even for simplest transaction**

# Critical Section Types

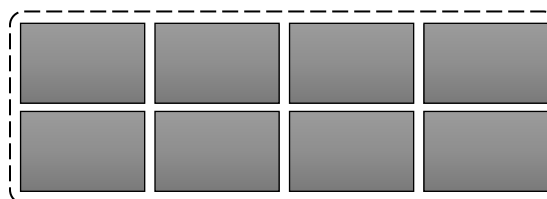
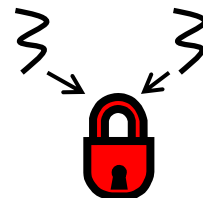
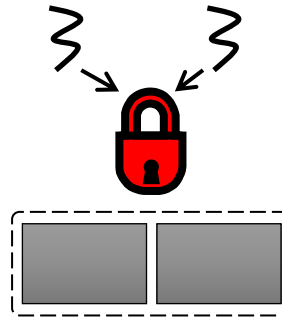
## unbounded



locking, latching

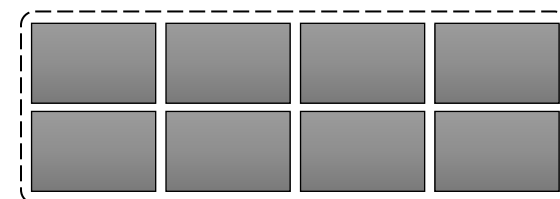
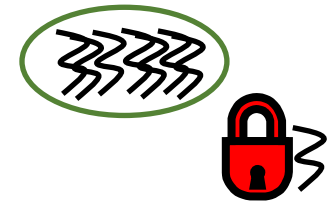
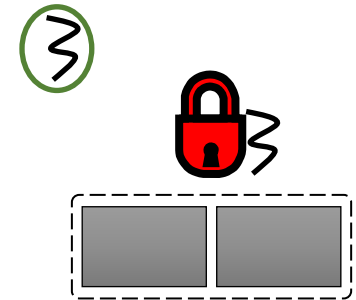


