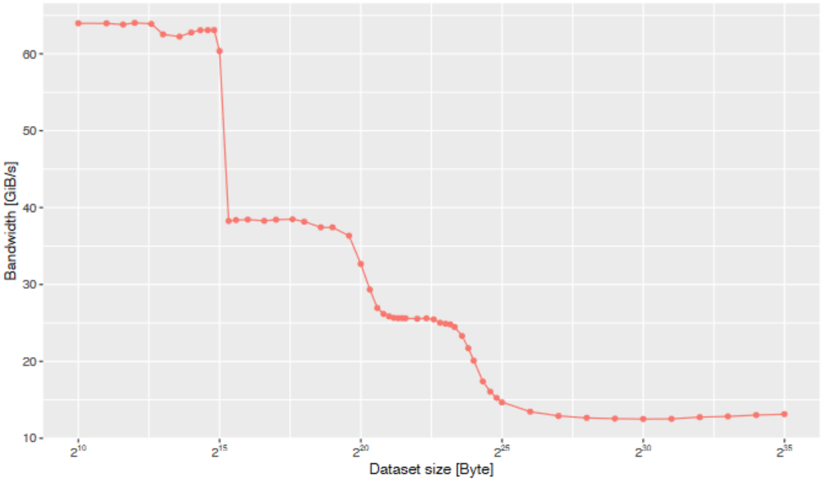
Data Processing on Modern Hardware

Jana Giceva

Lecture 4: In memory Joins

Recap: cache-awareness

- The plot shows the **bandwidth** of **scan** reading a dataset **sequentially** and **single-threaded.**
  The x-axis shows the size of the **dataset** (varied from 2^10 to 2^35).



- **Questions:**
  – Why is the performance of the scan not constant?
  – What do you see and what is the cause?
  – Can you use it to infer properties of the machine? What are they?

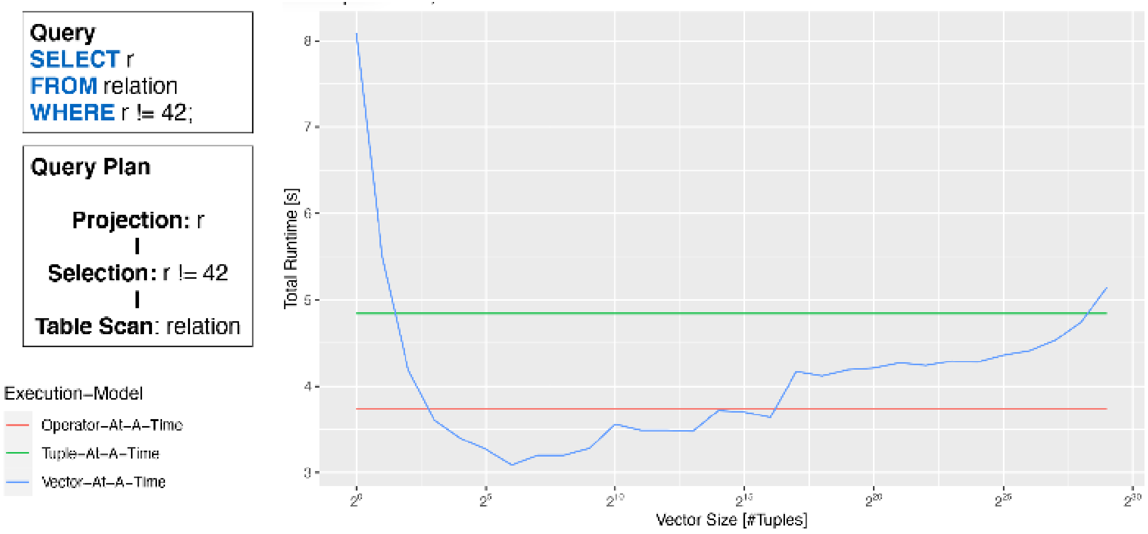
Recap: execution models

- When we compared the **compiled** vs. **vectorized** query execution models on a subset of **TPC-H** (analytical queries), we came to the conclusion that there is **no clear winner.**

- **Questions:**
  - What are the properties of a query that can benefit from compilation and (aggressive) operator fusion into pipelines?
  - What about vectorization?
  - Is there a way to get the best of both worlds? If yes, how? If not, describe the challenges. In both cases, give an example to support your claim.

Recap: execution models

- The plot shows the runtime of a query for different execution models.
- We vary the vector size from 1 to 2^30 tuples for the vector-at-a-time model.



- **Questions:**
  - – Describe the characteristics of the different execution models.
  - – Explain the curve of the vector-at-a-time model in relation to the vector size.
  - – Why is the operator-at-a-time faster than tuple-at-a-time for queries like this?
  - – For which queries do you expect the reverse behavior?

Hardware conscious in-memory joins

In-memory joins

After plain select queries, let us now look at **join queries:**

```
SELECT COUNT(*)
  FROM orders, lineitem
 WHERE o_orderkey = l_orderkey
```

We want to ignore result materialization for now, thus only **count** the result tuples.

Furthermore, we assume:
- No exploitable order
- No exploitable indices (input might be an intermediate result), and
- An equality join predicate (as above).
- No prior knowledge about key distribution

History of join processing: hashing vs. sorting

1970s – sorting

1980s – hashing

1990s – equivalent

2000s – hashing

2010s – hashing

2020s – ???

src: Andy Pavlo (Advanced Databases) CMU

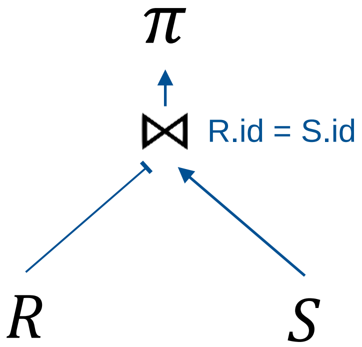
Hash Join

Hash Join is a good match for the equi-join example earlier

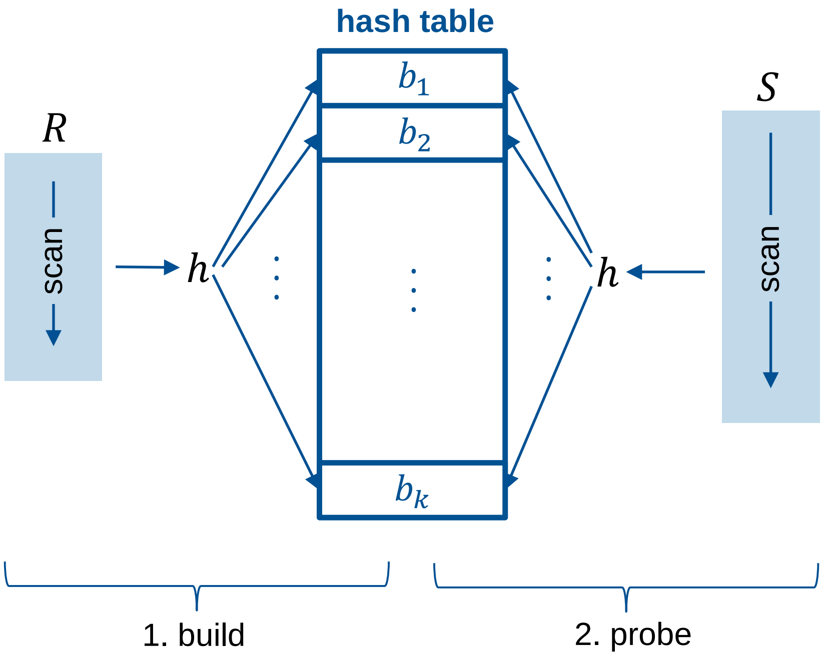
To compute $R \bowtie S$,

- 1. **Build a hash table** on the *outer* join relation R
- 2. **Scan** the *inner* relation S, and **probe** into the hash table for each tuple $s \in S$.

```
function: hash_join(R,S)
  // Build phase
for each tuple r ∈ R do
    insert r into hash table H
  // Join Phase
for each tuple s ∈ S do
    probe H and append matching tuples to result
```



Hash Join

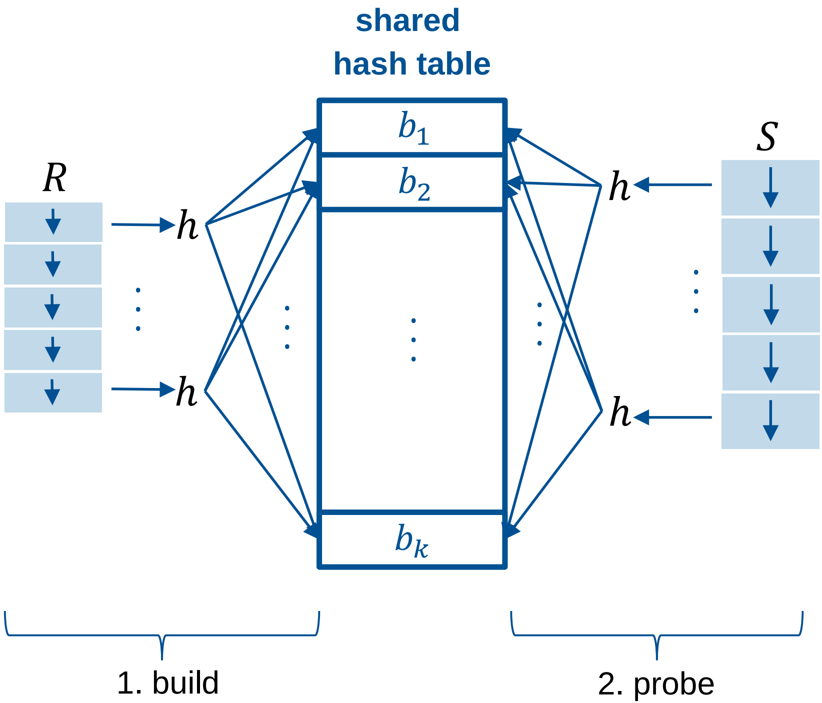


Complexity O(N)

Build is easy to ***parallelize***

Probe needs ***no synchronization***

Parallel Hash Join



Key characteristics:

- **Split** the input relations into chunks
- **Build**:
  - Each thread operates on its own input chunk and writes to a shared hash table
  - The shared hash table is protected using locks
  - Usually very low contention
- **Probe**:
  - Multiple readers – no synchronization needed
  - Each thread probes the hash table for its own chunk's tuples
  - Passes on the matched tuples

(Parallel) Hash Joins on Modern Hardware

Algorithm design goals for modern hardware:

- Minimize synchronization
  - avoid taking latches during execution
- Minimize memory access cost
  - ensure that data is local to worker thread
  - reuse data while it is still in the cache

The **naïve** parallel **hash join** has a lot of **random accesses**

- For large relations, every hash table access will likely be a **cache miss**
- The better the hash function, the more random the distribution of keys

Cost per tuple (build phase):

- 34 assembly instructions
- 1.5 cache misses
- 3.3 TLB misses

→ hash join is severely **latency bound**

Hardware-oblivious vs conscious dilemma

- **Hardware-conscious:**
  - Best performance can be achieved by fine-tuning to the underlying architecture:
    Cache hierarchy, translation lookaside buffer (TLB), non-uniform memory accesses (NUMA), etc.

