Data Processing on Modern Hardware

Jana Giceva

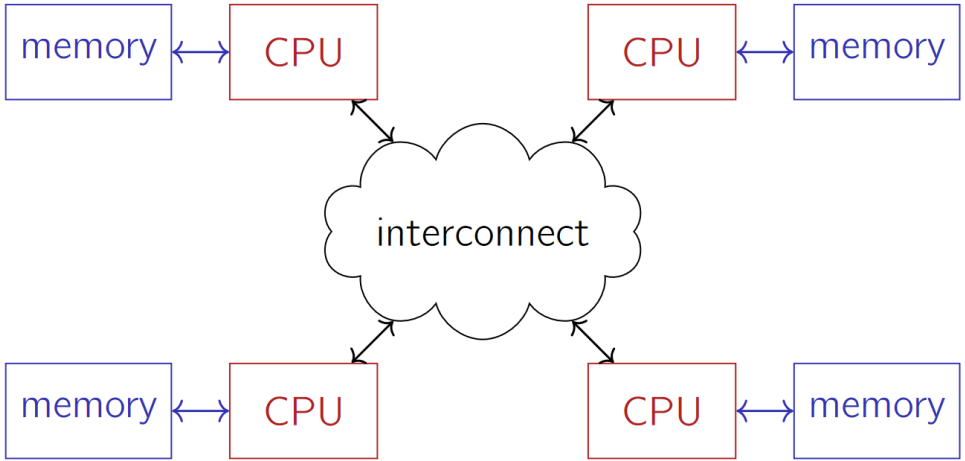
Lecture 8: Multicore CPUs

NUMA, interference and isolation

Non-uniform memory access (NUMA)

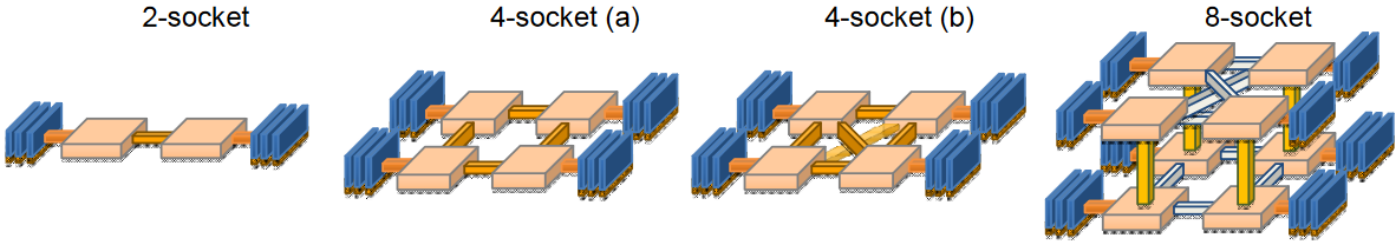
Distributed Shared Memory

- Single processor scalability (with shared memory) has limitations
- Idea: distribute memory



NUMA hardware

- Almost all mid-range and enterprise servers today are multi-socket



- Each server typically contains between 2-8 sockets

- Each socket contains:
  - between 4 and 24 cores (up to 64 cores on AMD EPYC)
  - a few memory DIMM modules attached through memory channels

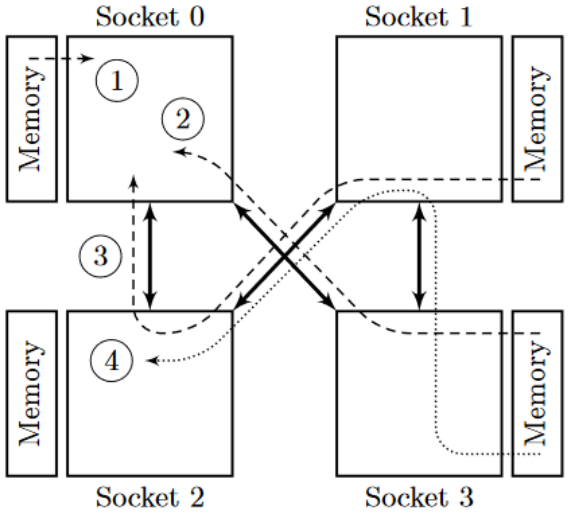
- An interconnect network among the sockets allows each core to access non-local memory

src. Li et al. “NUMA-aware algorithsm: the case of data shuffling” *CIDR 2013*

Effects of NUMA Hardware

- **Example multi-socket server:**
  - four 8-core Nehalem-EX processors, fully connected with 4 bi-directional 3.2 GHz QuickPath Interconnect (QPI)

- Measure performance for reading local vs. remote memory
  - **Flow 1: read locally** (from socket 0)
    - max. aggregate bandwidth (12 threads) is 24.7 GB/s
    - latency is 340 CPU cycles (~ 150ns)
  - **Flow 2: read remote** (over **1 QPI link**, from socket 3)
    - max. aggregate bandwidth is 10.9 GB/s
    - latency is 420 CPU cycles (~185ns)
  - **Flow 3: read remote** (over **2 QPI links**, from socket 1)
    - max. aggregate bandwidth is 10.9 GB/s
    - latency is 520 CPU cycles (~230ns)
  - **Flow 4: read remote** (over **2 QPI links**) **with cross traffic**
    - max. aggregate bandwidth is 5.3 GB/s
    - latency is 530 CPU cycles (~235ns)



src. Li et al. “NUMA-aware algorithsm: the case of data shuffling” *CIDR 2013*

What does that mean for data processing?

- Designing algorithms and data structures
  - Need to differentiate between local and remote memory
  - Local memory is faster and has higher bandwidth
- Concurrency
  - Synchronization within a socket / NUMA node is significantly faster
  - Concurrent data structures needs to scale across NUMA nodes
- NUMA effects in systems and databases

System‘s support for NUMA

- Modern **operating systems** are aware of NUMA architectures.
  - Linux partitions memory into NUMA zones, one for each socket.
  - For each NUMA zone, the kernel maintains separate management data structures.

- By **default** the Linux kernel **allocates memory** on the local NUMA node
  - The socket of the core on which the current thread is scheduled on.

- **Unless explicitly bound** to a specified socket (NUMA zone) through the **mbind() system call**

Memory allocation

- **Watch out for default OS**
  - **First touch** allocation **policy** (static and in place **until kernel version 2.6**)
  - **Today** there are two options:
    - **Transparent NUMA awareness**:
      - Allocate locally
      - Migrate thread or data to achieve good NUMA performance and balancing
    - **Explicit memory allocation policy**
      - Allocate memory (and do not migrate) based on the selected policy

- **NUMA memory policy**
  - System default policy
    - local (general), interleaved (during boot-up)
  - Task/Process policy – controls all page allocations made by or on behalf of the task
  - VMA policy – to a range of a task's virtual address space

Memory allocation

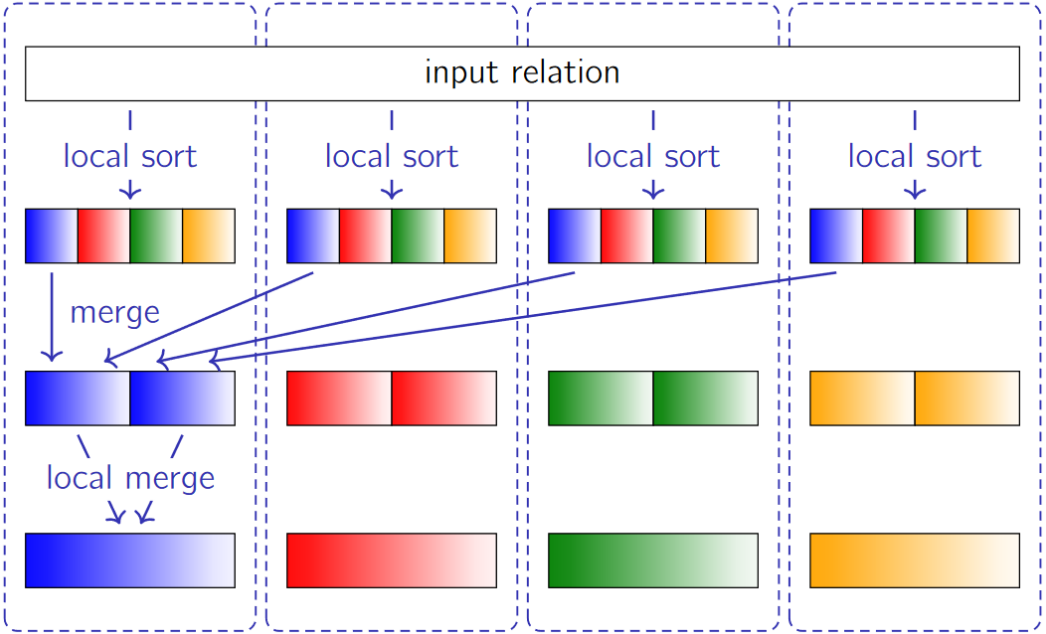
- **Allocation modes**
  - **Local** – memory from the local NUMA node
  - **Bind** – memory from the set of nodes specified by the policy
  - **Preferred** – memory from the set of nodes specified by the policy, if available
  - **Interleaved** – memory interleaved across all the NUMA nodes in the set provided by the policy