## fixed



transaction manager

## cooperative



logging



**unbounded → fixed / cooperative**



# Scaling up OLTP

## unscalable components

- locking
- latching
- logging

## synchronization

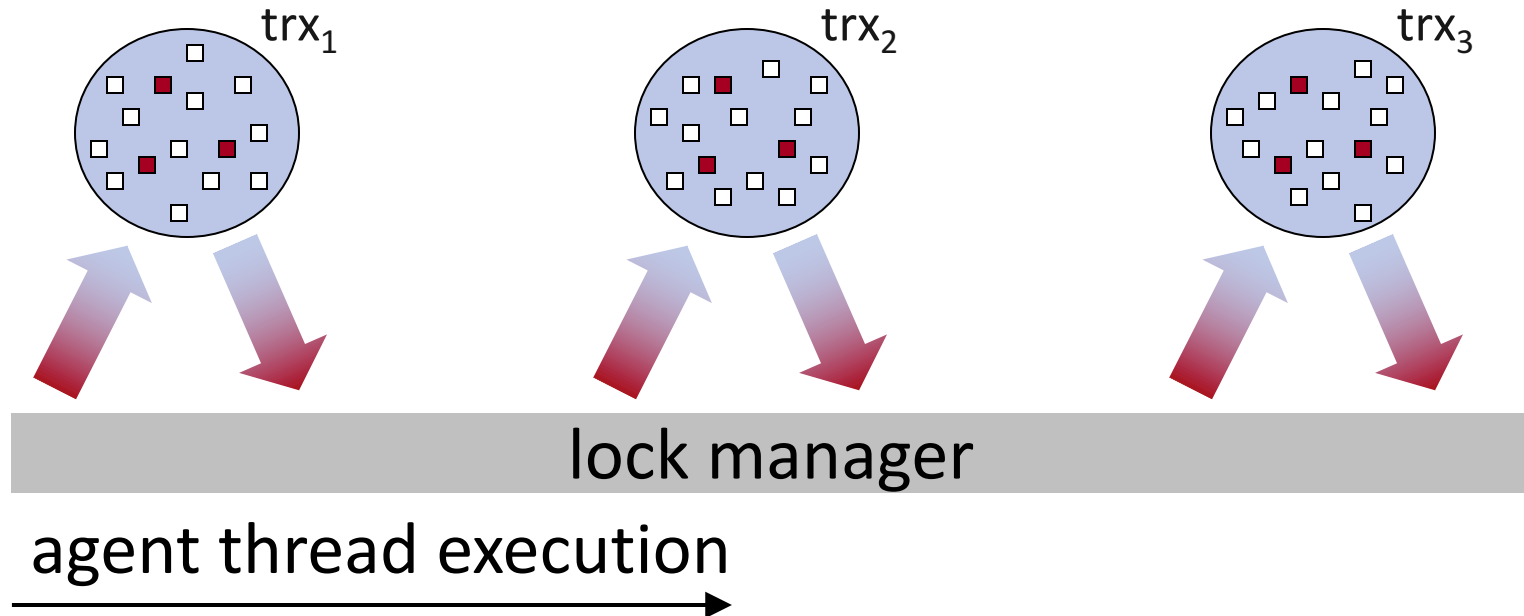
- tradeoffs
- best practices

## non-uniform communication

- hardware Islands

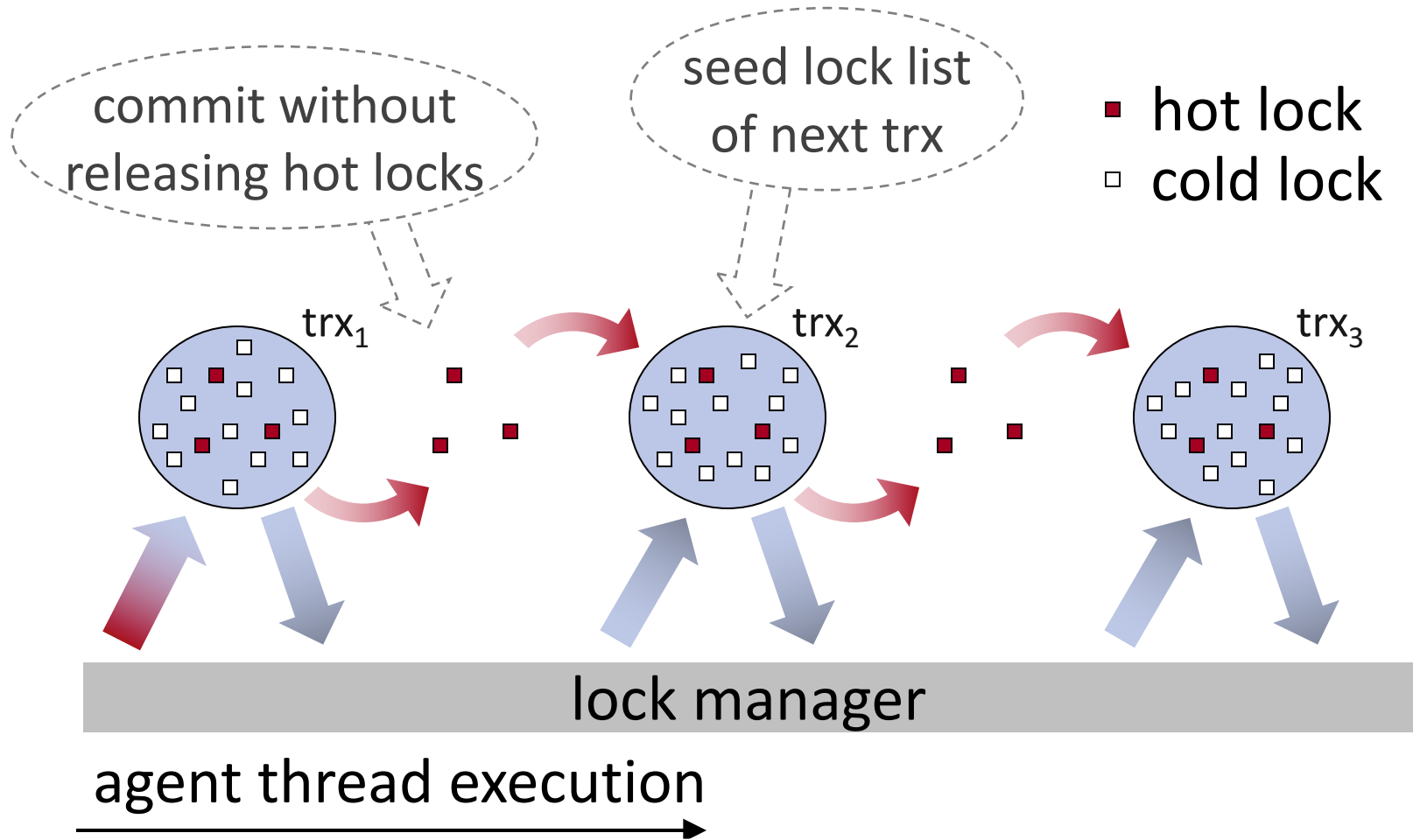
# Hot Shared Locks Cause Contention

- hot lock
- cold lock



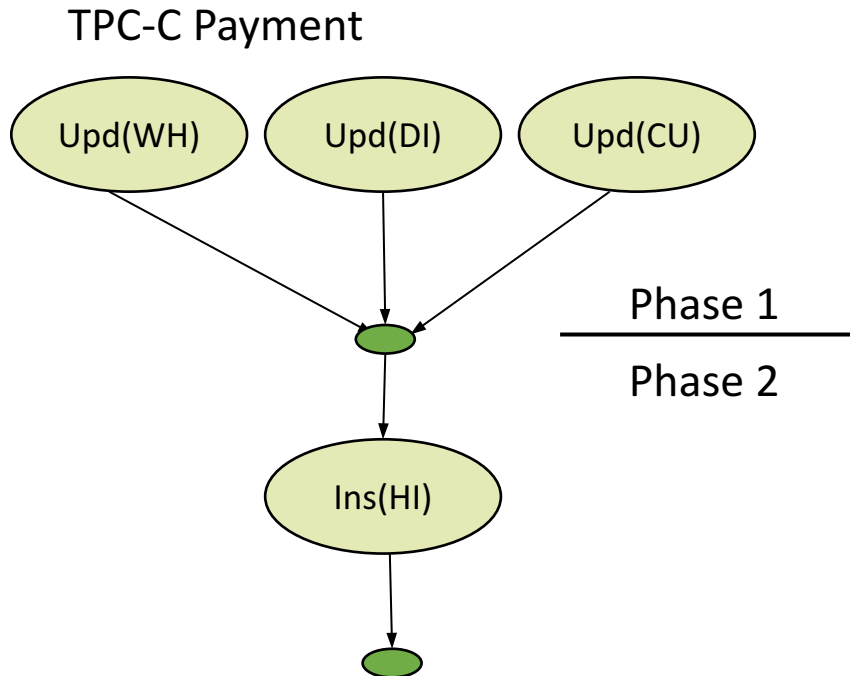
**release and request the same locks repeatedly**