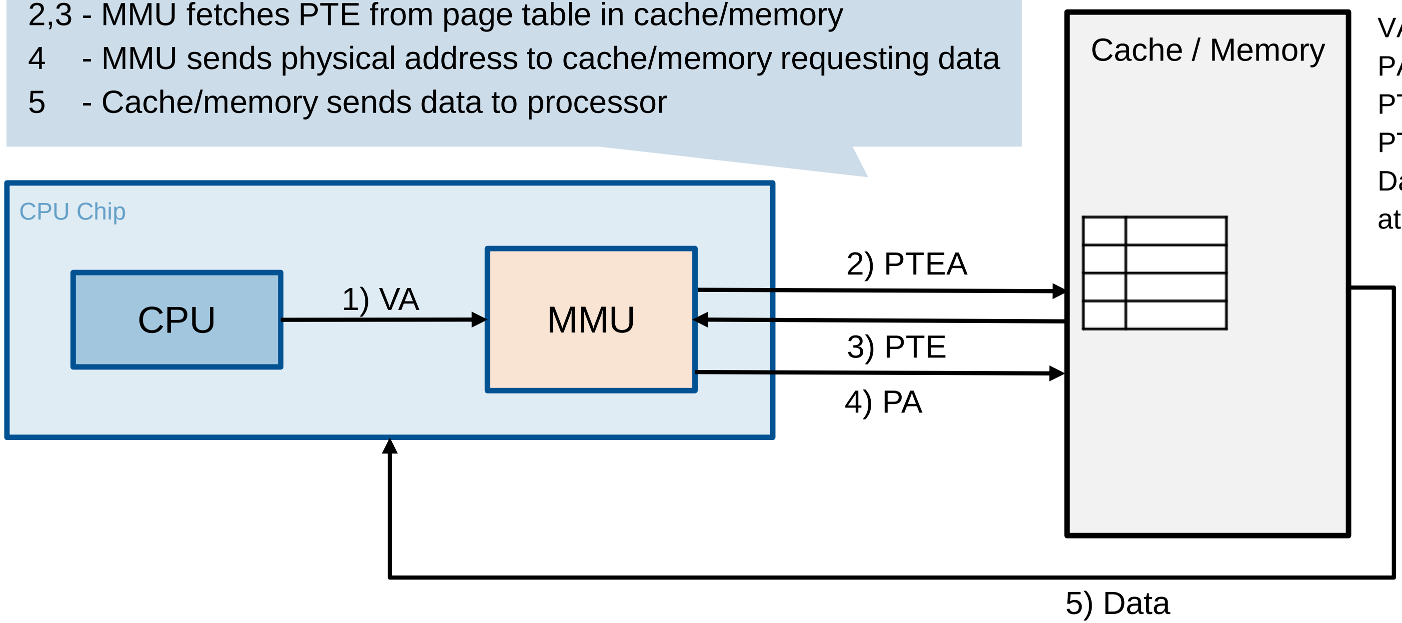
- **Hardware-oblivious:**
  - Algorithms can be efficient while remaining hardware oblivious because modern hardware hides the performance loss inherent in the multi-layer memory hierarchy with hyper-threads
  - Easily portable to different hardware
  - More robust to data-skew

Quick recap of virtual memory and address translation

Memory translation

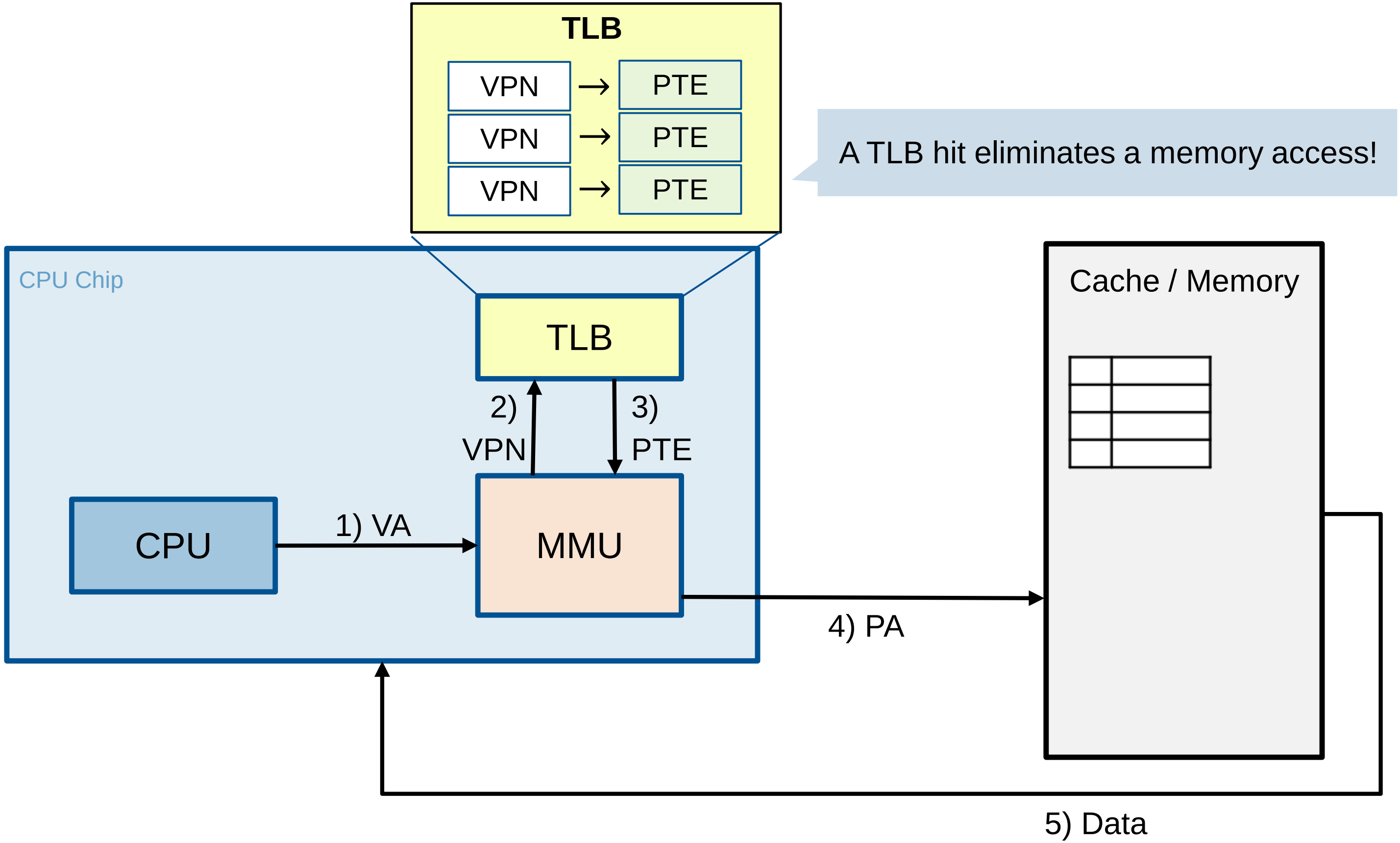
- Request is virtual address (VA), want physical address (PA)
- Use look-up table that we call ***page table (PT)***

1 - Processor sends virtual address to MMU
2,3 - MMU fetches PTE from page table in cache/memory
4 - MMU sends physical address to cache/memory requesting data
5 - Cache/memory sends data to processor