- Invoked from the process / thread itself or via the `numactl` library

- Memory policy APIs
  - `long set_mempolicy(int mode, const unsigned long *nmask, unsigned long maxnode)`
  - `long get_mempolicy(int *mode, const unsigned long *nmask, unsigned long maxnode, void *addr, int flags);`
  - `long mbind(void *start, unsigned long len, int mode, const unsigned long *nmask, unsigned long maxnode, unsigned flags)`

Where does it matter?

- Example 1: Sorting and NUMA



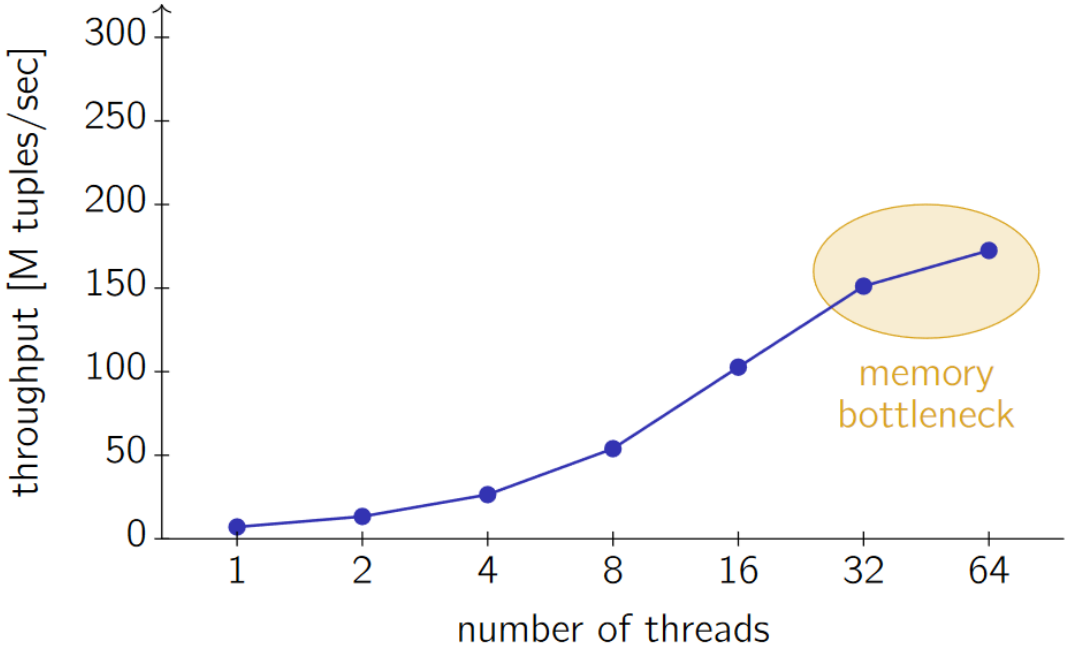
Step 1: Operate on local data as much as possible.

Step 2: Exchange data to gather local partitions on a local place.

Step 3: Merge the results locally.

src. Balkesen et al. “Multi-core, Main-Memory Joins: Sort vs Hash Revisited” *VLDB 2014*

Sorting and NUMA

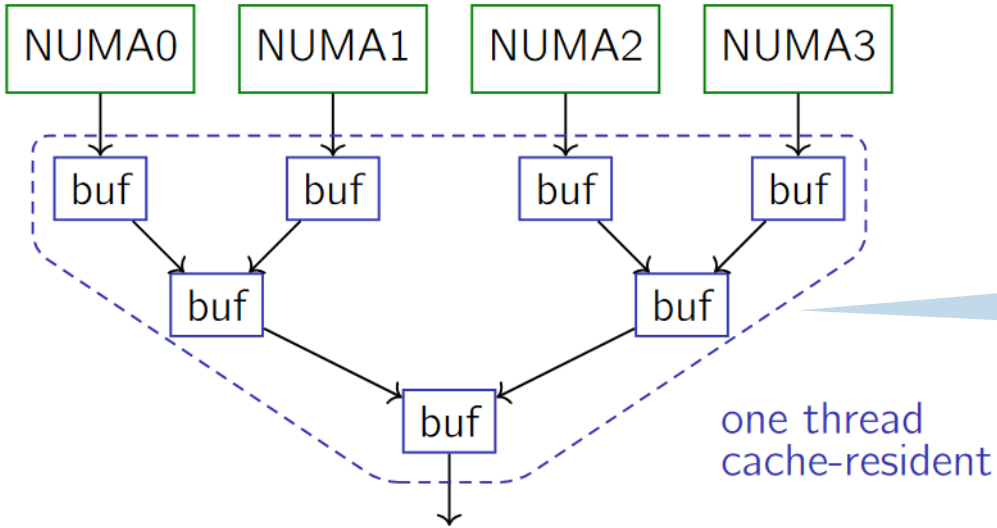


Even though NUMA-aware, the algorithm does not scale well.

Step 2 is very memory-bandwidth intensive and saturates the system‘s resources.

src. Balkesen et al. “Multi-core, Main-Memory Joins: Sort vs Hash Revisited” *VLDB 2014*

Multi-way merging as an alternative

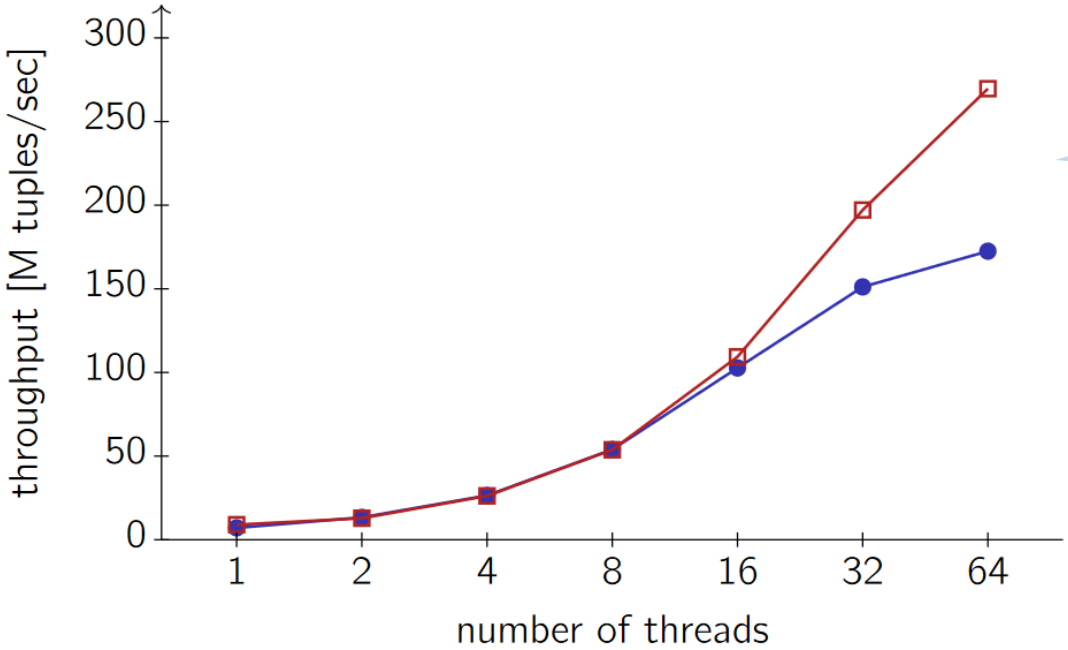


Step 1: Operate on local data as much as possible. (similar to before)