# Speculative Lock Inheritance



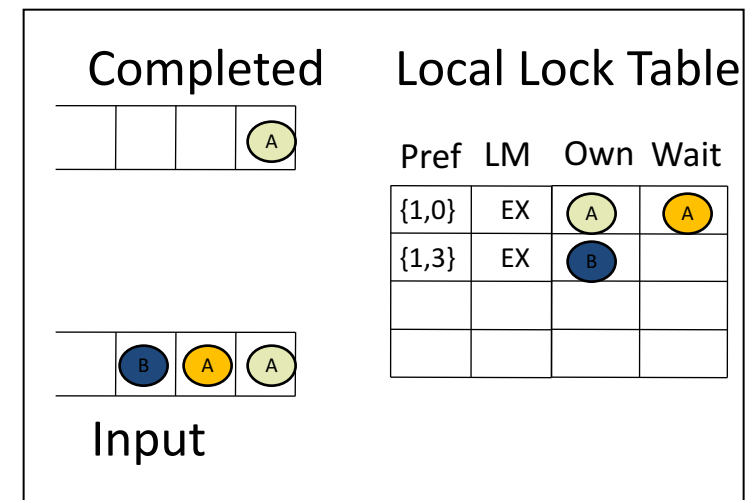
**significantly reduces lock contention**  
**co-locate atomic counters with data**