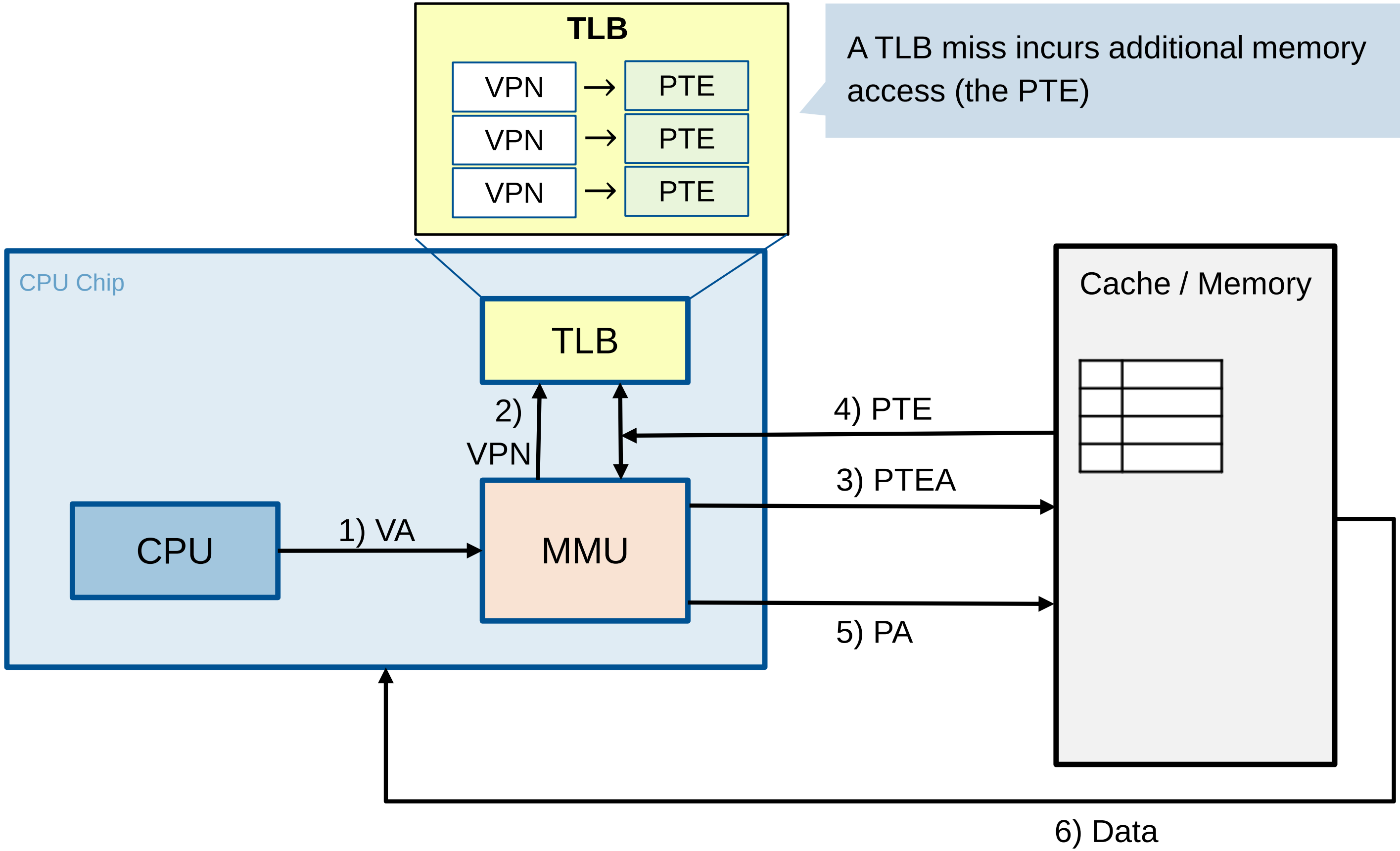


VA = Virtual Address
PA = Physical Address
PTEA = Page Table Entry Address
PTE = Page Table Entry
Data = Content of memory stored at VA originally requested by CPU

Translation Lookaside Buffer (TLB)

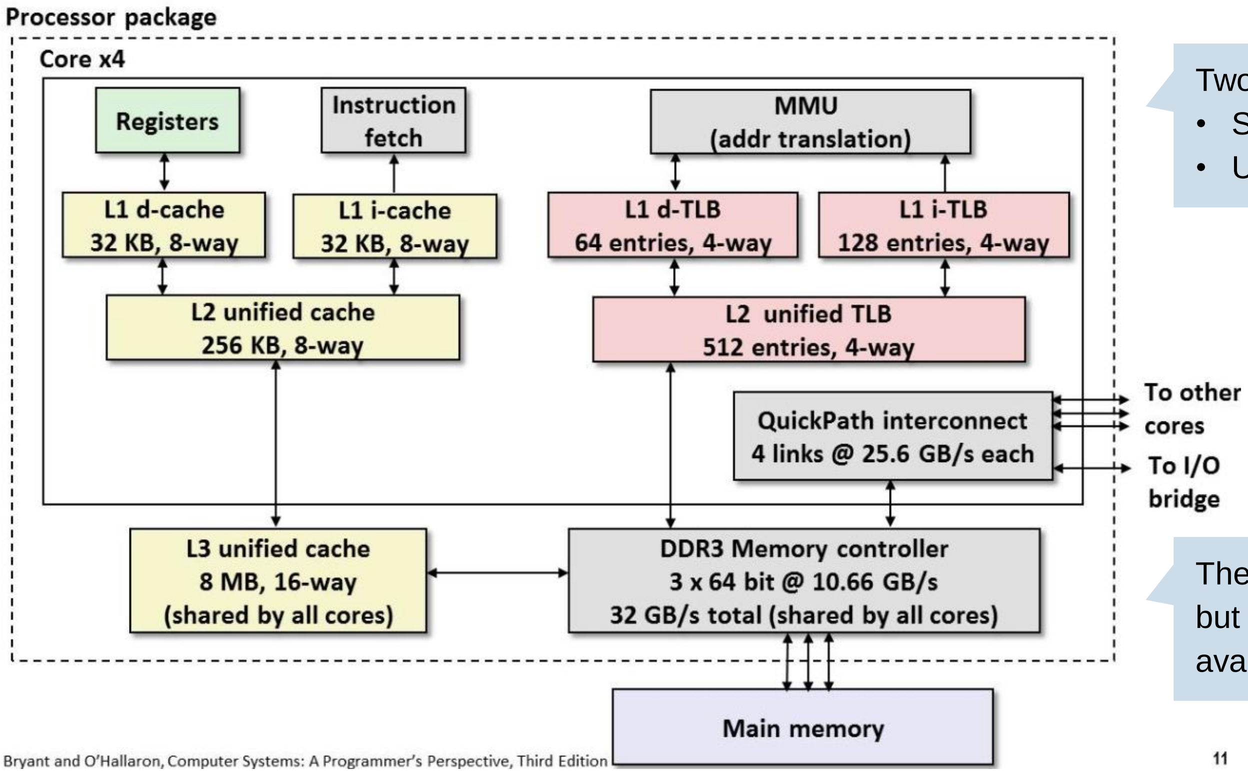


TLB Miss



A TLB miss incurs additional memory access (the PTE)

Intel Core i7 Memory System



Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Two level TLB caches

- Separate L1 for instruction and data
- Unified L2 for both

The standard page size on x86 is 4kB, but larger sizes 2MB and 1GB are also available (i.e., huge pages).

Back to hash joins

Improving the cache behavior

Factors that affect cache misses in a DBMS:

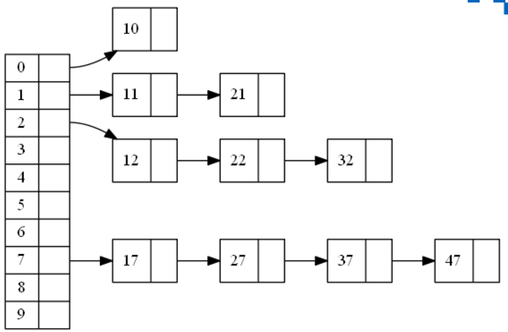
- Cache + TLB capacity
- Locality (temporal + spatial)

Key approaches to use:

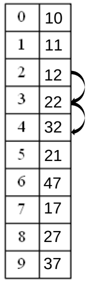
- **Sequential** (strided) access (*e.g.*, table scan):
  - Cluster and align data to a cache line
  - Execute more operations per cache line
- **Random** access (*e.g.*, index look-ups):
  - Pre-fetch data from memory manually
  - Use the blocking technique – partition data to fit in cache
  - Watch-out for the TLB cache

src: Johannes Gehrke (Main-memory database systems) http://www.cs.cornell.edu/courses/cs632/2001sp/slides/Main-memory%20database%20systems.ppt

Hashing schemes



- **Chained hashing:**
  - Maintain a linked list of buckets for each slot in the hash table
  - Resolve collisions by placing all elements with the same hash key into the same bucket



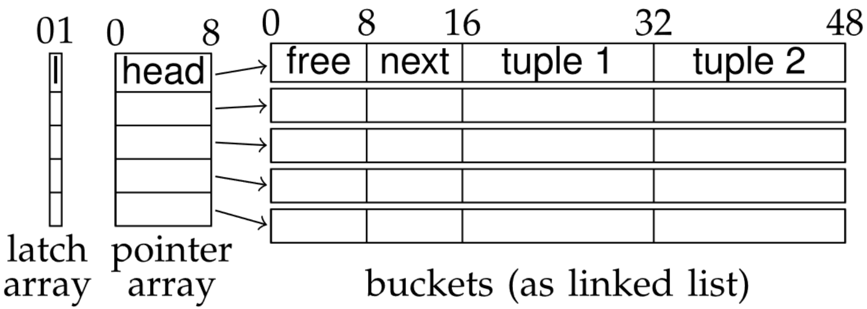
- **Open addressing:**
  - Use a single giant table of slobs
  - linear probing (LP) – resolve collisions by linearly searching for the next free slot in the table
  - other probe sequences (e.g., quadratic, robin-hood, hopscotch, etc.)

- **Different trade-offs:**
  - **Locality:** pointer chasing vs. sequential access
    - Chaining better performance during build phase
    - LP better throughput during probe phase
  - **Robustness:** on high load factors, LP suffers from primary clustering

Hash Table implementation

- Even for a simple chain hashing scheme, there are many things to consider.