Step 2: Gather data in a cache- and NUMA-conscous way. Recall the multi-way sort when we did SIMD?

src. Balkesen et al. “Multi-core, Main-Memory Joins: Sort vs Hash Revisited” *VLDB 2014*

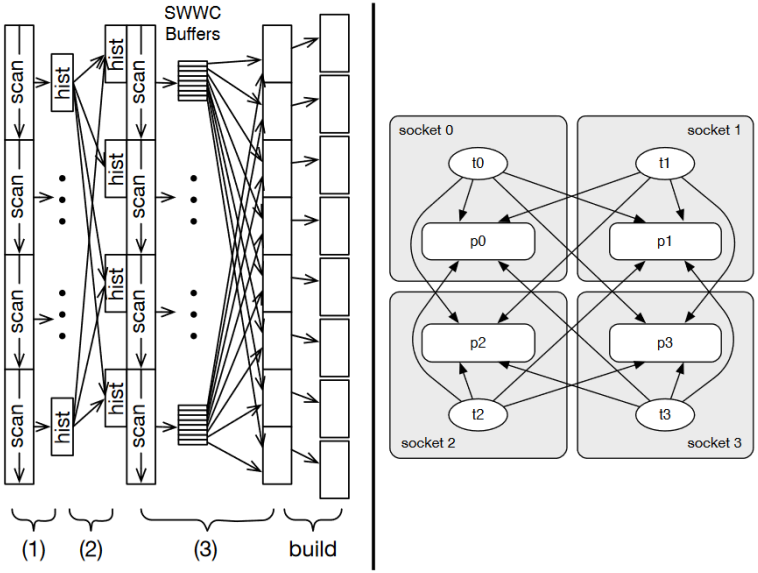
Sorting and NUMA

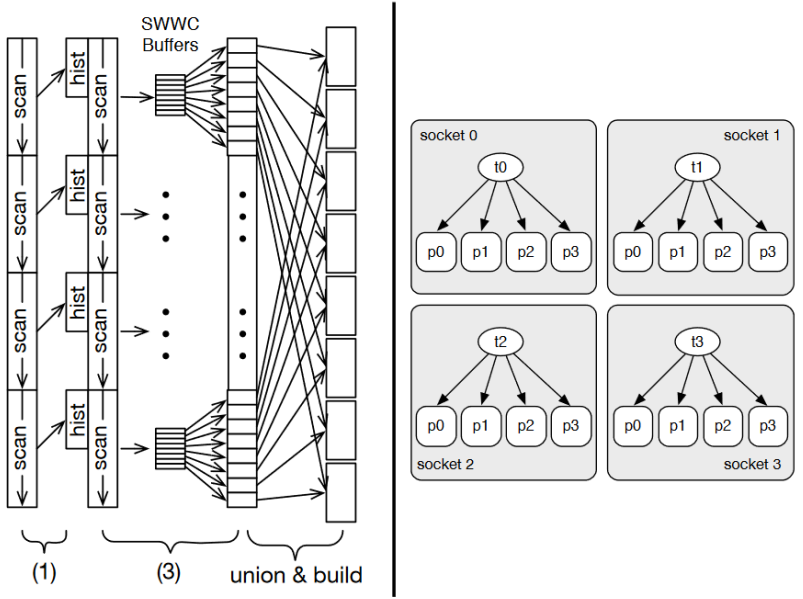


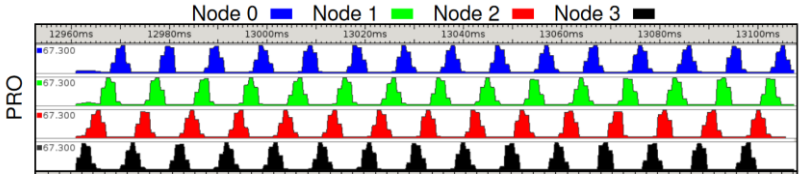
Performance speed-up much better than before because of NUMA-awareness.

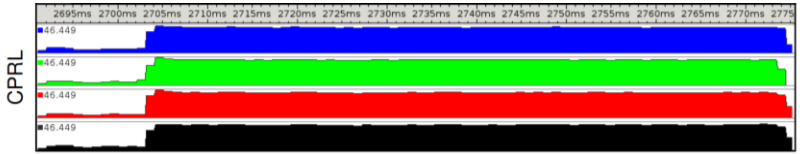
src. Balkesen et al. “Multi-core, Main-Memory Joins: Sort vs Hash Revisited” *VLDB 2014*

Radix-join and NUMA



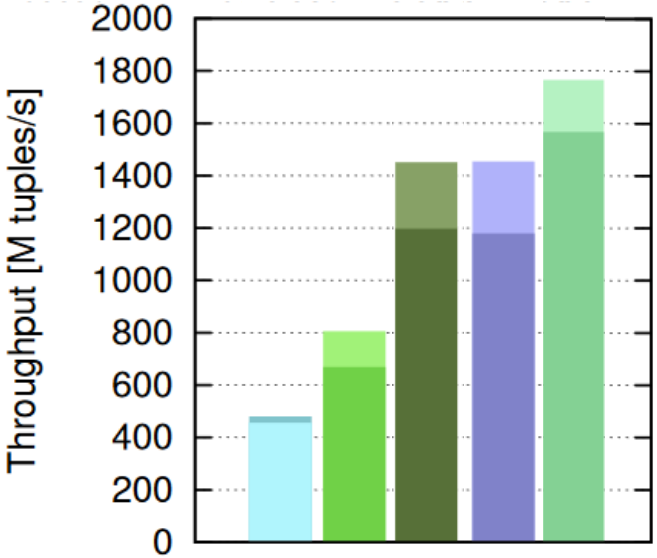






Src: Schul et al. "An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory." *SIGMOD 2016*

Radix-join and NUMA



- radix join (basic)
- no partitioned hash join with prefetching
- radix join (with chain hashing) with optimizations
- radix join (with linear hashing) with optimizations
- radix join (with LH) with optimizations and NUMA awareness

NUMA-awareness brings additional 20-30% improvement.

Src: Schul et al. “An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory.” *SIGMOD 2016*

