Data Warehousing und Data Mining: Modern Hardware

Stefan Sprenger

November 29, 2017

Content of this Lecture

- Introduction to Modern Hardware
  - CPUs, Cache Hierarchy
  - Branch Prediction
  - SIMD
  - NUMA
- Cache-Sensitive Skip List
  - Skip Lists
  - Our Contributions
  - Evaluation

Content of this Lecture

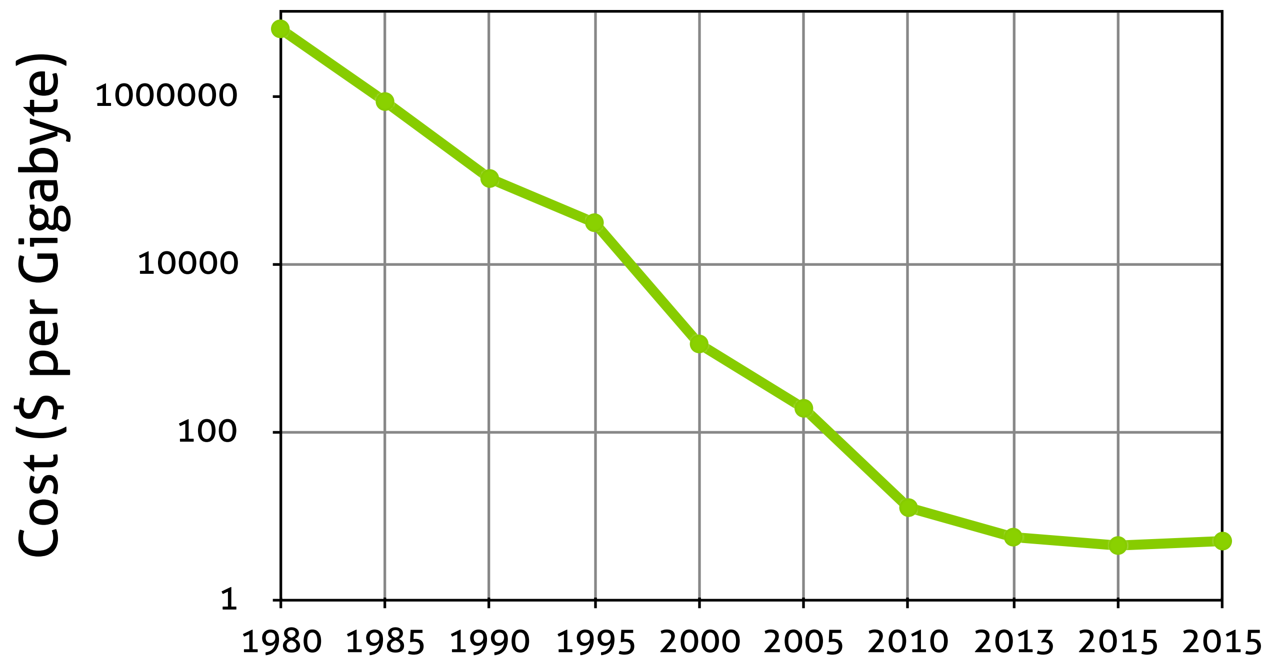
- Introduction to Modern Hardware
  - CPUs, Cache Hierarchy
  - Branch Prediction
  - SIMD
  - NUMA
- Cache-Sensitive Skip List
  - Skip Lists
  - Our Contributions
  - Evaluation

What is Modern Hardware?

- CPUs feature multiple cache levels
- many-core CPUs: massive parallelism
- multiple CPUs per machine: NUMA
- main memory is large and cheap enough to hold entire databases
  - elimination of the buffer pool
  - performance bottleneck moves up from disk/memory to memory/cpu caches
  - new goal: minimize cache misses
  - what about durability?
- specialized co-processors: GPUs, FPGAs

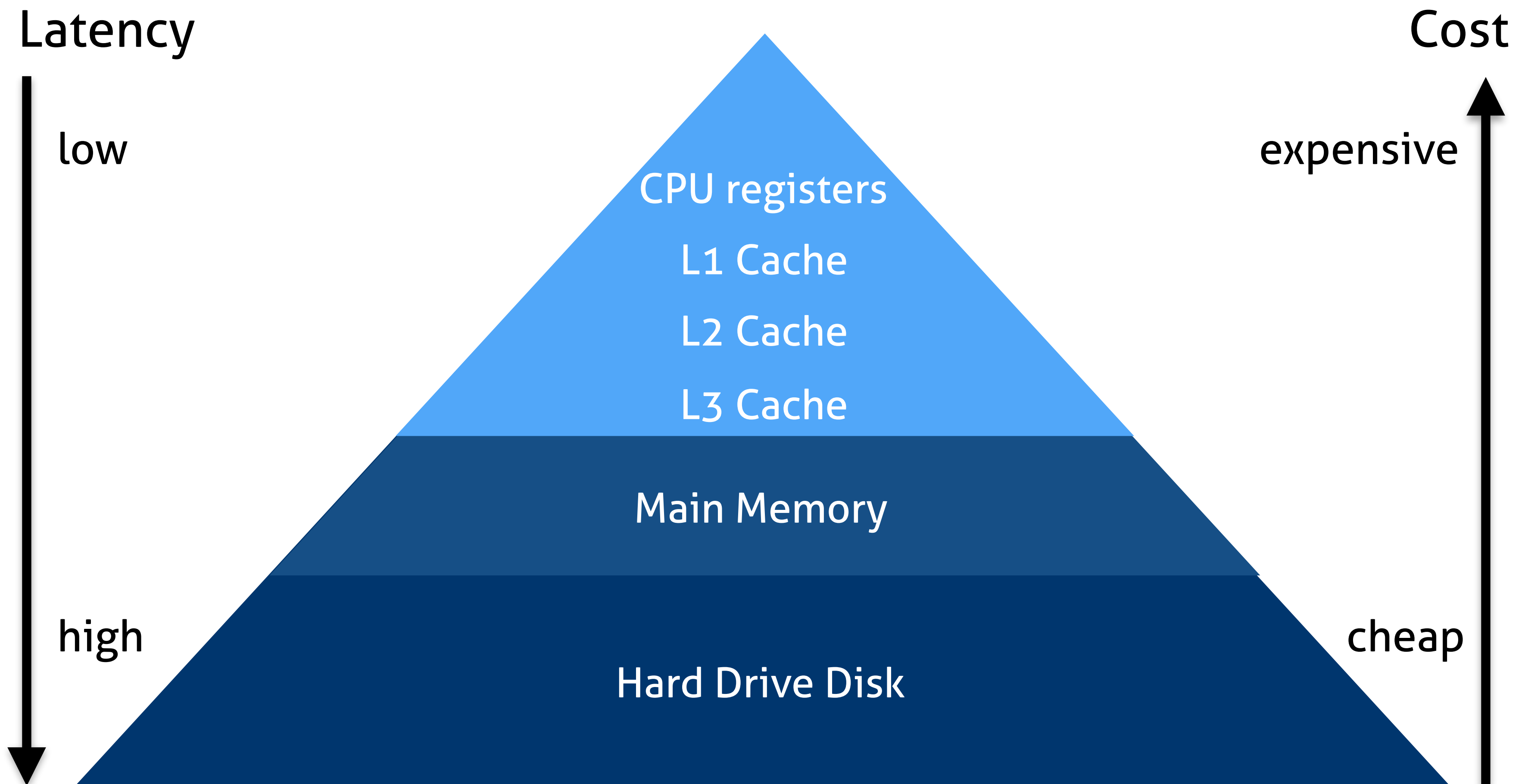
Main Memory

- today, systems can feature up to ~ 10-20 TB of RAM
- prices ($/GB) are dropping fast (2015: $4.37/GB RAM)



