How to stop underutilization and love multicores

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

goal: scalability
vertical dimension: cores & caches

memory matters
now: cores & cache utilization

at peak throughput on Shore-MT, Intel Xeon X5660

 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

http://tinyurl.com/tutorial2014feedback
modern parallelism
modern parallelism

instruction & data parallelism

- pipelining
- superscalar
- SIMD
modern parallelism

- instruction & data parallelism
  - pipelining
  - superscalar
  - SIMD
- core
- multithreading
- SMT
- i ♥ multicores
modern parallelism

instruction & data parallelism
- pipelining
- superscalar
- SIMD

multithreading
- SMT

horizontal parallelism
- multicores
subscalar CPUs

one instruction at a time

clock cycle

1  2  3  4  5  6  7  8  9  10  11  12
subscalar CPUs

one instruction at a time

... ten cycles to complete 2 instructions!
subscalar CPUs

one instruction at a time

... ten cycles to complete 2 instructions!
subscalar CPUs

one instruction at a time

... ten cycles to complete 2 instructions!
subscalar CPUs

one instruction at a time

... ten cycles to complete 2 instructions!
fundamental way to parallelize

**Instruction pipelining:**
multiple instructions can be partially overlapped
fundamental way to parallelize

**Instruction pipelining:**
multiple instructions can be partially overlapped

Increase the utilization of on-die execution resources
fundamental way to parallelize

**Instruction pipelining:**
multiple instructions can be partially overlapped

Increase the utilization of on-die execution resources
fundamental way to parallelize

**Instruction pipelining:**
multiple instructions can be partially overlapped

... six cycles to complete 2 instructions!
fundamental way to parallelize

**Instruction pipelining:**
multiple instructions can be partially overlapped

Instruction pipelining allows multiple instructions to execute concurrently. Each instruction stage (fetch, decode, execute, memory access, write) can be processed in parallel, reducing the overall execution time from six cycles to two cycles for two instructions.

**Increase the instruction throughput**

Increase the instruction throughput by overlapping the execution of multiple instructions, which is achieved through instruction pipelining.
superscalar cpu

more than one instructions during a clock cycle
superscalar cpu

4 instructions during a clock cycle
instructions

SISD

data

results
**SIMD** (single instruction multiple data)

Apply an instruction to multiple data elements

- allows parallelism
- process of \( K \) elements at a time → speedup of \( K \)

processing large arrays of numeric values

e.g., the same value is being added to a large number of data points
SIMD (single instruction multiple data)
SIMD (single instruction multiple data)

**K**-wide SIMD $\rightarrow$ **K** x faster
SISD to SIMD

traditionally (SISD)
SISD to SIMD

traditionally (SISD)
SISD to SIMD

traditionally (SISD)
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)

SIMD runs at the same time. Each `add` operation takes input 1 (128bits) and input 2 (128bits) and results in a 128bits result.

- Input 1: 128bits
- Input 2: 128bits
- Result: 128bits
**SIMD (single instruction multiple data)**

SIMD runs at the same time on multiple data points, allowing for efficient parallel processing.

**Input:**
- Input 1: 128 bits
- Input 2: 128 bits

**Result:** 128 bits

```
min

0, 4294967295, 0, 4294967295
```

```
min

3 5 10 3
```

```
min

2 21 1 2
```

Result: 128 bits
**SIMD** (single instruction multiple data)

runs at the same time

32-bit

11...1

result: 128bits

input 1: 128bits

input 2: 128bits

0, **4294967295**, 0, **4294967295**

```
3 5 10 3
2 21 1 2
```
**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) [SIGMOD02]

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])

what is next?
**SIMD** (single instruction multiple data)