NUMA-awareness engine-wide

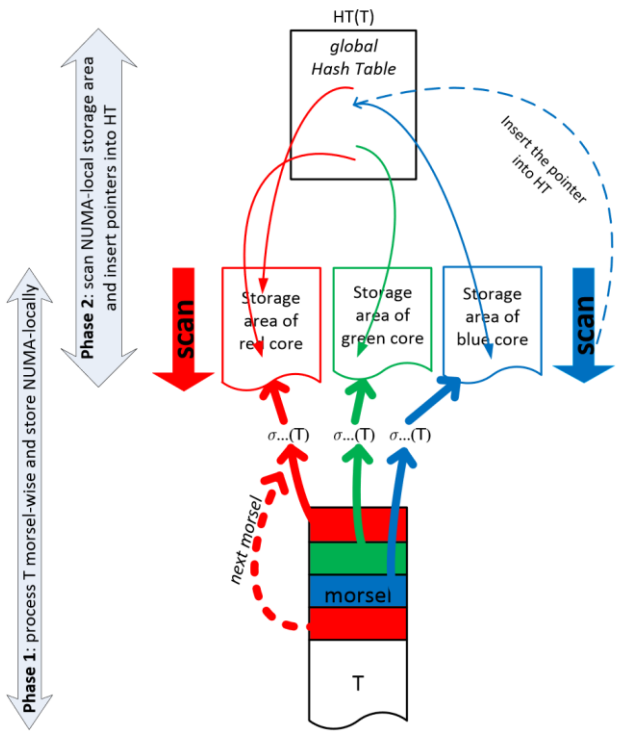
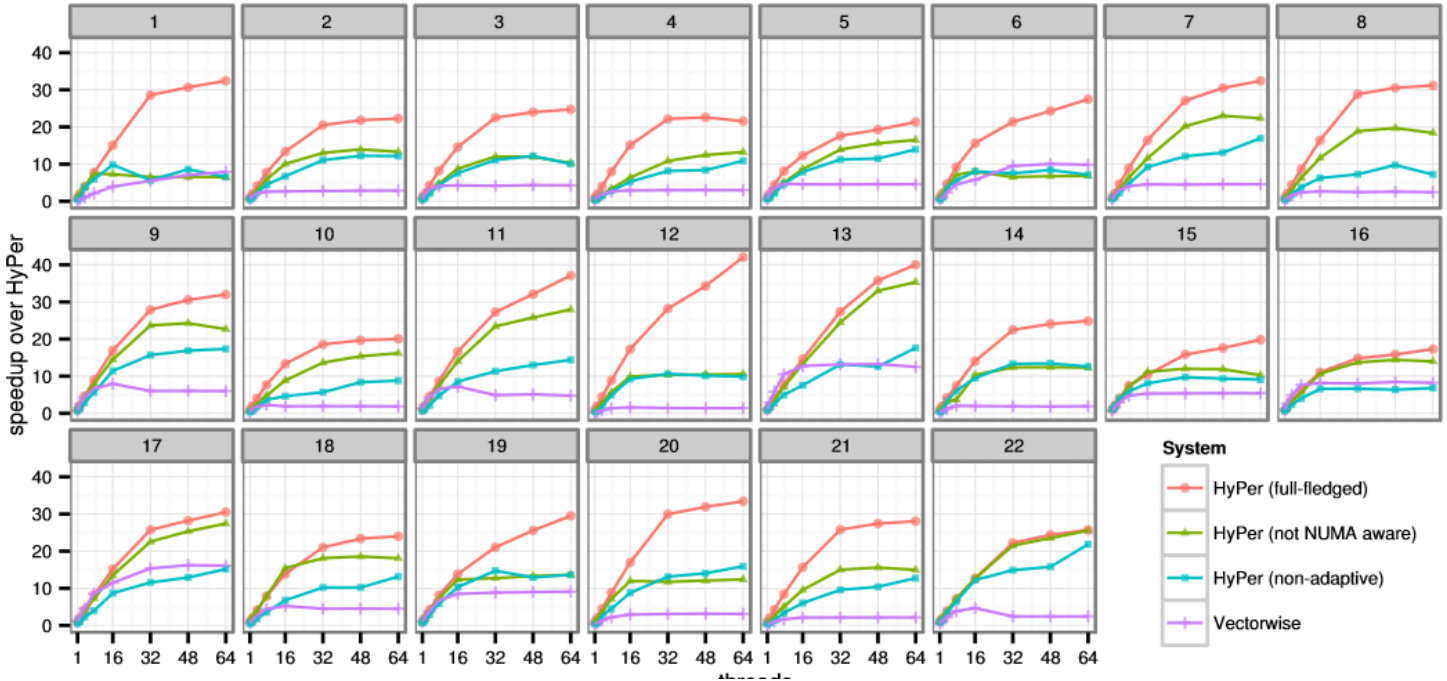


Figure 3: NUMA-aware processing of the build-phase

- Relation T is interleaved “morsel-wise” across the NUMA nodes
- The scheduler assigns a morsel located on the same NUMA node where the thread is executed.
- In the first phase the filtered tuples are inserted into NUMA-local storage areas, i.e., for each core there is a separate storage area in order to avoid synchronization.
- The global HT is probed by threads located on various sockets of a NUMA system.
  - To avoid contention, it is interleaved across all sockets.

src. Leis et al. “Morse-Driven Parallelism: A NUMA-aware query evaluation framework for the many-core age” *SIGMOD 2014*

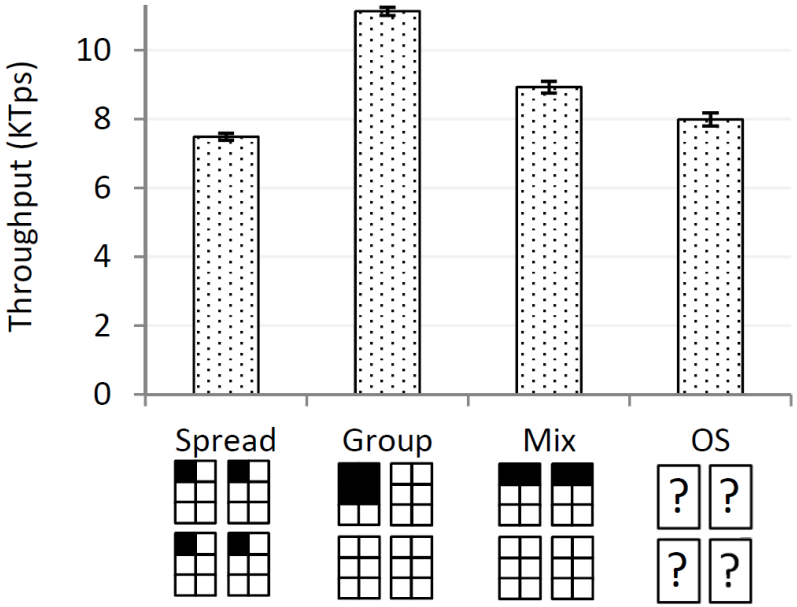
Engine-wide NUMA awareness



src. Leis et al. “Morse-Driven Parallelism: A NUMA-aware query evaluation framework for the many-core age” *SIGMOD 2014*

Impact of NUMA on DB synchronization

- Performance implications for synchronization and concurrency



- **Goal:** check the impact of NUMA latencies on OLTP transactions and the overall throughput.
- **OLTP workload**: TPC-C payment transaction
- **Machine:** 4 CPUs with 6 cores earch
- **Test:** Run the database with 4 worker threads, either using the default OS scheduling or pinning them to different cores.

- **Insights:**
  - DB threads collocated on the same NUMA node exhibit much better performance than alternatives.
  - Communication over the interconnect is expensive.
  - OS-scheduling can be unpredictable.

src. Porobic et al. “Analyzing the impact of system architecture on the scalability of OLTP engines for high-contention workloads” *VLDB 2017*

Locks and NUMA

- **Synchronization within the processor is cheaper than to synchronization over the interconnect**
  - due to **latency** concerns, but also for **increased memory traffic.**

- Two main approaches to make locks NUMA-aware locks (and concurrent data structures):
  - **Cohort locks** (hierarchical locks) [1]
  - **Combining** + remote core execution (select a leader, etc.) [2]

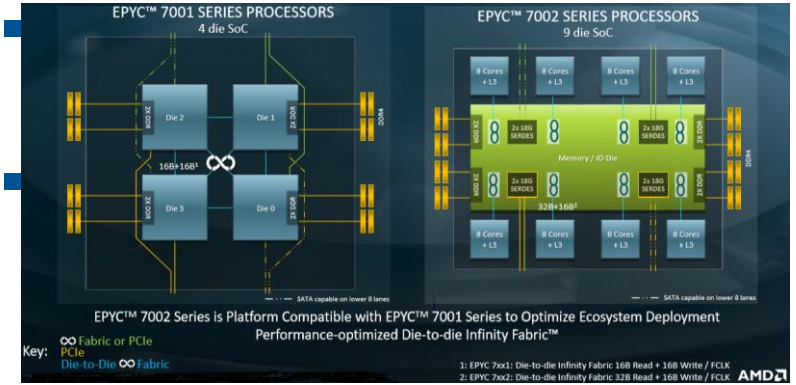
- Recent **black box approach** allows any **linear data structure** to be made NUMA-aware [3]

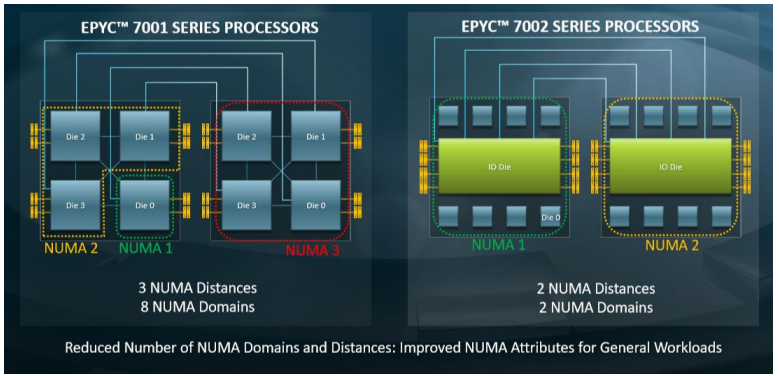
- Parking lock (e.g., optimized futex from last week) can be made a scalable and NUMA-aware blocking synchronization primitive: CST [4]

[1] Chabbi et al. *High Performance Locks for Multi-Level NUMA Systems.* PPoPP 2015
[2] Lozi et al. *Fast and Portable Locking for Multicore Architectures.* ACM Trans. Computing Systems 2016
[3] Calciu et al. *Black-box Concurrent Data Structures for NUMA Architectures.* ASPLOS 2017
[4] Kashyap et al. *Scalable NUMA-aware Blocking Synchronization Primitives.* USENIX ATC 2017
https://taesoo.kim/pubs/2017/kashyap:cst-slides.pdf

Latest generation hardware

- Intel scalable with UltraPath interconnect
- Succeeds Intel QuickPath Interconnect (QPI)
- Can connect each processor with up to 3 UPI links for connecting to other Intel Xeon processor.
- UPI uses a directory-based home snoop coherency protocol, operational speed of up to 10.4 GT/s
- Between 2- and 8-socket configurations





Performance isolation

Execution on Multiple cores

- Concurrency does not only affect correctness and hence the need for efficient synchronization.