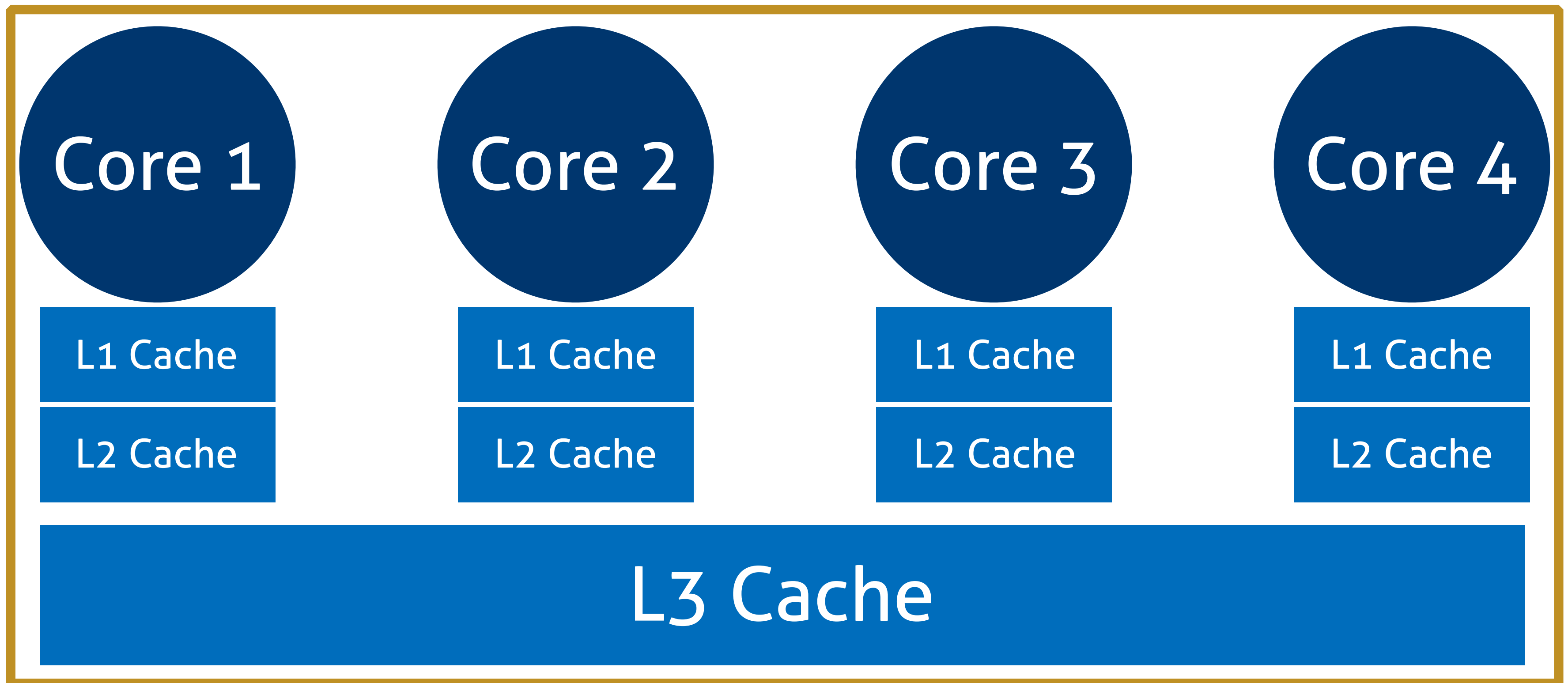
Data Source: http://www.statisticbrain.com/average-historic-price-of-ram/

Cache Hierarchy



Multi/Many-Core CPUs

Intel's Haswell architecture supports up to 18 cores per CPU.

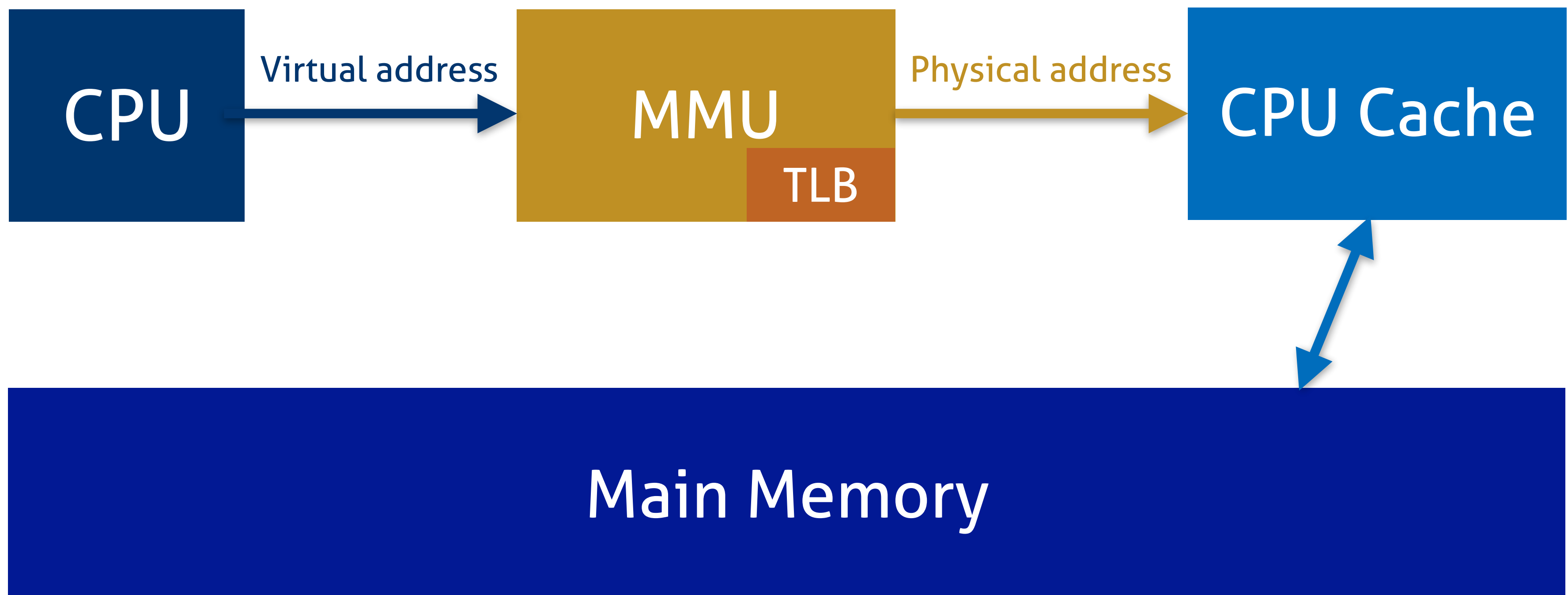


Many-Core CPUs: Intel Xeon Phi

- plenty of cores on one CPU (up to 72)
- rather low clock rates (~1-1.5 GhZ)
- no Level 3 Cache (only L1&L2)
- extra-wide SIMD registers: AVX-512
- available as standalone CPU
- competitive with modern GPUs while providing a conventional instruction set

Source: https://www.amazon.com/Intel-Xeon-Phi-7120P-Coprocessor/dp/B00FKG9R2Q

Reading data (from main memory)

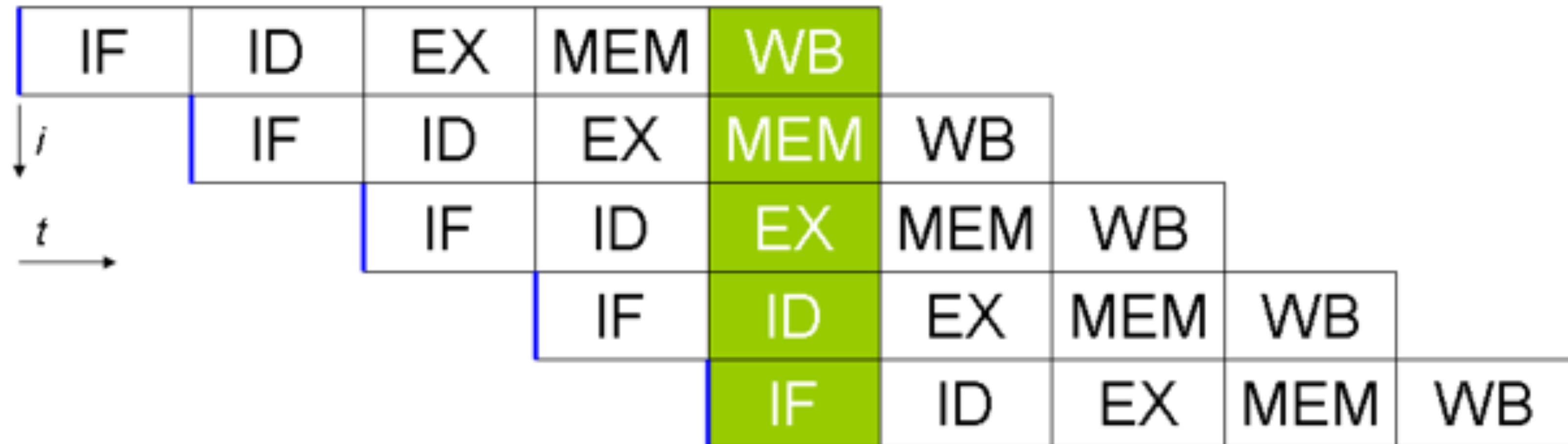


Expensive: TLB miss, CPU Cache misses

Cache Lines

- block in main memory
- basic unit for transfers from main memory to caches
- usually 64 bytes, e.g., 1 CL can hold 16 32-bit integers
- cache line utilization = portion of the transferred cache line that is actually used
- the successive cache line is usually prefetched, which is beneficial for sequential reads

Instruction Pipelining



Source: https://simple.wikipedia.org/wiki/Instruction_pipelining

- Instruction Fetch (IF)
- Instruction Decode (ID)
- Execute (EX)
- Memory Access (MEM)
- Register Write Back (WB)

Branch Prediction

- modern CPUs conduct pipelined execution
- for conditional branches, branches are predicted
- if the prediction was wrong, pipeline starts over with the correct branch, which is slow
- misprediction delays cost around 10-20 CPU cycles (depends on pipeline size)

```
for (int i = 0; i < n; i++)
  if (rand() % 2 == 0)
    do_something();
```

Good practices (performance-wise)

- read whole cache line
- access contiguous memory locations
  - decreases risk of cache & TLB misses
- avoid pointer chasing
- reduce conditional code
- use memory space efficiently

Example: Scanning a 2-dim array

Which variant is faster? Both have $O(n^2)$ complexity.

