Centrum Wiskunde & Informatica

Optimizing database architecture for machine architecture

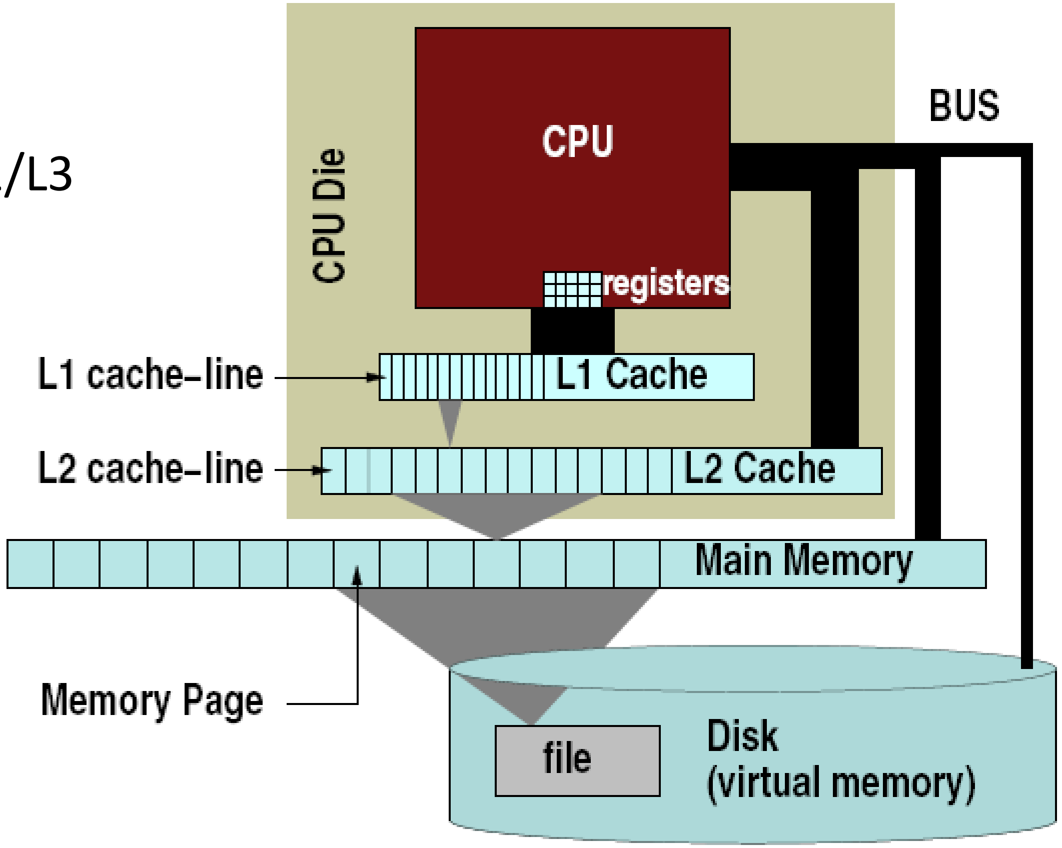
Peter Boncz

boncz@cwi.nl

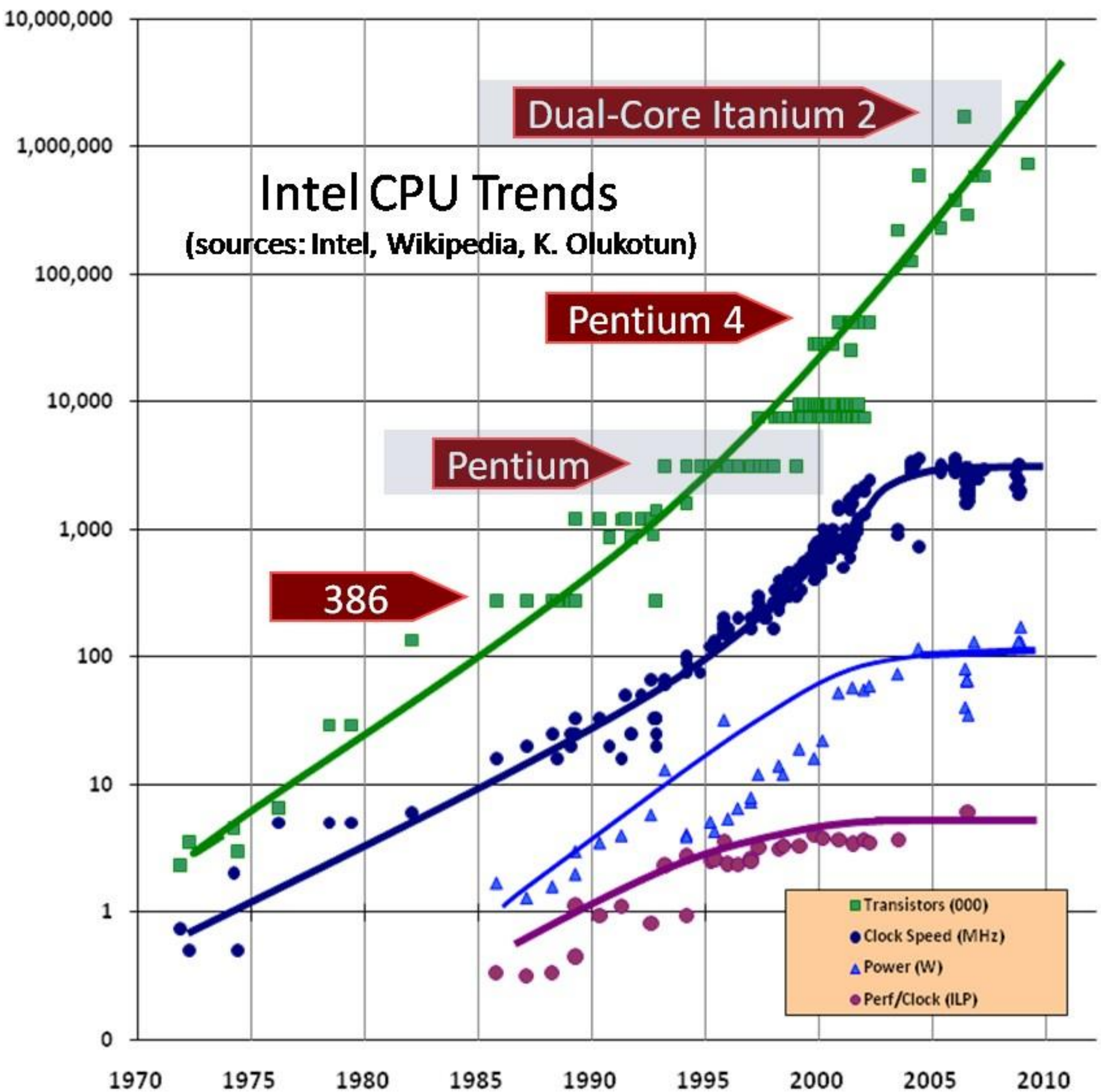
CPU Architecture

Elements:

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



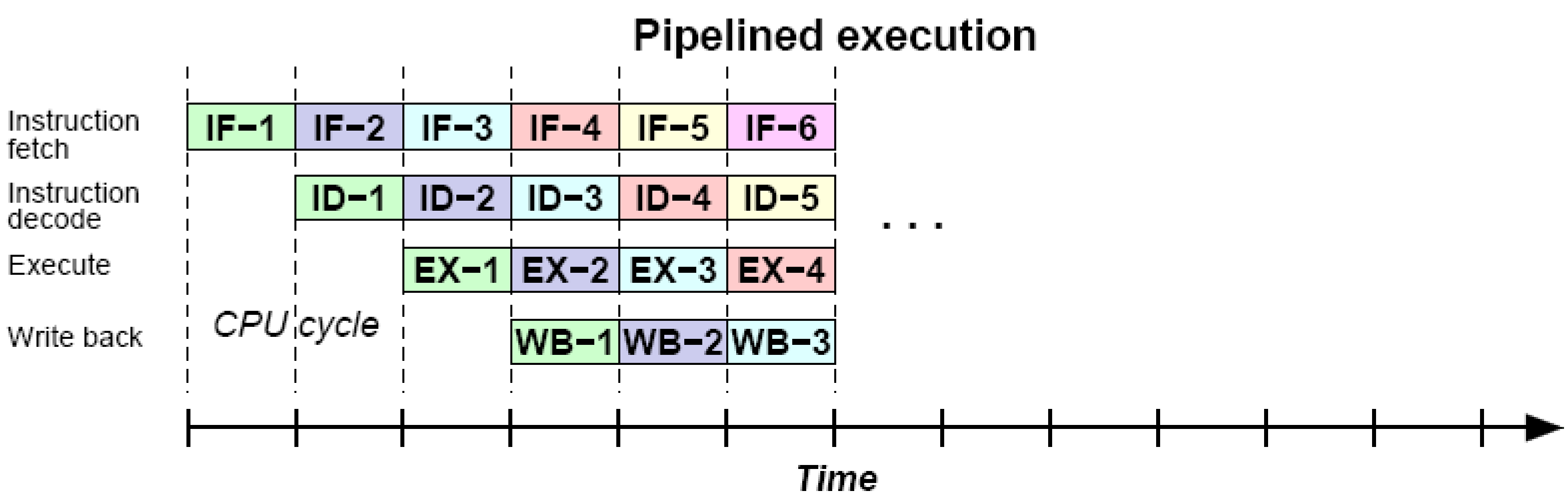
CPU Metrics



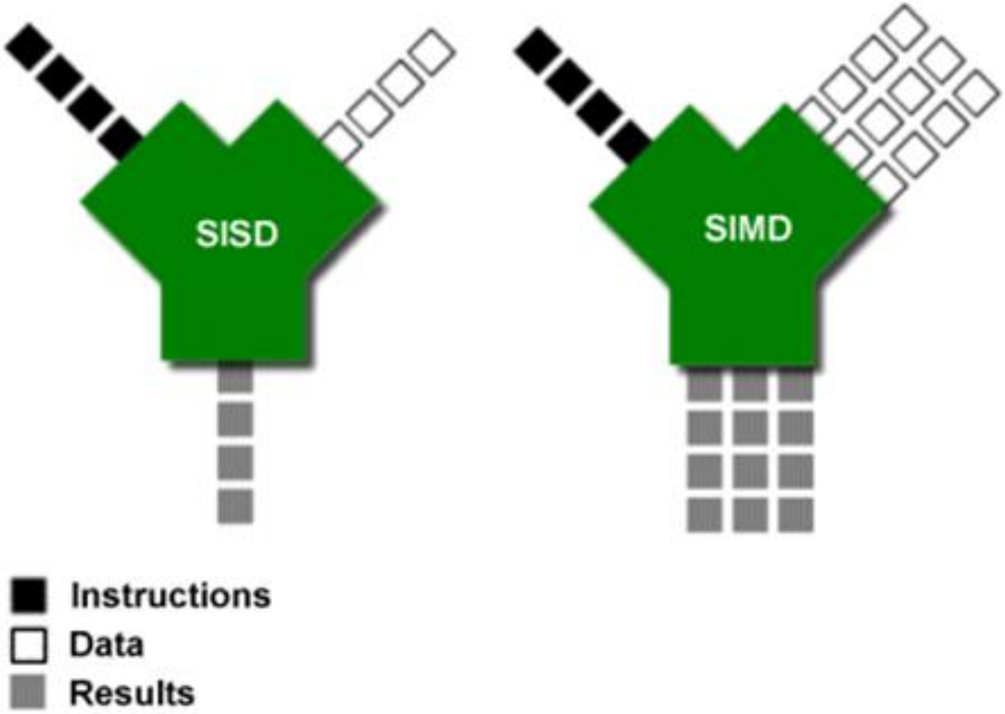
Haswell
2013
8MB L3 cache
4core (8SMT)
3.5GHz (3.9turbo)
8-way pipelines
256bits SIMD
SIMD scatter/gather
Transactional memory

Super-Scalar Execution (pipelining)