# Data-Oriented Transaction Execution



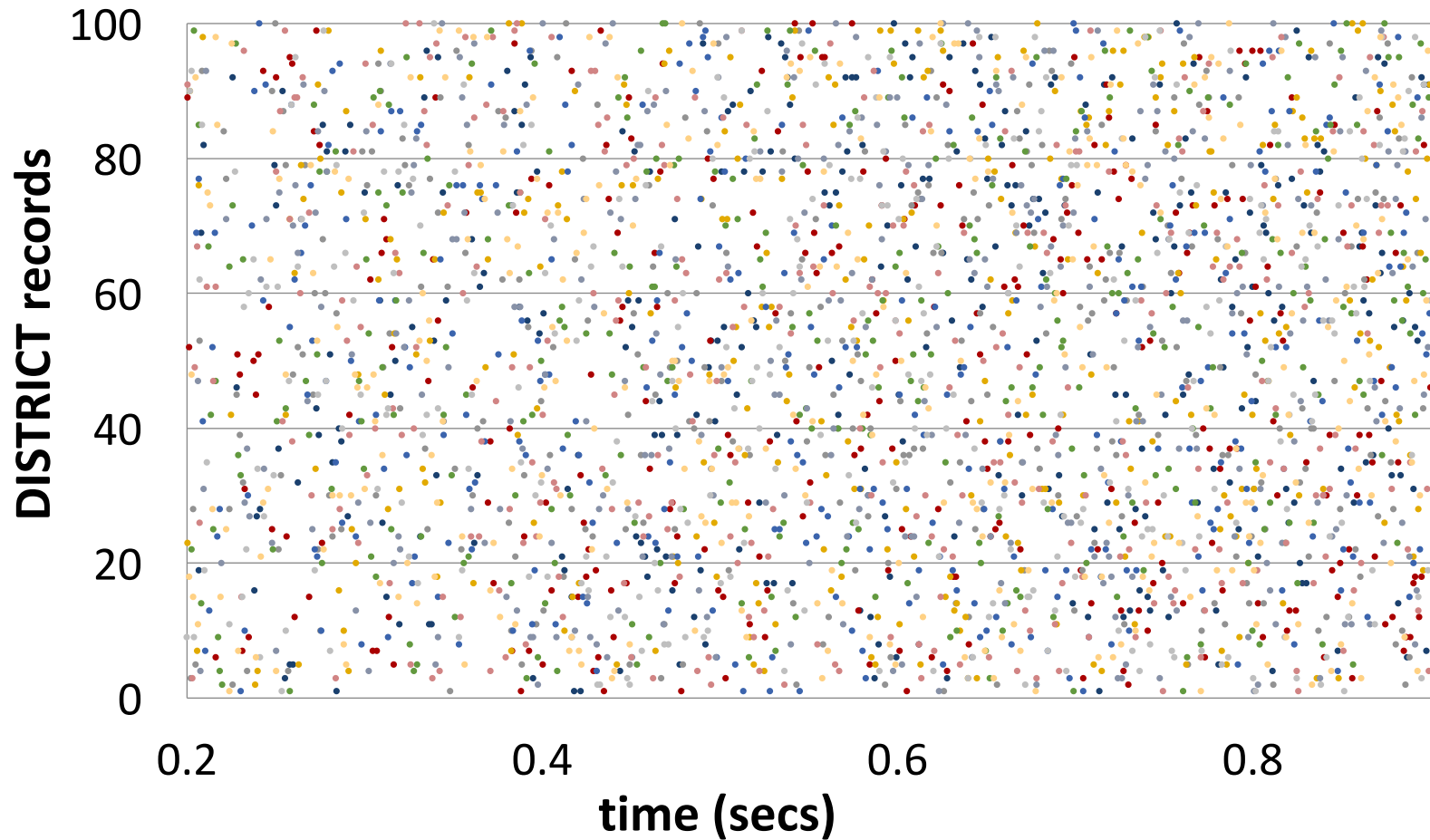
Routing fields: {WH\_ID, D\_ID}

Range	Executor
A-H	1
I-N	2

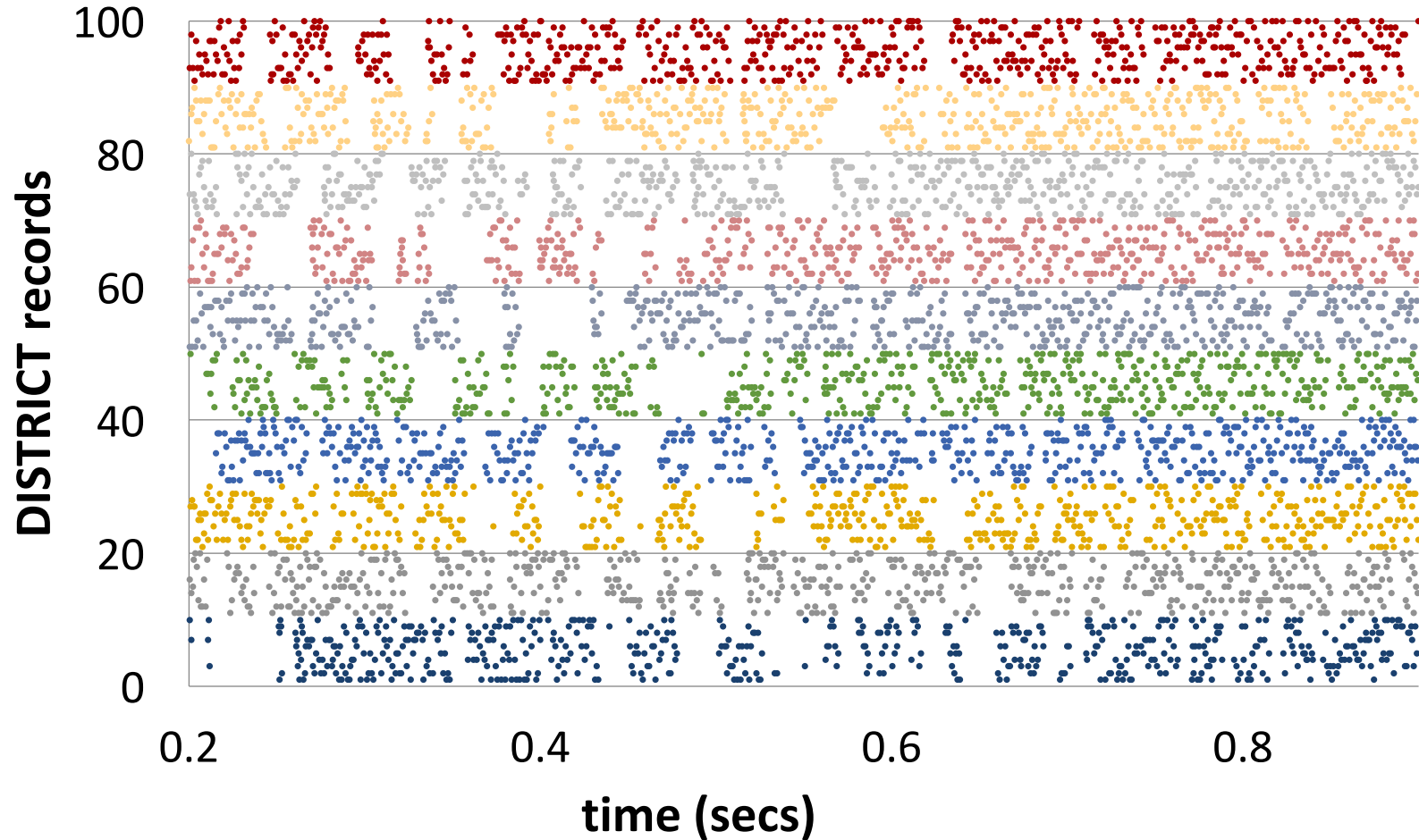


**convert centralized locking to thread-local**

# Thread-to-transaction - Access Pattern

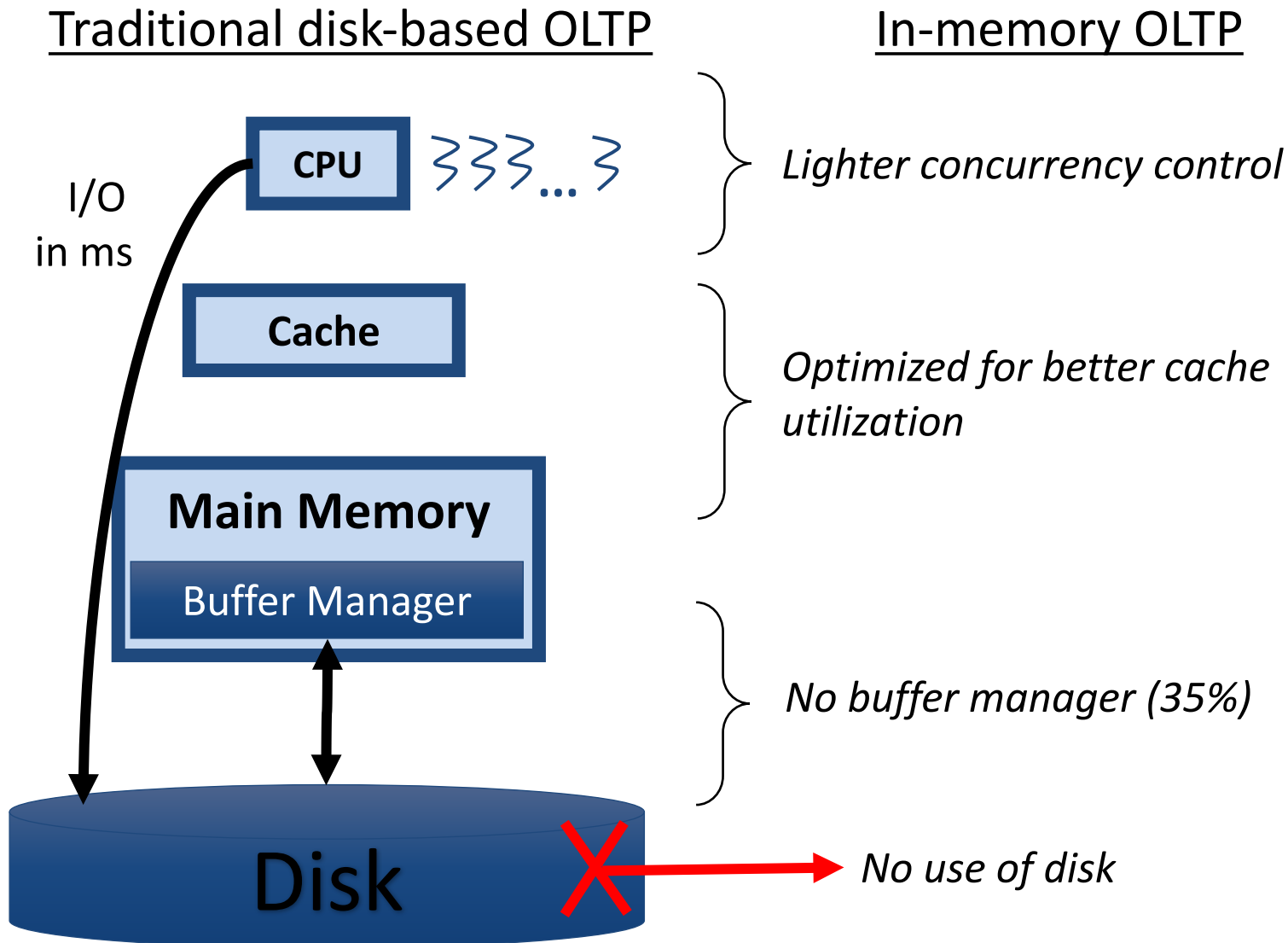


# Thread-to-data – Access Pattern



**predictable data accesses**