Variant 1:

```
int sum = 0;
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    sum += array[i][j] ;
```

Variant 2:

```
int sum = 0;
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    sum += array[j][i] ;
```

Answer: Variant 1: 0.33s Variant 2: 2.09s (for n=10k)

Example: Conditional Count

SELECT COUNT(*) FROM books WHERE genre = 3;

Variant 1 (with branch):

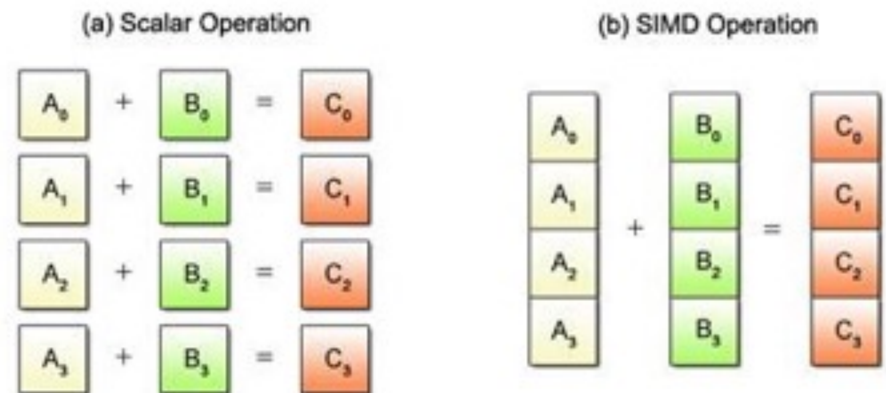
```
int count = 0;
for (int i = 0; i < n; i++)
  if (books_genre[i] == 3)
    count ++;
```

Variant 2 (branch-free):

```
int count = 0;
for (int i = 0; i < n; i++)
  count += (books_genre[i] == 3);
```

Single Instruction Multiple Data (SIMD)

- execute one instruction on multiple data elements
- extra-wide registers that hold multiple data elements
- degree of parallelism (DOP) is determined by SIMD register size S and data element size K: DOP = S/K
  - e.g., 256-bit SIMD register and 32-bit integers as operands allow 8 additions in parallel



Source: https://www.kernel.org/pub/linux/kernel/people/geoff/cell/ps3-linux-docs/CellProgrammingTutorial/BasicsOfSIMDProgramming.html

SIMD with Intrinsics

- executing a SIMD instruction using Intrinsics
  - 1.) load data into SIMD register
  - 2.) execute SIMD instruction
  - 3.) read result from SIMD register

SIMD segment 0 | SIMD segment 1 | SIMD segment 2 | SIMD segment 3

32-bit Integer | 32-bit Integer | 32-bit Integer | 32-bit Integer

128-bit SIMD register

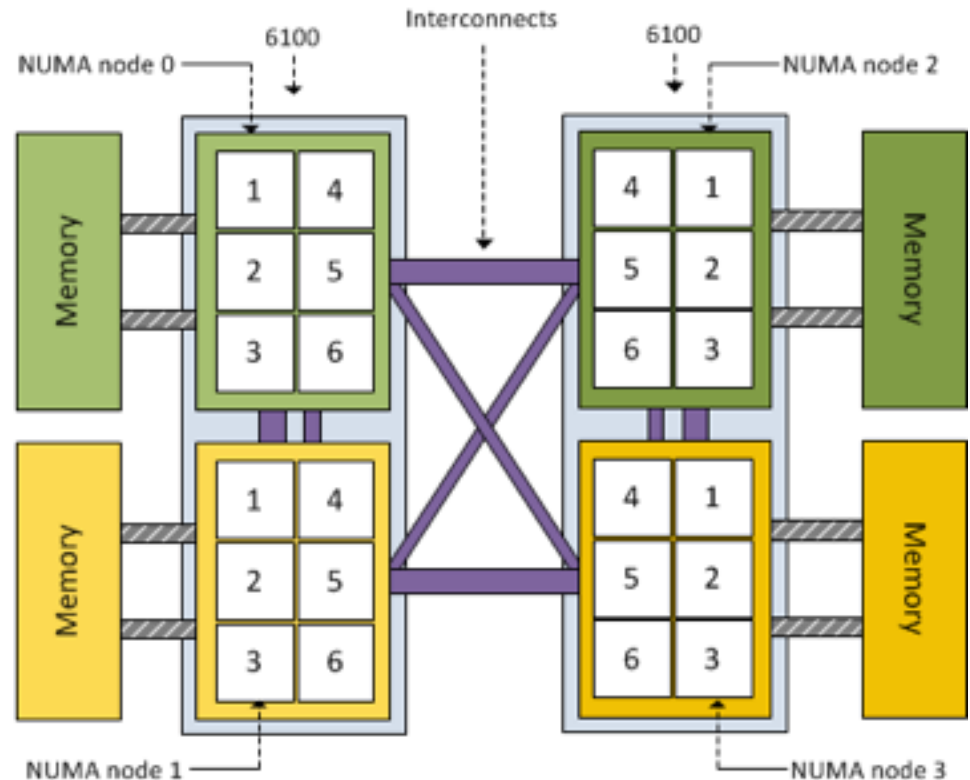
https://software.intel.com/sites/landingpage/IntrinsicsGuide/

Example: Linear Scan with SIMD

```
#include <immintrin.h>

__m256i s = _mm256_castsi256_ps(_mm256_set1_epi32(search_key));
__m256i keys, res;
int bitmask;
int match_pos = -1;

for (int i = 0; i < n; i += 8) {
    keys = _mm256_loadu_si256((__m256i const *) &array[i]);
    res = _mm256_cmpeq_epi32(keys, s);
    bitmask = _mm256_movemask_epi8(res);
    if (bitmask < 0xff) {
        match_pos = (i * 8) + pop_count(bitmask); break;
    }
}

if (match_pos >= 0) { /* search key found */ }
```

Non-Uniform Memory Access (NUMA)

- divide CPUs/cores into NUMA nodes
- each NUMA node is assigned to a certain memory partition ("local memory")
- access to local memory is faster than access to remote memory