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])
```

what is next?

- CPU registers
- SIMD registers

**SIMD**_shuffle32: [A,B,C,D] -> [B,A,D,C]

**SIMD**_shuffle64: [A,B,C,D] -> [C,D,A,B]
**SIMD** *(single instruction multiple data)*

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])

**what is next?**

SIMD_shuffle32: [A,B,C,D] -> [B,A,D,C]

SIMD_shuffle64: [A,B,C,D] -> [C,D,A,B]

\[\text{t1} = \text{SIMD_shuffle32}(\text{res})\]
\[\text{t2} = \text{SIMD_add}(\text{res}, \text{t1})\]
\[\text{t3} = \text{SIMD_shuffle64}(\text{t2})\]
\[\text{res} = \text{SIMD_add}(\text{t2}, \text{t3})\]
SIMD \textsuperscript{(single instruction multiple data)}

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

- Core

instruction & data parallelism

- Pipelining
- Superscalar
- SIMD
modern parallelism

- instruction & data parallelism
  - pipelining
  - superscalar
  - SIMD
modern parallelism

instruction & data parallelism
  pipelining
  superscalar
  SIMD

multithreading

SMT

core
SMT (simultaneous multithreading)

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

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

“30% performance gain” -- Intel

*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 - 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

beneficial for instruction & data cache performance

reimplementation of dbms operators

[Vldb05b]
SMT - multithreaded operators

share input and output data in the cache

- odd tuples
- even tuples
- separate output buffers
- merging step
- no longer preserving the order of input records

beneficial for instruction & data cache performance

reimplementation of dbms operators

partitioning and merging
preloading in SMT

preload data elements that will soon be needed

- use one thread for the computation
- and the other to manage resources
preloading in SMT

preload data elements that will soon be needed

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

[VLDB05b]
preloading in SMT

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

preload data elements that will soon be needed

main worker

helper

CPU

core

registers

L1

L2

memory

work-ahead set

data structure

memory address

preload

request
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
from single core to multi-cores

work in parallel

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

1 core for each query

Q1  Q2  Q3  Q4

c0  
c1  
c2  
c3  

reg  reg  reg  reg

L1  L1  L1  L1

L2

memory

[VLDB08b]
scan in multicores

1 core for each query

Q1  Q2  Q3  Q4

c0  c1  c2  c3
reg  reg  reg  reg
L1  L1  L1  L1

1 core for each query

Data blocks

Q1  Q2  Q3  Q4  Q5  Q6  Q7

memory

achieve limited I/O sharing via the convoy phenomenon
scan in multicores

1 core for each table scan

\{Q_1, \ldots, Q_n\}

\begin{itemize}
\item $c_0$ reg
\item $c_1$ reg
\item $c_2$ reg
\item $c_3$ reg
\end{itemize}

\begin{itemize}
\item $L_1$
\item $L_1$
\item $L_1$
\item $L_1$
\end{itemize}

L2

data

memory

[VLDB08b]
scan in multicores

1 core for each table scan

{Q1,...,Qn}  {Q1,...,Qn}  {Q1,...,Qn}  {Q1,...,Qn}

\[ \text{block1} \quad \text{block2} \quad \text{block3} \quad \text{block4} \]

\[ \text{L1} \quad \text{L1} \quad \text{L1} \quad \text{L1} \]

\[ \text{L2} \]

\[ \text{memory} \]
scan in multicores

1 core for each table scan

{Q1,...,Qn} {Q1,...,Qn} {Q1,...,Qn} {Q1,...,Qn}

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

\begin{align*}
&\begin{array}{c}
2 & 2 \\
5 & 5 \\
\end{array} \\
\text{low} & \text{high} \\
\end{align*}
sorting network
bitonic merge kernel

[VLDB08a]
bitonic merge kernel

[VLDB08a]
