Optimizing database architecture for machine architecture

Peter Boncz
boncz@cwi.nl
CPU Architecture

Elements:

- **Storage**
  - CPU caches L1/L2/L3
- **Registers**
- **Execution Unit(s)**
  - Pipelined
  - SIMD
CPU Metrics

Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)

Haswell
2013
8MB L3 cache
4 core (8SMT)
3.5GHz (3.9 turbo)
8-way pipelines
256bits SIMD
SIMD scatter/gather
Transactional memory
Super-Scalar Execution (pipelining)

- speculative + **out-of-order** execution
- use instructions further on to fill the pipelines
- >120 in-flight instructions by now
SIMD

- Single Instruction Multiple Data
  - Same operation applied on a vector of values
  - MMX: 64 bits, SSE: 128 bits, AVX: 256 bits
  - SSE, e.g. multiply 8 short integers
# Hazards

- **Data hazards**
  - Dependencies between instructions
  - L1 data cache misses

- **Control Hazards**
  - Branch mispredictions
  - Computed branches (late binding)
  - L1 instruction cache misses

Result: bubbles in the pipeline

<table>
<thead>
<tr>
<th>Instruction fetch</th>
<th>IF-1</th>
<th>IF-2</th>
<th>IF-3</th>
<th>IF-4</th>
<th>IF-5</th>
<th>IF-6</th>
<th>IF-7</th>
<th>IF-8</th>
<th>IF-9</th>
<th>IF-7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction decode</th>
<th>ID-1</th>
<th>ID-2</th>
<th>ID-3</th>
<th>ID-4</th>
<th>ID-5</th>
<th>ID-6</th>
<th>ID-7</th>
<th>ID-8</th>
<th>ID-7</th>
<th>ID-7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execute</th>
<th>EX-1</th>
<th>EX-2</th>
<th>EX-3</th>
<th>EX-4</th>
<th>EX-5</th>
<th>EX-6</th>
<th>EX-7</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write back</th>
<th>WB-1</th>
<th>WB-2</th>
<th>WB-3</th>
<th>WB-4</th>
<th>WB-5</th>
<th>WB-6</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Flushed instructions

Out-of-order execution addresses data hazards
- control hazards typically more expensive
Multi-Core: sustaining Moore's law

Source: Webinar by Dr. Tim Mattson, Intel Corp.
Non-uniform Memory Access (NUMA)

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,32)</td>
<td>(4,36)</td>
</tr>
<tr>
<td>Core 2</td>
<td>Core 3</td>
</tr>
<tr>
<td>(8,40)</td>
<td>(12,44)</td>
</tr>
<tr>
<td>Core 4</td>
<td>Core 5</td>
</tr>
<tr>
<td>(16,48)</td>
<td>(20,52)</td>
</tr>
<tr>
<td>Core 6</td>
<td>Core 7</td>
</tr>
<tr>
<td>(24,56)</td>
<td>(28,60)</td>
</tr>
<tr>
<td>24MB L3</td>
<td></td>
</tr>
</tbody>
</table>

HyperThread 12
HyperThread 44
L1 I 32KB | L1 D 32KB
L2 256KB

256 GB

CPU 0

CPU 1

CPU 2

CPU 3

QPI

256 GB

256 GB

256 GB

256 GB
DRAM Metrics

- Latency improvements lag bandwidth and size
Micro-Benchmark

• for(j=i=0; i<n; i++)  // CHASE
  j = table[j];

vs

• for(i=0; i<n; i++)  // FETCH
  result[i] = table[input[i]];
Memory Access Cost Model

TLB coverage:
TLB1: 64 entry, 4KB pages ➜ covers 256KB (2cyc)
TLB2: 1024 entry, 4KB pages ➜ covers 4MB (10cyc)

Memory Hierarchy:
L1: 16KB = 2cyc
L2: 2MB = 15cyc
L3: 8MB = 25cyc
RAM: 512GB = 200cyc

Cache misses due to TLB handling (page table cache misses -- PT)
- 8MB experiment ➜ 2048 pages occupy 16KB page table L1
- 1GB experiment ➜ 256K pages occupy 2MB page table L2
- 4GB experiment ➜ 1M pages occupy 8MB page table L3
- 8GB experiment ➜ 2M pages occupy 16MB page table L3 and 16KB page table L1

Predicted behavior (MEM + TB caused)
0-16KB: L1^{MEM} = 2
16KB-256KB: L2^{MEM} = 15
256KB-2MB: L2^{MEM} + TLB1^{MEM} = 17
2MB-4MB: L3^{MEM} + TLB1^{MEM} = 27
4MB-8MB: L3^{MEM} + TLB2^{MEM} + L1^{PT} = 37
8MB-1GB: RAM^{MEM} + TLB2^{MEM} + L2^{PT} = 225
1GB-4GB: RAM^{MEM} + TLB2^{MEM} + L3^{PT} = 235
4GB-8GB: RAM^{MEM} + TLB2^{MEM} + RAM^{PT} = 410
8GB+: RAM^{MEM} + TLB2^{MEM} + RAM^{PT} + L1^{PT} = 412
Micro-Benchmark Results
Out-of-order + Parallel Memory Access