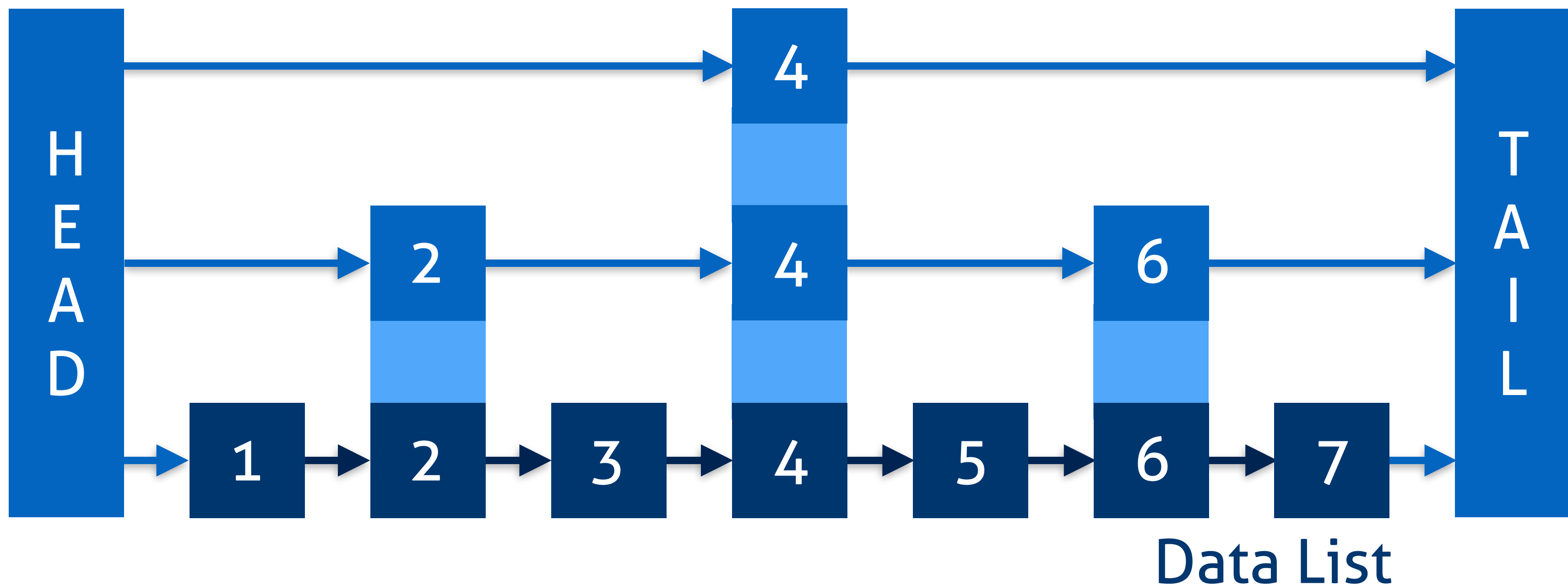


Source: http://frankdenneman.nl/2011/01/05/amd-magny-cours-and-esx/

Content of this Lecture

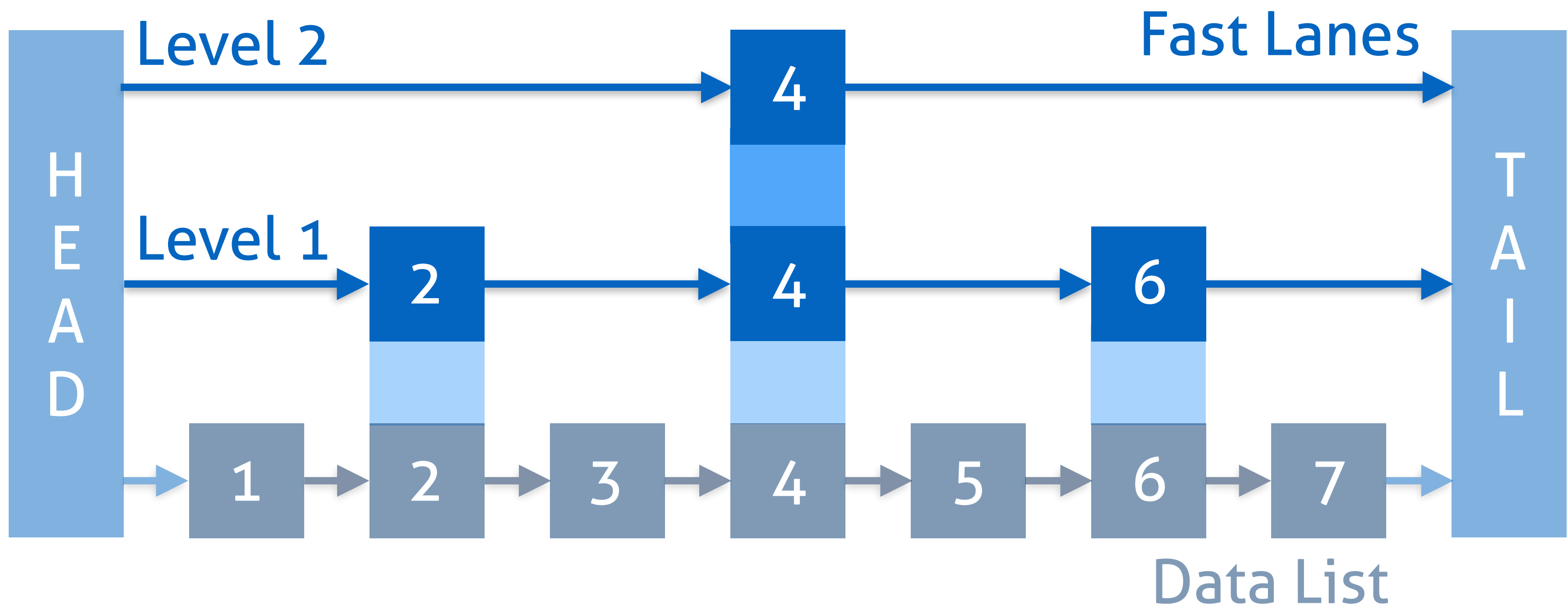
- Introduction to Modern Hardware
  - CPUs, Cache Hierarchy
  - Branch Prediction
  - SIMD
  - NUMA
- Cache-Sensitive Skip List
  - Skip Lists
  - Our Contributions
  - Evaluation

Skip Lists: Structure



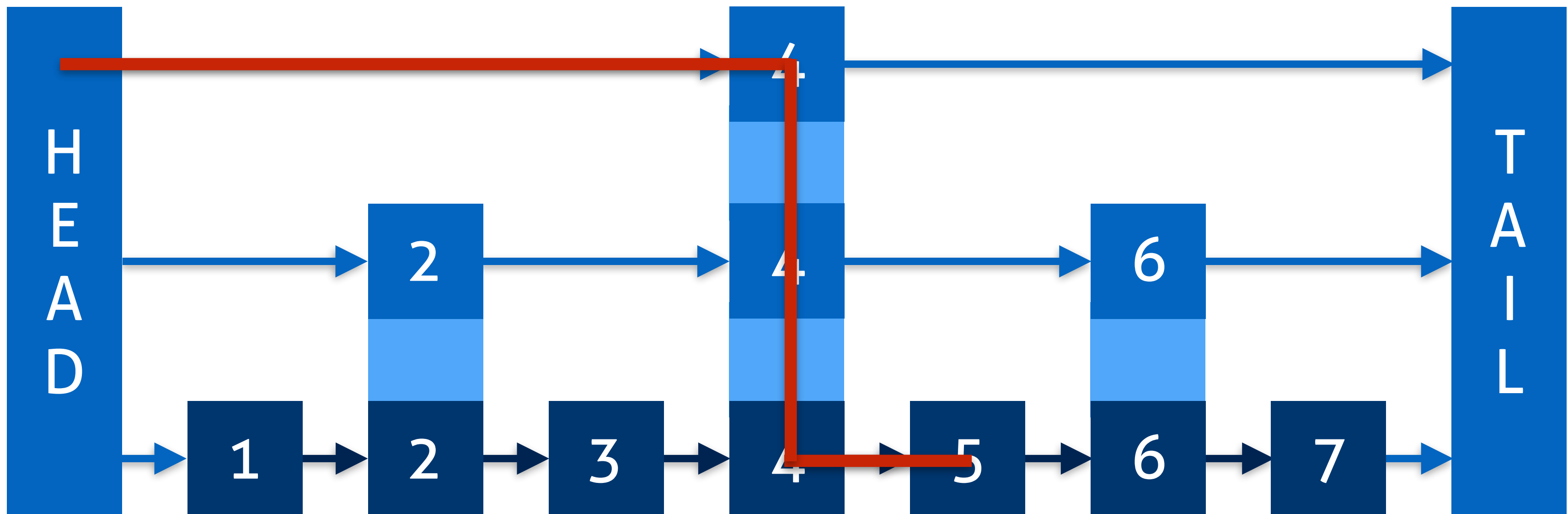
W. Pugh: "Skip lists: a probabilistic alternative to balanced trees" (1990)

Skip Lists: Fast Lanes



Fast lane i contains elements of fast lane i-1 with probability p.
In this case, p = 1/2.