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



SIMD

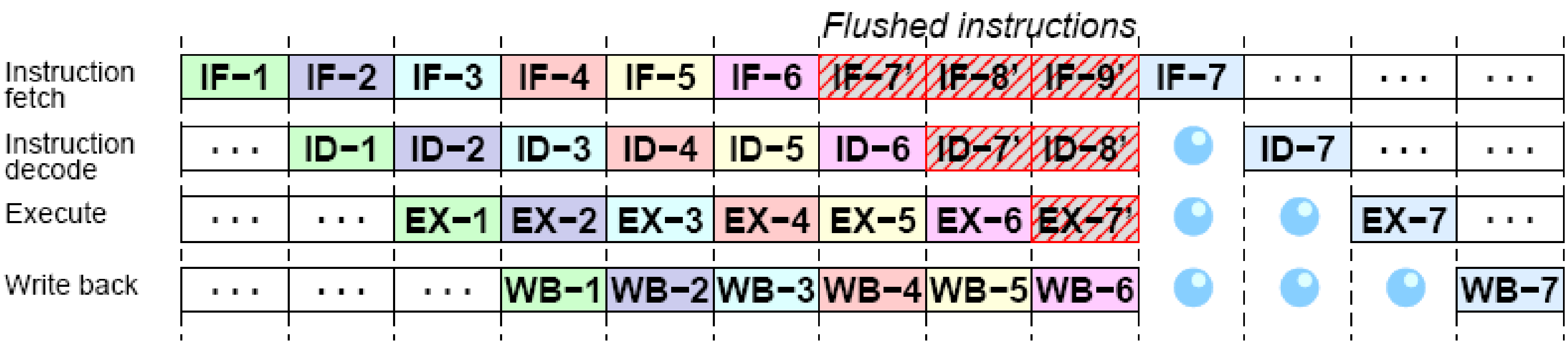


- Single Instruction Multiple Data
  - Same operation applied on a vector of values
  - MMX: 64 bits, SSE: 128bits, AVX: 256bits
  - SSE, e.g. multiply 8 short integers

Hazards

- Data hazards
  - Dependencies between instructions
  - L1 data cache misses
- Control Hazards
  - Branch mispredictions
  - Computed branches (late binding)
  - L1 instruction cache misses

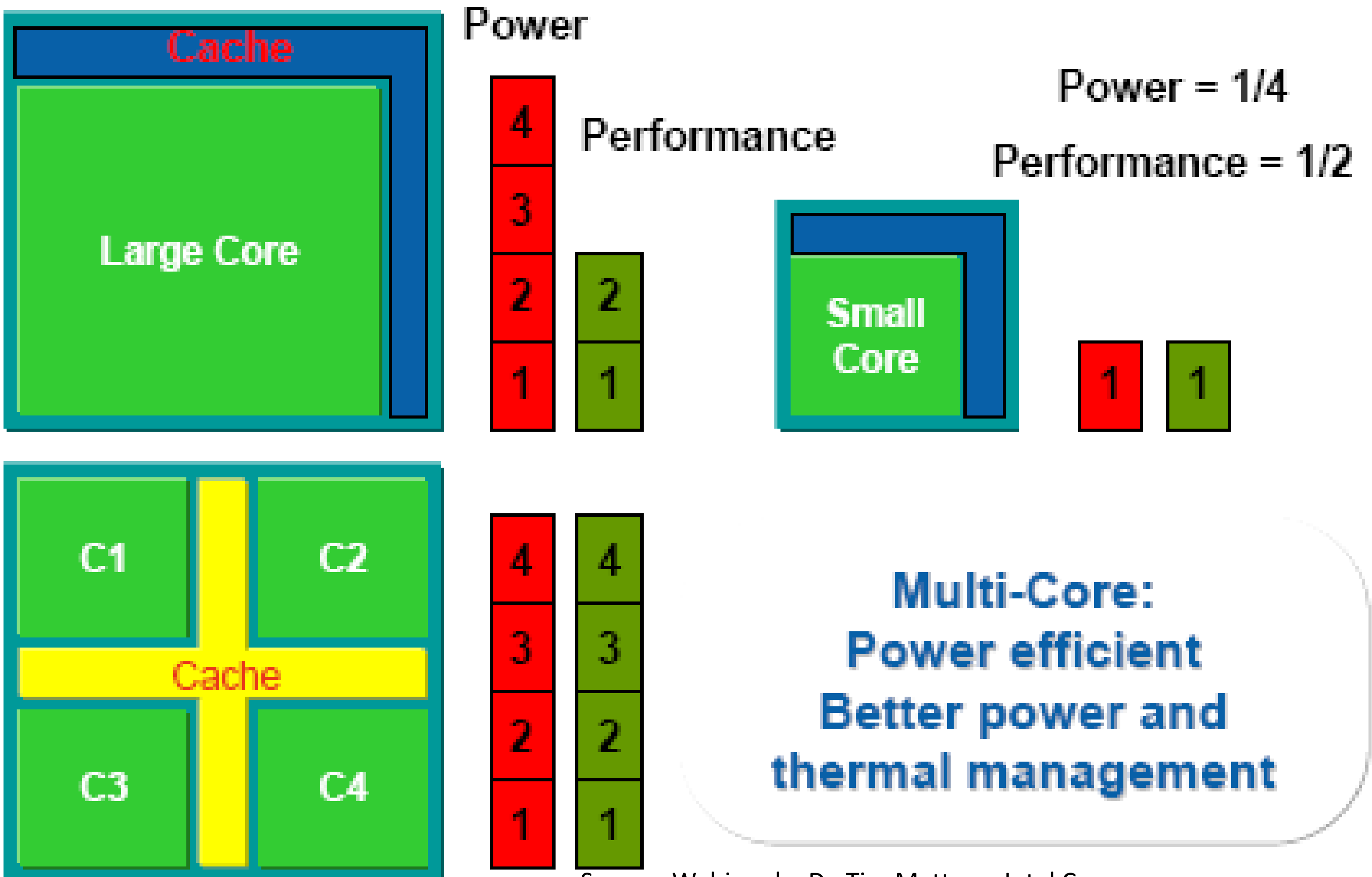
Result: bubbles in the pipeline



Out-of-order execution addresses data hazards

- control hazards typically more expensive

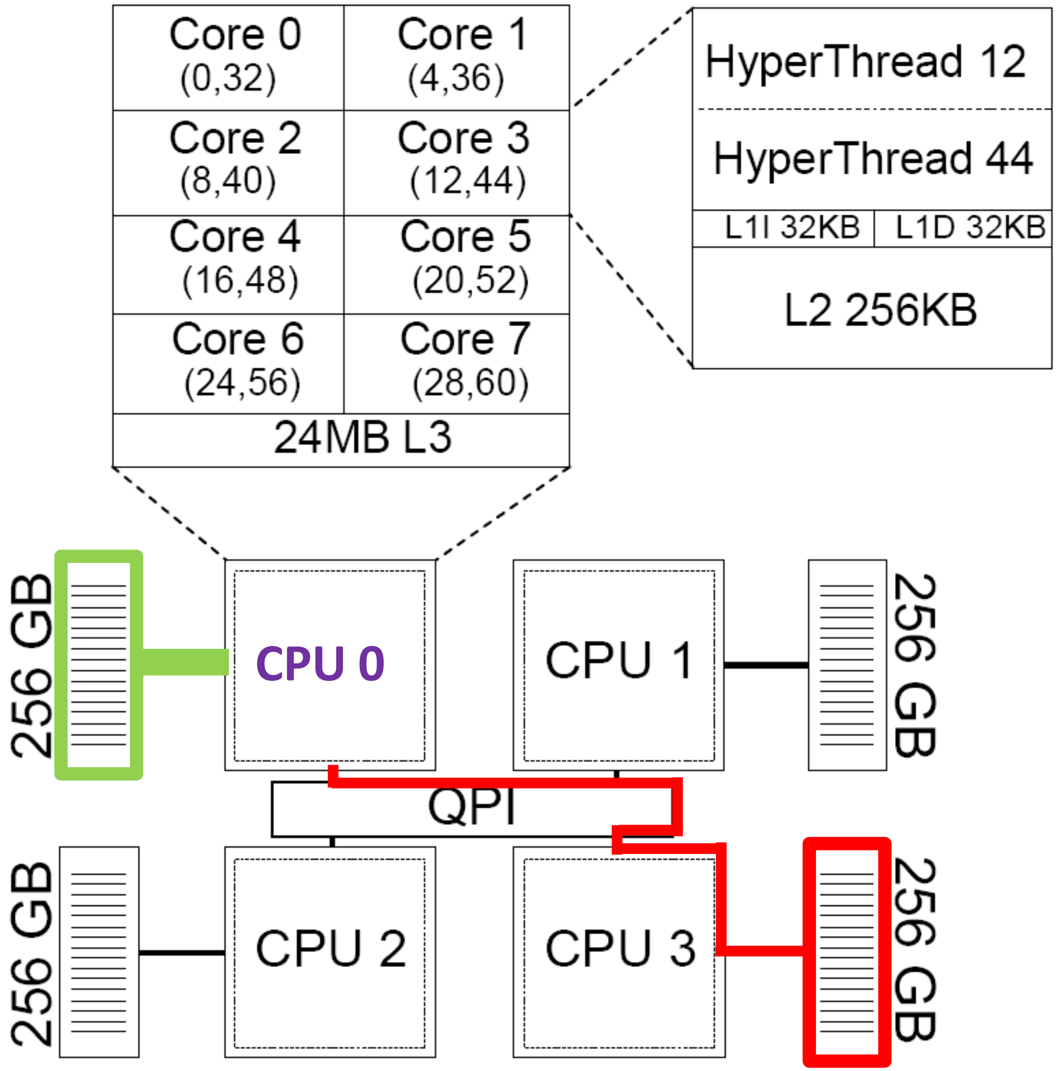
Multi-Core: sustaining Moore's law



Source: Webinar by Dr. Tim Mattson, Intel Corp.

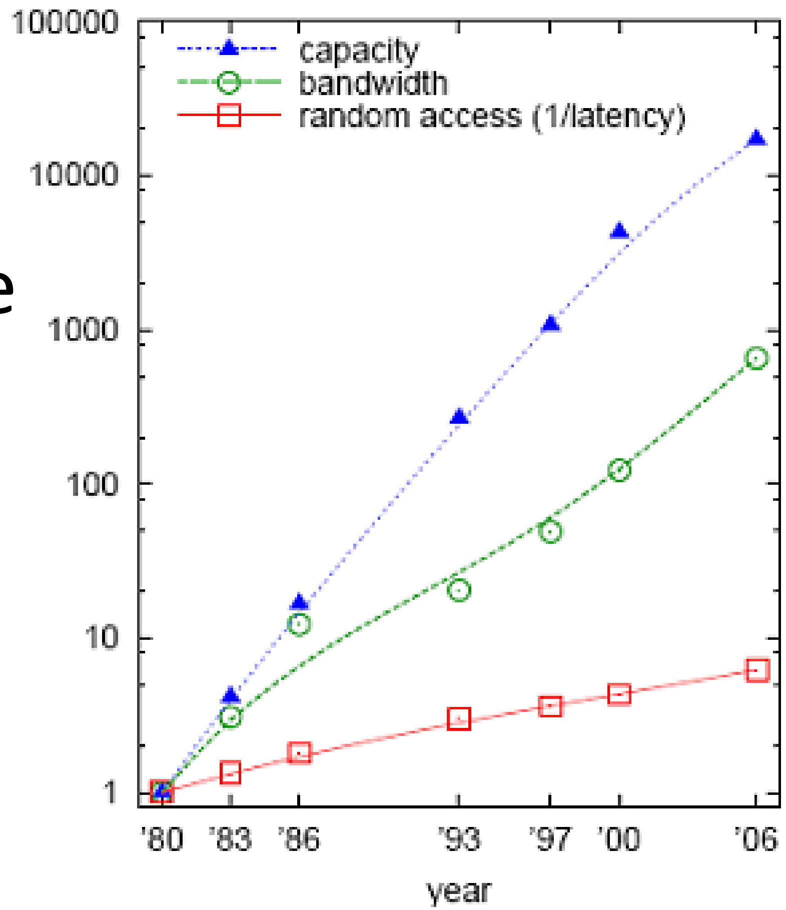
Centrum Wiskunde & Informatica

Non-uniform Memory Access (NUMA)