// CHASE
j = table[j]; \( \rightarrow \) wait for j

vs

// FETCH
result[i] = table[input[i]]; i++; (i<n) \( \rightarrow \) predict true
result[i] = table[input[i]]; i++; (i<n) \( \rightarrow \) predict true
result[i] = table[input[i]]; i++; (i<n) \( \rightarrow \) predict true
result[i] = table[input[i]]; i++; (i<n) \( \rightarrow \) predict true
result[i] = table[input[i]]; i++; (i<n) \( \rightarrow \) predict true

<mem req buffer full> \( \rightarrow \) wait
Typical Relational DBMS Engine

Query

```
SELECT name, salary*.19 AS tax
FROM employee
WHERE age > 25
```
Typical Relational DBMS Engine

Operators

- Iterator interface
  - open()
  - next(): tuple
  - close()
Database Architecture causes Hazards

DB workload execution on a modern computer

“DBMSs On A Modern Processor: Where Does Time Go?”
Ailamaki, DeWitt, Hill, Wood, VLDB’99
Optimizing database architecture for machine architecture

Vectorwise case
TPC-H 1GB, query 1

- selects 98% of fact table, computes net prices and aggregates all

- Results:
  - C program: ?
  - MySQL: 26.2s
  - DBMS “X”: 28.1s

“MonetDB/X100: Hyper-Pipelining Query Execution” Boncz, Zukowski, Nes, CIDR’05
DBMS Computational Efficiency

TPC-H 1GB, query 1
• selects 98% of fact table, computes net prices and aggregates all
• Results:
  – C program: 0.2s
  – MySQL: 26.2s
  – DBMS “X”: 28.1s

“MonetDB/X100: Hyper-Pipelining Query Execution ” Boncz, Zukowski, Nes, CIDR’05
Typical Relational DBMS Engine

Operators

- Iterator interface
- `open()`
- `next(): tuple`
- `close()`
Typical Relational DBMS Engine

Primitives

Provide computational functionality

All arithmetic allowed in expressions, e.g. multiplication

\[ \text{mult}(\text{int}, \text{int}) \Rightarrow \text{int} \]
“Vectorized Execution”
“Vectors”

Vector contains data of *multiple* tuples (~100)

All primitives are “vectorized”

Effect: much less Iterator.next() and primitive calls.
“Vectors”

Column slices to represent in-flow data

**NOT:**
Vertical is a better table storage layout than horizontal (though we still think it often is)

**RATIONALE:**
- Simple array operations are well-supported by compilers
- No record layout complexities
- SIMD friendly layout
- Assumed cache-resident
Vectorized Primitives

```c
int select_lt_int_col_int_val ( int *res, int *col, int val, int n) {
    for(int j=i=0; i<n; i++)
        if (col[i] < val) res[j++] = i;
    return j;
}
```

Many primitives take just 1-6 cycles per tuple

10-100x faster than Tuple-at-a-time
Selection

map_mul_flt_val_flt_col(float *res, int* sel, float val, float *col, int n)
{
    for(int i=0; i<n; i++)
        res[i] = val * col[sel[i]];  
}

selection vectors used to reduce vector copying

contain selected positions
Memory Hierarchy

**Vectorwise query engine**

- **CPU cache**
- **RAM**
- ColumnBM (buffer manager)
- (raid) Disk(s)

**Diagram Details**

- CPU Cache
  - Small, Fast, Expensive
  - ~10 GB/s, 2–20 cycles
- Main Memory
  - Large, Slow, Cheap
  - 2–3 GB/s, 150–250 cycles
  - 40–400 MB/s, millions of cycles
Memory Hierarchy

Vectors are only the in-cache representation

RAM & disk representation might actually be different

(we use both PAX and DSM)
Optimal Vector size?

All vectors together should fit the CPU cache

Optimizer should tune this, given the query characteristics.
Varying the Vector size

- "tuple at a time"
- DBMS "X" (28.1)
- MySQL 4.1 (26.6)

- VectorWise "vector at a time"
- low interpretation overhead in-cache materialization

- Hand-Coded C Program (0.22)

- interpretation dominates execution
- interpretation overhead decreases
- vectors start to exceed CPU cache, causing extra memory traffic
TPC-H 1GB, query 1

- selects 98% of fact table, computes net prices and aggregates all

- Results:
  - C program: 0.2s
  - MySQL: 26.2s
  - DBMS “X”: 28.1s
  - Vectorwise: ?