Skip Lists: Searching

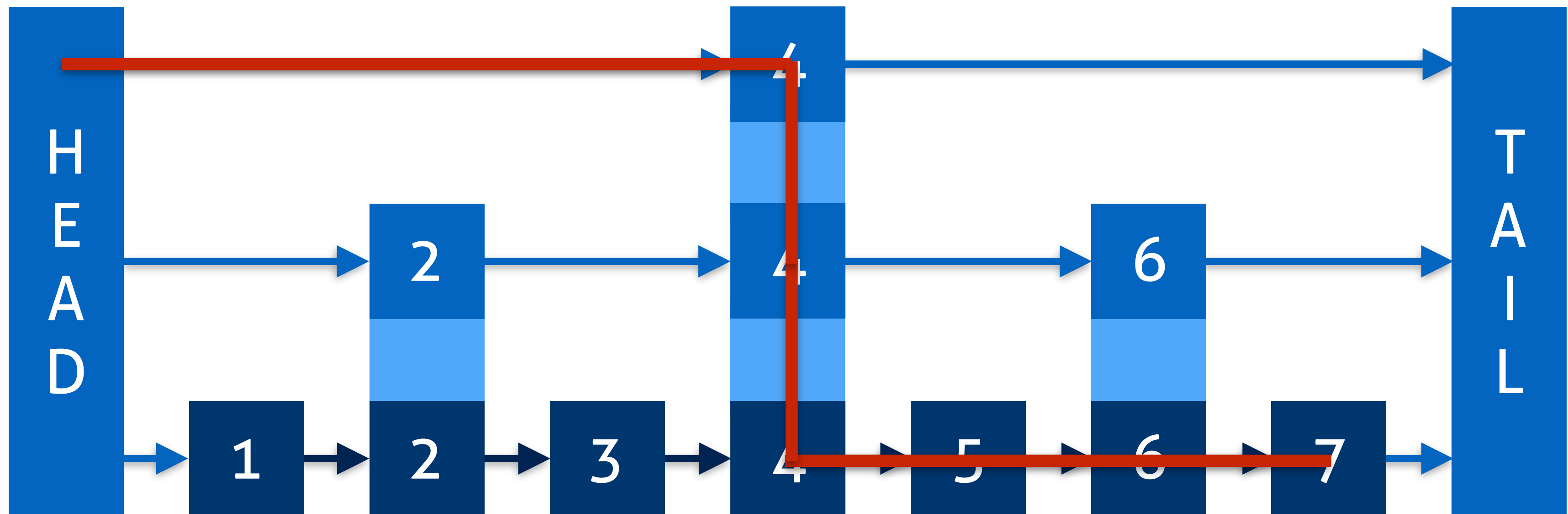


Skip Lists: Inserting

1. search corresponding position in data list that may hold the inserted key
2. insert key into data list
3. starting at the lowest fast lane, test for each fast lane if new key should be inserted into the fast lane;
   if p=1/2, this decision can be done by "flipping a coin"
4. abort as soon as the first coin flip "fails", i.e., the key is not inserted in the current fast lane

Compared to B-trees, skip lists don't require node splitting :-)

Skip Lists: Range Queries



Skip Lists

- probabilistic data structure
- logarithmic search access/insertions
- keys are stored in sorted order in linked list (“data list”)
- hierarchy of subsequences that skip over elements of lower levels (“fast lanes”)
- simpler implementation than B-trees (no node splits)

W. Pugh: “Skip lists: a probabilistic alternative to balanced trees” (1990)

Why do we use Skip Lists as base?

- originally built for main memory
- not aligned to disk blocks as in the case of B^+-tree nodes
- provide lookups and range queries
- (fast lane) keys are accessed sequentially
- structure of Skip Lists may be very beneficial w.r.t. “modern hardware”, most implementations are not

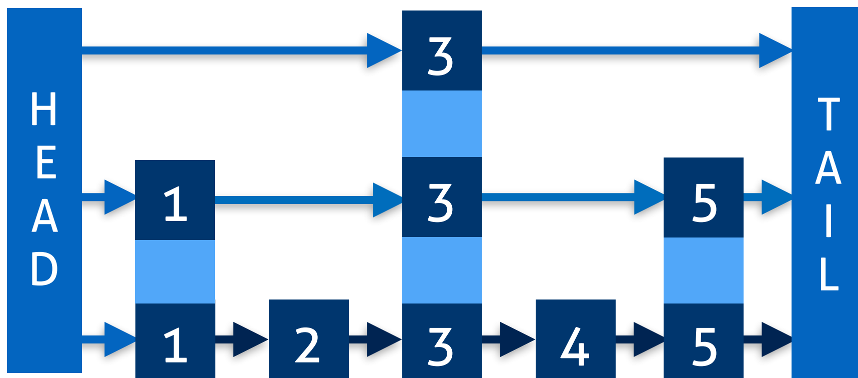
Deterministic Skip List

- deterministic variant of the probabilistic skip list
- number of elements each fast lane skips over is fixed
- fast lane i skips over 1/p elements of fast lane i-1
- provides a predictable memory layout
- implementation of insertions may be a bit more complex because you must always ensure that the fixed structure of the skip list is not violated

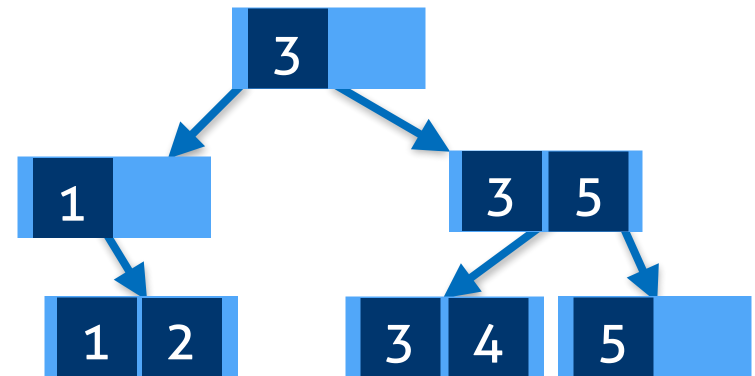
J.I. Munro et al.: "Deterministic skip lists" (1992)

Skip Lists vs. B^+-trees

The deterministic variant of a skip list can easily be transformed into a B^+-tree.