DRAM Metrics

- Latency improvements lag bandwidth and size

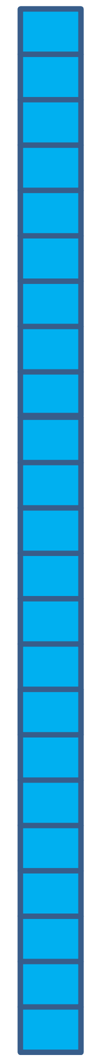


Micro-Benchmark

- ```
  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

Memory Hierarchy:

L1: 16KB = 2cyc

L2: 2MB = 15cyc

L3: 8MB = 25cyc

RAM:512GB = 200cyc

TLB coverage:

TLB1: 64 entry, 4KB pages → covers 256KB (2cyc)

TLB2: 1024 entry, 4KB pages → covers 4MB (10cyc)

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]; ➔ wait for j

vs

// FETCH

result[i] = table[input[i]];

i++; (i<n) ➔ predict true

result[i] = table[input[i]];

i++; (i<n) ➔ predict true

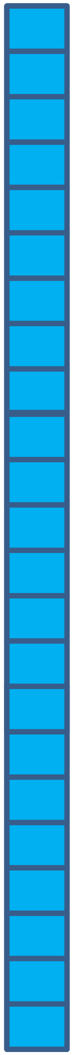
result[i] = table[input[i]];

i++; (i<n) ➔ predict true

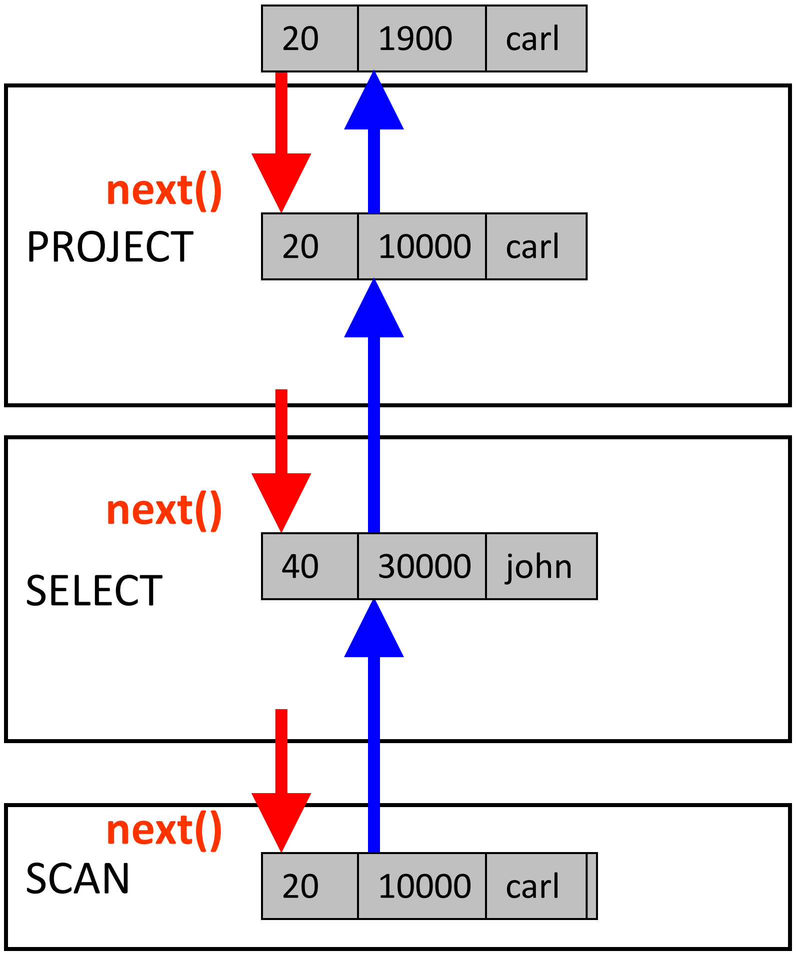
result[i] = table[input[i]];

i++; (i<n) ➔ predict true

<mem req buffer full> ➔ wait



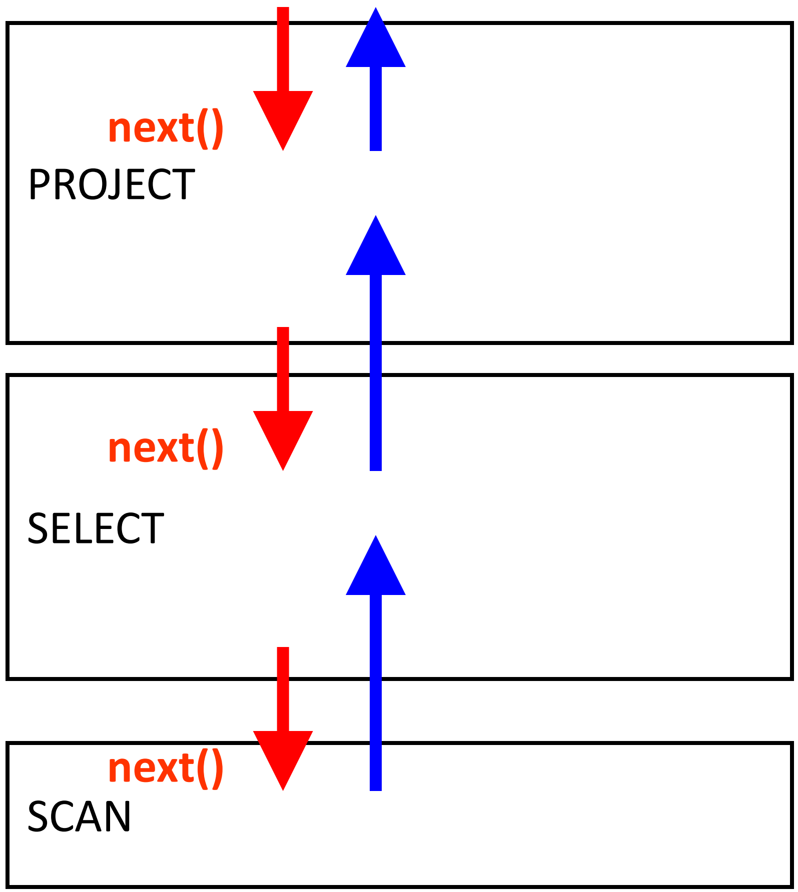
Typical Relational DBMS Engine



Query

SELECT
name,
salary*.19 AS tax
FROM
employee
WHERE
age > 25

Typical Relational DBMS Engine

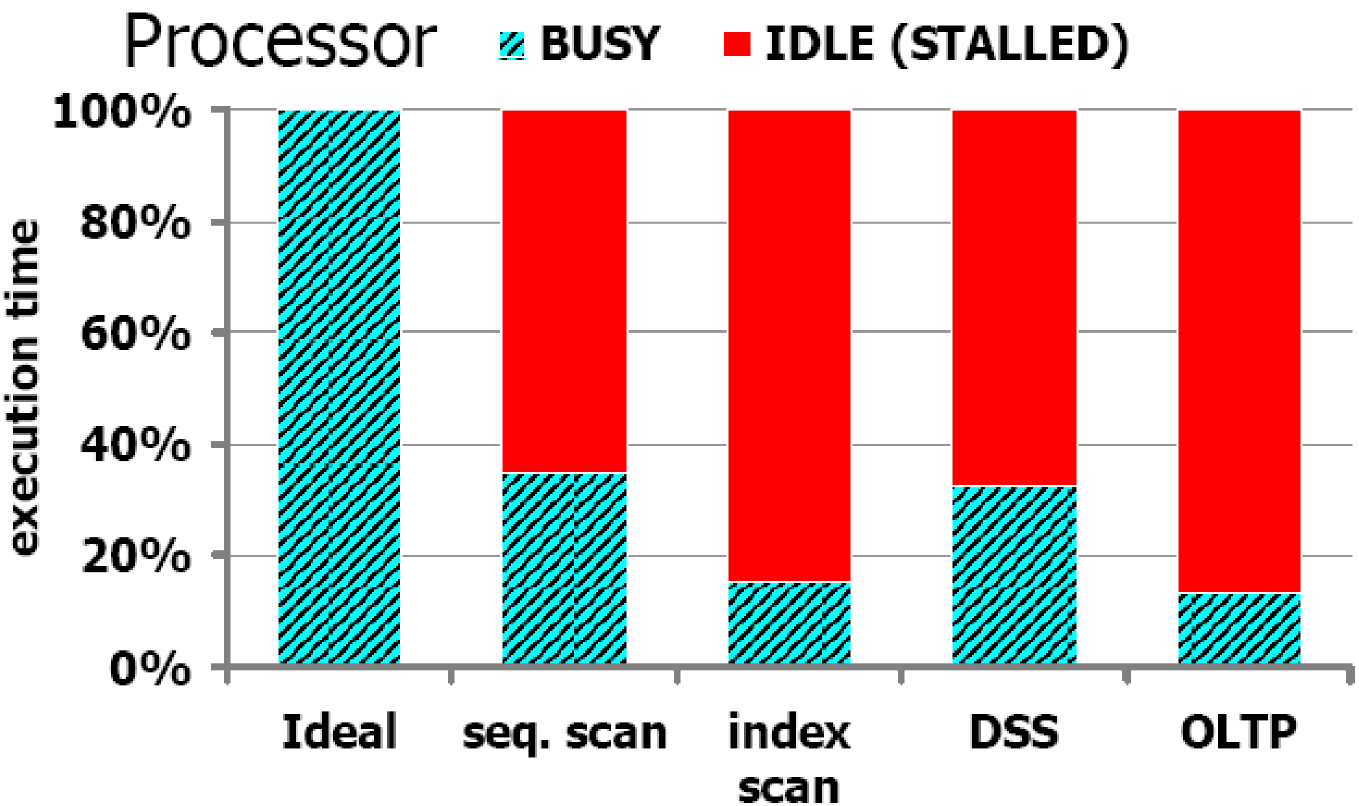


Operators

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

DBMS Computational Efficiency