“MonetDB/X100: Hyper-Pipelining Query Execution” Boncz, Zukowski, Nes, CIDR’05
TPC-H 1GB, query 1

- selects 98% of fact table, computes net prices and aggregates all

- Results:
  - C program: 0.2s
  - MySQL: 26.2s
  - DBMS “X”: 28.1s
  - Vectorwise: 0.6s

“MonetDB/X100: Hyper-Pipelining Query Execution” Boncz, Zukowski, Nes, CIDR’05
# TPC Results

The TPC defines transaction processing and database benchmarks and delivers trusted results to the industry.

## 100 GB Results

<table>
<thead>
<tr>
<th>Rank</th>
<th>Company</th>
<th>System</th>
<th>QphH</th>
<th>Price/QphH</th>
<th>Watts/KQphH</th>
<th>System Availability</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Lenovo</td>
<td>ThinkServer RD630</td>
<td>420.092</td>
<td>.11 USD</td>
<td>NR</td>
<td>05/13/13</td>
<td>VectorWise 3.0.0</td>
</tr>
<tr>
<td>2</td>
<td>Dell</td>
<td>PowerEdge R720</td>
<td>403.230</td>
<td>.12 USD</td>
<td>NR</td>
<td>05/08/12</td>
<td>Actian VectorWise 2.0.1</td>
</tr>
<tr>
<td>3</td>
<td>Cisco</td>
<td>UCS C250 M2 Extended-Memory Server</td>
<td>332.481</td>
<td>.15 USD</td>
<td>NR</td>
<td>02/14/12</td>
<td>Actian VectorWise 2.0.1</td>
</tr>
<tr>
<td>4</td>
<td>Dell</td>
<td>PowerEdge R610</td>
<td>303.289</td>
<td>.16 USD</td>
<td>1.28</td>
<td>06/30/13</td>
<td>Actian VectorWise 1.6</td>
</tr>
<tr>
<td>5</td>
<td>INGRES</td>
<td>ProLiant DL380 G7</td>
<td>251.561</td>
<td>.38 USD</td>
<td>NR</td>
<td>03/31/13</td>
<td>Actian VectorWise 1.5</td>
</tr>
</tbody>
</table>

## 300 GB Results

<table>
<thead>
<tr>
<th>Rank</th>
<th>Company</th>
<th>System</th>
<th>QphH</th>
<th>Price/QphH</th>
<th>Watts/KQphH</th>
<th>System Availability</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Lenovo</td>
<td>ThinkServer RD630</td>
<td>434.353</td>
<td>.24 USD</td>
<td>NR</td>
<td>05/10/13</td>
<td>VectorWise 3.0.0</td>
</tr>
</tbody>
</table>
DB2 with BLU Acceleration

Breakthrough analytics performance

Columnar store scans and locates relevant data based on columns instead of row formats, resulting in faster processing.

BLAZING-FAST PERFORMANCE:
A Technical Best Practices Tour with ColumnStore Index
Susan Price
Senior Program Manager

SAP HANA® Solution
Redefines In-memory Computing
Query Compilation

JIT Query Compilation?
Netteza, ParAccel, HIQUE, Hyper

"tuple at a time"
DBMS "X"
MySQL 4.1
interpretation dominates execution

"vector at a time"
low interpretation overhead
in-cache materialization

Hand-Coded C Program
Summary

• **Computer Architecture Trends**
  – CPU performance increased with many strings attached
  – Databases “difficult workload” do not profit fully

• **Database Architecture Response**
  – vectorized execution (Vectorwise- CWI)
  – compiled execution (Hyper - TUM)
    • Detailed discussion omitted (see appendix slides)
Query JIT Compilation
an alternative to vectorization?
vectorization || compilation?

- vectorization && compilation!!

- Damon2011: is it worth combining these?
  - In Vectorwise, should one add compilation?
  - In a JIT compilation database executor, can one add vectorization?

YES!
single-loop compilation approach

- Used in Netteza, ParAccel, HIQUE, Hyper, ...
- Compilation as proposed so far is “single-loop” compilation.
  - Processing as in tuple-at-a-time system.

```sql
SELECT SUM(price*(1+tax))
FROM orders
WHERE oid >= 100
  AND oid <= 200
GROUP BY category
```

for each tuple
```python
if(oid >= 100 && oid <= 200):
    result[category] += price*(1+tax);
```
vectorization = multi-loop

- Vectorization is “multi-loop” by definition.
  - Basic operations performed vector-at-a-time.
  - Interpretation overhead amortized.
  - Materialization of each step’s result.

```sql
SELECT SUM(price*(1+tax))
FROM orders
WHERE oid >= 100
AND oid <= 200
GROUPBY category
```

while(tuples)
Get vector of n tuples;
for(i = 0, m=0; i<n; i++)
  if(oid >= 100) sel[m++] = i;
for(i = 0, k=0; i<m; i++)
  { sel[k]=i; k+= (oid <= 200); }
for(i = 0; i < k; i++)
t1[sel[i]] = 1+tax[sel[i]];
for(i = 0; i < k; i++)
t2[sel[i]] = tmp1[sel[i]]*price[sel[i]];
for(i = 0; i < k;i++)
result[category[sel[i]] += t2[sel[i]];
```
multi-loop compilation

- Multi-loop compilation is often best!
  - Compiling small fragments takes less compilation time and is more reusable.
  - Sometimes benefits of a tight loop are bigger than materialization cost.