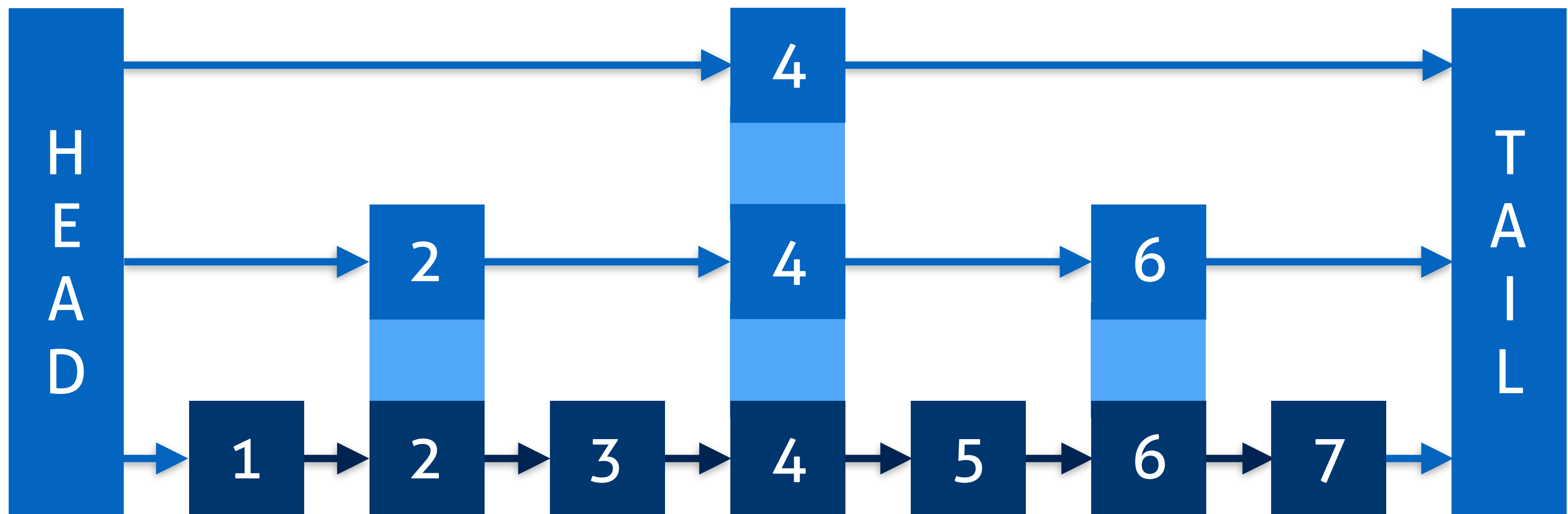
Deterministic Skip List



B^+-tree

M. G. Amoureux and B. G. Nickerson: "On the equivalence of B-trees and deterministic skip lists" (1996)

Issues of the conventional Skip List



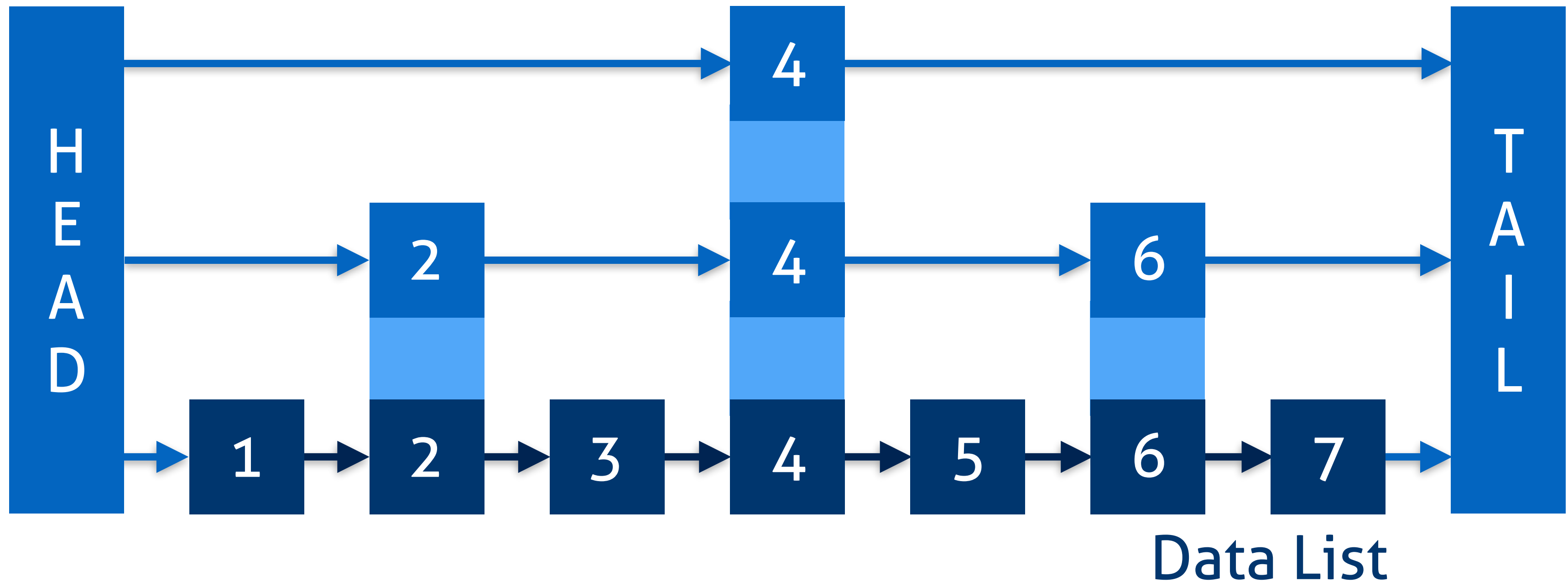
- heavy pointer chasing
- poor cache line utilisation
- ineffective prefetching
- no SIMD support

Cache-Sensitive Skip List (CSSL)

- based on the concepts behind skip lists
- improved memory layout to be more CPU-friendly
- sequential access of fast lane elements
  - improved cache line utilization
  - prefetching works better
  - less cache & TLB misses
- exploits SIMD instructions for searching
- implementation avoids branches if possible

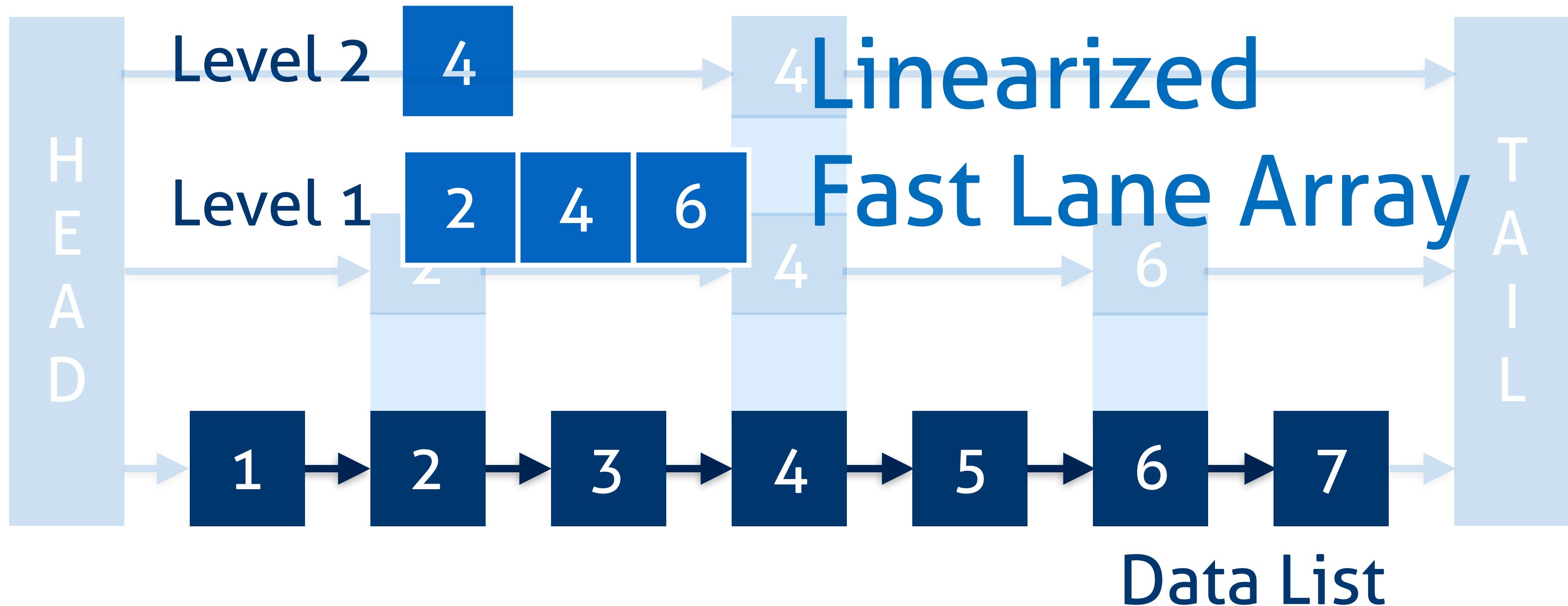
Stefan Sprenger, Steffen Zeuch, Ulf Leser:
"Cache-Sensitive Skip List: Efficient Range Queries on modern CPUs",
ADMS/IMDM @ VLDB, New Delhi, India, September 2016

CSSL is based on Skip Lists



We use the deterministic variant due to its predictable memory layout.

Linearized Fast Lanes

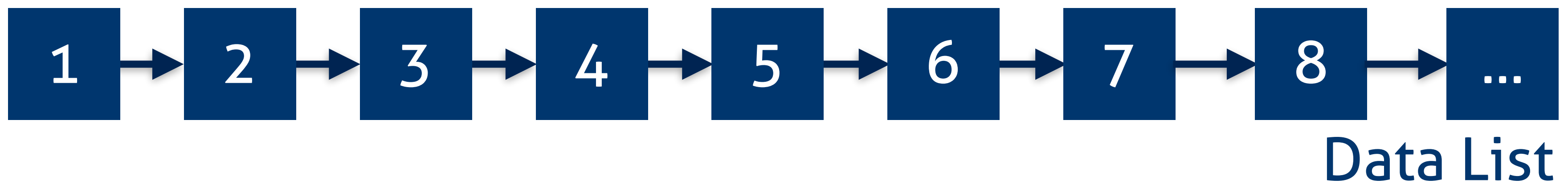


All fast lanes are stored in one dense array that is tailored to cache lines.

CSSL indexing 1..64 (p=1/2)

	0	1	2	3	4	5	6	7			
Level 3	1	9	17	25	33	41	49	57			
	8	9	10	11	12	13	14	15	16	17	
Level 2	1	5	9	13	17	21	25	29	33	37	...
	24	25	26	27	28	29	30	31	32	33	
Level 1	1	3	5	7	9	11	13	15	17	19	...