bitonic merge kernel

[VLDB08a]
bitonic merge kernel

[VLDB08a]
bitonic merge kernel

[VLDB08a]
bitonic merge kernel

[VLDB08a]
bitonic merge kernel

[VLDB08a]
**bitonic merge kernel**

![Diagram of bitonic merge kernel with levels and arrows indicating flow.](image-url)
bitonic merge kernel

bitonic merge network in SIMD multicores?
bitonic merge kernel with SIMD

simultaneous comparisons

L1 = SIMD_min(A, B);
H1 = SIMD_max(A, B);
L1p = SIMD_shuffle(L1);
H1p = SIMD_shuffle(H1);

[VLDB08a]
sorting on multicore SIMD

$N$
sorting on multicore SIMD

$N \rightarrow M$ [VLDB08a]
sorting on multicore SIMD

\[ N \quad \text{\[VLD08a]\]} \]

\[ M \]

\[ k \quad \text{...} \quad \text{...} \]
sorting on multicore SIMD

\[ N \]

\[ M \]

\[ k \]

[VLDB08a]
sorting on multicore SIMD

\[ N \]

\[ M \]

\[ k \]

\[ \frac{1}{3} \text{core} \]

\[ \frac{1}{3} \text{core} \]

\[ \frac{1}{3} \text{core} \]

\[ \frac{1}{3} \text{core} \]

\[ \ldots \]

\[ \frac{1}{3} \text{core} \]

\[ \ldots \]

\[ \frac{1}{3} \text{core} \]

\[ \ldots \]

\[ \frac{1}{3} \text{core} \]

\[ \ldots \]

\[ \frac{1}{3} \text{core} \]

\[ \ldots \]

[VLDB08a]
sorting on multicore SIMD

[VLDB08a]
sorting on multicore SIMD

[VLDB08a]
sorting on multicore SIMD

2 cores work simultaneously to merge the pair of lists
sorting on multicore SIMD

2 cores work simultaneously to merge the pair of lists
sorting on multicore SIMD

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

instruction & data parallelism
  - pipelining
  - superscalar
  - SIMD

multithreading

horizontal parallelism
  - multicores

SMT
utilization

exploiting core’s resources
minimizing memory stalls

scalability

scaling up OLTP
scaling up OLAP
conclusions

http://tinyurl.com/tutorial2014feedback
today’s memory hierarchy

in practice
no penalty
possible stalls

stalls \rightarrow \text{wasted power & $$$}$
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

L1-I & LLC data misses dominate the stall time

[DaMoN13, EDBT13]
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
- batching
- computation spreading
prefetching – lite

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

\[ \checkmark \text{favors sequential access & spatial locality} \]
\[ \times \text{instructions: branches, function calls} \]
  - branch prediction
\[ \times \text{data: pointer chasing} \]
  - stride: miss A, A+20 \(\rightarrow\) fetch A+40, A+60

preferred on real hardware due to simplicity
though, memory stalls are still too high

[ISCA90, MICRO00]... or text-book prefetching
temporal streaming

“EPFL”

lookup( )

time

cache accesses

traverse( )
cache blocks

slide courtesy of Cansu Kaynak

[ISCA05, MICRO13a]
temporal streaming

"SIGMOD"

cache accesses

time

exploits recurring control flow
high space cost
software-guided prefetching [Eurosys12, TOCS03]

only for data on real hardware
minimizing memory stalls

**Prefetching**
- lightweight
- temporal stream
- software-guided

**Being cache conscious**
- code optimizations
- alternative data structures/layout
- vectorized execution

**Exploiting common instructions**
- batching
- computation spreading
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
cache conscious data layouts

[SIGMOD85, CIDR05, VLDB05a]

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*

16 bytes 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>

cache lines (64bytes)

<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
cache conscious data structures

[SIGMOD02a, VLDB06]

index tree

in memory

lookup-heavy workload

scan-heavy workload

+ align nodes to cache lines