- Naïve implementation:
  - Hash table is an array of head pointers, each of which points to the head of a linked bucket chain.
  - Each bucket is implemented as a 48-byte record:
    - `free` points to the next available tuple space,
    - `next` pointer leads to the next overflow buffer
    - the bucket holds two `16-byte tuples`.
  - Since it is a shared hash table, latches are needed for synchronization. Implemented as a separate latch array.
  - 3 separate cache lines



Three steps to insert a new entry:

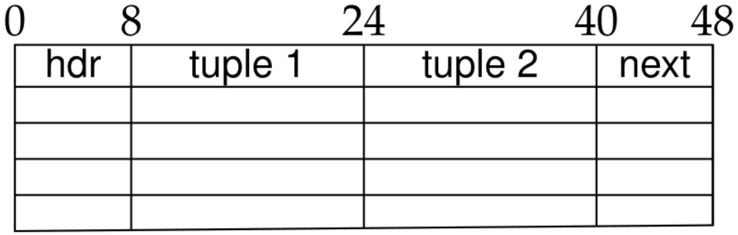
1. The latch must be locked from the latch array
2. The head must be read from the pointer array
3. The head pointer should be dereferenced to find the hash bucket

Each step could be a cache miss!

src: Blanas *et al.* Design and evaluation of main memory hash join algorithms for multi-core CPUs *SIGMOD 2011*

Hash Table implementation

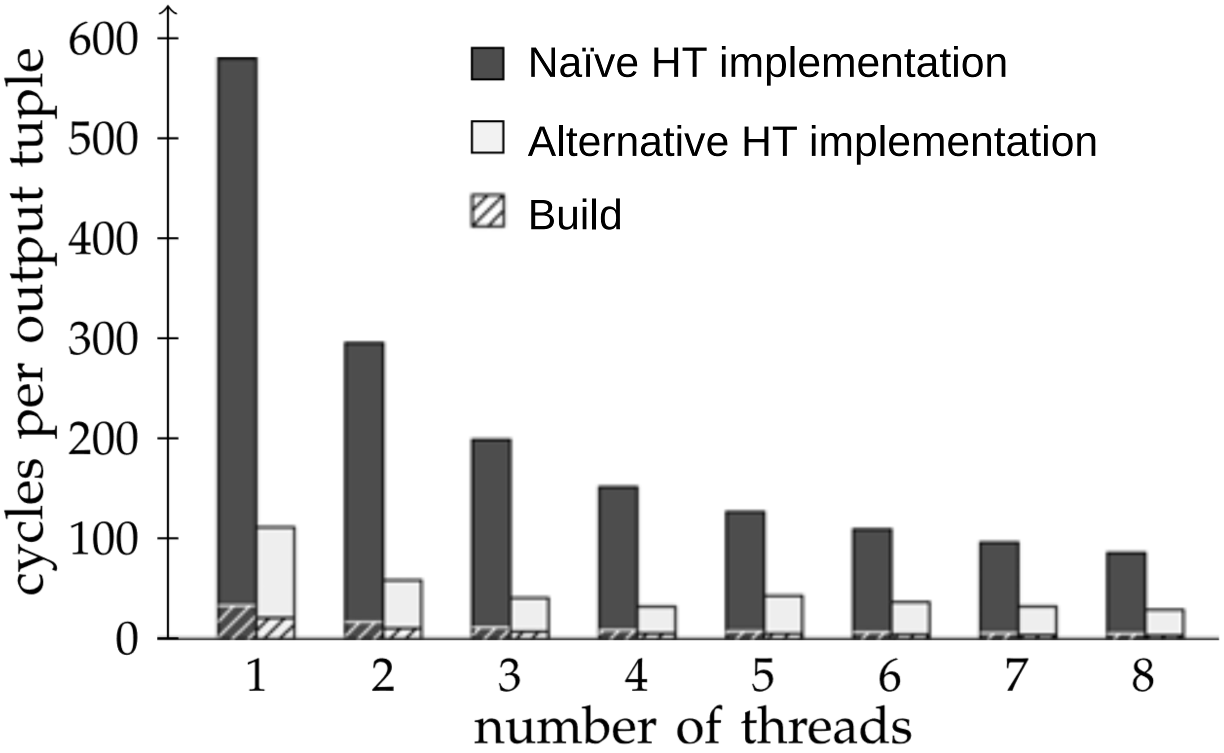
- An alternative chain hashing scheme:
  - The main hash table is a contiguous array of buckets.
  - Header contains 1-byte for latch, and a 7-byte counter indicating the number of tuples in the bucket.
  - Contains two 16-byte tuples.
  - For overflow, additional buckets are allocated outside the main hash table, referenced by the `next` pointer.
  - Fits in 1 cache line



Contiguous memory block can reduce the number of cache misses significantly.

src: Balkesen *et al.* Main-memory Hash Joins on Modern Processor Architectures *ICDE 2014*

Performance impact of HT implementation



src: Balkesen *et al.* Main-memory Hash Joins on Modern Processor Architectures *ICDE 2014*

Improving cache behavior for the hash join

The **hash join** has inherently a lot of **random accesses**, which is a problem when the **data** is large and **does not fit in the cache**.

There are two main options one could take:

- **Pre-fetching**
  - Recall assignment 1 → the hardware pre-fetcher cannot help with random accesses
  - But: a software pre-fetcher can issue memory requests ahead of time and hide latencies [1]
- **Partitioning**
  - Recall blocked matrix multiplication example →
  - Split the input relations into cache-resident buffers by hashing the tuples' join key(s) [2]
  - Insight: the cost of partitioning is often less than the overhead of cache misses for build and probe

[1] Chen *et al.* Improving Hash Join Performance through Prefetching. *ICDE 2004*
[2] Shatdal *et al.* Cache conscious algorithms for relational query processing. *VLDB 1994*

Case 1: Software based prefetching

- To hide cache miss latencies in hash joins, one can use **software pre-fetching.**
- Modify the source code using **special instructions** (compiler **intrinsic**) on any pointer in the program.

```
__mm_prefetch(void *p, enum __mm_hint h);
```

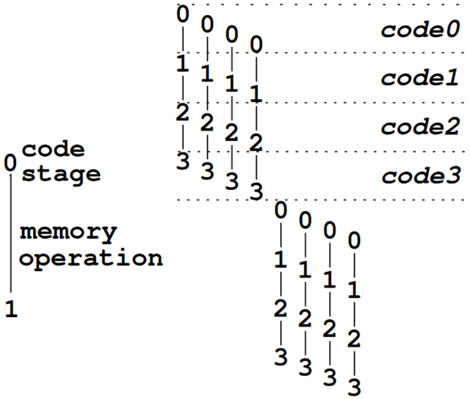
- **Group pre-fetching**
  - Modified forms of compiler transformations called *strip mining* and *loop distributions*
  - Restructure the code so that hash probe accesses resulting from groups of G consecutive probe tuples can be pipelined

- **Software pipelining**
  - Generate efficient schedules for loops by overlapping the execution of operations from different iterations of the loop.
  - Assume there are no inter-tuple dependencies (for simplicity)

src: Chen *et al.* Improving Hash Join Performance through Prefetching. *ICDE 2004*

Group pre-fetching (example)

```
for i=0 to N-1 do
  code 0;
  visit (m_i^1); code 1;
  visit (m_i^2); code 2;
  ...
  visit (m_i^k); code k;
end for
```



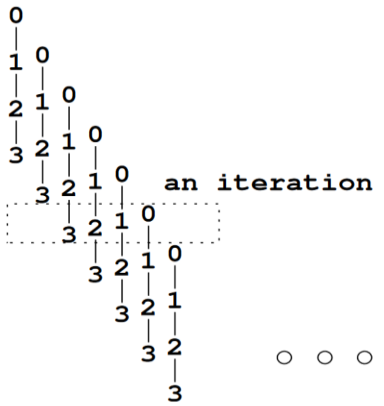

```
for j=0 to N-1 step G do
  for i=j to j+G-1 do
    code 0;
    prefetch (m_i^1);
  end for
  for i=j to j+G-1 do
    visit (m_i^1); code 1;
    prefetch (m_i^2);
  end for
  for i=j to j+G-1 do
    visit (m_i^2); code 2;
    prefetch (m_i^3);
  end for
  ...
  for i=j to j+G-1 do
    visit (m_i^k); code k;
  end for
end for
```

src: Chen *et al.* Improving Hash Join Performance through Prefetching. *ICDE 2004*

Software-pipelined pre-fetching

D is the prefetching distance.

```
for i=0 to N-1 do
  code 0;
  visit ($m_i^1$); code 1;
  visit ($m_i^2$); code 2;
  ...
  visit ($m_i^k$); code k;
end for
```




```
for j=0 to N-kD-1 do
  i=j+kD;
  code 0 for element i;
  prefetch($m_i^1$);

  i=j+(k-1)D;
  visit ($m_i^1$); code 1 for element i;
  prefetch($m_i^2$);

  i=j+(k-2)D;
  visit($m_i^2$); code 2 for element i;
  prefetch($m_i^3$);

  ...

  i=j;
  visit($m_i^k$); code k for element i;
end for
```

src: Chen *et al.* Improving Hash Join Performance through Prefetching. *ICDE 2004*

Group vs software-pipelined pre-fetching

Software-pipelined:

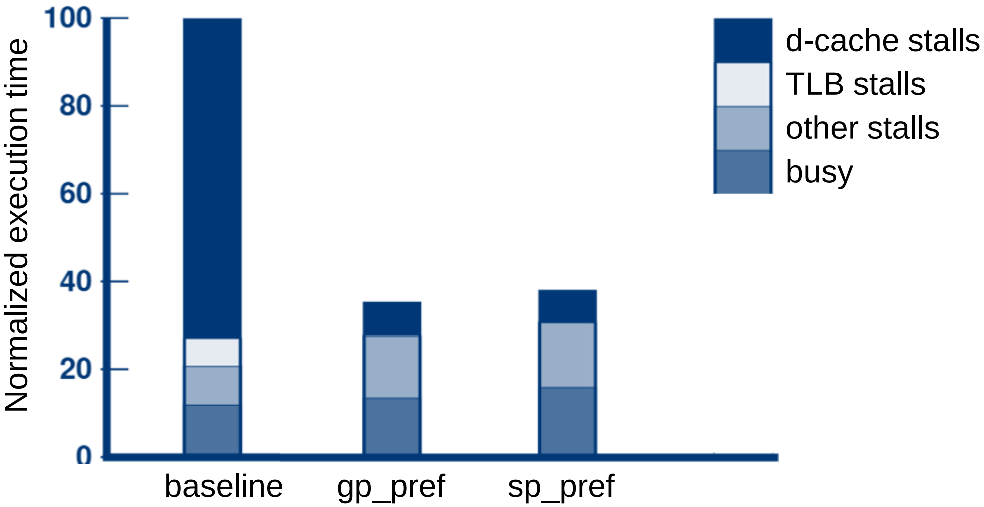
- Can always hide miss latencies
- But, has a larger book-keeping overhead and larger maintained state

Group:

- Easier to implement
- Not all cache misses can be hidden (esp. when code 0 is empty)
  - Can be amortized with large group of elements

src: Chen *et al.* Improving Hash Join Performance through Prefetching. *ICDE 2004*

Impact of prefetching on join performance



Big reduction of d-cache stalls
Eliminate TLB stalls

src: Chen *et al.* Improving Hash Join Performance through Prefetching. *ICDE 2004*

Case 2: partitioning

- Recall the *blocking* matrix multiplication example?