Linearized Fast Lane Array



Data List

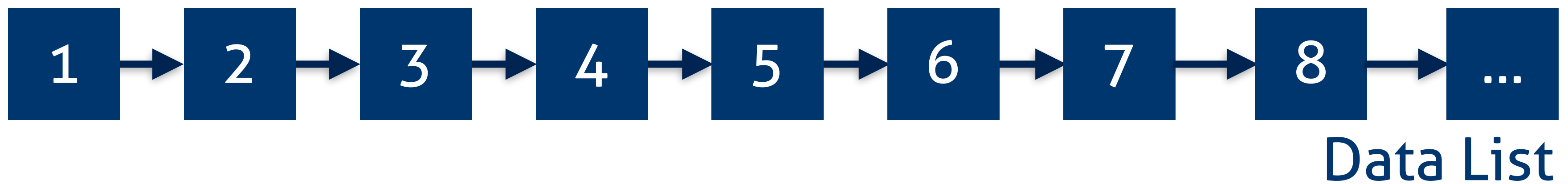
Cache Line Alignment

	0	1	2	3	4	5	6	7			
Level 3	1	9	17	25	33	41	49	57			

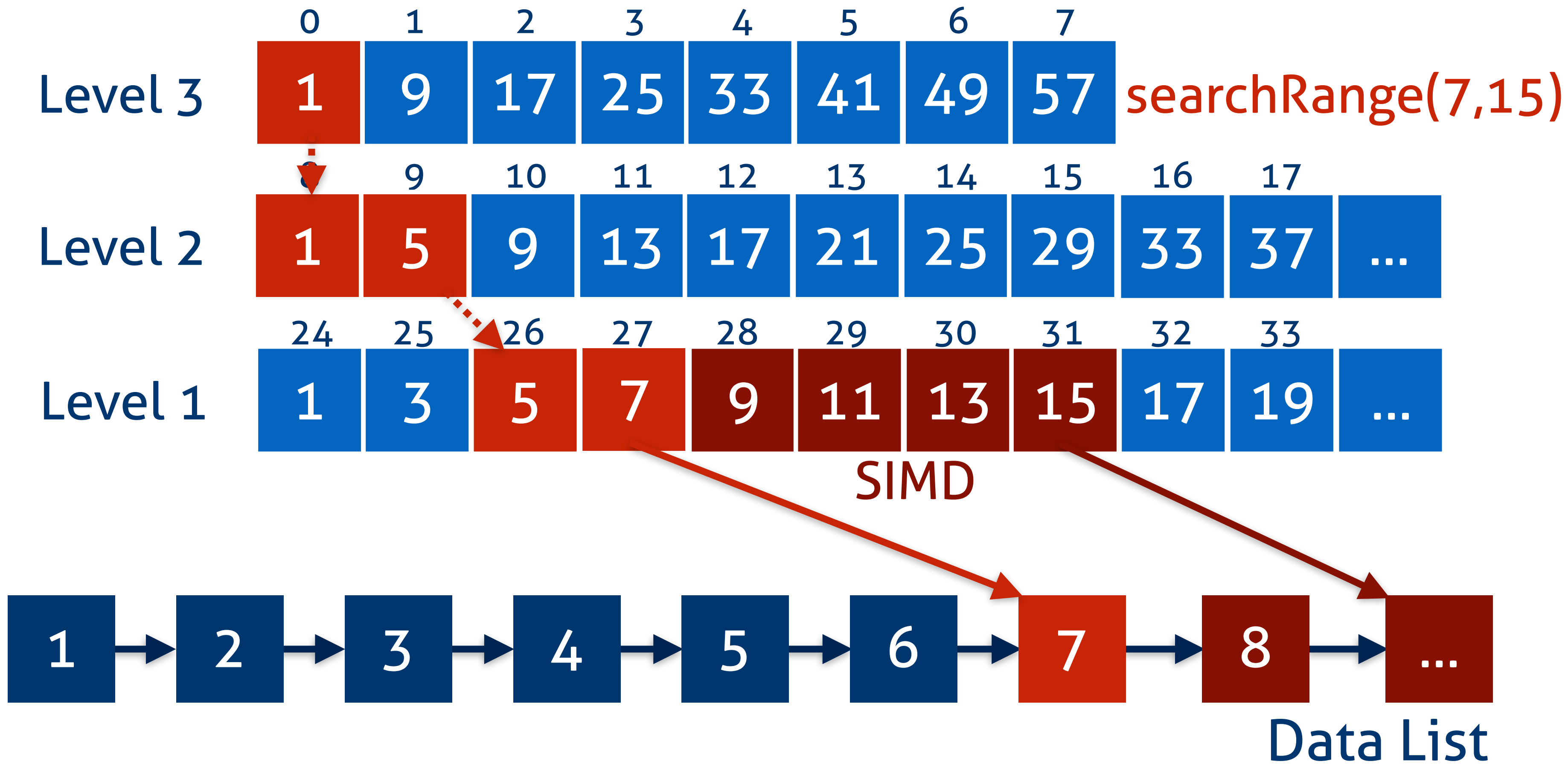
	8	9	10	11	12	13	14	15	16	17	
Level 2	1	5	9	13	17	21	25	29	33	37	...

	24	25	26	27	28	29	30	31	32	33	
Level 1	1	3	5	7	9	11	13	15	17	19	...

Linearized Fast Lane Array



Searching with SIMD



Branch-free evaluation of SIMD results

00000001 -> match at first position
00000011 -> match at second position

```
bitmask = _mm256_movemask_ps(result);
```

search key			
13	>=	9	yes
13	>=	13	yes
13	>=	15	no
13	>=	19	no
13	>=	24	no
13	>=	27	no
13	>=	32	no
13	>=	42	no

Branch-free evaluation of SIMD results

00000001 -> match at first position
00000011 -> match at second position

```
bitmask = _mm256_movemask_ps(result);
```

branches

```
if (bitmask == 0x1) {
   pos = 1;
} else if (bitmask == 0x3) {
   pos = 2;
} else if (bitmask == 0x7) {
   pos = 3;
} ....
```

branch-free

```
pos = __builtin_popcount(bitmask);
```

popcount returns number of set bits and is implemented branch-free (GCC Intrinsic)

Evaluation

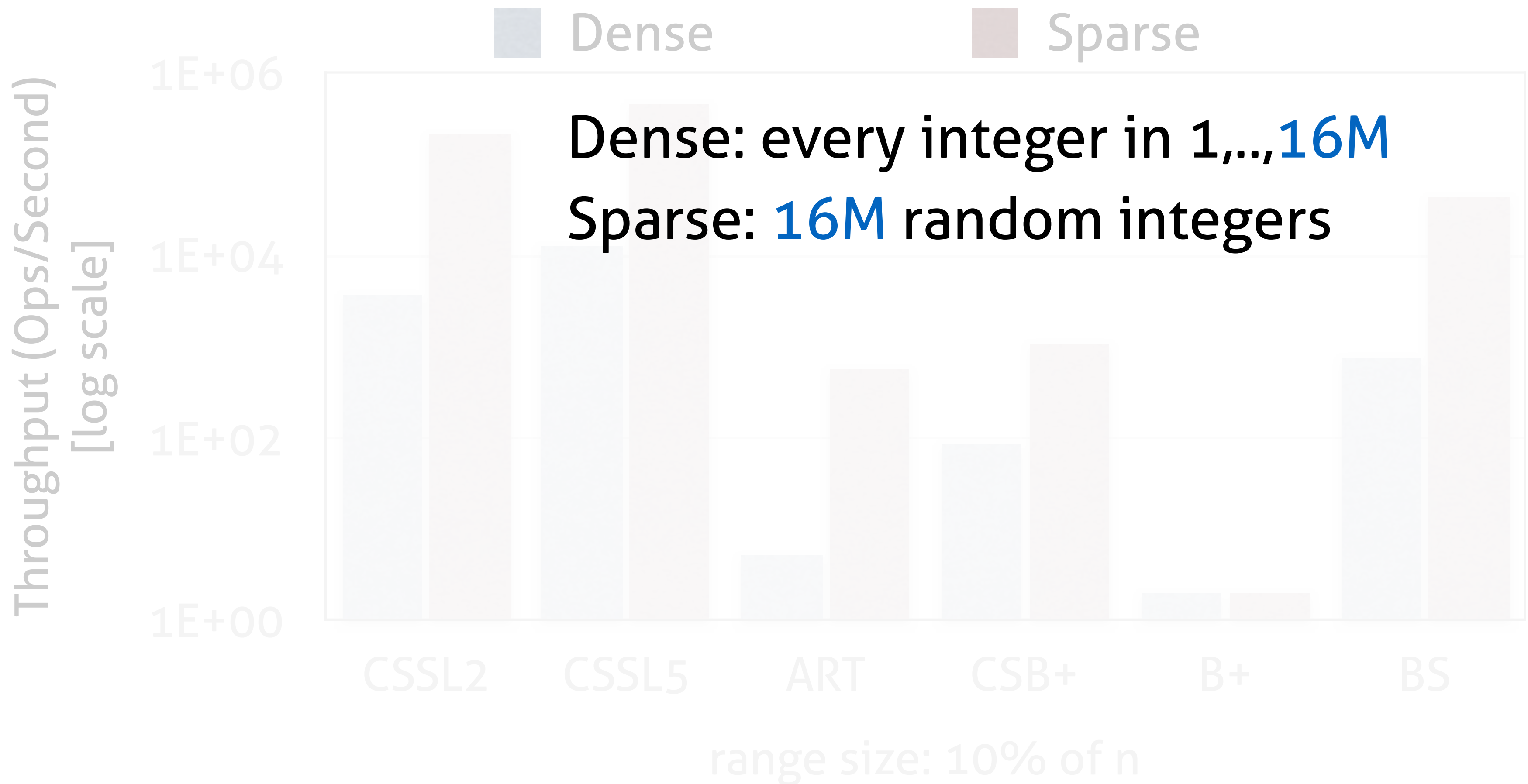
- range queries on 16M and 256M keys
- range queries on real-world data
- lookups on 16M keys
- mixed key/range workload
- space consumption

All experiments are conducted single-threaded on a Intel Xeon E5-2620 CPU (2 GHz) and 32 GB RAM.