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

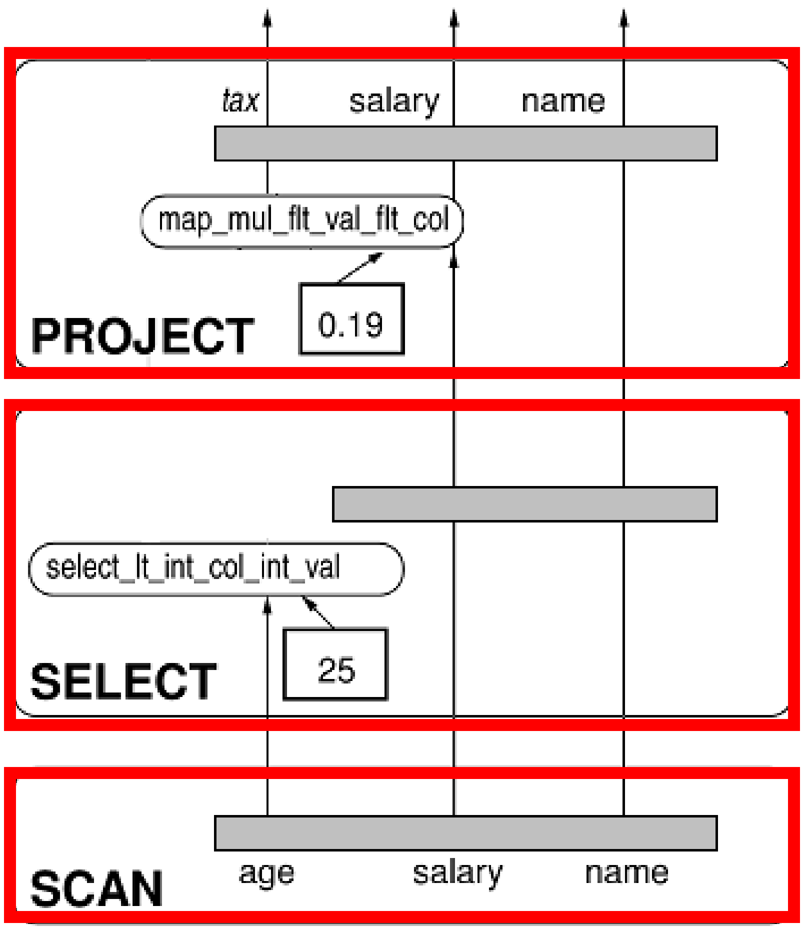
DBMS Computational Efficiency

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

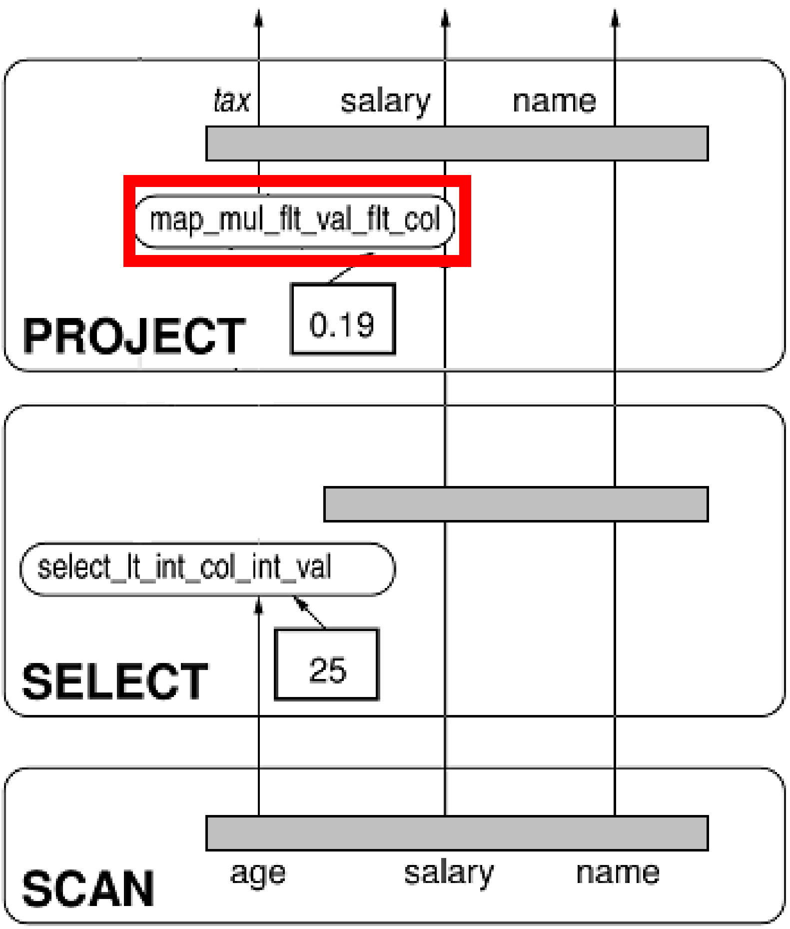
Typical Relational DBMS Engine



Operators

Iterator interface
-open()
-next(): tuple
-close()

Typical Relational DBMS Engine



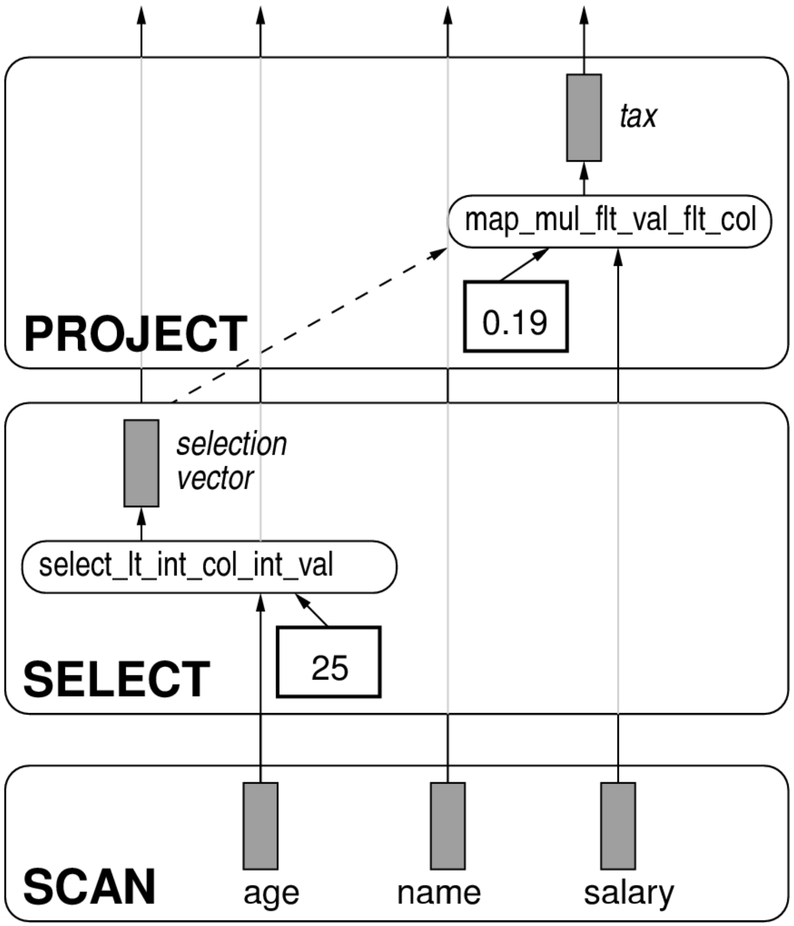
Primitives

Provide computational functionality

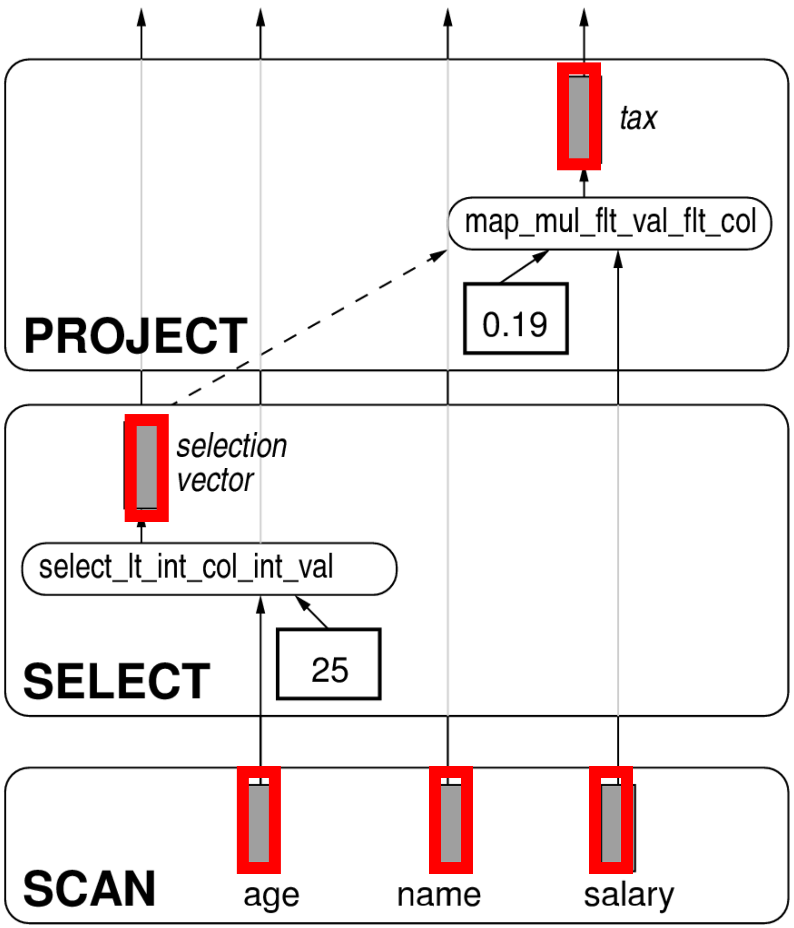
All arithmetic allowed in expressions,
e.g. multiplication

`mult(int,int)` ➔ `int`

"Vectorized Execution"



“Vectors”

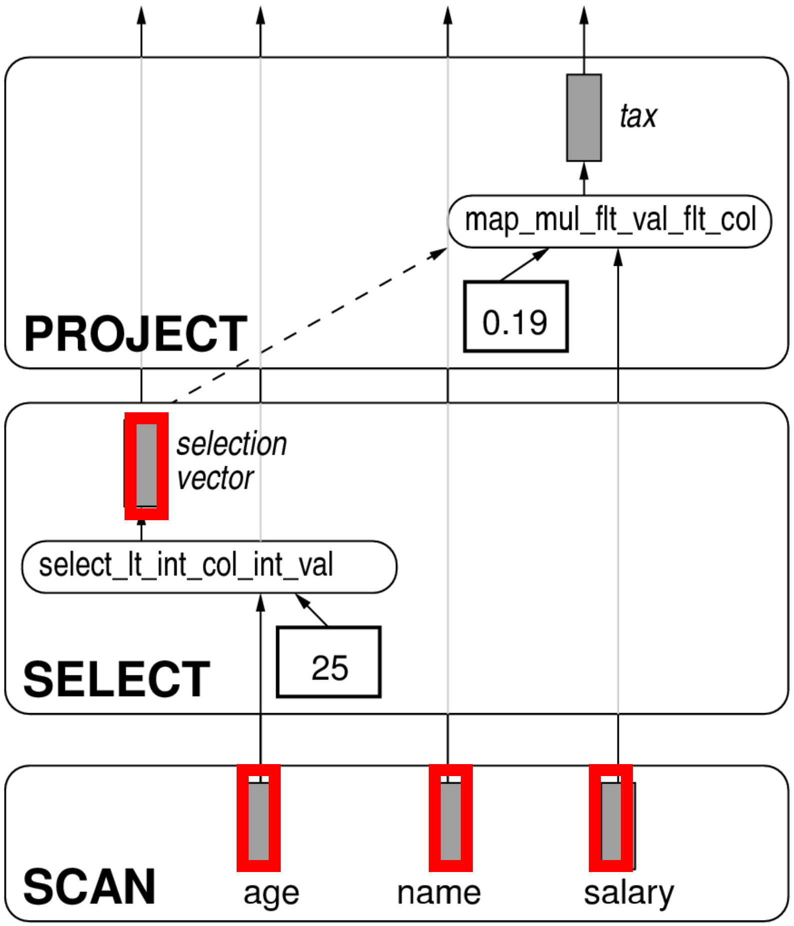


Vector contains data of ***multiple*** tuples (~100)

All primitives are “vectorized”

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

"Vectors"



Column slices to represent in-flow data

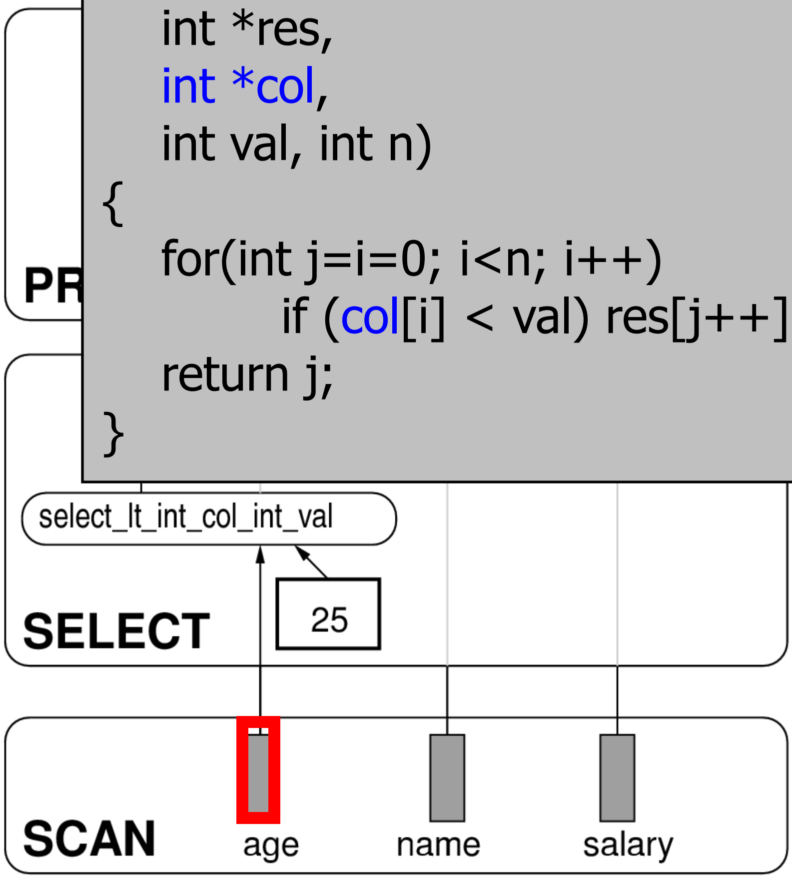
NOT:
Vertical is a better table **storage** layout than horizontal
(though we still think it often is)

RATIONALE:
- Simple array operations are well-supported by compilers
  No record layout complexities
- SIMD friendly layout

- Assumed cache-resident

Vectorized Primitives