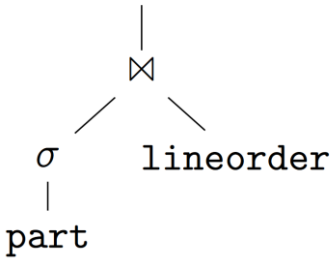
- The impact of resource sharing must not be overlooked:
  - *e.g.*, OLAP + OLAP *or* OLAP + OLTP

- Challenges of multi-programming (concurrency) due to interference:
  1. Restructuring the algorithm (less sensitive to noisy environment)
  2. Careful co-scheduling (victim and noisy neighbors)
  3. Isolation through pinning and running on a separate NUMA node
  4. Isolation with cache partitioning

Execution on Multiple cores

- Recall the example that we presented in the introductory lecture
- **Task: run parallel instances of the query**

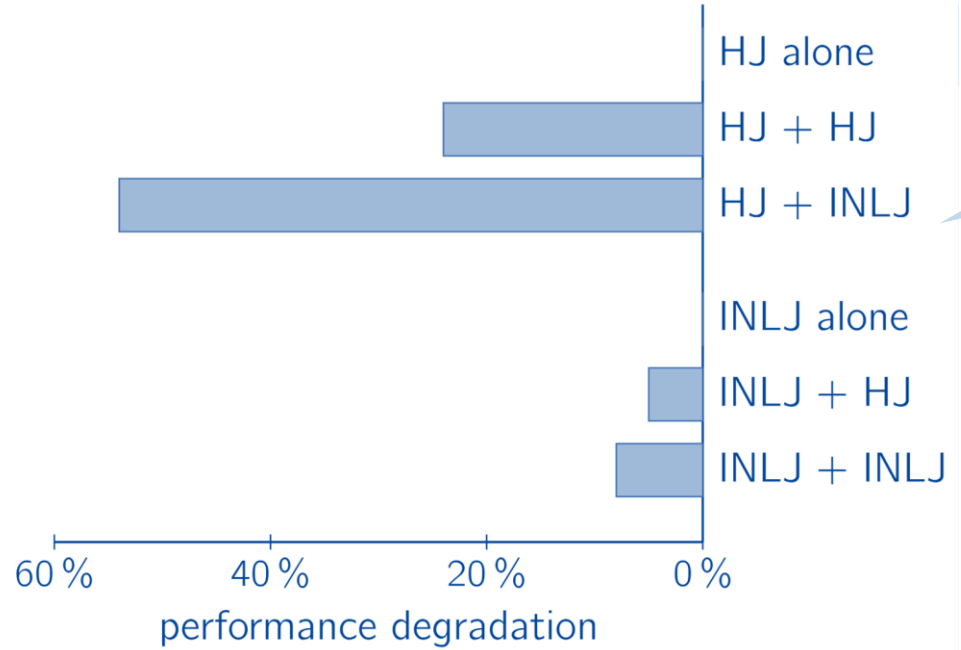
```
SELECT  SUM(lo_revenue)
FROM    part, lineorder
WHERE   p_partkey = lo_partkey
AND     p_category <= 5
```



- To implement the join use either
  - a **hash join** or
  - an **index nested loops join**
- Co-execute the independent instances on different CPUs and compare the performance to baseline when they are run in isolation.

Execution on independent CPU cores

- Concurrent queries may seriously affect each other's performance

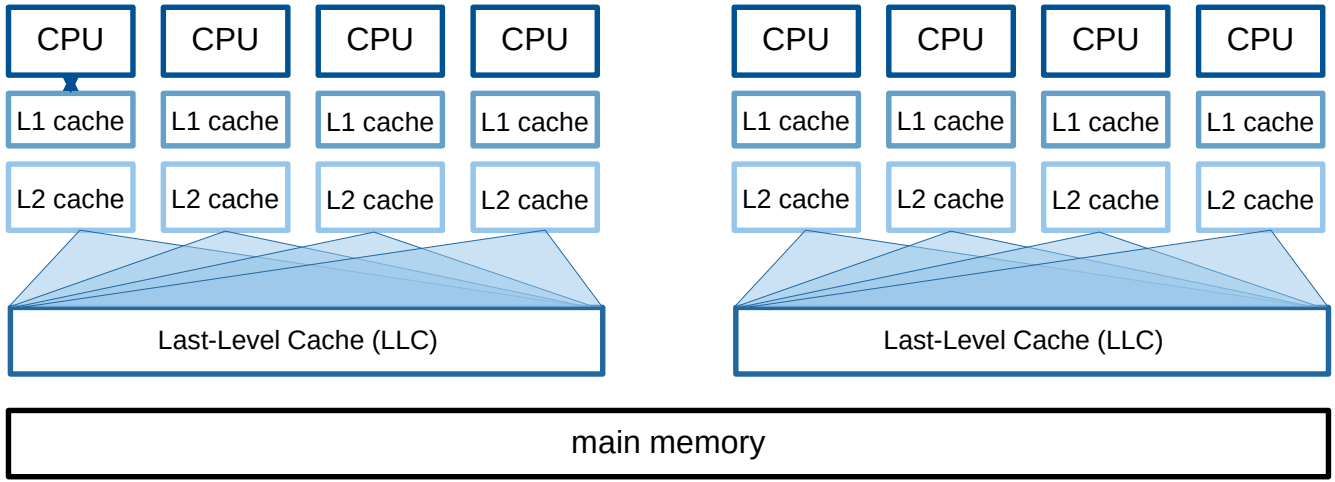


Some algorithms are more sensitive to noisy environments (victims) and their performance can be significantly affected if collocated with a bad neighbor.

Src: "Lee, Ding, Chen, Lu, and Zhang. MCC-DB: Minimizing Cache Conflicts in Multi-core Processors for Databases" *VLDB 2009*

Shared caches

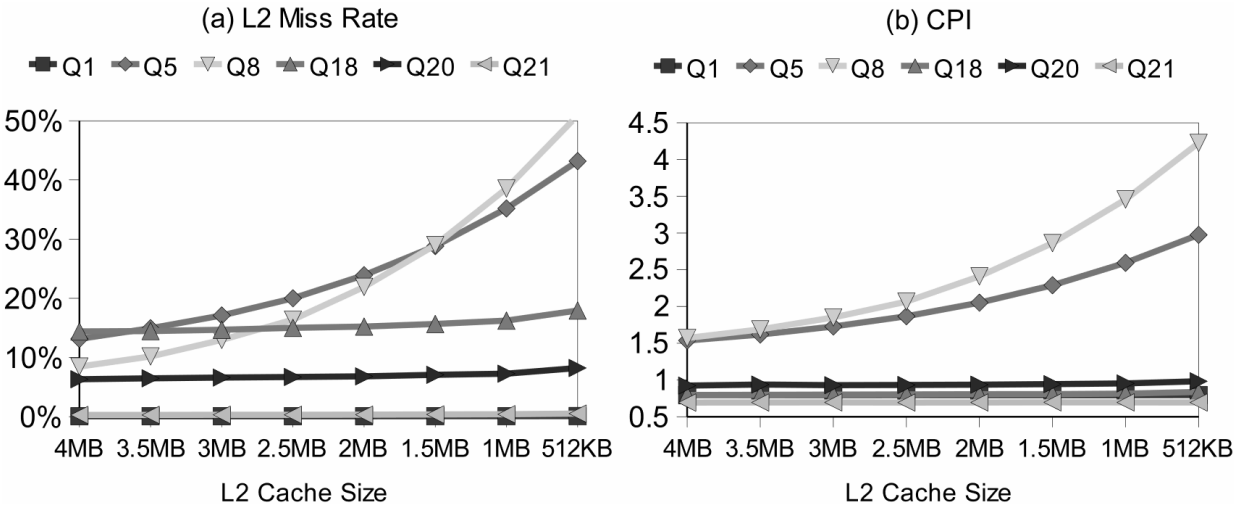
- More cores share the last-level cache (LLC)



- The problem we saw in the previous slide is **cache pollution**
  - How can we avoid it?