Competitors

- Cache-Sensitive Skip List (CSSL)
  - CSSL2: p=1/2
  - CSSL5: p=1/5
- adaptive radix tree (ART)
- Cache-Sensitive B+-tree (CSB+)
- binary search on a static array (BS)
- B+-tree (B+)

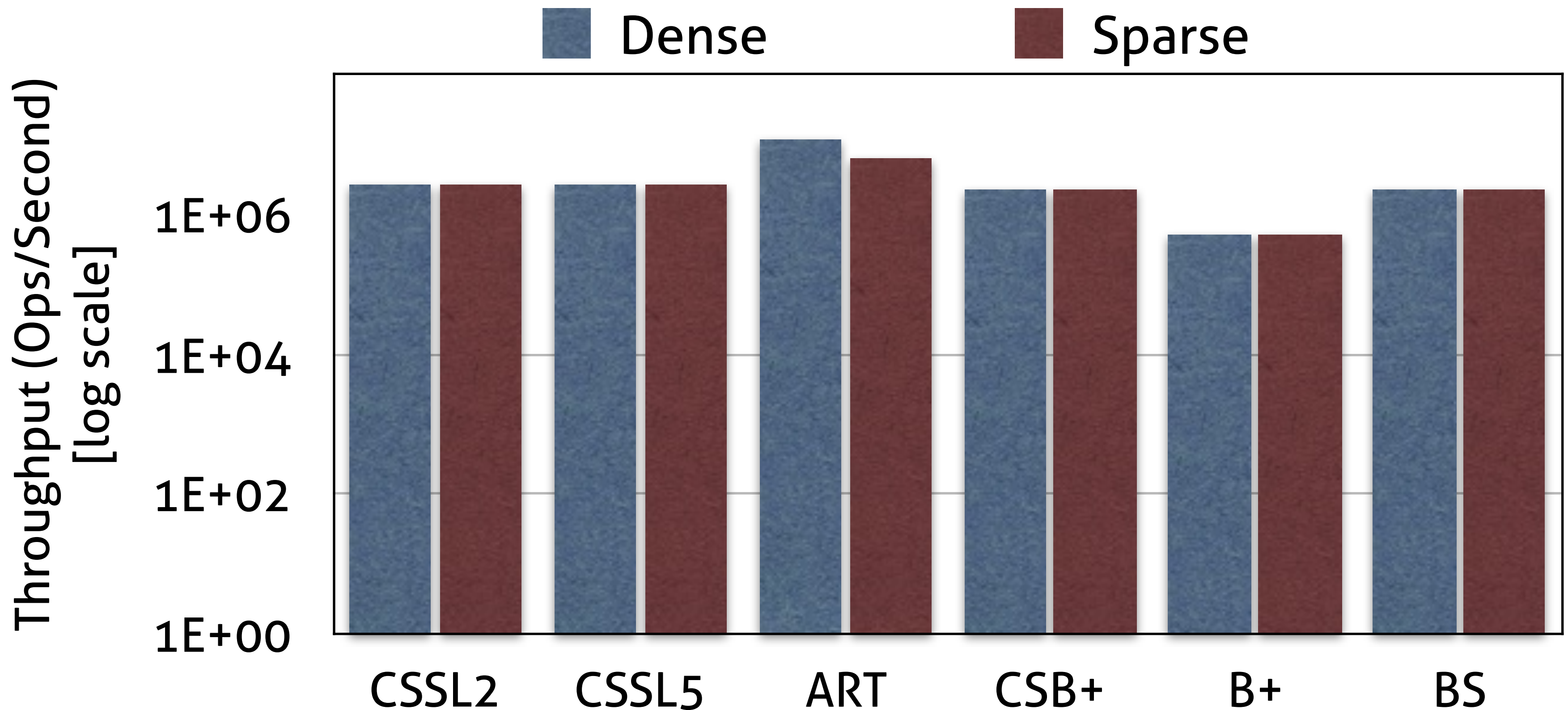
Range Queries on 16M Integer Keys



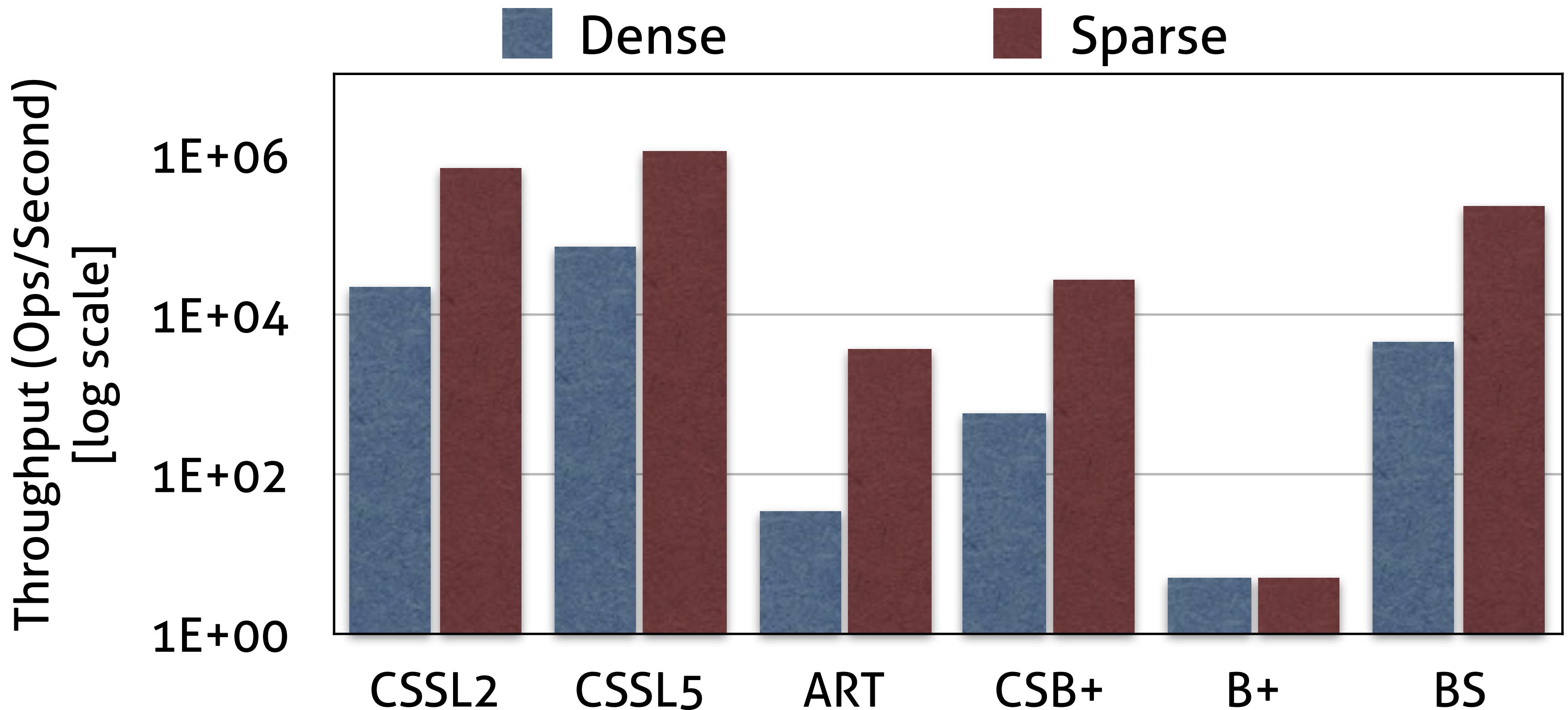
Performance Counters per Range Query

	CSSL2	CSSL5	ART	CSB+	B+	BS
L3 Cache Hits	373	139	14k	364	1.8k	325
L3 Cache Misses	165	23	28k	5.7k	7.4M	278
TLB Misses	5	3	19k	958	369k	10
Branch Mispredictions	16	13	16k	4.6k	832	13

Lookups on 16M Integer Keys



Mixed key-range Workload (50%-50%)



1M queries (500k lookups, 500k range queries) on 16M integer keys.

Evaluation Results

- CSSL provides very efficient range query implementation by tuning to the architecture of modern CPUs
- Impacts of our optimization are reflected in CPU performance counters
- Even for mixed workloads, CSSL benefits from efficient range queries

Summary

- introduction to Modern Hardware
- optimizing for main-memory and modern CPUs
- vectorized SIMD instructions
- Skip Lists as alternative to B-trees
- Cache-Sensitive Skip List, our main-memory variant of Skip Lists
- evaluation results

Questions?

- Introduction to Modern Hardware
  - CPUs, Cache Hierarchy
  - Branch Prediction
  - SIMD
  - NUMA
- Cache-Sensitive Skip List
  - Skip Lists
  - Our Contributions
  - Evaluation