- In ***blocking***, an *algorithm* is restructured to reuse chunks of data that fit in the cache.

```
for (i=0; i<M; i++)
  for (j=0; j<N, j++)
    process(a[i][j]);
```

➡

```
for (b=0; b<N/B; b++)
  for (i=0; i<M, i++)
    for (j=b+B; j<(b+1)*B; j++)
      process(a[i][j]);
```

- In ***partitioning***, the *layout* of the *input data* is reorganized to make maximum use of the cache
  - Make sure that partitions fit in the cache

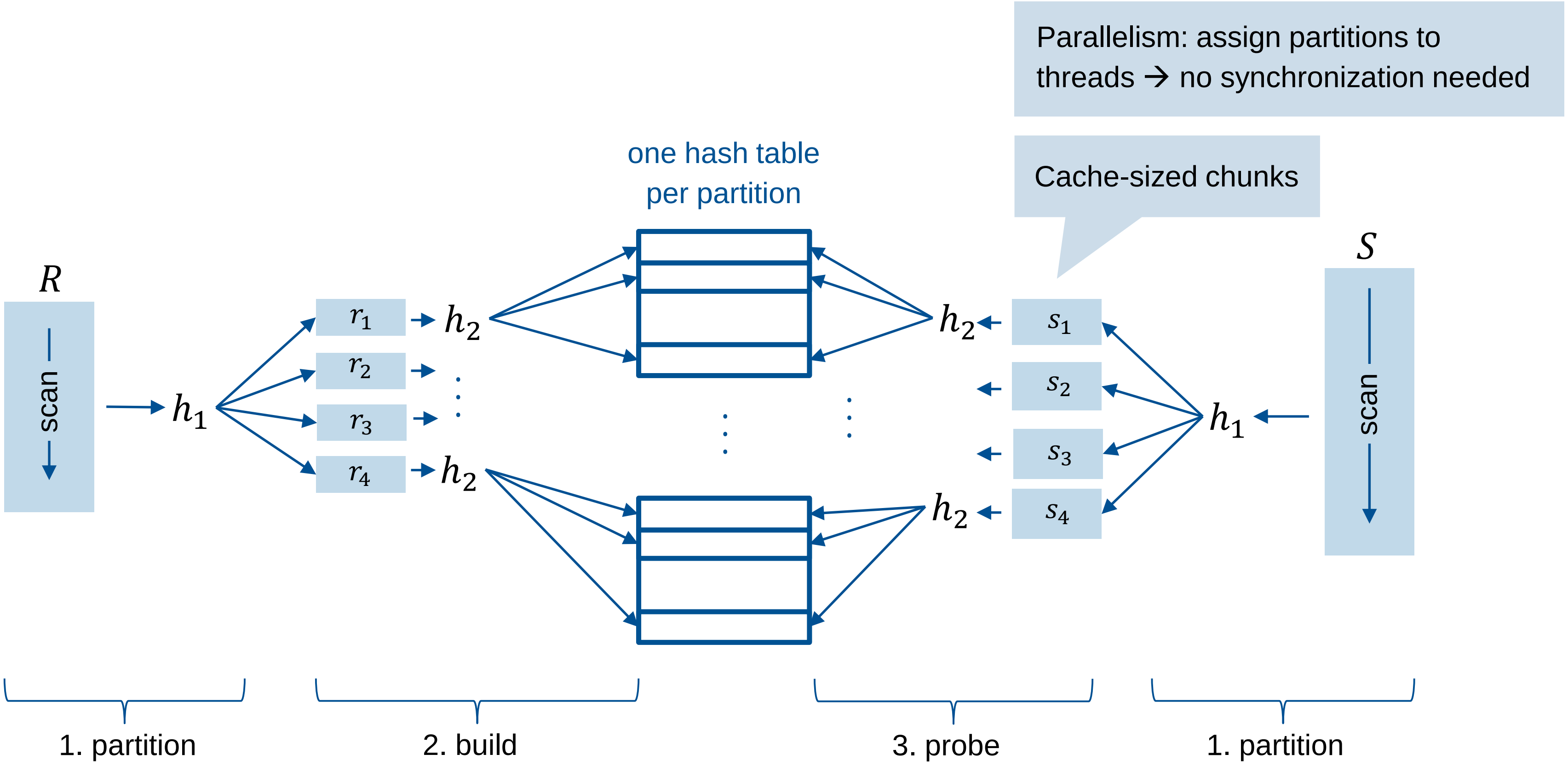
```
quicksort(relation[N])
```

➡

```
partition relation into blocks < cache size
for each partition r
  quicksort(relation[PARTITIONSIZE]);
merge all partitions
```

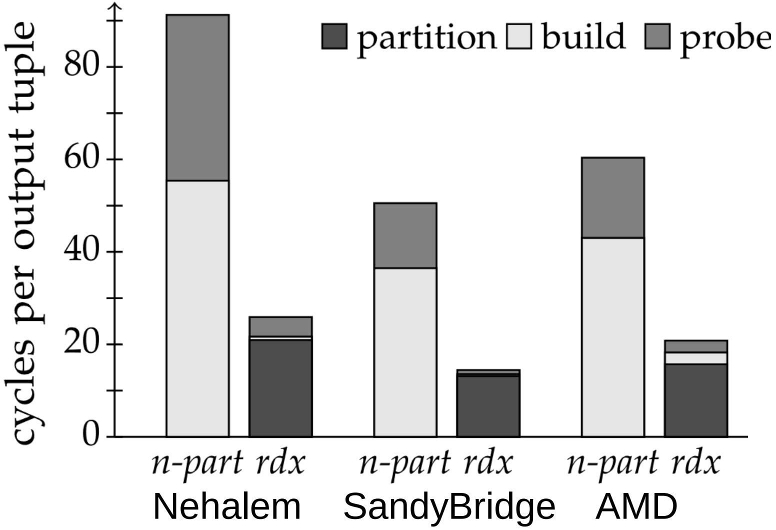
src: Nyberg *et al.* AlphaSort: A RISC Machine Sort. *SIGMOD 1994*

Partitioned Hash Join



Cache analysis of Partitioned Hash Joins

- **Build / Probe are now contained within the caches:**
  - From **34** down to **15/21** instructions per tuple (build/probe)
  - From **1.5** down to **0.01** cache misses per tuple
  - From **3.3** down to almost no TLB misses



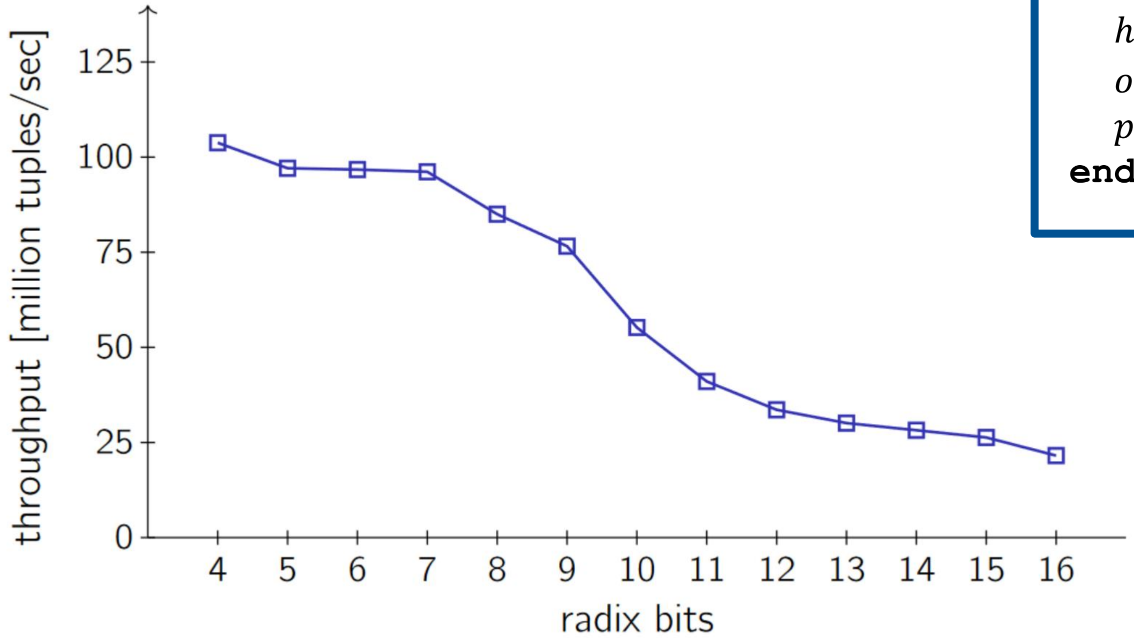
- Joining two relations with 8B key+payload and 128M tuples (total size 977MB)
- Measured on 3 different machines

- **Partitioning** is now critical
  - Many partitions are far apart
  - Each one will reside on its own page
  - Run out of **TLB entries** (100-500)

src: Balkesen *et al.* Main-memory Hash Joins on Modern Processor Architectures *ICDE 2014*

Cost of partitioning

Partitioning is expensive beyond ~ $2^8 - 2^9$ partitions



for all input tuples t **do**
  $h \leftarrow hash(t.key)$
  $out[pos[h]] \leftarrow t$
  $pos[h] \leftarrow pos[h] + 1$
end for

Due to TLB thrashing

src: Jens Teubner Lecture: *Data Processing on Modern Hardware.*

Radix partitioning (basic)

```
// Build a histogram
for i = 0 to N − 1 do
    ++histogram[h(input[i])];

// Calculate prefix-sum
offset = 0;
for i = 0 to num_partitions − 1 do
    dest[i] = offset;
    offset += histogram[i];

// Partition the data
for i = 0 to N − 1 do
    bucket_num = h(input[i]);
    output[dest[bucket_num]] = input[i];
    ++dest[bucket_num];
```

Partition a dataset into 2^R partitions.