# In-memory Databases



# Scaling up OLTP

## unscalable components

locking  
latching  
logging

## synchronization

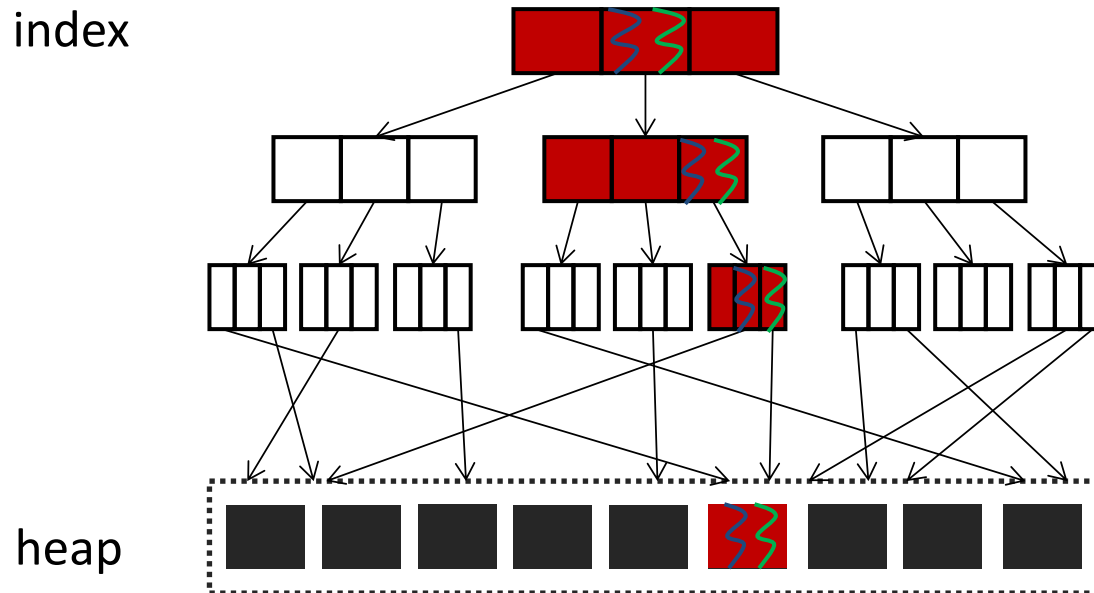
tradeoffs  
best practices

## non-uniform communication

hardware Islands



# Data Access in Centralized B-tree



**conflicts on both index and heap pages**

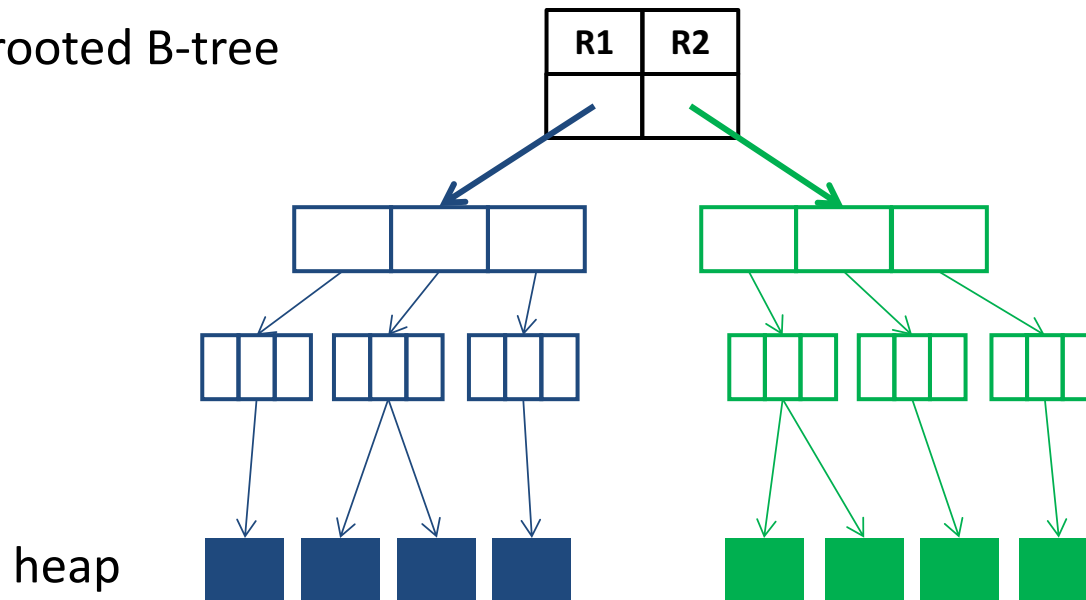
# Physiological Partitioning (PLP)