goal: maximize cache line utilization & exploit next-line prefetcher in tree probe
volcano iterator model

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

× poor data & instruction cache locality
vectorized execution

<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>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

next() next() next()

SELECT

SCAN

✓ good data & instruction cache locality
✓ allows exploiting SIMD

[CIDR05]
minimizing memory stalls

prefetching
  light
temporal stream
software-guided

being cache conscious
  code optimizations
  alternative data structures/layout
  vectorized execution

exploiting common instructions
  batching
  computation spreading
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

[ASPLOS06, MICRO12]

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

http://tinyurl.com/tutorial2014feedback
modern parallelism

- instruction & data parallelism
- multithreading
- horizontal parallelism
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

[PVLDB10b]
many critical sections even for simplest transaction
critical section types

unbounded
locking, latching

fixed
transaction manager

cooperative
logging

unbounded $\Rightarrow$ 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

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

agent thread execution

significantly reduces lock contention
lightweight intent locks

• hottest locks in the system are intent locks

• few intent locks -> high contention

• lightweight intent locks:
  – counters in data pages
  – updated atomically
  – lower overhead than SLI
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>

Completing

Local Lock Table

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

Input
thread-to-transaction - access pattern

[PVLDB10b]
thread-to-data – access pattern

Predictable data accesses

[PVLDB10b]
modern shared-nothing systems

- physical data partitioning
- single threaded execution: no locking or latching
- main-memory optimized: no buffer pool
- support persistence on disk
- durability through replication or logical logging

- main challenge: concurrency with multi-site and long running transactions

[VLDB07b, ICDE11, SIGMOD12]
modern shared-nothing systems

• H-Store/VoltDB
  – extreme fine-grained shared-nothing
  – speculative optimistic concurrency control

• HyPer
  – OLAP support through VM snapshots
  – strict timestamp ordering
  – tentative execution for long running transactions
  – implicit locking with hardware transactional memory

• Calvin
  – deterministic execution model with conflict detection
  – very lightweight locking
multiversion concurrency control

- scalable serializable snapshot isolation
  - latch-free validation phase using atomic ops

- distributed snapshot isolation in SAP HANA
  - snapshot tokens, local-only transactions and write buffering

- Hekaton
  - OCC with parallel validation and commit dependency tracking

- Silo
  - OCC with decentralized validation scheme

[ICDE14a, ICDE13b, SIGMOD13a, PVLDB12c, ICDE14a, ICDE13b]
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)

<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

heap

[PVLDB11b]
PALM: latch-free B-tree

- bulk synchronous parallel processing model
- point-to-point synchronization
- software-prefetching and SIMD
BW-tree

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

[ICDE13c, PVLDB13a]

figure courtesy of Justin Levandoski
scaling up OLTP

unscaleable components
- locking
- latching
- logging

synchronization
- tradeoffs
- best practices

non-uniform communication
- hardware Islands
WAL: gatekeeper of the DBMS

• write ahead logging is a performance enabler