- In the **first pass** over the data, for each partition we count the entries that will be sent to it.
- From this histogram, we calculate the start index of each partition (i.e., prefix sum).
- The **second pass** over the data copies the entries to their designated partition.

Optimizing the radix sort - partitioning

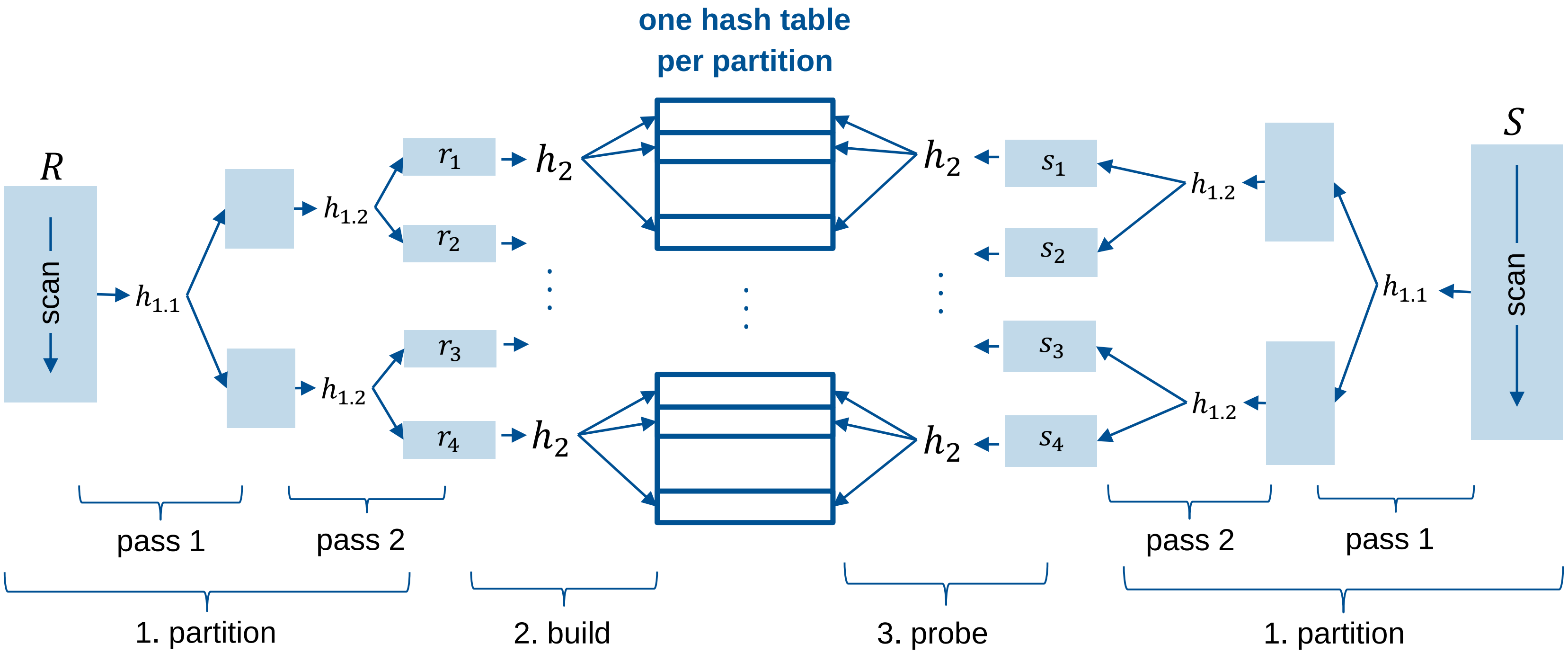
It's an art in itself and was studied extensively

- Single vs. multi-pass partitioning
- Software Write-Combine Buffers
- Non-temporal Streaming
- Using huge page tables
- NUMA awareness → covered in two weeks

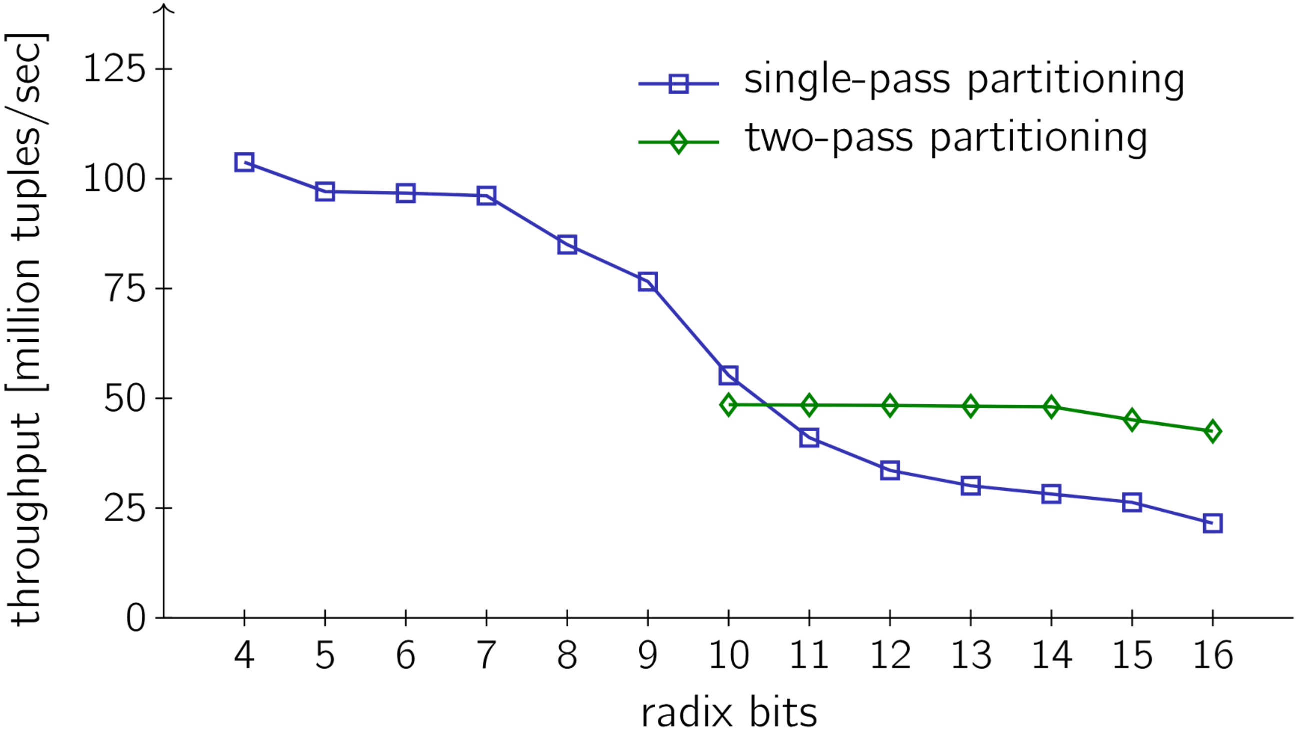
[1] Wassenberg and Sanders. Engineeringa multi-core radix-sort. *Euro-Par 2011*
[2] Polychroniou and Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison and radix-sort. *SIGMOD 2014*
[3] Schuhknecht *et al.* On the Surprising Difficulty of Simple Things: the Case of Radix Partitioning *VLDB 2015*

Multi-pass partitioning

- Creating too many partitions can easily thrash the TLB cache.
- Thus, do a multi-pass partitioning, and limit the fan-out of each partitioning pass



Multi-pass partitioning



Software managed buffers

Naïve partitioning

$$
\begin{array}{l}
\textbf{for all } \texttt{input tuples } t \textbf{ do} \\
\quad h \leftarrow hash(t.key) \\
\quad \texttt{copy } t \texttt{ to } out[pos[h]] \\
\quad pos[h] \leftarrow pos[h] + 1 \\
\textbf{end for}
\end{array}
$$

Memory access

Software managed buffers

$$
\begin{array}{l}
\textbf{for all } \texttt{input tuples } t \textbf{ do} \\
\quad h \leftarrow hash(t.key) \\
\quad buf[h][pos[h] \bmod bufsize] \leftarrow t \\
\quad \textbf{if } pos[h] \bmod bufsize = 0 \textbf{ then} \\
\quad\quad \texttt{copy } buf[h] \texttt{ to } out[pos[h] - bufsiz] \\
\quad \textbf{end if} \\
\quad pos[h] \leftarrow pos[h] + 1 \\
\textbf{end for}
\end{array}
$$

Memory access

- TLB miss only every *bufsize* tuples
- Choose *bufsize* to match **cache line size**

Non-temporal Streaming Stores

Key idea: keep the working set warm in cache, and issue memory writes that bypass the cache

Method: non-temporal streaming stores

```
__mm256_stream_si256(__m256i* mem, __m256i a)
```