```plaintext
while(tuples)
    Get vector of n tuples;
    for(i = 0, m=0; i<n; i++)
        if(oid >= 100) sel[m++] = i;
    for(i = 0, k=0; i<m; i++)
        { sel[k]=i; k+= (oid <= 200); }
    for(i = 0; i < k; i++)
        result[category[sel[i]]] += price[sel[i]]*(1+tax[sel[i]);

SELECT SUM(price*(1+tax))
FROM orders
WHERE oid >= 100
AND oid <= 200
GROUPBY category
```

* Just an example. Not necessarily optimal.
Case studies

see: Damon2011 Sompolski et al.

• Projections
• Selections
• Hash lookups
Case studies

see: Damon2011 Sompolski et al.

• Projections
  • Easier SIMD
  • Avoids branch mispredictions
  • Improves memory access pattern

• Selections

• Hash lookups
Hash lookup algorithm

```plaintext
pos = B[hash_keys(probe_keys)]
if (pos) {
    do { // pos == 0 reserved for miss.
        if (keys_equal(probe_keys, V[pos].keys)) {
            fetch_value_columns(V[pos]);
            break; // match
        }
    } while (pos = next in chain); // collision or miss
}
```

Bucket-chained Hash Table
Hash lookup algorithm

```
pos = B[hash_keys(probe_keys)]
if (pos) {
    do { // pos == 0 reserved for miss.
        if (keys_equal(probe_keys, V[pos].keys)) {
            fetch_value_columns(V[pos]);
            break; // match
        }
    } while (pos = next in chain); // collision or miss
}
```

**Interpretation:**
- Type of keys.
- Multi-attribute keys.
- Type of fetched columns.
- Number of fetched columns.
single-loop compiled hash lookup: avoids interpretation

```plaintext
for (i=0; i<n; i++) {
    pos = B[hash(key1[i]) ^ hash(key2[i]) & SIZE];
    if (pos) {
        do {
            if (key1[i] == V[pos].key1 && key2[i] == V[pos].key2) {
                res1[i] = V[pos].val1;
                res2[i] = V[pos].val2;
                res3[i] = V[pos].val3;
                break; // match
            }
        } while (pos = V[pos].next); // miss
    }
}
```

Avoid interpretation:
- Hard-coded hashing and comparing keys
- Hard-coded fetching values
single-loop compiled hash lookup: dependencies

for (i=0; i<n; i++) {
    pos = B[HASH(key1[i]) ^ HASH(key2[i]) & SIZE];
    if (pos) {
        do {
            if (key1[i] == V[pos].key1 && key2[i] == V[pos].key2) {
                res1[i] = V[pos].val1;
                res2[i] = V[pos].val2;
                res3[i] = V[pos].val3;
                break; // match
            }
        } while (pos = V[pos].next); // miss
    }
}
single-loop compiled hash lookup: dependencies

```c
for (i=0; i<n; i++) {
    pos = B[HASH(key1[i]) ^ HASH(key2[i]) & SIZE];
    if (pos) {
        do {
            if (key1[i] == V[pos].key1 && key2[i] == V[pos].key2) {
                res1[i] = V[pos].val1;
                res2[i] = V[pos].val2;
                res3[i] = V[pos].val3;
                break; // match
            }
        } while (pos = V[pos].next);
    }
}
```

**High random access cost:**
- Both B and V are huge arrays
  - Cache miss
  - TLB miss
single-loop compiled hash lookup: dependencies

```c
for (i=0; i<n; i++) {
    pos = B[HASH(key1[i]) ^ HASH(key2[i]) & SIZE];
    if (pos) {
        do {
            if (key1[i] == V[pos].key1 && key2[i] == V[pos].key2) {
                res1[i] = V[pos].val1;
                res2[i] = V[pos].val2;
                res3[i] = V[pos].val3;
                break;  // match
            }
        } while (pos = V[pos].next);
    }
}
```

**High random access cost:**
- Both B and V are huge arrays
- Cache miss
- TLB miss

**Poor performance:**
- Modern processor needs multiple memory fetches in parallel to fully utilize memory bandwidth.
- No independent instructions that can hide memory latency.
single-loop compiled hash lookup:
branch predictability