Cache sensitivity

- Dependence on cache sizes for some TPC-H queries



- Some queries are more sensitive to cache sizes than others:
- For example:
  - **Cache sensitive**: hash joins
  - **Cache insensitive:** index nested loop joins, hash joins with very small or very large hash tables

Src: “Lee, Ding, Chen, Lu, and Zhang. MCC-DB: Minimizing Cache Conflicts in Multi-core Processors for Databases” *VLDB 2009*

Locality strength

This behavior is related to the **locality strength** of execution plans:

- **Strong locality**
  - Small data structure; reused very frequently
    - *e.g.,* a small hash table
- **Moderate locality**
  - Frequently reused data structure; data structure ~ cache size
    - *e.g.,* moderate-sized hash table
- **Weak locality**
  - Data not reused frequently or data structure >> cache size
    - *e.g.,* large hash table, index lookups

Execution plan characteristics

- Locality effects how caches are used:

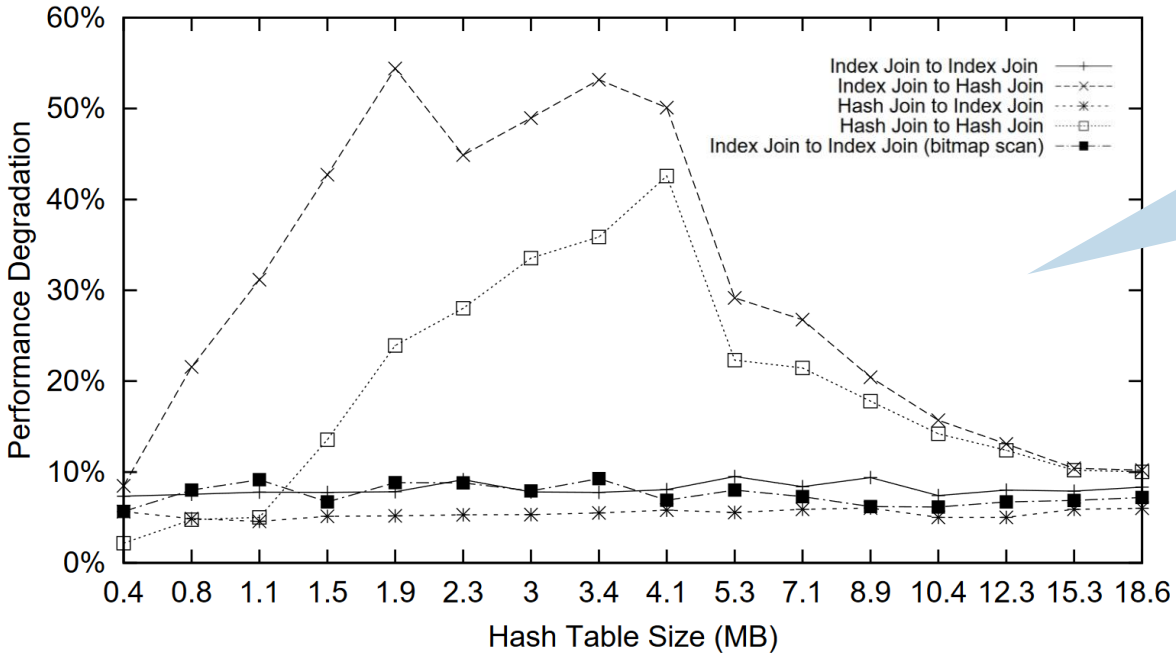
Cache pollution	strong	moderate	weak
amount of cache used	small	large	large
amount of cache needed	small	large	small

- Plans with **weak locality** have most severe impact on co-running queries.

- **Impact of co-runner on query:**

query / co-runner	strong	moderate	weak
strong	low	moderate	high
moderate	moderate	high	high
weak	low	low	low

Experiments: locality strength



Hash join is the only algorithm sensitive to sharing the caches.

The index join is not affected, regardless of the co-runner query.

Lee *et al*. MCC-DB: Minimizing Cache Conflicts in Multi-core Processors for Databases. *VLDB 2009*

Locality-aware scheduling

An **optimizer** could use knowledge about localities to **schedule** queries:

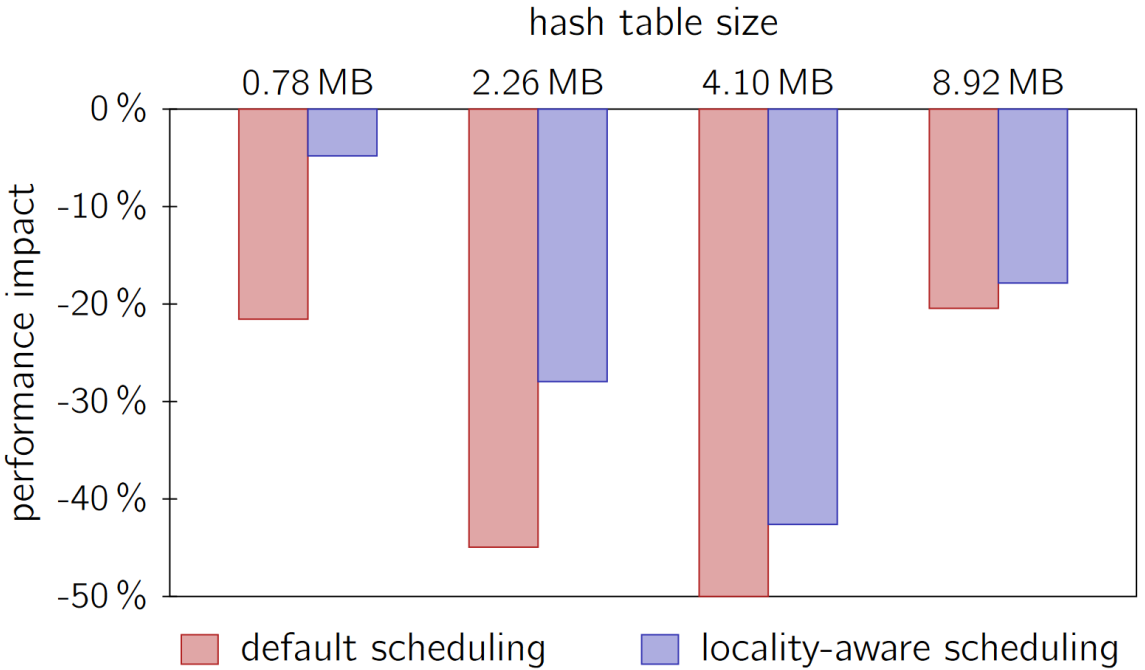
- **Estimate** locality during query analysis
  - Index nested loop join → weak locality
  - Hash join:
    - Hash table ≪ cache size → strong locality
    - Hash table ≈ cache size → moderate locality
    - Hash table ≫ cache size → weak locality

- **Co-schedule** queries to minimize (the impact of) cache pollution

- **Which queries should be co-scheduled, which ones not?**
  - Only run weak-locality queries alongside other weak-locality queries.
    → They cause high pollution, but are not affected by pollution.
  - Try to co-schedule queries with small hash tables.