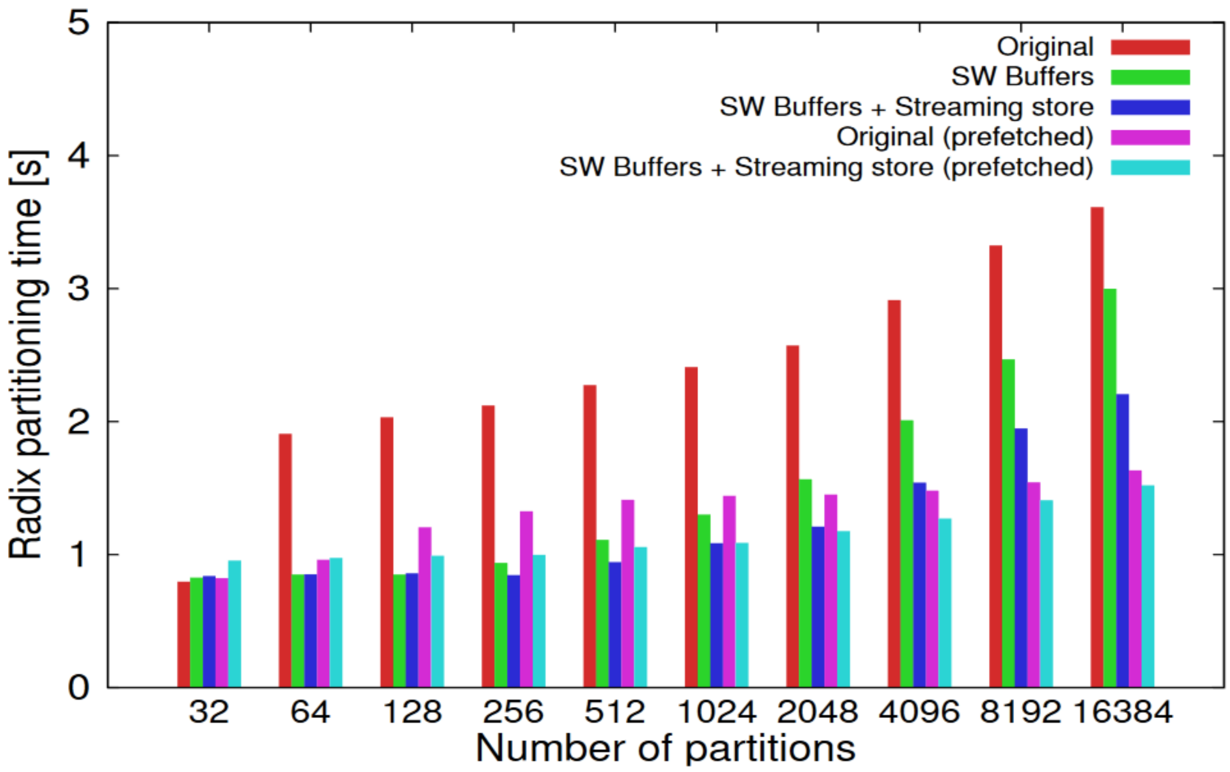
- This AVX intrinsic writes 4 buffered 64-bit entries to a partition at once (i.e., half a cache line).

- The processor tries to fill a cache line in its own *write-combine buffer* before writing to memory
- As soon as it is filled, it is flushed out without reading the corresponding cache-line from memory.

- **Caveat:** the memory address must be aligned to 32 Bytes = 256 bits

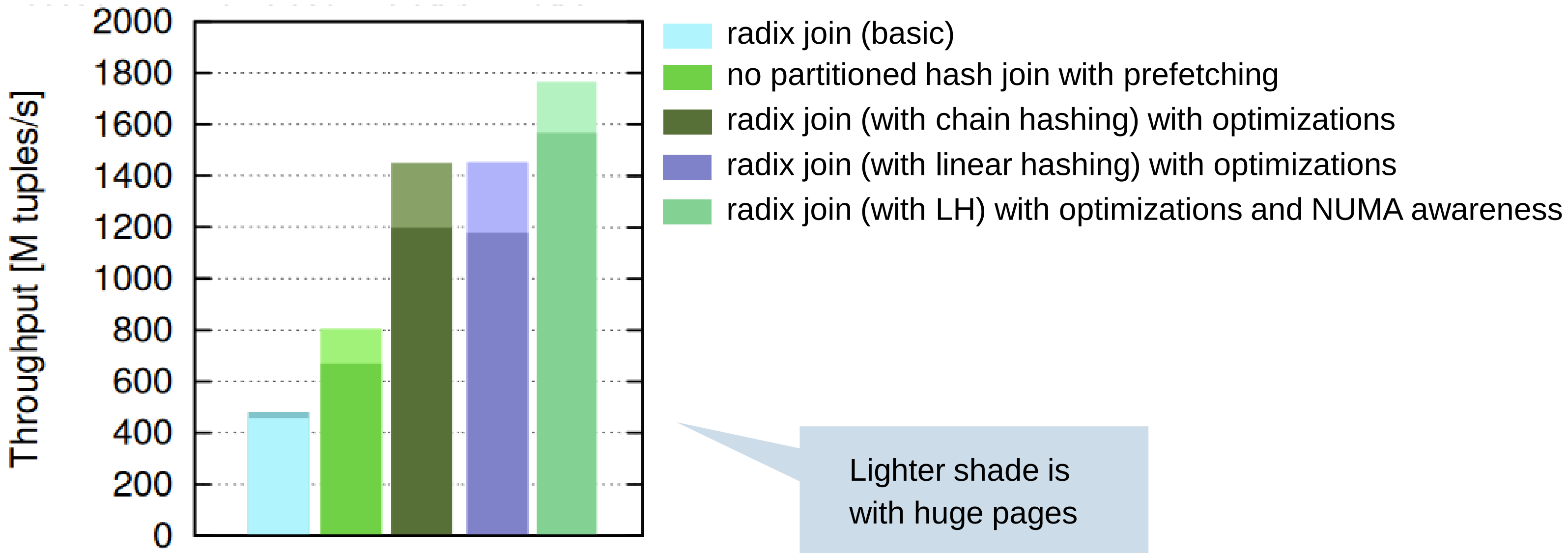
- For AVX 512, we can fill a full cache line per call ☺

Partitioning performance



src: Schuhknecht *et al.* On the Surprising Difficulty of Simple Things: the Case of Radix Partitioning *VLDB 2015*

Results



Lighter shade is
with huge pages

src: Schuh *et al.* An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory *SIGMOD 2016*

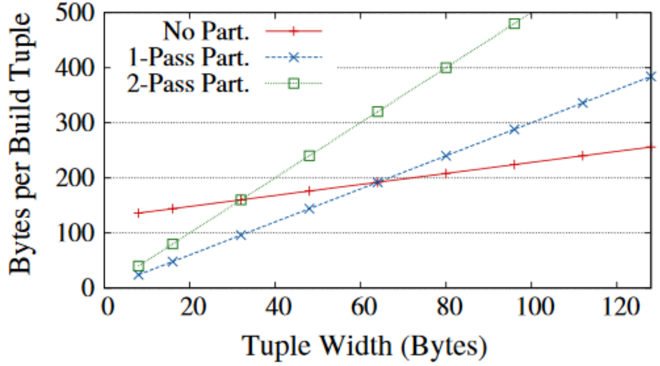
So far, join on narrow tuples

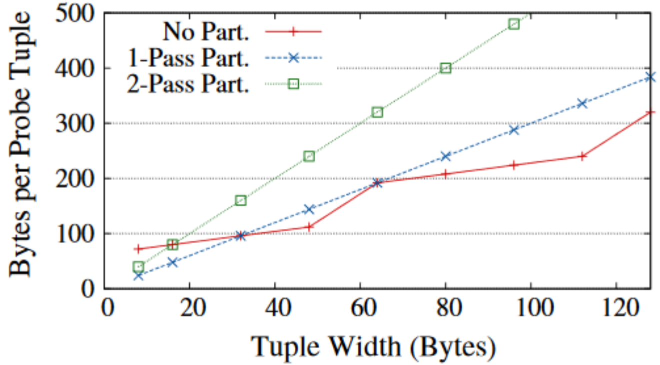
- If optimized well, with prefetching or SWWCB and streaming instructions, the join quickly becomes memory bound

- A simple analytical model can tell us when to use which type of join (no-partitioning, or radix-join).

Table 1 Model for memory bandwidth consumed per tuple for subop-erations of hash join algorithms

	Bytes read	Bytes written
Out-of-cache build	$CL + t$	CL
Out-of-cache probe	$CL \cdot \lceil \frac{t+m}{CL} \rceil + t$	0
In-cache build	t	0
In-cache probe	t	0
Partition	t	t

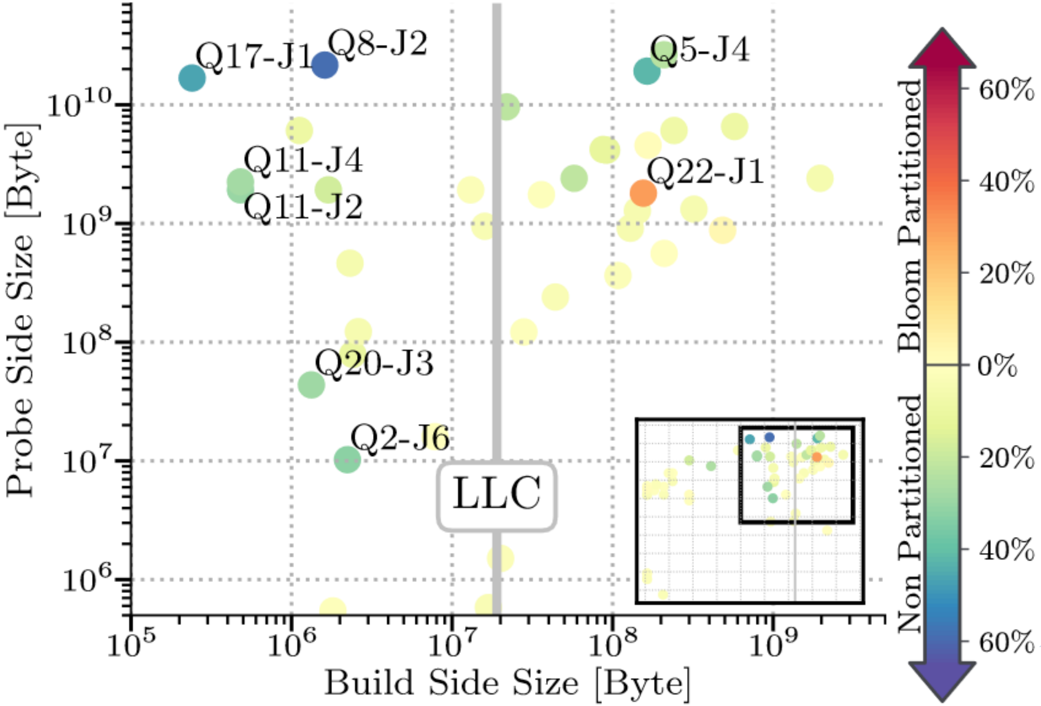




src: Makreshanski *et al.* Many-query join: efficient shared execution of relational joins on modern hardware *VLDBJ 2018*

Radix-join in a full-fledged database

- When is it worth to partition the data beforehand?



