How to stop underutilization and love multicores

Anastasia Ailamaki (EPFL)
Erietta Liarou (EPFL)
Pınar Tözün (IBM Almaden)
Danica Porobic (EPFL)
Iraklis Psaroudakis (EPFL, SAP)

Longer version: http://tinyurl.com/LoveMulticores
once upon a time ...

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
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

TPC-C  TPC-E

Maximum

Execution Cycles Breakdown

Stalled  Busy

TPC-C  TPC-E

IPC < 1 on a 4-issue machine
70% of the execution time goes to stalls
horizontal dimension: cores & sockets

exploit abundant parallelism
workload scalability on multicores

OLTP

throughput

number of threads

access latency

OLAP

throughput

number of threads

memory bandwidth
stopping underutilization

• how to adapt traditional execution models to fully exploit modern hardware?

• how to maximize data & instruction locality at the right level of the memory hierarchy?

• how to continue scaling-up despite many cores and non-uniform topologies?
utilization

exploiting core’s resources
minimizing memory stalls

scalability

scaling up OLTP
scaling up OLAP
conclusions
modern parallelism

multithreading

horizontal parallelism
multicores

instruction & data parallelism
pipelining
superscalar
SIMD

SMT
fundamental way to parallelize

Instruction pipelining:
multiple instructions can be partially overlapped

Instruction pipelining:
multiple instructions can be partially overlapped

increase the instruction throughput
superscalar cpu

more than one instructions during a clock cycle
SISD

SIMD
SISD to SIMD

traditionally (SISD)

SIMD

apply the same action on multiple data values with the same cost as for 1 value
**SIMD** (single instruction multiple data)

How:

assembly or compilers provide special commands

```c
for(i=0;i<N;i++)
res+=a[i]
```

```c
for (i=0;i<N;i+=4)
res[i,i+1,i+2,i+3]=SIMD_add(res[i,i+1,i+2,i+3], a[i,i+1,i+2,i+3])
```

+ corner cases

**ignoring the for-loop code**

**we will do 4 times less instructions**
**SIMD** (single instruction multiple data)

Accelerate many data management primitives:
- filtering
- partitioning
- compression
- data movement with scatter/gather instructions

column-store model helps as data is already packed in dense arrays
modern parallelism

multithreading

instruction & data parallelism

- pipelining
- superscalar
- SIMD

SMT
SMT (simultaneous multithreading)

A SMT processor pretends to be multiple *logical* processors (one per instruction stream).

If one thread stalls another one can continue.
SMT – treat logical as physical

- minimal code changes
- ignorance of resource sharing
- competition for execution units

[VLDB05b]
SMT – multithreaded operators

- share input and output data in the cache
  - odd tuples
  - even tuples
  - separate output buffers
  - merging step

- beneficial for instruction & data cache performance
- reimplemention of dbms operators
- partitioning and merging

[VLDB05b]
preloading in SMT

preload data elements that will soon be needed

use one thread for the computation
and the other to manage resources

[VLDB05b]
SMT (simultaneous multithreading)

A SMT processor pretends to be multiple *logical* processors (one per instruction stream).

better than single threaded:
- increase thread-level parallelism
- improve processor utilization when one thread blocks

not as good as two physical cores
- cpu resources are shared, not replicated
from single core to multi-cores

work in parallel

how do we keep cpu at 100% ?
scan in multicores

1 core for each query

limited opportunities I/O sharing
scan in multicores

1 core for each table scan

load into caches once + share => reduce cache misses
sorting on multicore SIMD
sorting on multicore SIMD

\[ \frac{3}{2} \text{ core} \]

[VLDB08a]
sorting on multicore SIMD

2 cores work simultaneously to merge the pair of lists
modern parallelism

instruction & data parallelism
- pipelining
- superscalar
- SIMD

horizontal parallelism

multithreading
SMT

multicores
utilization

exploiting core’s resources
minimizing memory stalls

scalability

scaling up OLTP
scaling up OLAP
conclusions
today’s memory hierarchy

in practice

no penalty

possible stalls

MAIN MEMORY

stalls $\rightarrow$ wasted power & $$$$

i ❤️ multicores

latency

~4 cycles

~12 cycles

~30 cycles

~200 cycles

L1-I L1-D

L2

L3 / LLC

L1-I L1-D

L2
stalls in cloud workloads

CloudSuite on Intel Xeon X5670

~1 instructions per cycle

> 50% of the time goes to stalls on average

graph courtesy of Ferdman et al.