Locality-aware scheduling

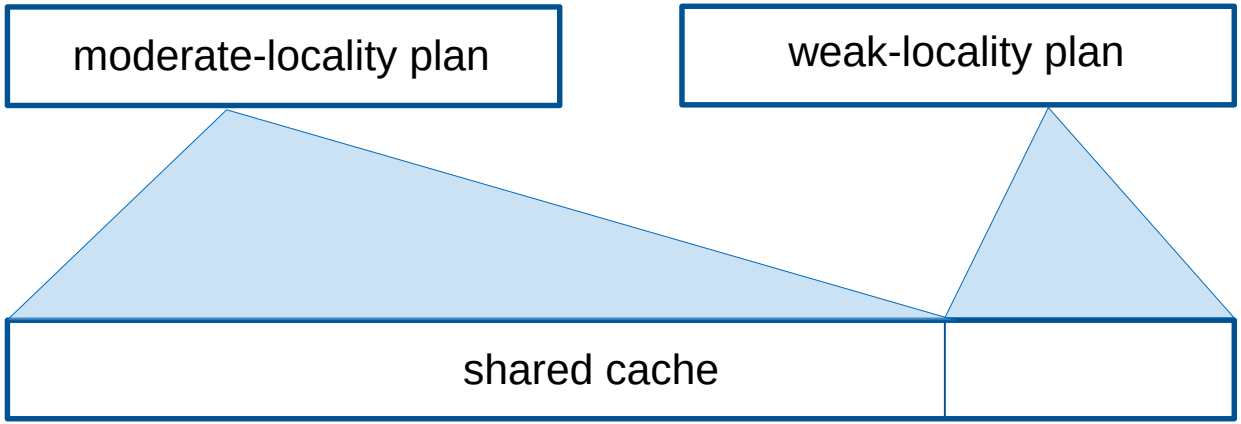
- PostgreSQL
- 4 queries (different `p_categorys`); for each query:
  - 2 x hash join,
  - 2 x INLJ;
- Performance impact reported for the hash joins



Src: “Lee, Ding, Chen, Lu, and Zhang. MCC-DB: Minimizing Cache Conflicts in Multi-core Processors for Databases” *VLDB 2009*

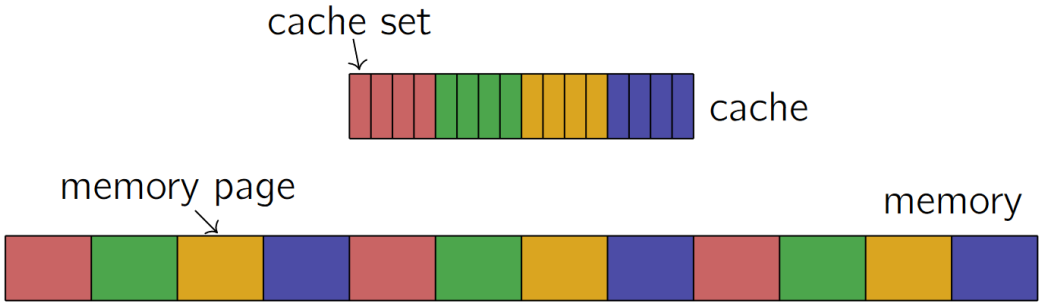
Cache pollution

- Weak-locality plans cause cache pollution, because they **use** much cache space even though they do not strictly need it or benefit from it.
- By partitioning the cache we could reduce the pollution with little impact on the weak locality plan.



Cache partitioning

- **In the past**, people had to rely on page coloring to achieve cache partitioning from the software side
  - The address <-> cache set relationship inspired the idea of page colors



Each memory page is assigned a **color**.

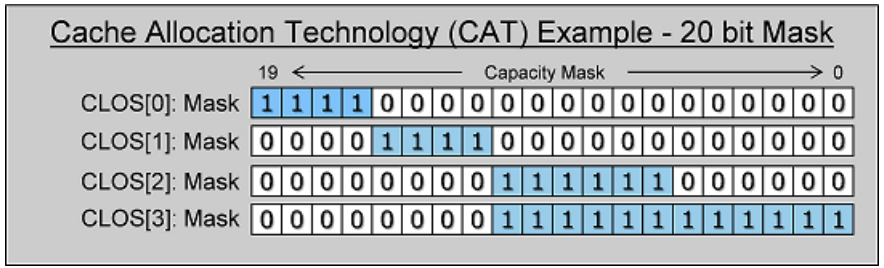
Pages that map to the **same cache sets** get the **same color** with OS support.

- **Today**, Intel provides the **Resource Directory Technology** (RDT)
  - **Cache Monitoring and Allocation Technology** (CMT and CAT)
  - CAT is a software programmable control over the space that can be consumed by a given thread, application, virtual machine (VM), or a container.

Intel Cache Allocation Technology (CAT)

- Class of service (CLOS) or an application priority class
  - resource control tag that allows us to group threads or applications.

- Associate the CLOS with resource capacity bitmasks (CBMs) indicating how much of the cache can be used by a given CLOS.
  - The CBMs indicate the relative amount of cache available, the degree of overlap or isolation.



CLOS[1] has less cache available than CLOS[3], even though it has higher priority

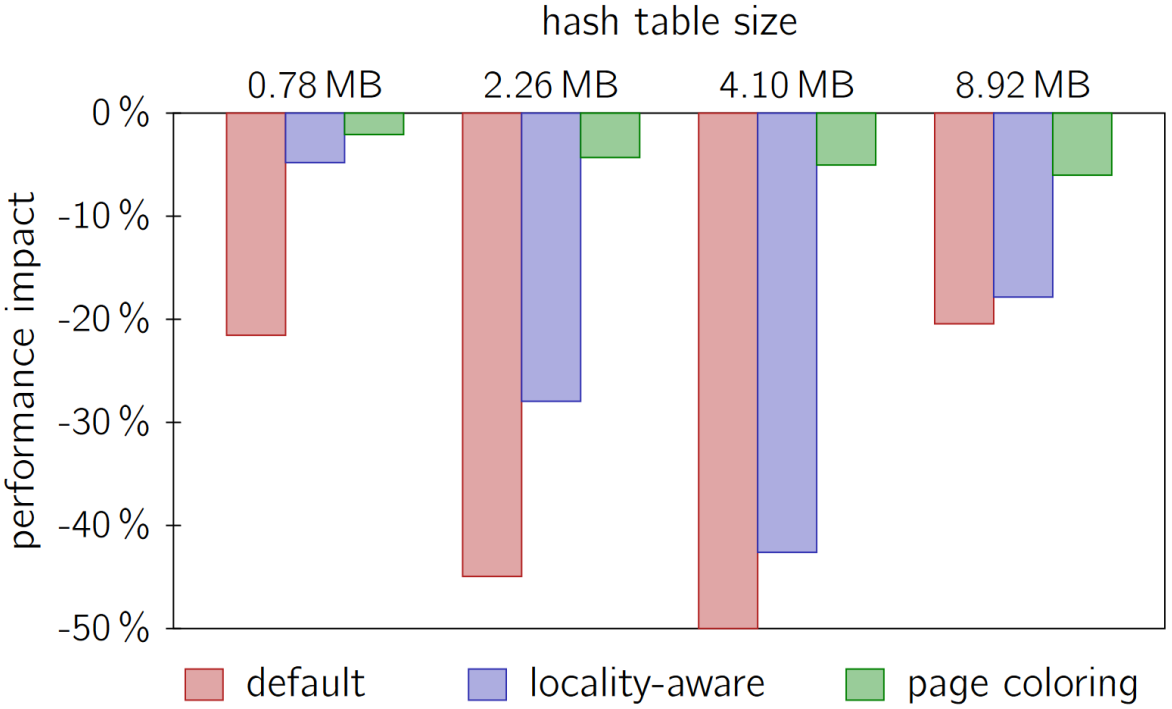
CLOS[2] and CLOS[3] have overlapping bitmasks, can achieve higher throughput than in isolation, but relative priorities will be preserved.