Factors	Workable	Beneficial
Selectivity	handled by Bloom filter	
Payload Size[9]	≤ 32B	≤ 16B
Pipeline Depth	< 8 Joins	< 2 Joins
Skew (Zipf)	≤ 1	≤ 0.5
Build Size	> *LLC*	≫*LLC*
Size Difference	< ×50	< ×10

- Radix Join is optimal in rare cases
- Not competitive for real-world workloads

- TPC-H joins performance comparison of RJ wrt non-partitioned join (w/ prefetching)

References

- Various papers cross-referenced in the slides

  - Wassenberg and Sanders. Engineeringa multi-core radix-sort. *Euro-Par 2011*
  - Chen *et al.* Improving Hash Join Performance through Prefetching. *ICDE 2004*
  - Shatdal *et al.* Cache conscious algorithms for relational query processing. *VLDB 1994*
  - Blanas *et al.* Design and evaluation of main memory hash join algorithms for multi-core CPUs *SIGMOD 2011*
  - Balkesen *et al.* Main-memory Hash Joins on Modern Processor Architectures *ICDE 2014*
  - Polychroniou and Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison and radix-sort. *SIGMOD 2014*
  - Schuhknecht *et al.* On the Surprising Difficulty of Simple Things: the Case of Radix Partitioning *VLDB 2015*
  - Schuh *et al.* An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory *SIGMOD 2016*
  - Makreshanski *et al.* Many-query join: efficient shared execution of relational joins on modern hardware *VLDBJ 2018*

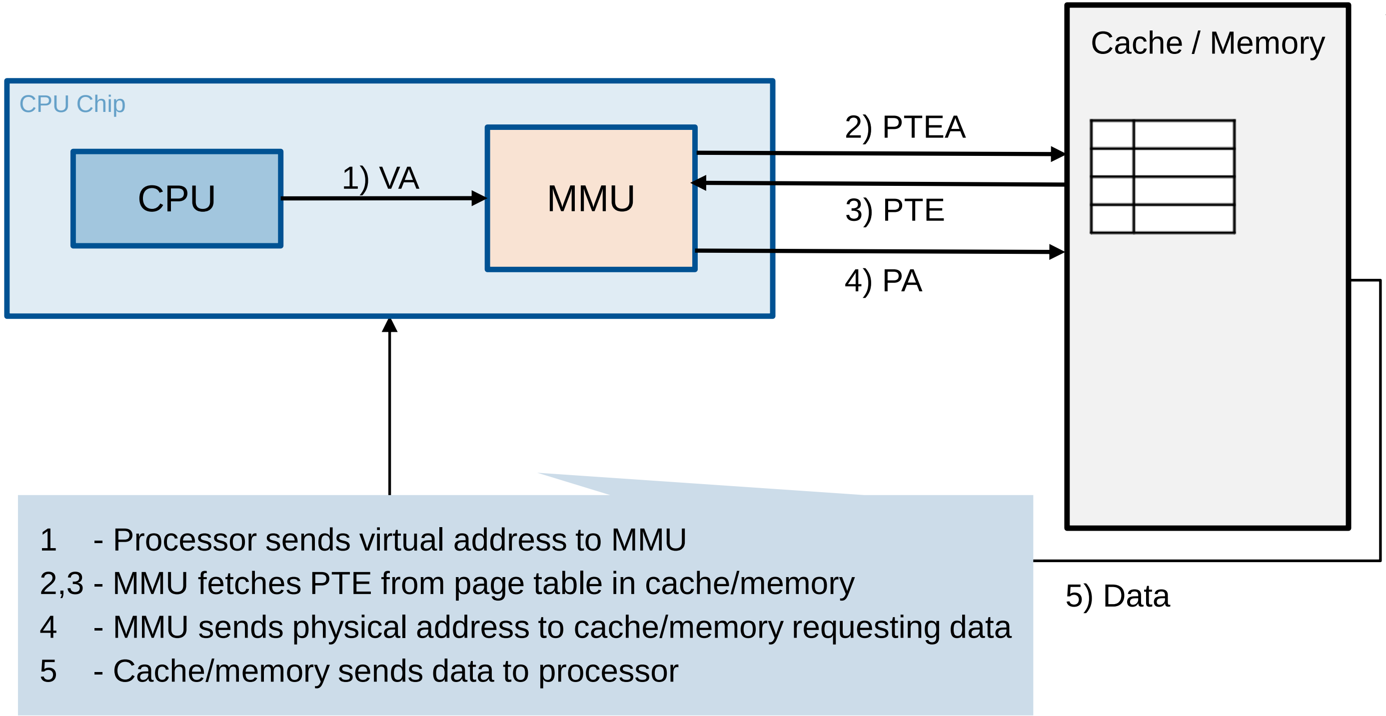
- Lecture: *Database Systems on Modern CPU Architectures* by Prof. Thomas Neumann (TUM)
- Lecture: *Data Processing on Modern Hardware* by Prof. Jens Teubner (TU Dortmund, past ETH)
- Lecture: *Advanced Databases* by Prof. Andy Pavlo (CMU)
- Book: *Computer Systems: A Programmer's Perspective* 3rd edition by Bryant and O'Hallaron
- Book: *What every programmer should know about memory* by Ulrich Drepper

- Intel manuals for software write combining, streaming instructions, software-based prefetching
  - https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html
- Check out the code from Cagri Balkesen for high performance radix join implementation:
  - https://www.systems.ethz.ch/node/334

Appendix – Address Translation

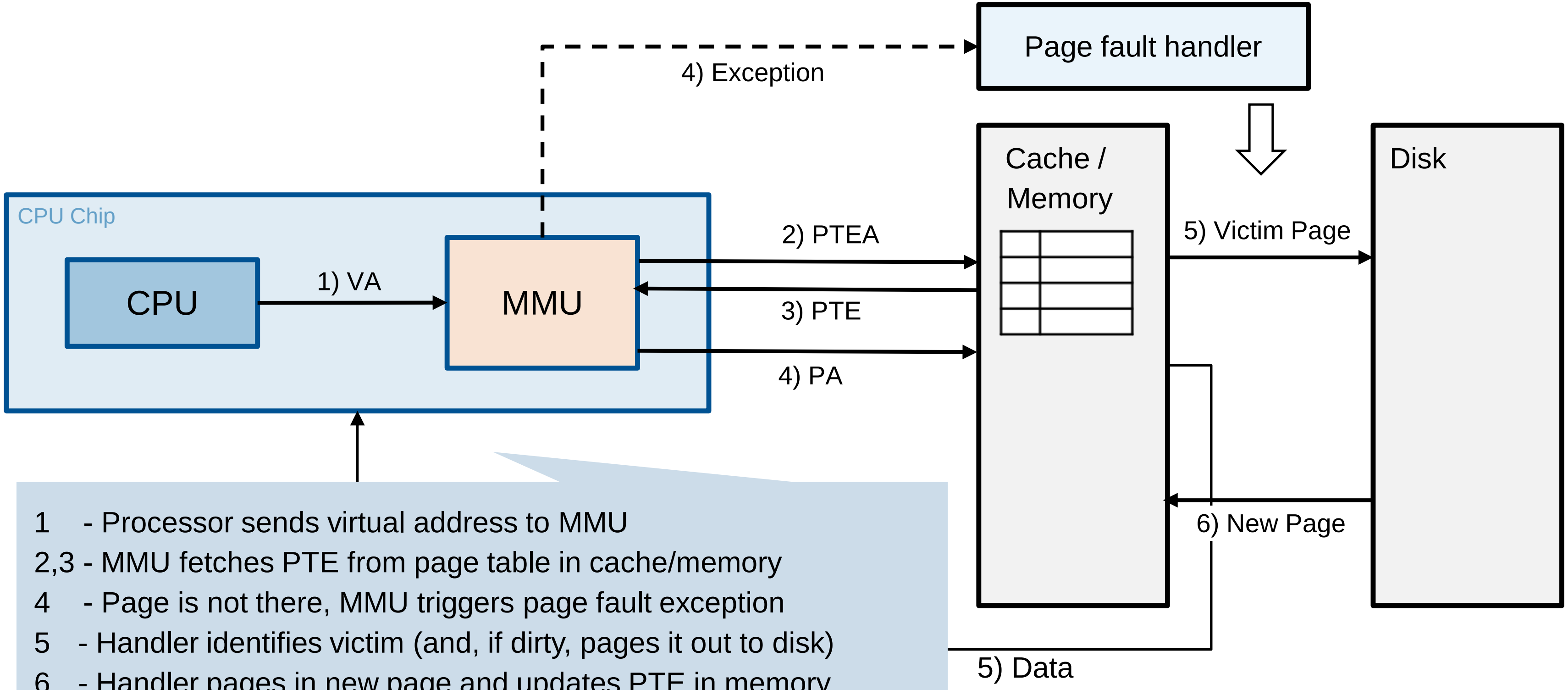
Address Translation: Page Hit

- Request is virtual address (VA), want physical address (PA)
- Use look-up table that we call ***page table (PT)***



VA = Virtual Address
PA = Physical Address
PTEA = Page Table Entry Address
PTE = Page Table Entry
Data = Content of memory stored at VA originally requested by CPU

Address Translation: Page Fault



1 - Processor sends virtual address to MMU
2,3 - MMU fetches PTE from page table in cache/memory
4 - Page is not there, MMU triggers page fault exception
5 - Handler identifies victim (and, if dirty, pages it out to disk)
6 - Handler pages in new page and updates PTE in memory
7 - Handler returns to original process, restarting faulting instructions

Address Translation