range	worker
A – M	⚡
N – Z	⚡

logical

physical

multi-rooted B-tree



# Scaling up OLTP

## unscalable components

locking  
latching  
logging

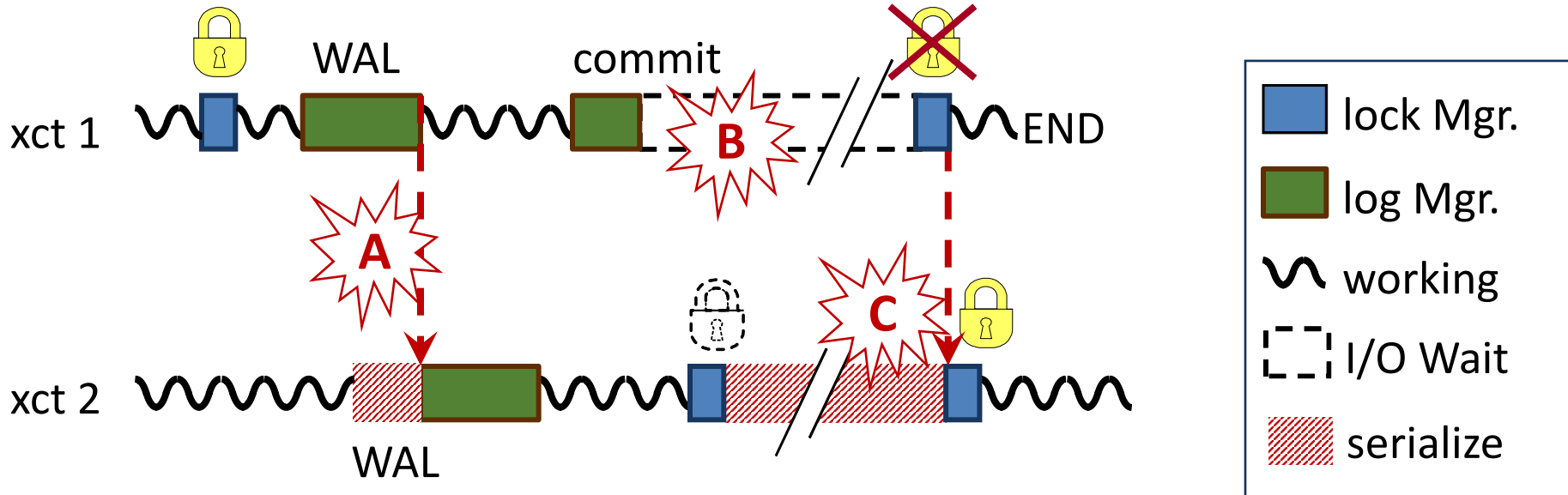
## synchronization

tradeoffs  
best practices

## non-uniform communication

hardware Islands

# A Day in the Life of a Serial Log



**A** serialize at the log head

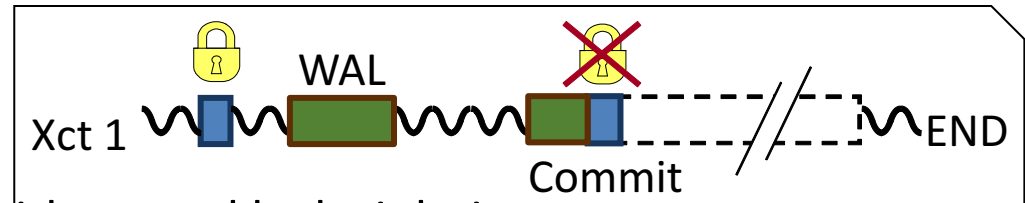
**B** I/O delay to harden the commit record