```
for (i=0; i<n; i++) {
    pos = B[HASH(key1[i]) ^ HASH(key2[i]) & SIZE];
    if (pos) {
        do {
            if (key1[i]==V[pos].key1 && key2[i]==V[pos].key2) {
                res1[i] = V[pos].val1;
                res2[i] = V[pos].val2;
                res3[i] = V[pos].val3;
                break; // match
            }
        } while (pos = V[pos].next); // miss
    }
}
```

• Always match and no collisions: A

Save the day with processor speculation?
single-loop compiled hash lookup: branch predictability

\[
\text{for } (i=0; i<n; i++) \{ \\
\quad \text{pos} = B[\text{HASH(key1[i])} \land \text{HASH(key2[i])} \land \text{SIZE}]; \\
\quad \text{if } (\text{pos}) \{ \\
\qquad \text{do } \{ \\
\qquad\quad \text{if } (\text{key1[i]}==V[\text{pos}].\text{key1} \land \text{key2[i]}==V[\text{pos}].\text{key2}) \{ \\
\qquad\qquad \text{res1[i]} = V[\text{pos}].\text{val1}; \\
\qquad\qquad \text{res2[i]} = V[\text{pos}].\text{val2}; \\
\qquad\qquad \text{res3[i]} = V[\text{pos}].\text{val3}; \\
\qquad\quad \text{break}; // match \\
\qquad\} \\
\qquad \} \text{ while } (\text{pos} = V[\text{pos}].\text{next}); // miss \\
\qquad \} \\
\} \\
\]

- Always match and no collisions: ABC
for (i=0; i<n; i++) {
    pos = B[HASH(key1[i]) ^ HASH(key2[i]) & SIZE];
    if (pos) {
        do {
            if (key1[i]==V[pos].key1 && key2[i]==V[pos].key2) {
                res1[i] = V[pos].val1;
                res2[i] = V[pos].val2;
                res3[i] = V[pos].val3;
                break; // match
            }
        } while (pos = V[pos].next); // miss
    }
}

• Always match and no collisions: ABCD
single-loop compiled hash lookup: branch predictability

```java
for (i=0; i<n; i++) {
    pos = B[HASH(key1[i]) ^ HASH(key2[i]) & SIZE];
    if (pos) {
        do {
            if (key1[i] == V[pos].key1 && key2[i] == V[pos].key2) {
                res1[i] = V[pos].val1;
                res2[i] = V[pos].val2;
                res3[i] = V[pos].val3;
                break; // match
            }
        } while (pos = V[pos].next); // miss
    }
}

• Always match and no collisions: ABCD ABCD ...
```
single-loop compiled hash lookup: branch predictability

```c
for (i=0; i<n; i++) {
    pos = B[HASH(key1[i]) ^ HASH(key2[i]) & SIZE];
    if (pos) {
        do {
            if (key1[i]==V[pos].key1 && key2[i]==V[pos].key2) {
                res1[i] = V[pos].val1;
                res2[i] = V[pos].val2;
                res3[i] = V[pos].val3;
                break; // match
            }
        } while (pos = V[pos].next); // miss
    }
}
```

- Always match and no collisions: ABCD ABCD ...

Speculate and execute out-of-order to fetch data from arrays B and V for next iterations of outer loop.
single-loop compiled hash lookup: branch predictability

```c
for (i=0; i<n; i++) {
    pos = B[HASH(key1[i]) ^ HASH(key2[i])] & SIZE;
    if (pos) {
        do {
            if (key1[i]==V[pos].key1 && key2[i]==V[pos].key2) {
                res1[i] = V[pos].val1;
                res2[i] = V[pos].val2;
                res3[i] = V[pos].val3;
                break; // match
            }
        } while (pos = V[pos].next); // miss
    }
}
```

• Misses or collisions: AB AB A..
single-loop compiled hash lookup: branch predictability

```c
for (i=0; i<n; i++) {
    pos = B[HASH(key1[i]) ^ HASH(key2[i]) & SIZE];
    if (pos) {
        do {
            if (key1[i] == V[pos].key1 && key2[i] == V[pos].key2) {
                res1[i] = V[pos].val1;
                res2[i] = V[pos].val2;
                res3[i] = V[pos].val3;
                break; // match
            }
        } while (pos = V[pos].next); // miss
    }
}
```

- Misses or collisions: AB AB AB ABCECE..
single-loop compiled hash lookup: branch predictability