```
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

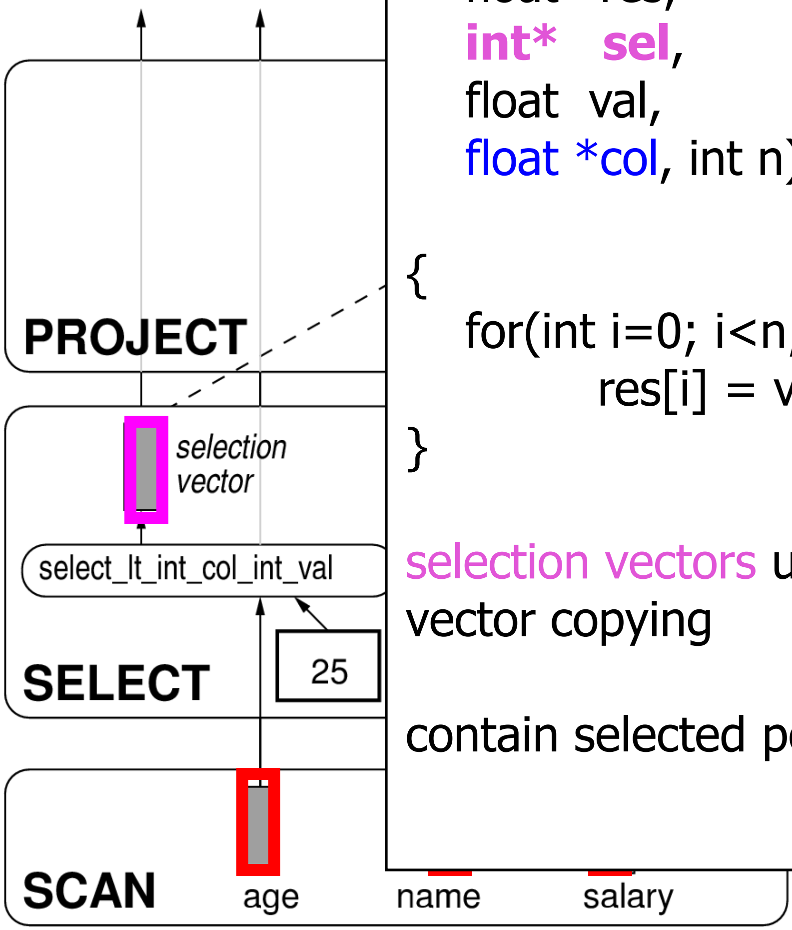
10-100x faster than Tuple-at-a-time

Selection

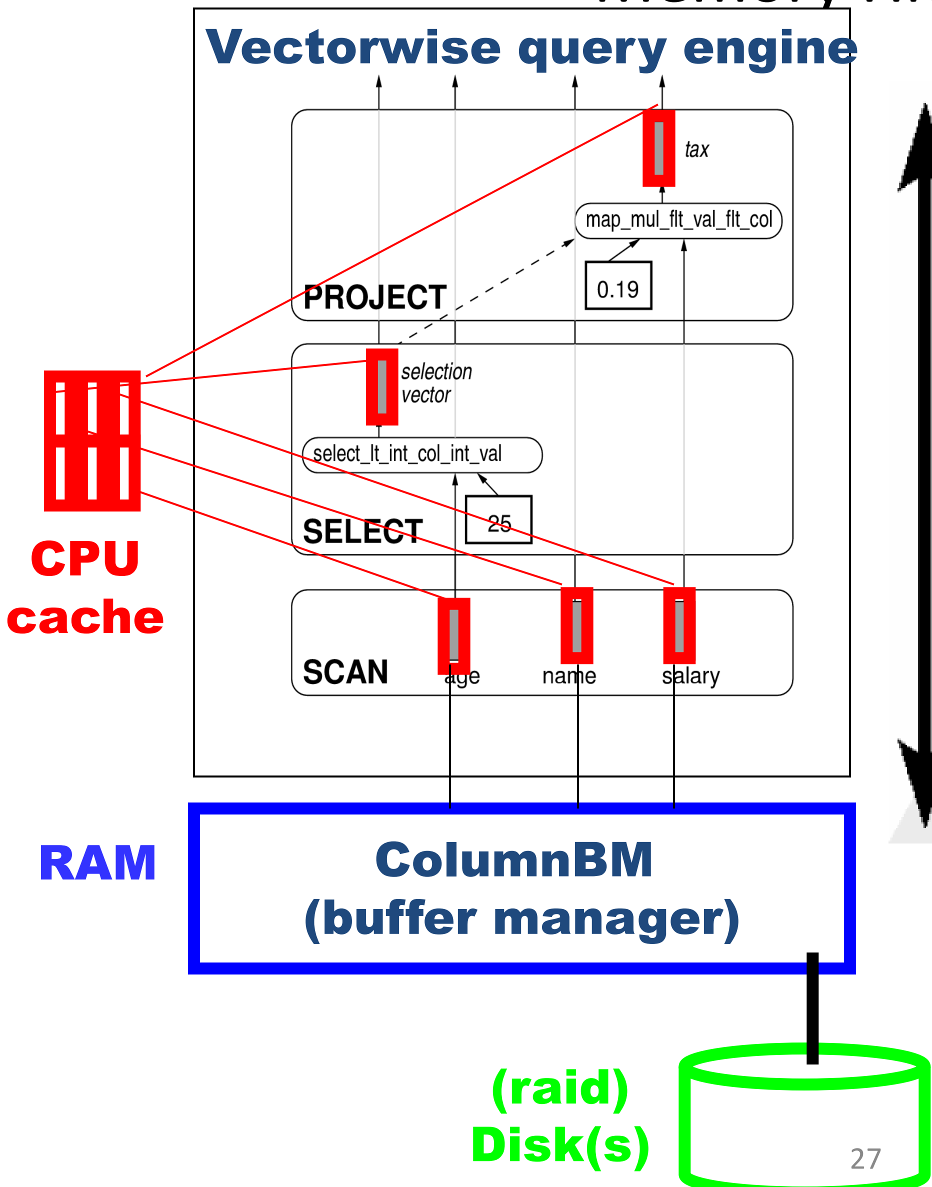
```
map_mul_flt_val_flt_col(
    float *res,
    int*   sel,
    float  val,
    float *col, int n)

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

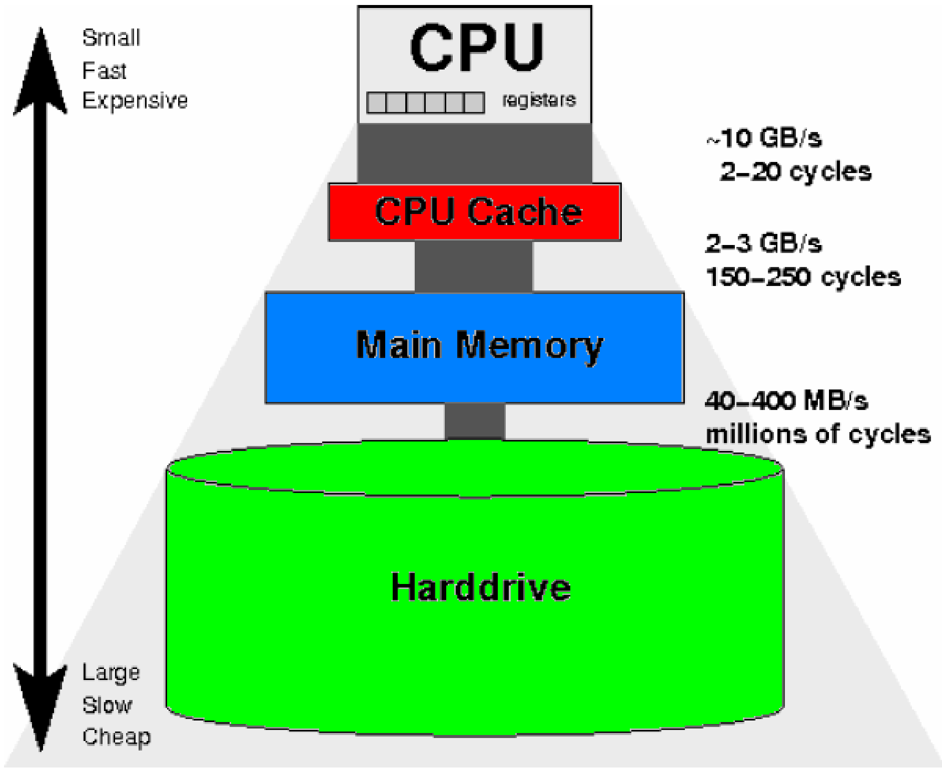
selection vectors used to reduce vector copying

contain selected positions

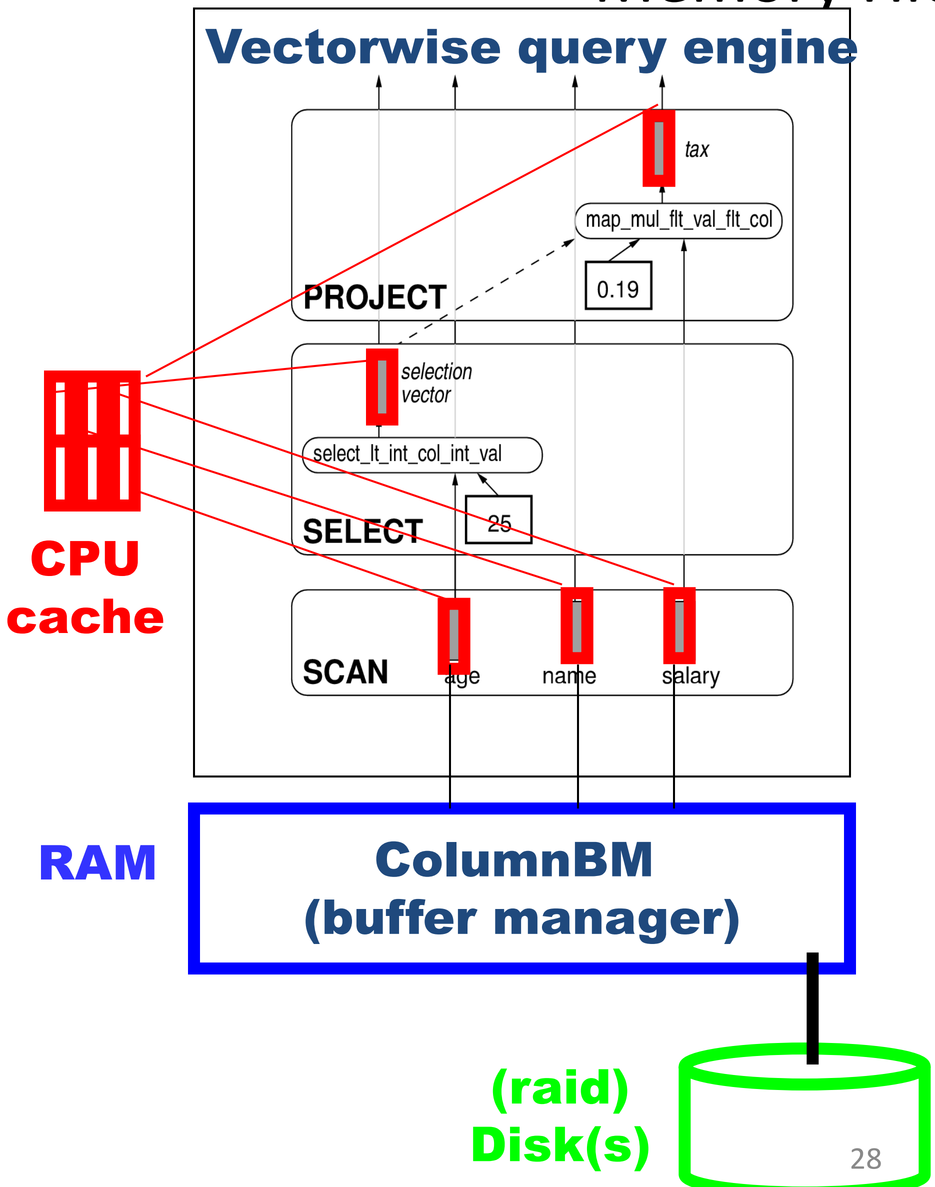


Memory Hierarchy





Memory Hierarchy

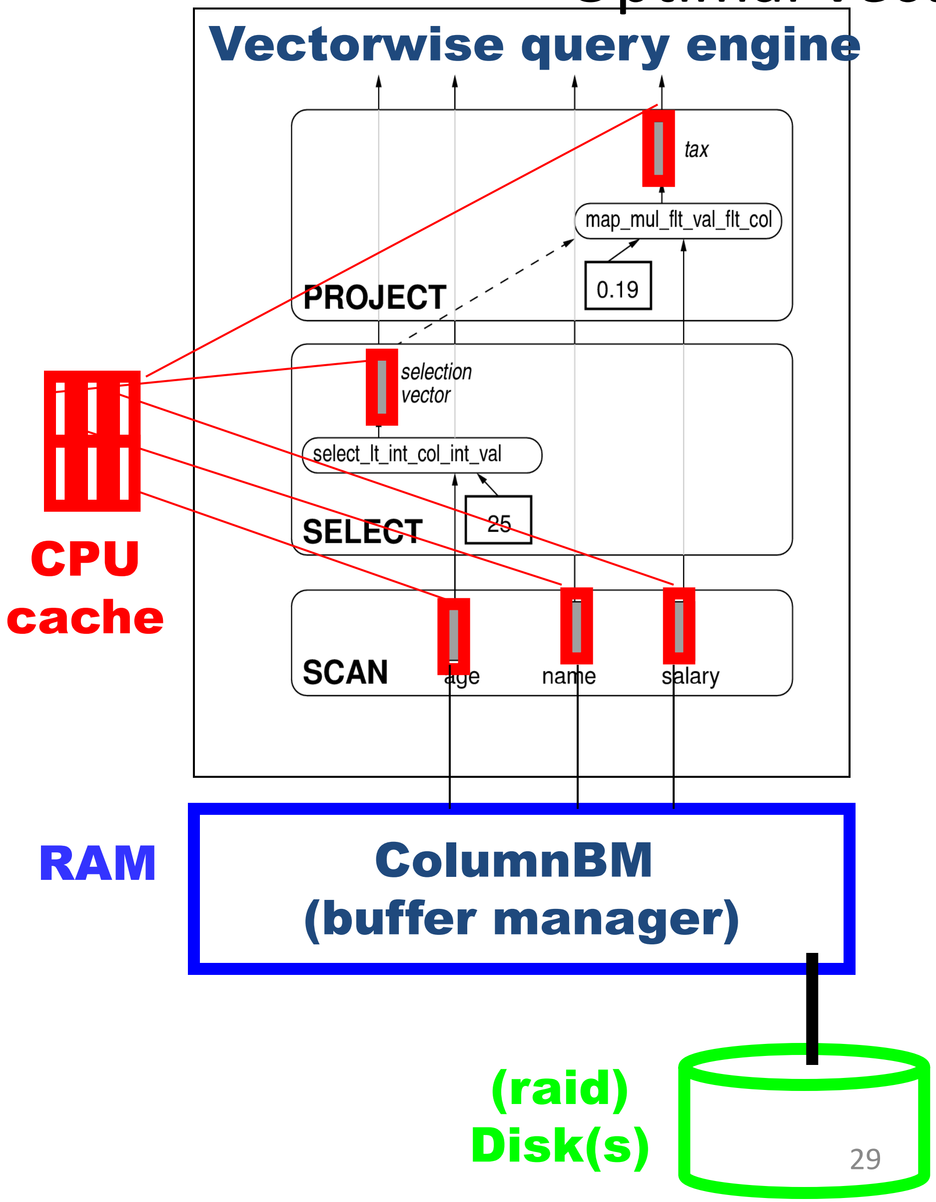


Vectors are only the in-cache representation

RAM & disk representation might actually be different

(we use both PAX and DSM)

Optimal Vector size?

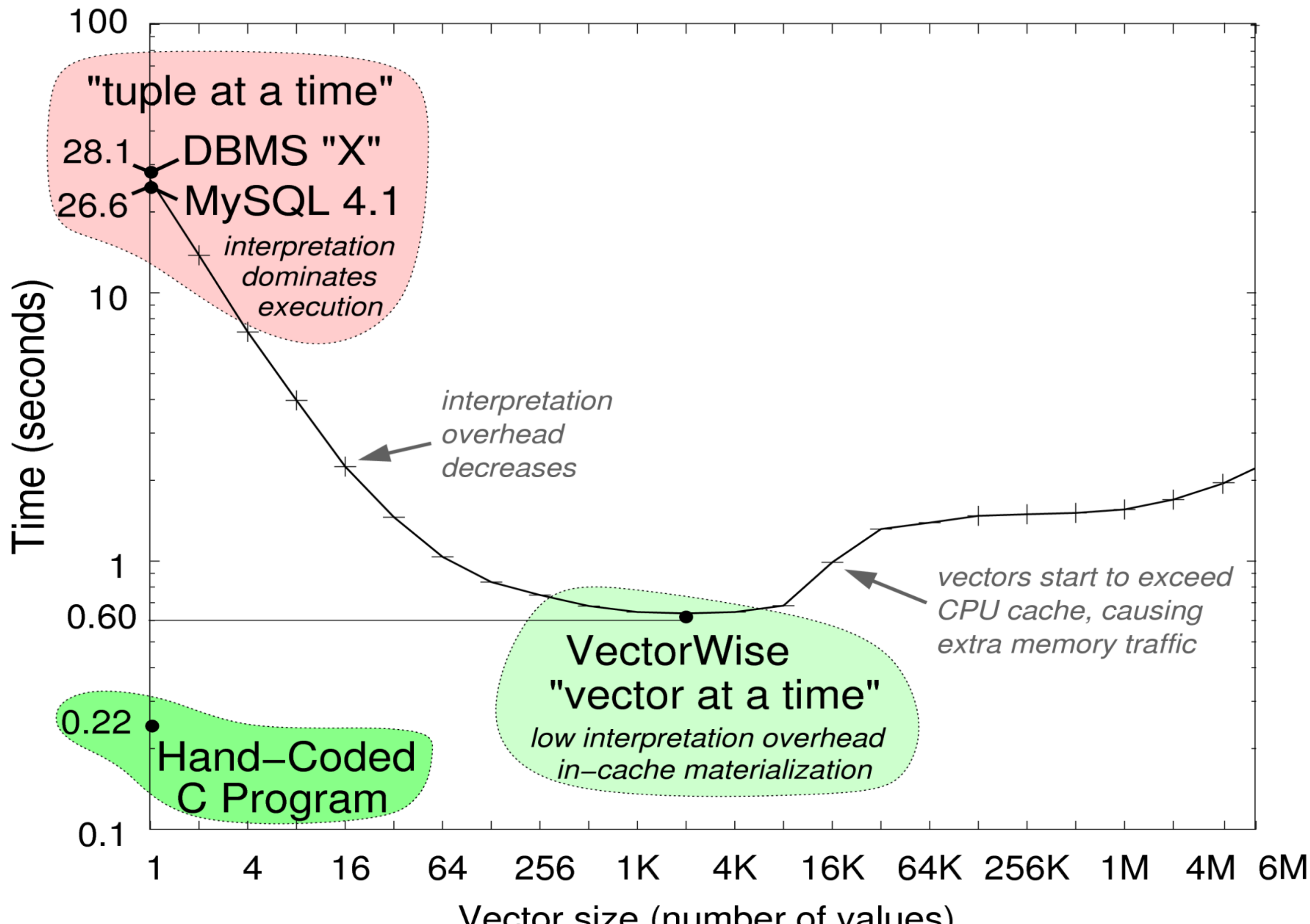


All vectors together should fit the CPU cache

Optimizer should tune this, given the query characteristics.

Centrum Wiskunde & Informatica

Varying the Vector size



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
  - Vectorwise: **?**

“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
  - Vectorwise: **0.6s**

"MonetDB/X100: Hyper-Pipelining Query Execution " Boncz, Zukowski, Nes, CIDR'05

CWI Centrum Wiskunde & Informatica

vectorwise

TPC Transaction Processing Performance Council

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

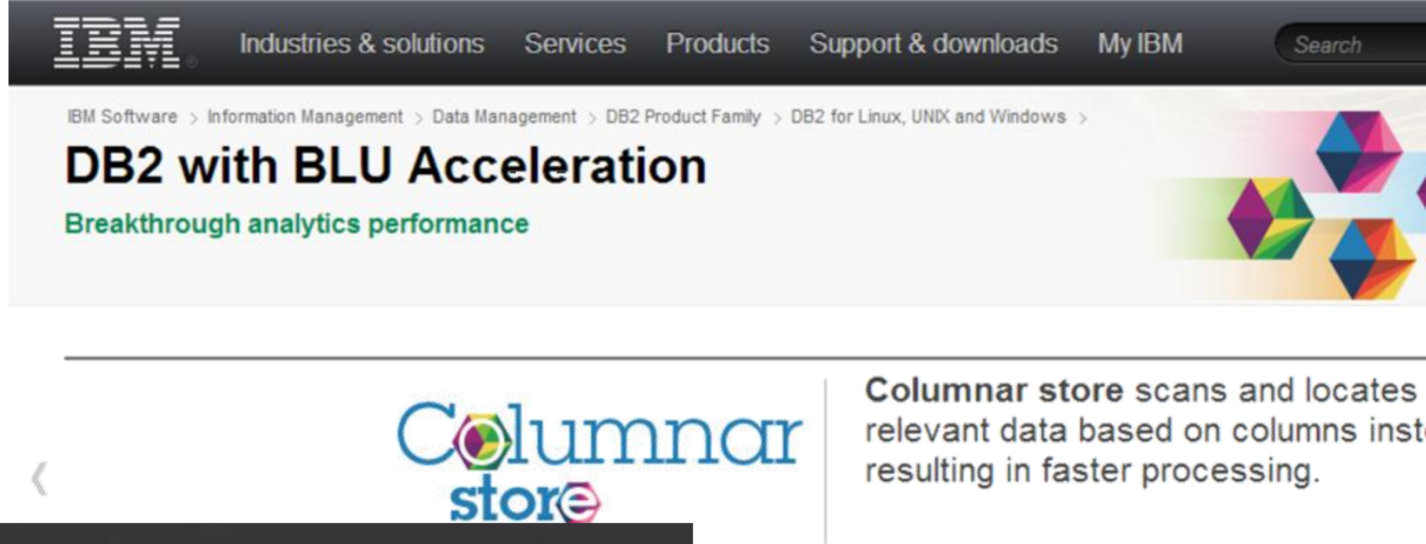
- Home
- Results
- Benchmarks
  - TPC-C
    - Results
    - Description
    - FAQ
  - TPC-DS
  - TPC-E
  - TPC-H
  - TPC-VMS
  - TPC-Pricing
  - TPC-Energy
  - Obsolete BMs
    - TPC-A
    - TPC-B
    - TPC-D
    - TPC-R
    - TPC-W
    - TPC-App
- Technical Articles
- Related Links
- What's New
- About the TPC
- Who We Are
- Contact Us
- Join the TPC
- Useful Links
- TPCTC
- Search
- Member Login

100 GB Results

Rank	Company	System	QphH	Price/QphH	Watts/KQphH	System Availability	Database	Op S