• xct update:

• xct commit:

logging is completely serial – by design
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

---

References:

- [PVLDB10a]
- [SIGMOD13b]
scaling up OLTP

unscalable components
  locking
  latching
  logging

synchronization
  tradeoffs
  best practices

non-uniform communication
  hardware Islands
other unbounded communication

synchronization required for one index probe

• critical sections protect log buffer, stats, lock and latch internal state, thread coordination...

diverse use cases – how to select the best primitive?
lock-based approaches

blocking OS mutex
  ✔ simple to use
  ✗ overhead, unscalable

test and set spinlock (TAS)
  ✔ efficient
  ✗ unscalable

queue-based spinlock ("MCS")
  ✔ scalable
  ✗ memory management

reader-writer lock
  ✔ concurrent readers
  ✗ overhead
lock-free approaches

atomic updates

✓ efficient

× limited applicability

lock-free algorithms

✓ scalable

× special-purpose algos

optimistic concurrency control (OCC)

✓ low read overhead

× writes cause livelock

hardware transactional memory

✓ efficient, scalable

× not widely available
synchronization “cheat sheet”

- OS blocking mutex: only for scheduling
- reader-writer lock: dominated by OCC/MCS
- lock-free: sometimes (but be very, very careful)
scaling up OLTP

unscalable components
  locking
  latching
  logging

synchronization
  tradeoffs
  best practices

non-uniform communication
  hardware Islands
multisocket multicore

<10 cycles

50 cycles

500 cycles

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

[PVLDB12d]
A TraPos: Adaptive Transaction Processing

[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

http://tinyurl.com/tutorial2014feedback
Scaling up OLAP

parallelizing a single aggregation

sharing across queries

OLAP is concerned also with resources saturation

[DaMoN14, 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)
6. I/O

numerous points to consider for NUMA-awareness
scaling up OLAP

sharing
  common sub-plans
  shared operators

scheduling
  task scheduling
  NUMA-aware task scheduling

NUMA-awareness
  application-agnostic
  database operators
sharing is caring...

in the era of big data

...for resources
sharing techniques

<table>
<thead>
<tr>
<th>query-centric</th>
<th>reactive sharing</th>
<th>proactive sharing</th>
</tr>
</thead>
<tbody>
<tr>
<td>• caching</td>
<td>• query-centric</td>
<td>• global query plan</td>
</tr>
<tr>
<td>• materialized views</td>
<td>• shares common sub-plans</td>
<td>with shared operators</td>
</tr>
<tr>
<td>• multi-query optimization</td>
<td>• shared scans</td>
<td>• shared scans</td>
</tr>
<tr>
<td>• buffer pool management</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

how and when should we use each technique?
reactive sharing: how to react?

[VLDB07a, PVLDB13b]

query-centric

Q1  Q2

Σ    Σ

FIFO buffer

:: common sub-plans

by pulling shared intermediate results

Push

forward results

Pull

move independently

Serialization point
proactive sharing

\[
Q_1: \text{SELECT * FROM A, B WHERE } A.c_1 = B.c_1 \text{ AND } \sigma(A) \text{ AND } \sigma(B)
\]

\[
Q_2: \text{SELECT * FROM A, B WHERE } A.c_1 = B.c_1 \text{ AND } \sigma'(A) \text{ AND } \sigma'(B)
\]

shared operators can support high throughput
proactive + reactive sharing

SELECT * FROM A, B
WHERE A.c_1 = B.c_1
AND \( \sigma(A) \) AND \( \sigma(B) \)

QL_1

\[ Q_1 \]

SELECT * FROM A, B
WHERE A.c_1 = B.c_1
AND \( \sigma(A) \) AND \( \sigma(B) \)

QL_2

\[ Q_2 \]

\( \Join \)

\( \Join \)

bits are always the same

+ bitwise AND

\[ [PVLDB13b] \]
proactive + reactive sharing