[ASPLOS12]
sources of memory stalls

TPC-C, 100GB data on Intel Xeon E5-2640 v2

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 → fetch A+1
• stream: miss A, A+1 → fetch A+2, A+3

✓ favors sequential access & spatial locality

✗ instructions: branches, function calls
  • branch prediction

✗ data: pointer chasing
  • stride: miss A, A+20 → fetch A+40, A+60

preferred on real hardware due to simplicity

though, memory stalls are still too high
temporal streaming

exploits recurring control flow
more accurate → higher space cost
code optimizations

• simplified code
  – in-memory databases have smaller instruction footprint

• better code layout
  – minimize jumps \(\rightarrow\) exploit next line prefetcher
  – profile-guided optimizations (static)
  – just-in-time (dynamic)

• query compilation into machine/naïve code
  – e.g., HyPer, Hekaton, MemSQL, Impala
# cache conscious data layouts

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**

---

<table>
<thead>
<tr>
<th>erietta</th>
<th>blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>pinar</td>
<td>black</td>
</tr>
<tr>
<td>danica</td>
<td>green</td>
</tr>
<tr>
<td>iraklis</td>
<td>orange</td>
</tr>
</tbody>
</table>

16 bytes columns

---

<table>
<thead>
<tr>
<th>erietta</th>
<th>blue</th>
<th>pinar</th>
<th>black</th>
</tr>
</thead>
<tbody>
<tr>
<td>erietta</td>
<td>pinar</td>
<td>danica</td>
<td>iraklis</td>
</tr>
</tbody>
</table>

row store  
column store
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

[PLDB14f]
computation spreading

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
utilization

exploiting core’s resources
minimizing memory stalls

scalability

scaling up OLTP
scaling up OLAP
conclusions
challenges when scaling up

OLTP

throughput

number of threads

access latency

OLAP

throughput

number of threads

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

many critical sections even for simplest transaction

Updating 1 row
critical section types

unbounded

fixed

cooperative

locking, latching

transaction manager

logging

unbounded $\rightarrow$ fixed / cooperative

[VLDBJ14]
scaling up OLTP

unscalable components
- locking
- latching
- logging

non-uniform communication
- hardware Islands
hot shared locks cause contention

- hot lock
- cold lock

release and request the same locks repeatedly
speculative lock inheritance

- commit without releasing hot locks
- seed lock list of next trx

- hot lock
- cold lock

significantly reduces lock contention
LIL: co-locate atomic counters with data
data-oriented transaction execution

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

<table>
<thead>
<tr>
<th>Range</th>
<th>Executor</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-H</td>
<td>1</td>
</tr>
<tr>
<td>I-N</td>
<td>2</td>
</tr>
</tbody>
</table>

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

Completed

Local Lock Table

<table>
<thead>
<tr>
<th>Pref</th>
<th>LM</th>
<th>Own</th>
<th>Wait</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1,0}</td>
<td>EX</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>{1,3}</td>
<td>EX</td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

Input
thread-to-transaction - access pattern

[PVLDB10b]

DISTRICT records

time (secs)
thread-to-data – access pattern

[pvlb10b]
in-memory databases

Traditional disk-based OLTP

I/O in ms

Main Memory

Buffer Manager

Cache

CPU

Disk

No buffer manager (35%)

Lighter concurrency control

Optimized for better cache utilization

No use of disk

In-memory OLTP
multicore optimized OLTP

- **H-Store/VoltDB and HyPer**
  - single-threaded execution

- **Calvin**
  - deterministic execution model with conflict detection

- **Hekaton and Silo**
  - OCC with parallel validation schemes

[VLDB07b, ICDE11]

[SIGMOD12]