1	lenovo FOR THOSE WHO DO.	Lenovo ThinkServer RD630	420,092	.11 USD	NR	05/13/13	VectorWise 3.0.0	Re En Lir
2	DELL	Dell PowerEdge R720	403,230	.12 USD	NR	05/08/12	Actian VectorWise 2.0.1	Re En Lir
3	CISCO	Cisco UCS C250 M2 Extended-Memory Server	332,481	.15 USD	NR	02/14/12	Actian VectorWise 2.0.1	Re En Lir
4	DELL	Dell PowerEdge R610	303,289	.16 USD	1.28	06/30/11	Actian VectorWise 1.6	Re En Lir
5	INGRES	HP ProLiant DL380 G7	251,561	.38 USD	NR	03/31/11	Actian VectorWise 1.5	Re En Lir

300 GB Results

Rank	Company	System	QphH	Price/QphH	Watts/KQphH	System Availability	Database	Op S
1	lenovo FOR THOSE WHO DO.	Lenovo ThinkServer RD630	434,353	.24 USD	NR	05/10/13	VectorWise 3.0.0	Re En Lir

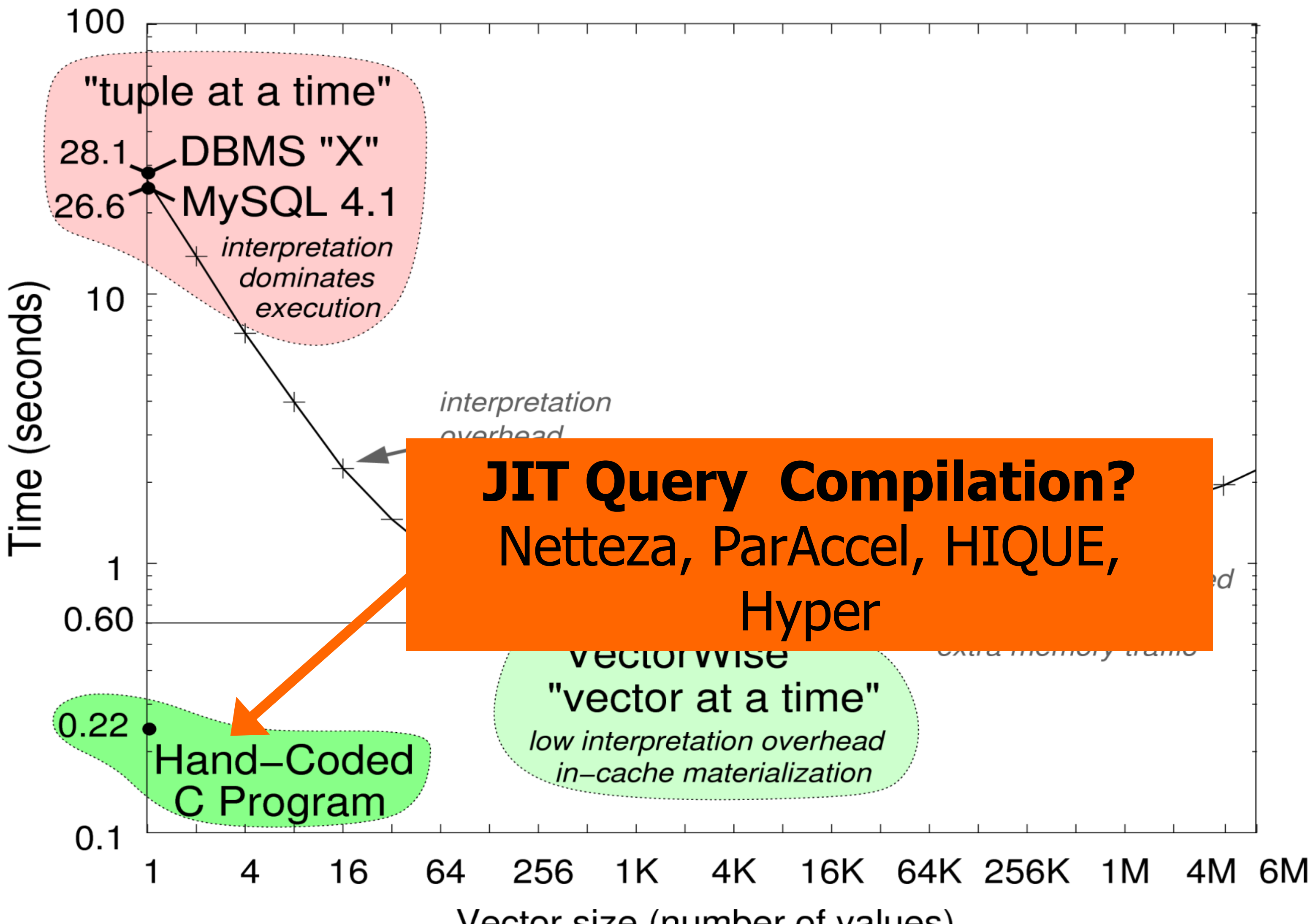

IBM
Industries & solutions
Services
Products
Support & downloads
My IBM
Search
IBM Software > Information Management > Data Management > DB2 Product Family > DB2 for Linux, UNIX and Windows >
DB2 with BLU Acceleration
Breakthrough analytics performance
Columnar store
Columnar store scans and locates relevant data based on columns inst
resulting in faster processing.


BLAZING-FAST PERFORMANCE:
A Technical Best Practices Tour with ColumnStore Index
Susan Price
Senior Program Manager
SQL Server 2012


Optimizing Transaction and Query Performance
Row Format Databases versus Column Format Databases
Transactions run faster on row format
Insert or query a sales order
Fast processing few rows, many columns
Analytics run faster on column format
Report on sales totals by state
Fast accessing few columns, many rows
Oracle 12c: Stores Data in Both Formats Simultaneously


SAP HANA® Solution
Redefines In-memory Computing

Query Compilation



JIT Query Compilation?
Netteza, ParAccel, HIQUE, Hyper

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.

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

→