```c
for (i=0; i<n; i++) {
    pos = B[HASH(key1[i]) ^ HASH(key2[i]) & SIZE];
    if (pos) {
        do {
            if (key1[i]==V[pos].key1 && key2[i]==V[pos].key2) {
                res1[i] = V[pos].val1;
                res2[i] = V[pos].val2;
                res3[i] = V[pos].val3;
                break; // match
            }
        } while (pos = V[pos].next); // miss
    }
}
```

- Misses or collisions: AB AB AB ABCECE A...
single-loop compiled hash lookup: branch predictability

for (i=0; i<n; i++) {
    pos = B[HASH(key1[i])] ^ HASH(key2[i]) & SIZE;
    if (pos) {
        do {
            if (key1[i]==V[pos].key1 && key2[i]==V[pos].key2) {
                res1[i] = V[pos].val1;
                res2[i] = V[pos].val2;
                res3[i] = V[pos].val3;
                break; // match
            }
        } while (pos = V[pos].next); // miss
    }
}

- Misses or collisions: AB AB ABABCECE ABABCECD A...
single-loop compiled hash lookup: branch predictability

```
for (i=0; i<n; i++) {
    pos = B[HASH(key1[i]) ^ HASH(key2[i]) & SIZE];
    if (pos) {
        do {
            if (key1[i]==V[pos].key1 && key2[i]==V[pos].key2) {
                res1[i] = V[pos].val1;
                res2[i] = V[pos].val2;
                res3[i] = V[pos].val3;
                break; // match
            }
        } while (pos = V[pos].next); // miss
    }
}
```

- No reliable speculation! Memory stalls:
  - `pos = B[...]` must finish before “B”
  - `pos = V[pos].next must` finish before “C”

- Misses or collisions:
  ```
  • AB AB ABCECE ABCECD A...
  ```
vectorized hash lookup

Good:
• Independent loop iterations at each step.

Bad:
• Each step accessing a vector of positions all over again

// base = &V[0].key1;
for(i=0;i<n;i++)
    res[i] = (key[i] != base[stride * pos[i]]);

// base = &V[0].key2;
for(i=0;i<n;i++)
    res[i] |= (key[i] != base[stride * pos[i]]);

// base = &V[0].val3
for(i=0;i<n;i++)
    res[match[i]] = base[stride * pos[match[i]]];
**vectorized hash lookup**

<table>
<thead>
<tr>
<th>key1</th>
<th>key2</th>
<th>val1</th>
<th>val2</th>
<th>val3</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>1003</td>
<td>3</td>
<td>a</td>
<td>May</td>
</tr>
<tr>
<td>100</td>
<td>2004</td>
<td>7</td>
<td>x</td>
<td>Jan</td>
</tr>
<tr>
<td>102</td>
<td>1005</td>
<td>2</td>
<td>w</td>
<td>Oct</td>
</tr>
<tr>
<td>103</td>
<td>1100</td>
<td>6</td>
<td>d</td>
<td>Nov</td>
</tr>
<tr>
<td>120</td>
<td>1234</td>
<td>9</td>
<td>e</td>
<td>Dec</td>
</tr>
<tr>
<td>111</td>
<td>1010</td>
<td>0</td>
<td>r</td>
<td>Jan</td>
</tr>
<tr>
<td>150</td>
<td>1203</td>
<td>1</td>
<td>t</td>
<td>Jun</td>
</tr>
<tr>
<td>105</td>
<td>1003</td>
<td>3</td>
<td>g</td>
<td>Oct</td>
</tr>
<tr>
<td>103</td>
<td>1110</td>
<td>5</td>
<td>h</td>
<td>Sep</td>
</tr>
</tbody>
</table>

// base = &V[0].key1;  
for(i=0;i<n;i++)  
res[i] = (key[i] != base[stride * pos[i]]);

// base = &V[0].key2;  
for(i=0;i<n;i++)  
res[i] |= (key[i] != base[stride * pos[i]]);
vectorized hash lookup

<table>
<thead>
<tr>
<th>key1</th>
<th>key2</th>
<th>val1</th>
<th>val2</th>
<th>val3</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>1003</td>
<td>3</td>
<td>a</td>
<td>May</td>
</tr>
<tr>
<td>100</td>
<td>2004</td>
<td>7</td>
<td>x</td>
<td>Jan</td>
</tr>
<tr>
<td>102</td>
<td>1005</td>
<td>2</td>
<td>w</td>
<td>Oct</td>
</tr>
<tr>
<td>103</td>
<td>1100</td>
<td>6</td>
<td>d</td>
<td>Nov</td>
</tr>
<tr>
<td>120</td>
<td>1234</td>
<td>9</td>
<td>e</td>
<td>Dec</td>
</tr>
<tr>
<td>111</td>
<td>1010</td>
<td>0</td>
<td>r</td>
<td>Jan</td>
</tr>
<tr>
<td>150</td>
<td>1203</td>
<td>1</td>
<td>t</td>
<td>Jun</td>
</tr>
<tr>
<td>105</td>
<td>1003</td>
<td>3</td>
<td>g</td>
<td>Oct</td>
</tr>
<tr>
<td>103</td>
<td>1110</td>
<td>5</td>
<td>h</td>
<td>Sep</td>
</tr>
</tbody>
</table>

// base = &V[0].key1;