[PVLDB12c, SOSP13]
data access in centralized B-tree

conflicts on both index and heap pages
physiological partitioning (PLP)

<table>
<thead>
<tr>
<th>range</th>
<th>worker</th>
</tr>
</thead>
<tbody>
<tr>
<td>A – M</td>
<td>3</td>
</tr>
<tr>
<td>N – Z</td>
<td>3</td>
</tr>
</tbody>
</table>

multi-rooted B-tree

logical physical

heap

multi-rootsen B-tree

[PVLDB11b]
BW-tree

- latch-free log-structured B-tree
- optimized for both main memory and flash
- no updates in place -> delta updates

[ICDE13c, PVLDB13a]
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
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

- adaptive OLTP design for Islands
  - no unnecessary inter-socket synchronization
  - workload & hardware-aware partitioning
  - lightweight monitoring and repartitioning

References: [PVLDB12d, ICDE14d]
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
utilization

exploiting core’s resources
minimizing memory stalls

scalability

scaling up OLTP
scaling up OLAP

conclusions
Scaling up OLAP

*parallelizing a single aggregation*

Sharing across queries

OLAP is concerned also with resources saturation

[DaMoN14b, SIGMOD14b]
bottlenecks in NUMA architectures

(1) underutilization, oversubscription
(2) cache efficiency
(3) remote access latency (1.5x local)
(4) interconnect bandwidth (12GB/s)
(5) memory bandwidth (25GB/s)

numerous points to consider for NUMA-awareness

[USENIX11]
scaling up OLAP

sharing
  common sub-plans
  shared operators

NUMA-awareness
  application-agnostic
database operators

scheduling
  task scheduling
  NUMA-aware task scheduling
sharing is caring...

in the era of big data

...for resources
sharing techniques

### query-centric
- caching
- materialized views
- multi-query optimization
- buffer pool management

### reactive sharing
- query-centric
- shares common sub-plans
- shared scans

### proactive sharing
- global query plan with shared operators
- shared scans

<table>
<thead>
<tr>
<th>Sharing Type</th>
<th>QPipe</th>
<th>CJOIN</th>
<th>DataPath</th>
<th>SharedDB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>reactive</td>
<td>proactive (global query plan)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution</td>
<td>dynamic</td>
<td>dynamic</td>
<td>dynamic</td>
<td>batched</td>
</tr>
<tr>
<td>Schema</td>
<td>general</td>
<td>star</td>
<td>general</td>
<td>general (pre-comp.)</td>
</tr>
</tbody>
</table>

[SIGMOD14b]  

QPipe [SIGMOD05]  
CJOIN [VLDB09a]  
DataPath [SIGMOD10a]  
SharedDB [PVLDB12b, PVLDB14b]
reactive sharing: how to react?

[VLDB07a, PVLDB13b]

query-centric

Q1

Σ

FIFO buffer

Pull

Σ

by pulling shared intermediate results

common sub-plans

Pull

Σ

serialization point

forward results

Push

Σ

move independently
proactive sharing

SELECT * FROM A, B
WHERE A.c₁ = B.c₁
AND σ(A) AND σ(B)

SELECT * FROM A, B
WHERE A.c₁ = B.c₁
AND σ'(A) AND σ'(B)

shared operators can support high throughput
application-agnostic NUMA-awareness

- black box approach
  - monitoring to predict behavior
- DINO scheduler
  - moves threads and their data to balance cache load
- Carrefour
  - re-organizes data to avoid memory bottlenecks
    - by: replicating, interleaving or co-locating data

not always optimal for DBMS
when to partition across sockets?

1 hot table

3 hot tables

16 sockets x 15 cores

adapt partitioning to access patterns

[PVLDB15e]
data shuffling

- $N$ threads, each partitions its local data into $N$ equally-sized pieces, transmitted to the rest
- naïve method:

saturates memory and interconnects
coordinated shuffling

balances memory and interconnect traffic
scheduling work

- OS scheduler

[ADMS13]
scheduling work

- OS scheduler
- task scheduler

Oversubscription

Context switch

Cache thrashing

task queues

socket 1

socket 2

a solution for DBMS to efficiently utilize resources
task scheduling for OLAP

- **HyPer**
  - Partition data by small morsels across sockets
  - Pass morsels through whole operator pipelines
embrace...