**C** serialize on incompatible lock

# Aether Holistic Logging

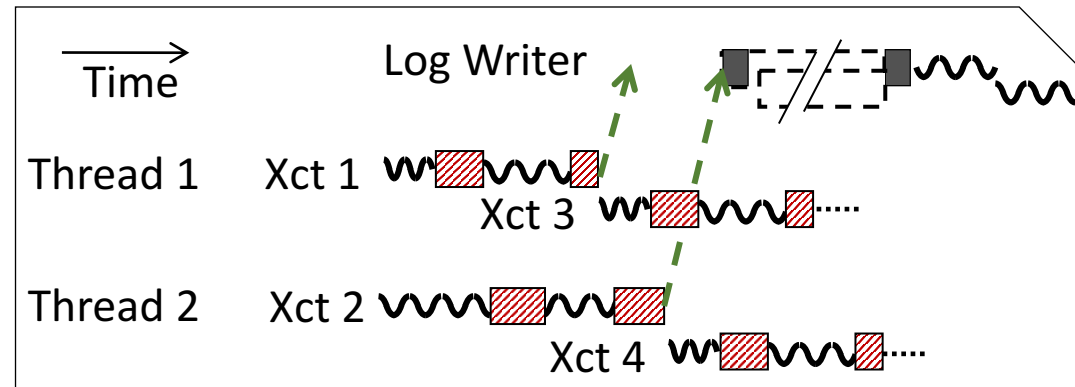
- early lock release

- can be improved further with control lock violation



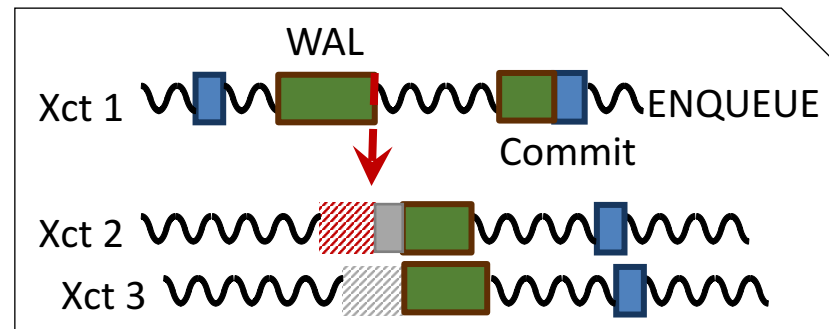
- flush pipelining

- reduces context switches



- consolidation array

- minimize log contention



# Scaling up OLTP

unscalable components

locking  
latching  
logging

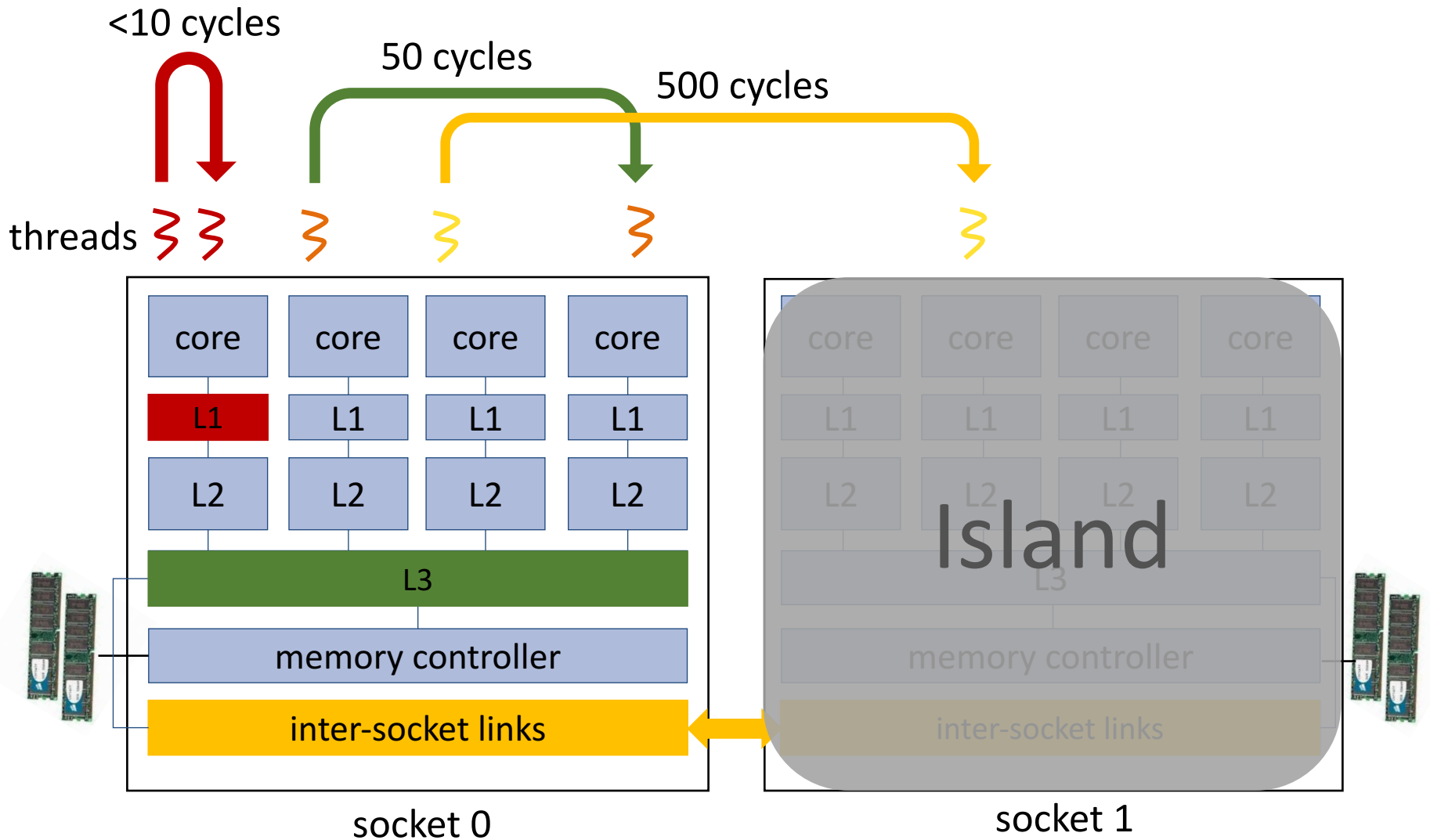
synchronization

tradeoffs  
best practices

non-uniform communication

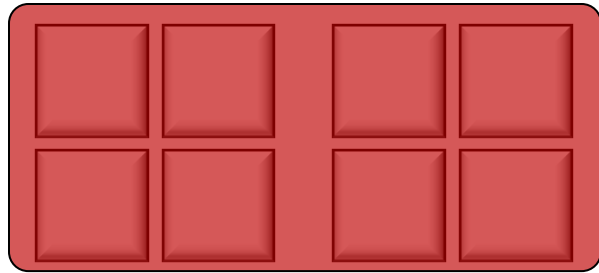
hardware Islands

# Multisocket Multicores



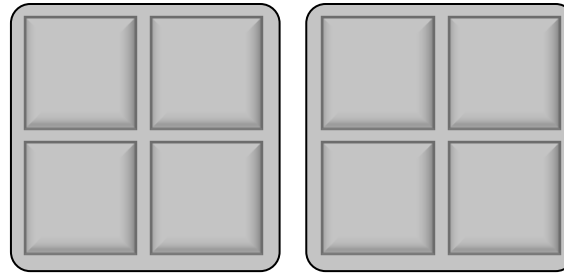
**communication latencies vary by order-of-magnitude**

# OLTP on Hardware Islands



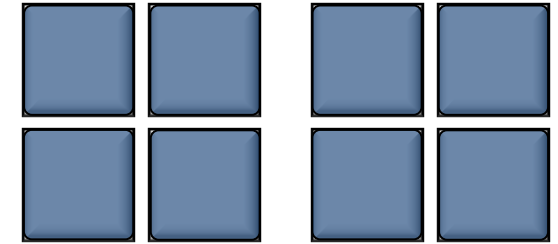