```
for each tuple
   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.

```
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.

```
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++)
     result[category[sel[i]]] +=
        price[sel[i]]*(1+tax[sel[i]]);
```

Single-loop (last for-loop)

Multi-loop (all for-loops)

* Just an example. Not necessarily optimal.

Case studies

see: Damon2011 Sompolski et al.

- Projections
- Selections
- Hash lookups

Case studies

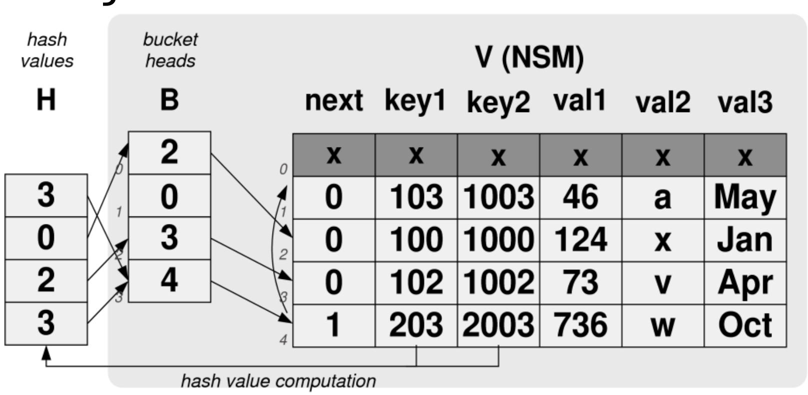
see: Damon2011 Sompolski et al.

Multi-loop on modern hardware:

- Projections → Easier SIMD
- Selections → Avoids branch mispredictions
- Hash lookups → Improves memory access pattern

Hash lookup algorithm

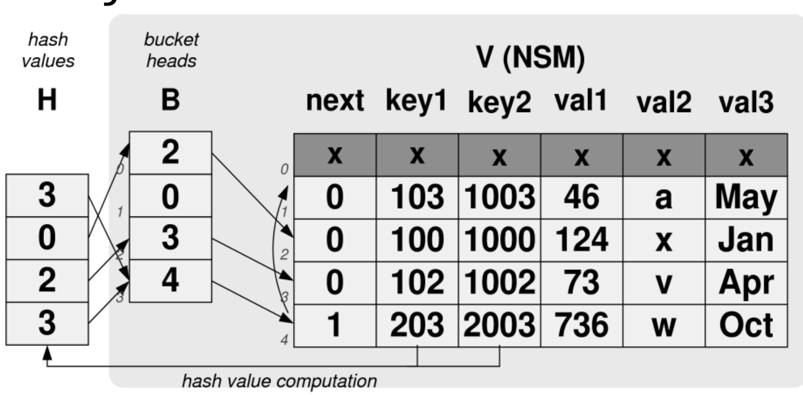
```
pos = B[hash_keys(probe_keys)]
if (pos) {
    do { // pos == 0 reserved for miss.
        if (keys_equal(probe_keys, V[pos].keys)) {
            fetch_value_columns(V[pos]);
            break; // match
        }
    } while(pos = next in chain); // collision or miss
}
```



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

```
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

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

High random access cost:

- **Both B and V are huge arrays**
    - Cache miss
    - TLB miss

single-loop compiled hash lookup: dependencies

```
for (i=0; i<n; i++) {
    pos = B[HASH(key1[i]
    if (pos) {
        do {
            if (key1[i]==V[p
                res1[i] = V[po
                res2[i] = V[pos].val2;
                res3[i] = V[pos].val3;
                break; // matc
            }
        } while (pos = V[p
    }
}
```

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++) {      A
    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
    }
}
```

Save the day with processor speculation?

- Always match and no collisions: A

single-loop compiled hash lookup: branch predictability

```
for (i=0; i<n; i++) {                A
        pos = B[HASH(key1[i]) ^ HASH(key2[i]) & SIZE];
   B    if (pos) {
            do {
       C        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: ABC

single-loop compiled hash lookup: branch predictability

```
for (i=0; i<n; i++) {                                           A
      pos = B[HASH(key1[i]) ^ HASH(key2[i]) & SIZE];
   B  if (pos) {
         do {
       C    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;
          D     break; // match
            }
         } while (pos = V[pos].next); // miss
      }
   }
```

- Always match and no collisions: ABCD

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: ABCD ABCD ...

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 mat

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

```
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

```
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 ABCECE..

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 ABCECE A...

single-loop compiled hash lookup: branch predictability

```
for (i=0; i<n; i++) {                                    A
      pos = B[HASH(key1[i]) ^ HASH(key2[i]) & SIZE];
B     if (pos) {
         do {
C           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                  D
            }
E        } while (pos = V[pos].next); // miss
      }
}
```

- Misses or collisions: AB AB ABCECE ABCECD 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 co

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

vectorized hash lookup

Good:

- **Independent loop iterations at each step.**

Bad:

- **Each step accessing a vector of positions all over again**

match[]

```
// 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]]);
```

} while (pos = V[pos].next); // miss

Fetch new pos[] from next in miss[]

Fetch v3 for match[]

Loop until pos[] empty

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

vectorized hash lookup

pos[0] →

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

```
// 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

pos[0]

pos[1]

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

```
// 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

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

pos[0]

pos[2]

pos[1]

```
// 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

	key1	key2	val1	val2	val3
	123	1003	3	a	May
pos[0]	100	2004	7	x	Jan
	102	1005	2	w	Oct
	103	1100	6	d	Nov
pos[2]	120	1234	9	e	Dec
	111	1010	0	r	Jan
pos[1]	150	1203	1	t	Jun
	105	1003	3	g	Oct
pos[n-1] →	103	1110	5	h	Sep

```
// 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

	key1	key2	val1	val2	val3
	123	1003	3	a	May
pos[0] →	100	2004	7	x	Jan
	102	1005	2	w	Oct
	103	1			
pos[2]	120	1			
	111	1			
pos[1]	150	1			
	105	1003	3	g	Oct
pos[n-1]	103	1110	5	h	Sep