for(i=0;i<n;i++)
res[i] = (key[i] != base[stride * pos[i]]);

// base = &V[0].key2;
for(i=0;i<n;i++)
res[i] |= (key[i] != base[stride * pos[i]]);

// pos[0]
// pos[1]
vectorized hash lookup

<table>
<thead>
<tr>
<th>key1</th>
<th>key2</th>
<th>val1</th>
<th>val2</th>
<th>val3</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>1003</td>
<td>3</td>
<td>a</td>
<td>May</td>
</tr>
<tr>
<td>100</td>
<td>2004</td>
<td>7</td>
<td>x</td>
<td>Jan</td>
</tr>
<tr>
<td>102</td>
<td>1005</td>
<td>2</td>
<td>w</td>
<td>Oct</td>
</tr>
<tr>
<td>103</td>
<td>1100</td>
<td>6</td>
<td>d</td>
<td>Nov</td>
</tr>
<tr>
<td>120</td>
<td>1234</td>
<td>9</td>
<td>e</td>
<td>Dec</td>
</tr>
<tr>
<td>111</td>
<td>1010</td>
<td>0</td>
<td>r</td>
<td>Jan</td>
</tr>
<tr>
<td>150</td>
<td>1203</td>
<td>1</td>
<td>t</td>
<td>Jun</td>
</tr>
<tr>
<td>105</td>
<td>1003</td>
<td>3</td>
<td>g</td>
<td>Oct</td>
</tr>
<tr>
<td>103</td>
<td>1110</td>
<td>5</td>
<td>h</td>
<td>Sep</td>
</tr>
</tbody>
</table>

// base = &V[0].key1;
for(i=0;i<n;i++)
res[i] = (key[i] != base[stride * pos[i]])

// base = &V[0].key2;
for(i=0;i<n;i++)
res[i] |= (key[i] != base[stride * pos[i]])
vectorized hash lookup

<table>
<thead>
<tr>
<th>pos[0]</th>
<th>key1</th>
<th>key2</th>
<th>val1</th>
<th>val2</th>
<th>val3</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>1003</td>
<td>3</td>
<td>a</td>
<td>May</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>2004</td>
<td>7</td>
<td>x</td>
<td>Jan</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>1005</td>
<td>2</td>
<td>w</td>
<td>Oct</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>1100</td>
<td>6</td>
<td>d</td>
<td>Nov</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pos[2]</th>
<th>key1</th>
<th>key2</th>
<th>val1</th>
<th>val2</th>
<th>val3</th>
</tr>
</thead>
<tbody>
<tr>
<td>120</td>
<td>1234</td>
<td>9</td>
<td>e</td>
<td>Dec</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>1010</td>
<td>0</td>
<td>r</td>
<td>Jan</td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>1203</td>
<td>1</td>
<td>t</td>
<td>Jun</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pos[1]</th>
<th>key1</th>
<th>key2</th>
<th>val1</th>
<th>val2</th>
<th>val3</th>
</tr>
</thead>
<tbody>
<tr>
<td>105</td>
<td>1003</td>
<td>3</td>
<td>g</td>
<td>Oct</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pos[n-1]</th>
<th>key1</th>
<th>key2</th>
<th>val1</th>
<th>val2</th>
<th>val3</th>
</tr>
</thead>
<tbody>
<tr>
<td>103</td>
<td>1110</td>
<td>5</td>
<td>h</td>
<td>Sep</td>
<td></td>
</tr>
</tbody>
</table>

// base = &V[0].key1; 
for(i=0;i<n;i++) 
res[i] = (key[i] != base[stride * pos[i]]);

// base = &V[0].key2; 
for(i=0;i<n;i++) 
res[i] |= (key[i] != base[stride * pos[i]]);
vectorized hash lookup

<table>
<thead>
<tr>
<th>key1</th>
<th>key2</th>
<th>val1</th>
<th>val2</th>
<th>val3</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>1003</td>
<td>3</td>
<td>a</td>
<td>May</td>
</tr>
<tr>
<td>100</td>
<td>2004</td>
<td>7</td>
<td>x</td>
<td>Jan</td>
</tr>
<tr>
<td>102</td>
<td>1005</td>
<td>2</td>
<td>w</td>
<td>Oct</td>
</tr>
<tr>
<td>103</td>
<td>1003</td>
<td>1</td>
<td>g</td>
<td>Oct</td>
</tr>
<tr>
<td>120</td>
<td>1005</td>
<td>2</td>
<td>w</td>
<td>Oct</td>
</tr>
<tr>
<td>111</td>
<td>1003</td>
<td>1</td>
<td>g</td>
<td>Oct</td>
</tr>
<tr>
<td>150</td>
<td>1005</td>
<td>2</td>
<td>w</td>
<td>Oct</td>
</tr>
<tr>
<td>105</td>
<td>1005</td>
<td>3</td>
<td>g</td>
<td>Oct</td>
</tr>
<tr>
<td>103</td>
<td>1110</td>
<td>5</td>
<td>h</td>
<td>Sep</td>
</tr>
</tbody>
</table>

Bad:
- Has to fetch V[pos[0]] again.
- Already evicted from TLB cache.

// base = &V[0].key1;  
for(i=0;i<n;i++)  
res[i] = (key[i] != base[stride * pos[i]]);  

// base = &V[0].key2;  
for(i=0;i<n;i++)  
res[i] |= (key[i] != base[stride * pos[i]]);
single-loop compiled hash lookup

