• Modern Hardware
  – CPUs, Cache Hierarchy
  – Branch Prediction
  – SIMD

• Cache-Sensitive Skip List (CCSL)
Modern Hardware

- CPUs with multiple cache levels
- Many-core CPUs
- **Large main memory**
  - Large and cheap enough to hold entire databases
  - Performance bottleneck moves up from disk/memory to memory / CPU caches
  - New goal: minimize cache misses
- **Pipelining** and branch prediction
- **SIMD**
- Specialized co-processors: GPUs, FPGAs
Many-Core CPUs

- **Current systems**
  - Up to 18 cores per CPU (normal: 4)
  - 4-8 CPUs per server + hyperthreading
  - 32 – 244 threads

- **Sizes**
  - L1: 8KB – 512KB
    - Split data / instruction
  - L2: 512KB – 4MB
  - L3: 4MB – 128MB
Expensive Operations

- **TLB misses** (Translocation lookaside buffer)
  - Need to compute physical page ID in Memory Management Unit
- **Cache misses**
  - Need to fetch data from main memory into cache hierarchy
Cache Lines

• **Basic unit** for transfers from main memory to caches
  – Also memory is a “block” device
• Usually 64 bytes, enough for 16 32-bit integers
• Cache line utilization = fraction of the transferred cache line that is actually used
  – Goal is **maximizing utilization** – load nothing that you don’t need
• Successive cache line is usually prefetched
  – Use **sequential reads** where possible
Example

Variant 1:

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

Variant 2:

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

- Variant 1: **0.33sec**  
  - Variant 2: **2.09sec**
- Test with n=1E4
  - Note: In this simple case, GCC will most likely recognize the issue and reorder automatically
  - Depending on optimizer level
One more Trick: Branch Prediction

- Modern CPUs conduct **pipelined execution**
  - Every CPU command consist of 3-5 parts
  - Pipelined execution: Parallelize at the part level
- Requires knowing the next command
- For **conditional branches**, branches are predicted
- If prediction was wrong, **pipeline starts over** with the correct branch
  - Very expensive: Around 10-20 CPU cycles
- **Avoid conditionals** wherever possible
Example

```sql
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);
```

- You need to know which **CPU commands** emerge from your programming language constructs
Single Instruction Multiple Data (SIMD)

- Special command set in current CPUs
- Execute one instruction on multiple data elements
- Data must first be stored in special, extra-wide registers
- **Degree of parallelism** (DOP) determined by SIMD register size $S$ and data element size $K$: $DOP = S/K$

**Example**
- 256-bit SIMD register
- 32-bit integers
- Allows 8 additions in parallel

**Also**: Vectorized execution
• Introduction to Modern Hardware
• **Cache-Sensitive Skip List (CCSL)**
  – Skip Lists
  – CCSL
  – Evaluation
Skip-Lists

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

- **Static skip lists**: Every 2\textsuperscript{nd}/4\textsuperscript{th}/8\textsuperscript{th} element has skip pointer
  - Disadvantage: Costly inserts / deletions
- **Probabilistic skip lists**: Fast lane i contains elements of fast lane i-1 with probability p
  - Example corresponds roughly to p = \( \frac{1}{2} \)
  - Same average-case search complexity, faster ins/dels
Searching

- WC / AC complexity: $O(\log(n))$
- Also range queries are supported well
  - Search first element, scan to last element
- Often used as alternative to tree-like structures in memory
Skip Lists on Modern Hardware

- Heavy pointer chasing
- Poor cache line utilization
- Ineffective prefetching
- Unsure branch predictions
- No SIMD support
Cache-Sensitive Skip List (CSSL)

- Main-memory optimized variant of deterministic skip lists
- CPU-friendly memory layout
- Sequential access of fast lane elements
- Improved cache line utilization & prefetching
- Less cache & TLB misses
- SIMD instructions for searching
- Implementation avoids branches if possible

Linearized Storage

- No pointers, few TLB misses
- Good cache utilization & prefetching
- Searching within each lane with bin-search / seq. scans
  - Or SIMD – next slide
- Data list only retrieved for payload access
Searching with SIMD

- Load max number of values into SIMD register
  - Consider STRIDE in recent SIMD
- Compare all at once to search key(s) in one cycle
- Scans “hop” through lanes
Systems Compared

- CSSL2: $p=1/2$
- CSSL5: $p=1/5$
- **Adaptive radix tree** (ART)
  - Prefix-Tree with mem-optimized, adaptive inner nodes
  - Small memory footprint, SOTA for key-searches
- Cache-Sensitive B+-tree (CSB+)
- Binary search on a static array (BS)
  - Essentially **immutable**
- B+-tree (B+) in main memory
- Data
  - Dense: All integers from 1 to $n$
  - Sparse: $n$ random integers
Evaluation – Range Queries

- Note the log-scale
- CCSL faster than all competitors in all configurations
- B+ (not MM-optimized) suffers a lot
- ART not suitable for range queries
  - Pointer-chasing
# Explanation by CPU Performance Counters

<table>
<thead>
<tr>
<th></th>
<th>CSSL2</th>
<th>CSSL5</th>
<th>ART</th>
<th>CSB+</th>
<th>B+</th>
<th>BS</th>
</tr>
</thead>
<tbody>
<tr>
<td>L3 Cache Hits</td>
<td>373</td>
<td>139</td>
<td>14k</td>
<td>364</td>
<td>1.8k</td>
<td>325</td>
</tr>
<tr>
<td>L3 Cache Misses</td>
<td>165</td>
<td>23</td>
<td>28k</td>
<td>5.7k</td>
<td>7.4M</td>
<td>278</td>
</tr>
<tr>
<td>TLB Misses</td>
<td>5</td>
<td>3</td>
<td>19k</td>
<td>958</td>
<td>369k</td>
<td>10</td>
</tr>
<tr>
<td>Branch Mispredictions</td>
<td>16</td>
<td>13</td>
<td>16k</td>
<td>4.6k</td>
<td>832</td>
<td>13</td>
</tr>
</tbody>
</table>
Evaluation – Single Key Lookups

- Lookups: Selectivity of 1/n
- CCSL: Slower only than ART
- Research prototypes
  - We never implemented updates
  - We never implemented payloads