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]]);
```

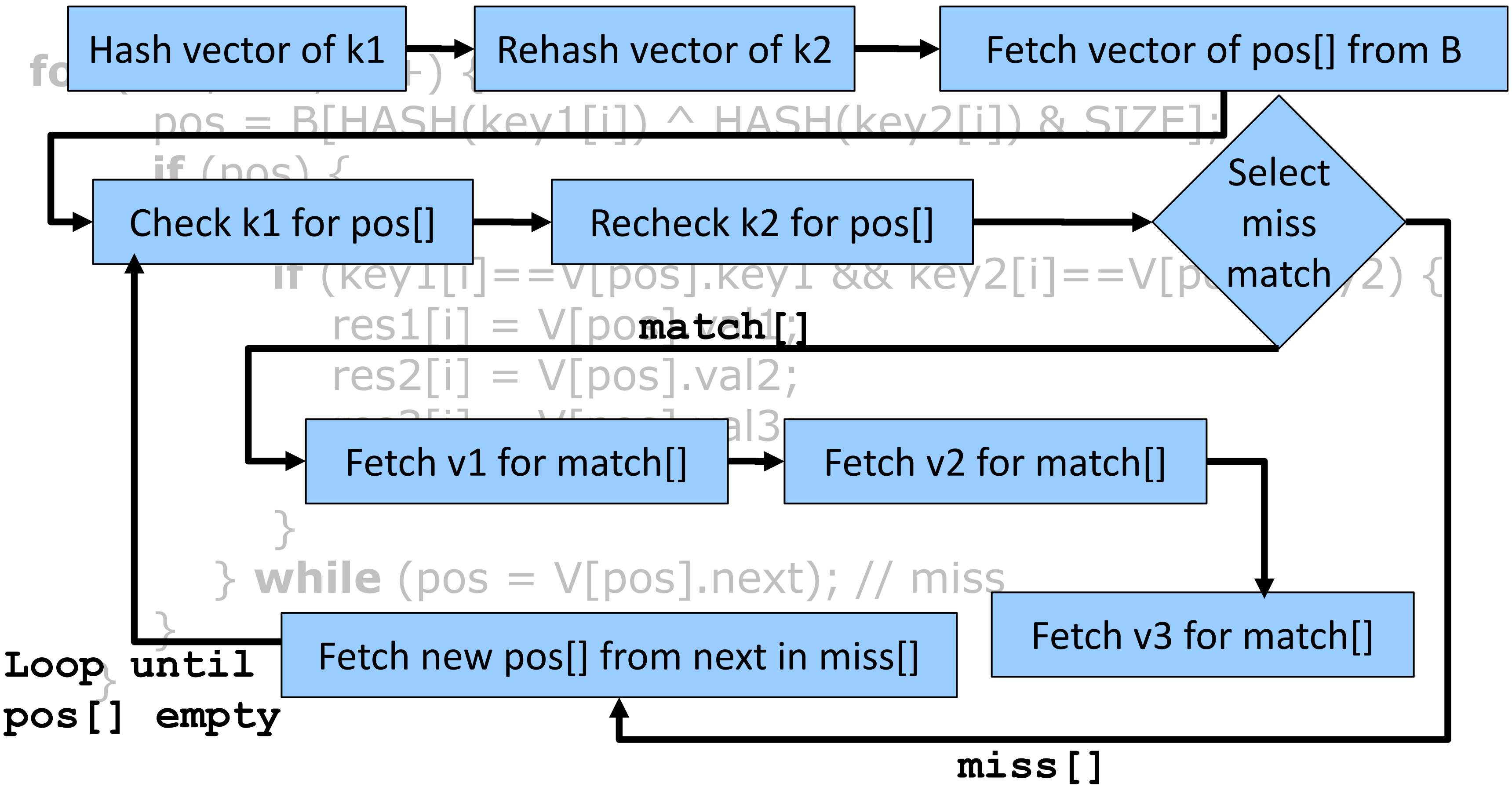
```
// base = &V[0].key2;
for(i=0;i<n;i++)
   res[i] |= (key[i] != base[stride * pos[i]]);
```

single-loop compiled hash lookup

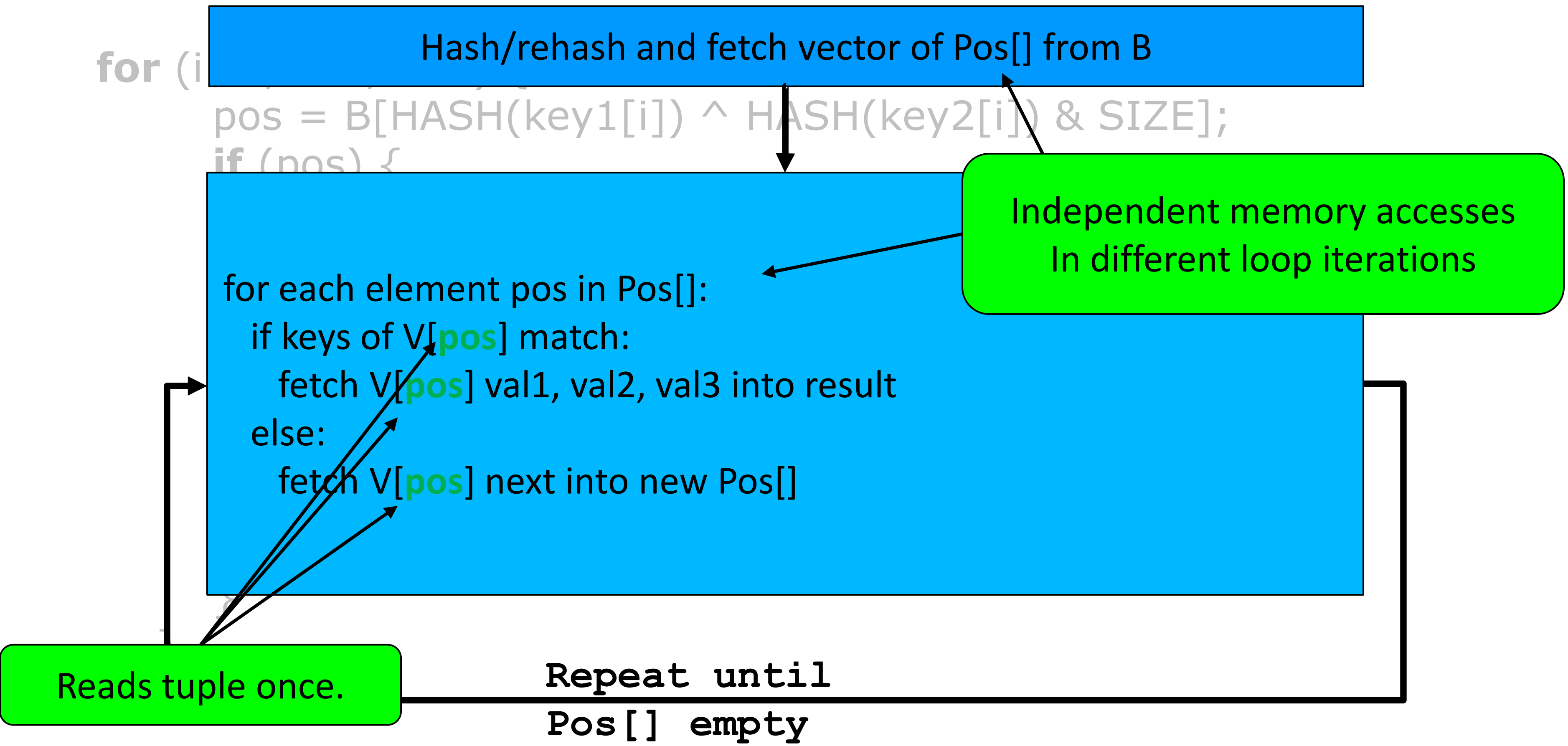
```
for (i=0; i<n; i++) {
        pos = B[HASH(key1[i]) ^ HASH(key2[i]) & SIZE];
        if (pos) {
            do {
                if (key1[i]==V[pos].key1 && key2[i]==V[pos].key2) {
                    res1[i] = V[pos].val1;
                    res2[i] = V[pos].val2;
                    res3[i] =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

Reads tuple once.

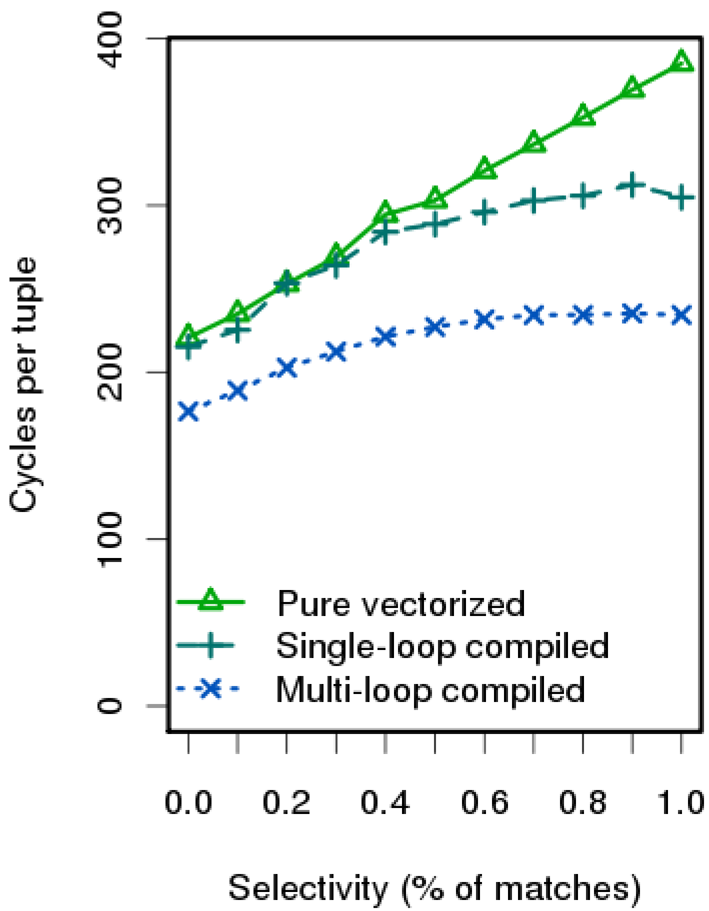
vectorized hash lookup



multi-loop compiled hash lookup

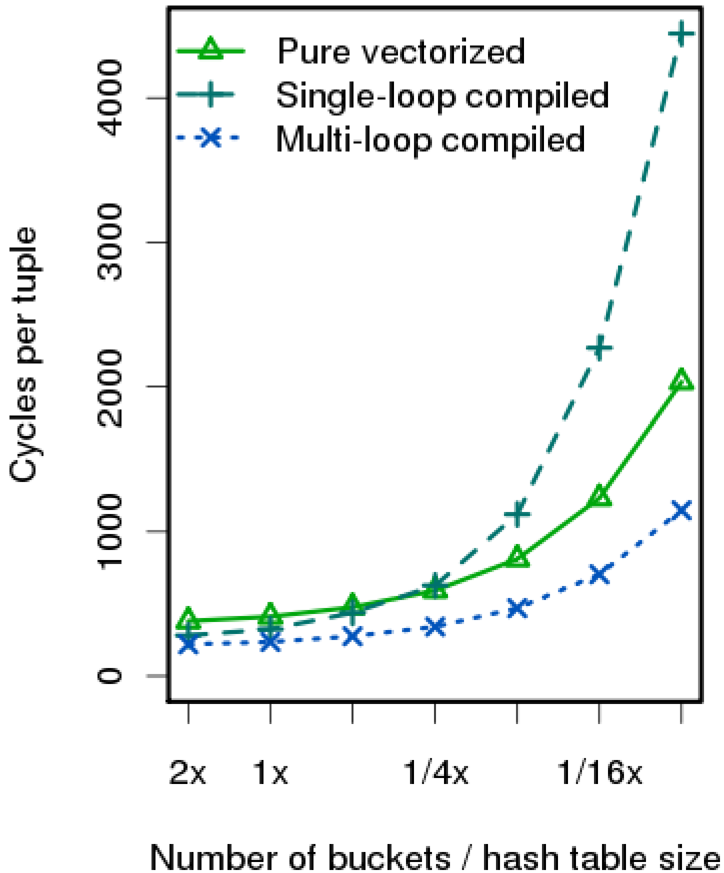


Hash lookup benchmarks



- 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**

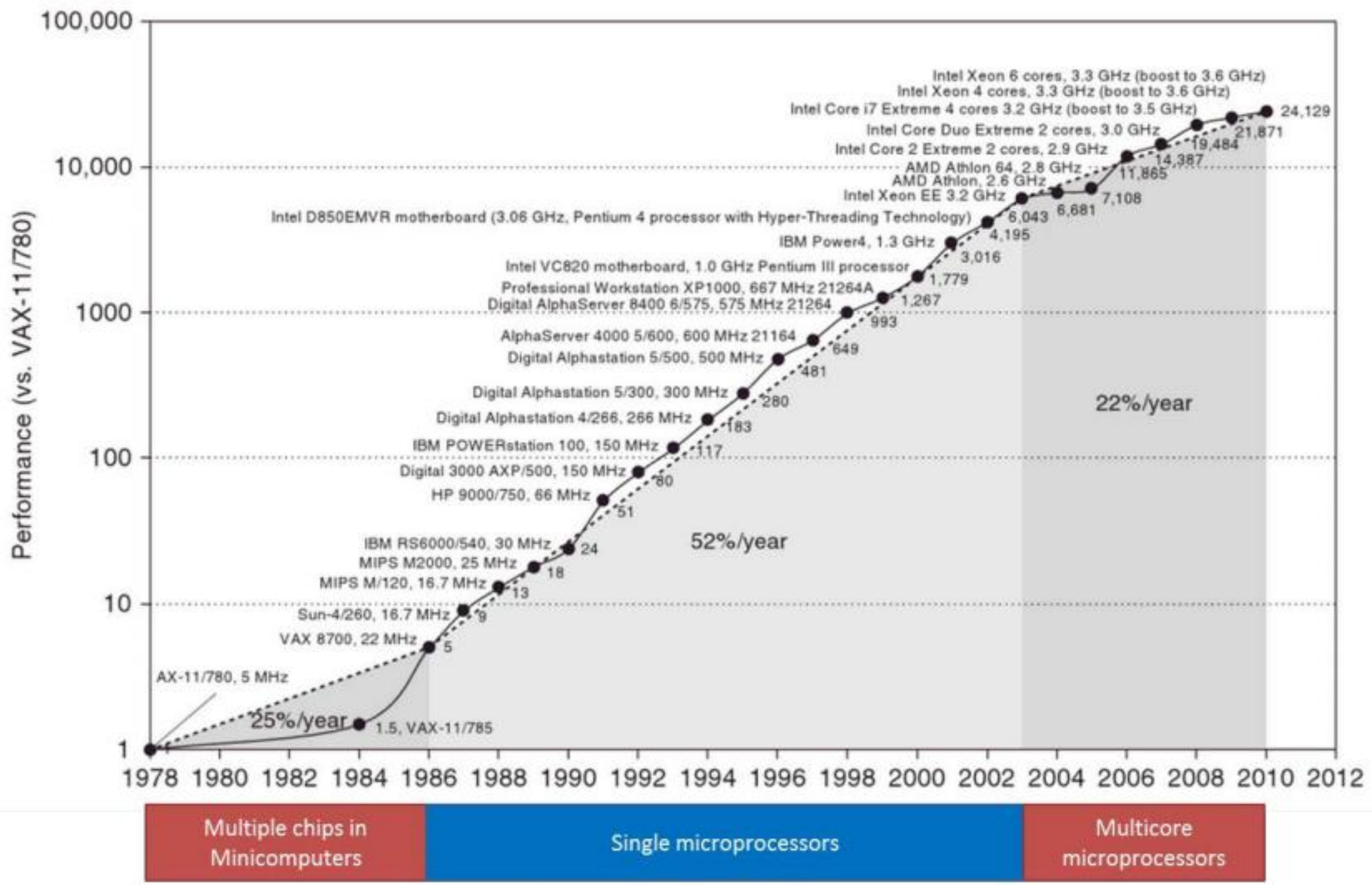
robust = '**good enough**' performance **all-the-time**

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



Centrum Wiskunde & Informatica

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?

Centrum Wiskunde & Informatica

Thank You!