- Can be further refied with code and data- prioritization (CDP) technology

src: Intel

Experiments: MCC-DB with page coloring

- PostgreSQL
- 4 queries (different `p_categorys`); for each query:
  - 2 x hash join,
  - 2 x INLJ;
- Performance impact reported for the hash joins



Src: “Lee, Ding, Chen, Lu, and Zhang. MCC-DB: Minimizing Cache Conflicts in Multi-core Processors for Databases” *VLDB 2009*

But, it is not only the cache is shared

websearch

	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%
LLC (small)	134%	103%	96%	96%	109%	102%	100%	96%	96%	104%	99%	100%	101%	100%	104%	103%	104%	103%	99%
LLC (med)	152%	106%	99%	99%	116%	111%	109%	103%	105%	116%	109%	108%	107%	110%	123%	125%	114%	111%	101%
LLC (big)	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	264%	222%	123%	102%
DRAM	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	270%	228%	122%	103%
HyperThread	81%	109%	106%	106%	104%	113%	106%	114%	113%	105%	114%	117%	118%	119%	122%	136%	>300%	>300%	>300%
CPU power	190%	124%	110%	107%	134%	115%	106%	108%	102%	114%	107%	105%	104%	101%	105%	100%	98%	99%	97%
Network	35%	35%	36%	36%	36%	36%	36%	37%	37%	38%	39%	41%	44%	48%	51%	55%	58%	64%	95%
brain	158%	165%	157%	173%	160%	168%	180%	230%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%

ml_cluster

	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%
LLC (small)	101%	88%	99%	84%	91%	110%	96%	93%	100%	216%	117%	106%	119%	105%	182%	206%	109%	202%	203%
LLC (med)	98%	88%	102%	91%	112%	115%	105%	104%	111%	>300%	282%	212%	237%	220%	220%	212%	215%	205%	201%
LLC (big)	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	276%	250%	223%	214%	206%
DRAM	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	287%	230%	223%	211%
HyperThread	113%	109%	110%	111%	104%	100%	97%	107%	111%	112%	114%	114%	114%	119%	121%	130%	259%	262%	262%
CPU power	112%	101%	97%	89%	91%	86%	89%	90%	89%	92%	91%	90%	89%	89%	90%	92%	94%	97%	106%
Network	57%	56%	58%	60%	58%	58%	58%	58%	59%	59%	59%	59%	59%	63%	63%	67%	76%	89%	113%
brain	151%	149%	174%	189%	193%	202%	209%	217%	225%	239%	>300%	>300%	279%	>300%	>300%	>300%	>300%	>300%	>300%

memkeyval

	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%
LLC (small)	115%	88%	88%	91%	99%	101%	79%	91%	97%	101%	135%	138%	148%	140%	134%	150%	114%	78%	70%
LLC (med)	209%	148%	159%	107%	207%	119%	96%	108%	117%	138%	170%	230%	182%	181%	167%	162%	144%	100%	104%
LLC (big)	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	280%	225%	222%	170%	79%	85%
DRAM	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	252%	234%	199%	103%	100%
HyperThread	26%	31%	32%	32%	32%	32%	33%	35%	39%	43%	48%	51%	56%	62%	81%	119%	116%	153%	>300%
CPU power	192%	277%	237%	294%	>300%	>300%	219%	>300%	292%	224%	>300%	252%	227%	193%	163%	167%	122%	82%	123%
Network	27%	28%	28%	29%	29%	27%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%
brain	197%	232%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%

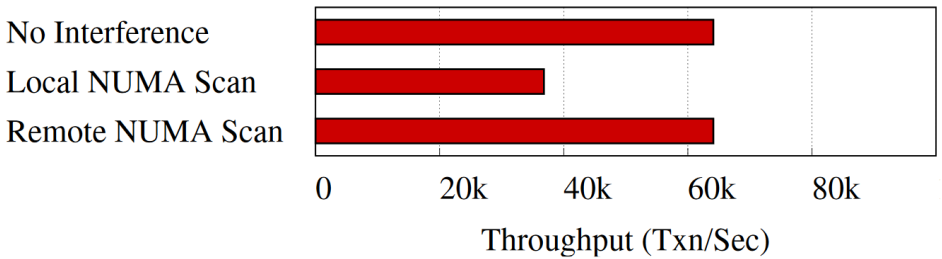
Src: Lo et al. “Heracles: Improving Resource Efficiency at Scale” *ISCA 2015*

Impact of interference on transactions

- Goal: Measure the performance impact that a local NUMA scan can have on an OLTP workload.

- No explicit sharing of resources
  - the scan runs on a separate dataset and in a separate process from the OLTP workload.

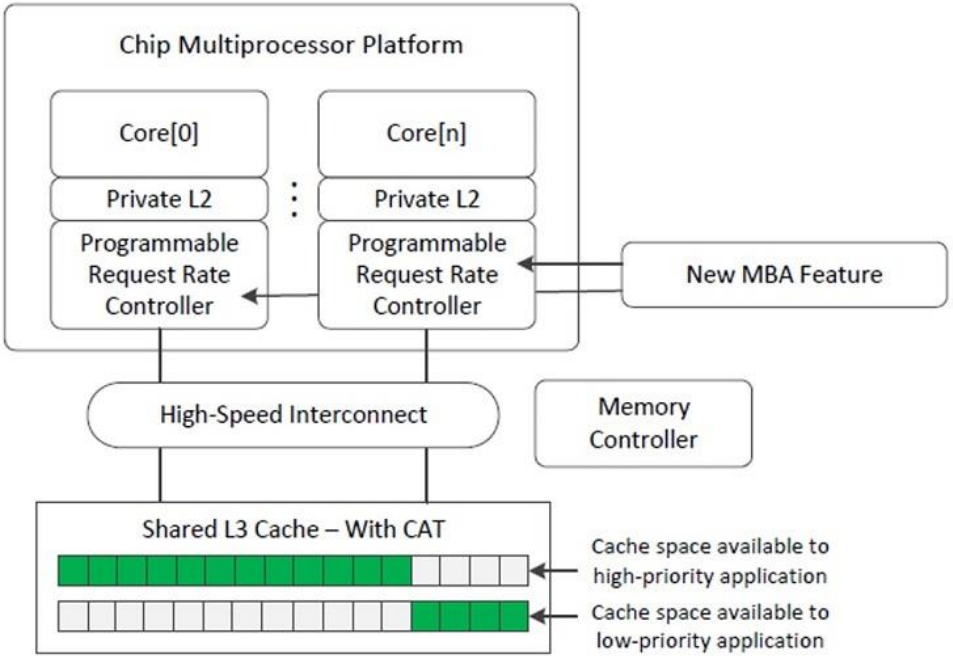
- Set-up: 5/10 cores allocated to the OLTP process and measure its performance when:
  - runs alone (in isolation, no interference)
  - runs co-located with the bandwidth intensive scan running on the other 5 cores (i.e., local NUMA scan)
  - the scan runs on 5 cores on another CPU and reads data locally (i.e., remote NUMA scan)



src: Makreshanski et al. “BatchDB: Efficient Isolated Execution of Hybrid OLTP and OLAP workloads for Interactive Applications” *SIGMOD’17*

Bandwidth allocation and partitioning

- In concurrent data processing workloads (and complex data center and enterprise deployments), we can easily get memory-bound (e.g., bottlenecked on the memory bandwidth).
- Need to ensure that the performance critical tasks (e.g., OLTP transactions) still meet their SLAs.
- New addition to Intel's RDT is **Memory Bandwidth Allocation** (in Intel Xeon Scalable processors), which extends the CAT
  - Also groups threads and applications into CLOS
  - Throttles them based on priorities



src: https://software.intel.com/content/www/us/en/develop/articles/introduction-to-memory-bandwidth-allocation.html

References

- Various papers cross-referenced in the slides
  - Li et al. “NUMA-aware algorithsm: the case of data shuffling” *CIDR 2013*
  - Balkesen et al. “Multi-core, Main-Memory Joins: Sort vs Hash Revisited” *VLDB 2014*
  - Schul et al. “An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory.” *SIGMOD 2016*
  - Leis et al. “Morse-Driven Parallelism: A NUMA-aware query evaluation framework for the many-core age” *SIGMOD 2014*
  - Porobic et al. “Analyzing the impact of system architecture on the scalability of OLTP engines for high-contention workloads” *VLDB 2017*
  - Lee et al. “MCC-DB: Minimizing Cache Conflicts in Multi-core Processors for Databases” *VLDB 2009*
  - Lo et al. “Heracles: Improving Resource Efficiency at Scale” *ISCA 2015*
  - Makreshanski et al. “BatchDB: Efficient Isolated Execution of Hybrid OLTP and OLAP workloads for Interactive Applications” *SIGMOD’17*

- Lecture: *Data Processing on Modern Hardware* by Prof. Jens Teubner (TU Dortmund, past ETH)

- Book: *What every programmer should know about memory?* by Ulrich Drepper
  - Chapters 5 and 6.5

- Intel Architectures Software Developer Manuals
  - Optimizing Applications for NUMA (https://software.intel.com/content/dam/develop/external/us/en/documents/3-5-memmgt-optimizing-applications-for-numa-184398.pdf)
  - Volume 3b: chapters 17.16 and 17.16 (for Intel RDT)