```c
for (i=0; i<n; i++) {
    pos = B[HASH(key1[i]) ^ HASH(key2[i]) & SIZE];
    if (pos) {
        do {
            if (key1[i]==V[pos].key1 && key2[i]==V[pos].key2) {
                res1[i] = V[pos].val1;
                res2[i] = V[pos].val2;
                res3[i] = V[pos].val3;
                break; // match
            }
        } while (pos = V[pos].next); // miss
    }
}
```

Reads tuple once.
vectorized hash lookup

for \(i=0; i<n; i++\) {
    \(pos = B[\text{HASH}(\text{key1}[i]) \oplus \text{HASH}(\text{key2}[i]) \& \text{SIZE}]\);
    if (pos) {
        do {
            if (\text{key1}[i]==V[pos].key1 && key2[i]==V[pos].key2) {
                res1[i] = V[pos].val1;
                res2[i] = V[pos].val2;
                res3[i] = V[pos].val3;
                break; // match
            }
        } while (pos = V[pos].next); // miss
    }
}

Check k1 for pos[]
Recheck k2 for pos[]
Fetch vector of pos[] from B
Select miss match
Fetch v1 for match[]
Fetch v2 for match[]
Fetch v3 for match[]
Fetch new pos[] from next in miss[]
Loop until pos[] empty

Hash vector of k1
Rehash vector of k2

Fetch vector of pos[] from B

Hash vector of k1
Rehash vector of k2
Fetch vector of pos[] from B
Select miss match
Fetch v1 for match[]
Fetch v2 for match[]
Fetch v3 for match[]
Fetch new pos[] from next in miss[]
Loop until pos[] empty
multi-loop compiled hash lookup

for (int i = 0; i < n; i++) {
    pos = B[HASH(key1[i]) ^ HASH(key2[i]) & SIZE];
    if (pos) {
        for each element pos in Pos[]:
            if keys of V[pos] match:
                fetch V[pos] val1, val2, val3 into result
            else:
                fetch V[pos] next into new Pos[]
        Independent memory accesses
        In different loop iterations
    }
    Repeat until Pos[] empty
    Reads tuple once.
}
Hash lookup benchmarks

• Experiment 1: Probing with varying match-ratio.

• Multi-loop compiled is most robust.
Hash lookup benchmarks

- Experiment 2: Reduced size of B[ ] array = more hash collisions
- Multi-loop compiled is most robust.
Final Thoughts
The Quest for Performance Robustness

robust = ‘good enough’ performance all-the-time
robust ≠ ‘perfect’ performance in one experiment & subpar performance in many others

The problem is getting worse
• Computer architects do more radical things to use the transistors
• Database architecture is challenged to react to the diversifying hardware platforms
SPEC benchmark progress

Figure 3: SPEC Benchmark
Diversifying Hardware Architectures

- architectural split between mobile and server?
- multi-core trend
  - Multi-fat-core vs Many in-order simple core?
    - Niagara (+SMT), Larrabee/Intel PHI (+SIMD)
  - NUMA
    - Memory locality = database problem
  - Cache coherency ↔ scaling
    - Transactional memory, atomic instructions
- different beasts on the CPU chip
  - CPU-GPU integration
  - On-chip FPGA
  - Special purpose offloading (encryption, network, joins?), “dark silicon”
- storage diversification
  - Tape + magnetic disk + SSD + flash memory cards
  - “storage class memory”
Some Research Questions

• What are the common underlying algorithmic properties of data management methods that allow to properly utilize parallel hardware across its diverse forms?

• How to map data management methods automatically onto efficient programs in a way that makes them applicable on very diverse hardware platforms (e.g. across fat/slim many-cores, GPUs, FPGA)?

• How to use machine architectures that are heterogeneous themselves (consist of architecturally different units, e.g. CPU + GPU)?

• Can possible (sub-) answers to the above questions be united into a new database architecture?
  – adaptive to different platform properties?
  – provides robust performance?
Thank You!