shared-everything

- ✓ stable
- ✗ not optimal



Island shared-nothing

- ✓ robust middle ground



shared-nothing

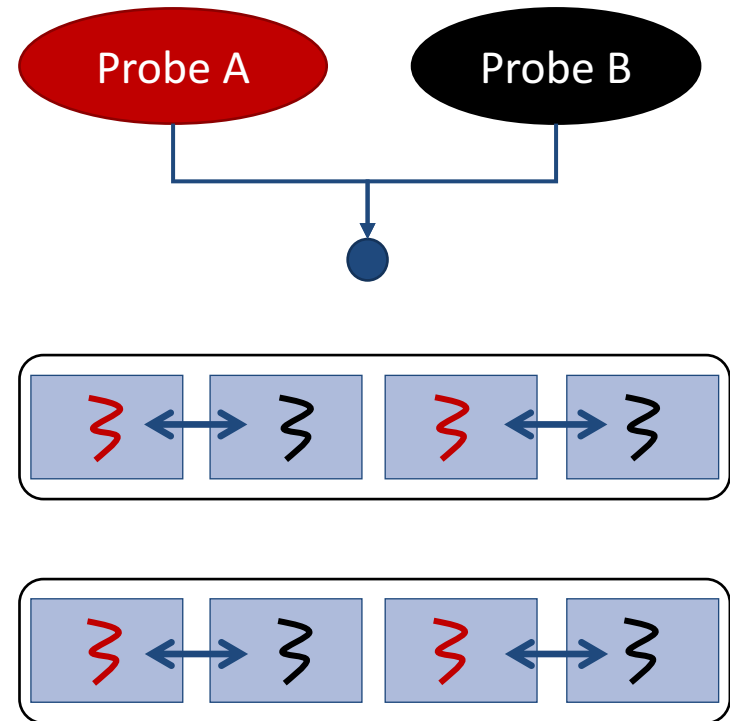
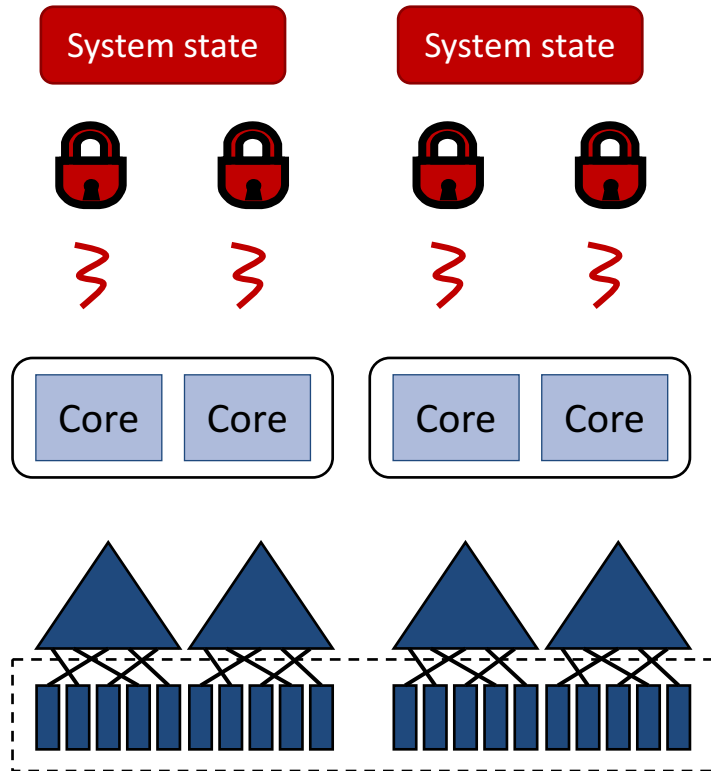
- ✓ fast
- ✗ sensitive to workload

## Challenges:

- optimal configuration depends on workload and hardware
- expensive repartitioning due to physical data movement



# Adaptive Transaction Processing



# Scaling up OLTP

- identify bottlenecks in existing systems
  - eliminate bottlenecks systematically and holistically
- design new system from the ground up
  - without creating new bottlenecks
- do not assume uniformity in communication
- choose the right synchronization mechanism