```sql
Q_1
SELECT * FROM A, B
WHERE A.c_1 = B.c_1
AND \sigma(A) AND \sigma(B)

Q_2
SELECT * FROM A, B
WHERE A.c_1 = B.c_1
AND \sigma(A) AND \sigma(B)
```

reactive sharing avoids redundant computations

reactive sharing can improve proactive sharing

[PVLDB13b]
sharing in practice

<table>
<thead>
<tr>
<th>sharing type</th>
<th>QPipe [SIGMOD05]</th>
<th>CJOIN [VLDB09a]</th>
<th>DataPath [SIGMOD10a]</th>
<th>SharedDB [PVLDB12b, PVLDB14b]</th>
</tr>
</thead>
<tbody>
<tr>
<td>execution</td>
<td>reactive</td>
<td>proactive (global query plan)</td>
<td>dynamic</td>
<td>dynamic</td>
</tr>
<tr>
<td>schema</td>
<td>general</td>
<td>star</td>
<td>general</td>
<td>general (pre-comp.)</td>
</tr>
<tr>
<td>I/O</td>
<td>circular scans</td>
<td>circular scans</td>
<td>linear scan of a disk array</td>
<td>main-memory circ. scans</td>
</tr>
</tbody>
</table>
share responsibly

[ PVldb13b, Sigmod14b ]

demo on wed 15:00 & thu 10:30

<table>
<thead>
<tr>
<th>when to share</th>
<th>how to share</th>
</tr>
</thead>
<tbody>
<tr>
<td>low concurrency</td>
<td>query-centric operators</td>
</tr>
<tr>
<td></td>
<td>+ reactive sharing</td>
</tr>
<tr>
<td>high concurrency</td>
<td>proactive sharing</td>
</tr>
<tr>
<td></td>
<td>+ reactive sharing</td>
</tr>
</tbody>
</table>
scaling up OLAP

sharing
common sub-plans
shared operators

NUMA-awareness
application-agnostic
database operators

scheduling
task scheduling
NUMA-aware task scheduling
application-agnostic NUMA-awareness

[HPCA13, USENIX11, ASPLOS13]

• 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
impact of NUMA

- data partitions accessed by different clients
  - co-locate threads and data they access

up to 75% improvement
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
radix hash join

partitions (by key) are small enough to fit into cache

charge-efficient but not NUMA-aware

figure courtesy of Kim et al.
massively parallel sort-merge join

• NUMA-awareness rules:
  – no remote random writes
  – sequential remote reads
  – no synchronization

remote random accesses > remote scans
sort-merge join forever?

- Massively parallel sort merge join (mPSM) suffers from bandwidth saturation for general schemas.
- Multi-way merging with task scheduling to balance CPU and memory.
- Radix hash join still superior.

A long-standing battle.
scaling up OLAP

sharing
  common sub-plans
  shared operators

NUMA-awareness
  application-agnostic
  database operators

scheduling
  task scheduling
  NUMA-aware task scheduling
scheduling work

• OS scheduler

[ADMS13]
scheduling work

- OS scheduler

  Overutilization

- task scheduler

  task queues

  socket 1

  socket 2

  a solution for DBMS to efficiently utilize resources
opportunities and challenges

[ADMS13, DSAA14, ICDE13a, PCS13]

<table>
<thead>
<tr>
<th>opportunities</th>
<th>advantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>decouple from OS</td>
<td>full control and predictability</td>
</tr>
<tr>
<td>task granularity</td>
<td>balance CPU and memory parallelism</td>
</tr>
<tr>
<td>task prioritization</td>
<td>workload management</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>challenges</th>
<th>solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>unbalanced task queues</td>
<td>stealing</td>
</tr>
<tr>
<td>NUMA-awareness</td>
<td>affinities</td>
</tr>
<tr>
<td></td>
<td>restricted stealing</td>
</tr>
<tr>
<td>blocking tasks</td>
<td>co-operative scheduling</td>
</tr>
<tr>
<td></td>
<td>flexible #threads</td>
</tr>
<tr>
<td>task granularity</td>
<td>depending on saturation</td>
</tr>
</tbody>
</table>
task scheduling for OLAP

figure courtesy of Leis et al.

[SIGMOD14a]
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 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

http://tinyurl.com/tutorial2014feedback
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 [MICRO13b]
• Database processing unit [ASPLOS14]
• Programmable accelerators [VLDB09d]
• Bionic databases [CIDR13a]
• Reconfigurable datacenters [ISCA14]
• Commercial: RAPID [ORACLE]

toward specialized hardware
open questions – How to ...

• fit NVRAM to memory hierarchy? [PVLDB14e, PVLDB14f]
• exploit HTM? [ICDE14b, Eurosys14]
• adapt the whole software stack (OS + applications) to hardware specialization?
• take advantage of compilers? [PVLDB14c, PVLDB14d]
• design concurrency-control for many-cores? [MITCMU14]
• ...

http://tinyurl.com/tutorial2014feedback
references


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


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


[Eurosyst14] Z. Wang, H. Qian, J. Li, and H. Chen: Using Restricted Transactional Memory to Build a Scalable In-Memory Database.

references

[ICDE13a] J. Dees and P. Sanders: Efficient many-core query execution in main memory column-stores
[ICDE14c] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker: Rethinking Main Memory OLTP Recovery.
references


references


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


references


references


references


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


references