• sharing
  – reduces contention for resources
  – reactive and proactive

• NUMA-awareness
  – reduce latency and avoid bottlenecks
  – data placement and thread scheduling
  – black box approach not always optimal
  – algorithms

• task scheduling
  – abstract resources and utilize them efficiently

...to scale up OLAP
utilization

exploiting core’s resources
minimizing memory stalls

scalability

scaling up OLTP
scaling up OLAP

conclusions
Map of the jungle

**Hardware**
- Prefetching
- Superscalar
- Specialization

**Cache locality**
- Data layout
- Computation spreading
- Code optimizations
- Vectorized execution

**Multicores**
- HTM
- NVram
- Fast durability
- Lightweight concurrency control

**Multisockets**
- Data placement
- Task scheduling
- Topology-aware OTLP
- RAVs (black-box)
- NUMA-aware operators

**Software**
- Operators & algorithms
- Software-guided prefetching
- Cache-conscious structures
- Multithreaded operators

**Operators & algorithms**
- DORA
- Synchronization
- Adaptive indexing
- Latch-free structures
- Sharing multi-query scheduling

**Energy effic. (DVFS etc.)**
concluding remarks

exploiting hardware requires
– utilizing the resources of a core
– taking advantage of parallelism
– optimally managing the memory

art of scheduling
– adjust your task granularity
– optimize locality at the right level
– avoid saturation

road to scalability
– eliminate all unbounded communication

bridge the gap between software & hardware
winter is coming...

- Transistor Scaling (Moore's Law)
- Supply Voltage (ITRS)

exponential increase in unusable area on chips
age of dark silicon is upon us!
exploiting dark silicon

• Meet the walkers
• Database processing unit
• Programmable accelerators
• Bionic databases
• Reconfigurable datacenters
• Commercial: RAPID

toward specialized hardware
open questions – How to ... 

- fit NVRAM to memory hierarchy?
- exploit HTM?
- adapt the whole software stack (OS + applications) to hardware specialization?
- take advantage of compilers?
- design concurrency-control for many-cores?

[PVldb14c, PVldb14d, PVldb15f]
references


references


[CIDR13a] R. Johnson and I. Pandis: The bionic DBMS is coming, but what will it look like?


[DaMoN13] P. Tözün, B. Gold, and A. Ailamaki: OLTP in Wonderland -- Where do cache misses come from in major OLTP components?

[DaMoN14a] H. Pirk, E. Petraki, S. Idreos, S. Manegold, and M.L. Kersten: Database Cracking: Fancy Scan, not Poor Man’s Sort!


references


[Eurosys14] Z. Wang, H. Qian, J. Li, and H. Chen: Using Restricted Transactional Memory to Build a Scalable In-Memory Database.


[ICDE13a] J. Dees and P. Sanders: Efficient many-core query execution in main memory columnstores


references


[ICDE14c] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker: Rethinking Main Memory OLTP Recovery.


references

[PVLDB10b] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki: Data-Oriented Transaction Execution.
references


references


[PVLDB15a] A. Chatzistergiou, M. Cintra, S. Viglas: REWIND: Recovery Write-Ahead System for In-Memory Non-Volatile Data-Structures

[PVLDB15b] S. Chen, Q. Jin: Persistent B+Trees in NonVolatile Main Memory


references

references


[TOCS03] M. Annavaram, J. M. Patel, E. S. Davidson: Call graph prefetching for database applications.


references


[VLDB07a] R. Johnson, S. Harizopoulos, N. Hardavellas, K. Sabirli, I. Pandis, A. Ailamaki, N. G. Mancheril, and B. Falsafi: To share or not to share?

[VLDB07b] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland: The end of an architectural era: (it’s time for a complete rewrite).


[VLDB08b] Lin Qiao, Vijayshankar Raman, Frederick Reiss, Peter J. Haas, and Guy M. Lohman. Main-Memory Scan Sharing For Multi-Core CPUs


