

Data Processing on Modern Hardware

Jens Teubner, TU Dortmund, DBIS Group
`jens.teubner@cs.tu-dortmund.de`

Summer 2015

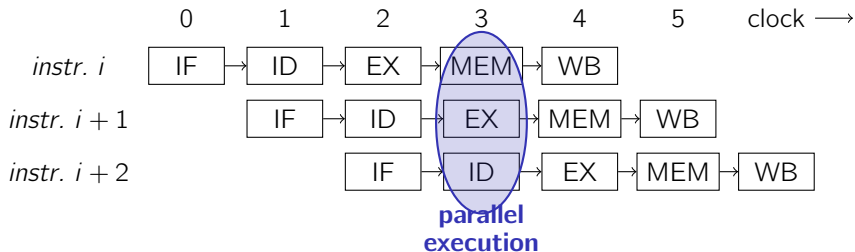
Part III

Instruction Execution

Pipelining in CPUs

Pipelining is a CPU implementation technique whereby multiple instructions are **overlapped in execution**.

- Break CPU instructions into smaller units and pipeline.
- *E.g.*, classical five-stage pipeline for RISC:



Ideally, a k -stage pipeline improves performance by a factor of k .

- Slowest (sub-)instruction determines clock frequency.
 - Ideally, break instructions into k equi-length parts.
- Issue one instruction per clock cycle ($\text{IPC} = 1$).

Example: Intel Pentium 4: 31+ pipeline stages.

Hazards

The effectiveness of pipelining is hindered by **hazards**.

Structural Hazard

Different pipeline stages need same **functional unit**
(resource conflict; *e.g.*, memory access \leftrightarrow instruction fetch)

Data Hazard

Result of one instruction not ready before access by later instruction.

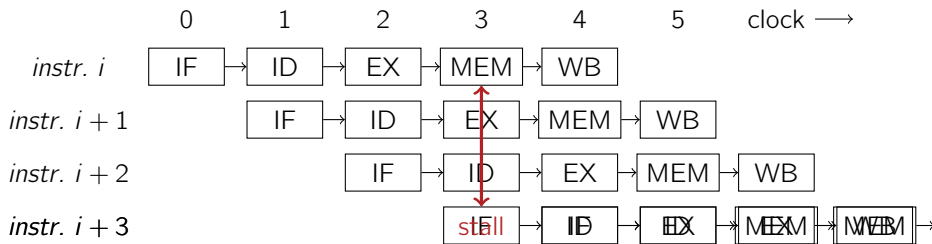
Control Hazard

Arises from branches or other instructions that modify PC
("data hazard on PC register").

Hazards lead to **pipeline stalls** that decrease IPC.

Structural Hazards

A **structural hazard** will occur if a CPU has only one memory access unit and *instruction fetch* and *memory access* are scheduled in the same cycle.



Resolution:

- **Provision** hardware accordingly (e.g., separate fetch units)
- **Schedule** instructions (at compile- or runtime)

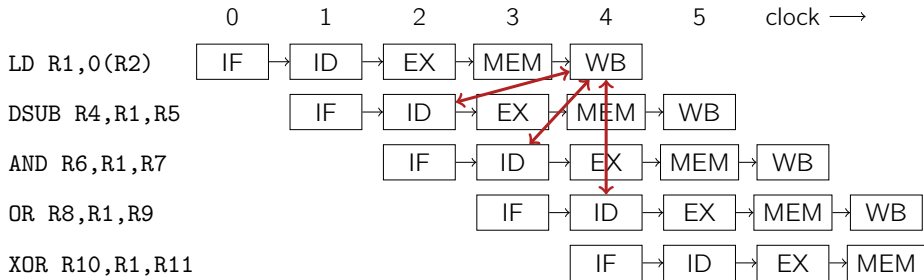
Structural hazards can also occur because functional units are **not fully pipelined**.

- *E.g.*, a (complex) floating point unit might not accept new data on every clock cycle.
- Often a space/cost \leftrightarrow performance trade-off.

Data Hazards

```
LD    R1, 0(R2)
DSUB  R4, R1, R5
AND   R6, R1, R7
OR    R8, R1, R9
XOR   R10, R1, R11
```

- Instructions read R1 before it was written by LD (stage WB writes register results).
- Would cause incorrect execution result.



Resolution:

- **Forward** result data from instruction to instruction.
 - Could resolve hazard LD \leftrightarrow AND on previous slide (forward R1 between cycles 3 and 4).
 - **Cannot** resolve hazard LD \leftrightarrow DSUB on previous slide.
- **Schedule** instructions (at compile- or runtime).
 - Cannot avoid all data hazards.
- Detecting data hazards can be hard, *e.g.*, if they go through memory.

```
SD  R1, 0(R2)
LD   R3, 0(R4)
```

Tight **loops** are a good candidate to improve instruction scheduling.

```
for (i = 1000; i > 0; i = i - 1)
    x[i] = x[i] + s;
```

```
1: L.D      F0, 0(R1)
   ADD.D    F4, F0, F2
   S.D      F4, 0(R1)
   DADDUI   R1, R1, #-8
   BNE      R1, R2, 1
```

naïve code

```
1: L.D      F0, 0(R1)
   DADDUI   R1, R1, #-8
   ADD.D    F4, F0, F2
   stall
   stall
   S.D      F4, 8(R1)
   BNE      R1, R2, 1
```

re-schedule

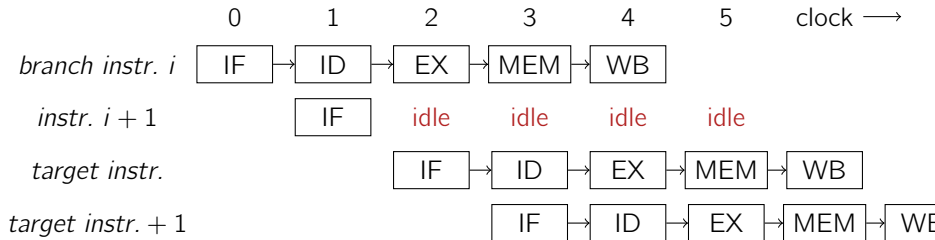
```
1: L.D      F0, 0(R1)
   L.D      F6, -8(R1)
   L.D      F10, -16(R1)
   L.D      F14, -24(R1)
   ADD.D    F4, F0, F2
   ADD.D    F8, F6, F2
   ADD.D    F12, F10, F2
   ADD.D    F16, F14, F2
   S.D      F4, 0(R1)
   S.D      F8, -8(R1)
   DADDUI   R1, R1, #-32
   S.D      F12, 16(R1)
   S.D      F16, 8(R1)
   BNE      R1, R2, 1
```

loop unrolling

Control Hazards

Control hazards are often more severe than are data hazards.

- Most simple implementation: **flush pipeline, redo instr. fetch**



With increasing pipeline depths, the penalty gets **worse**.

A simple optimization is to **only** flush if the branch was **taken**.

- Penalty only occurs for taken branches.
- If the two outcomes have different (known) likeliness:
 - Generate code such that a non-taken branch is more likely.
- Aborting a running instruction is harder when the branch outcome is known late.
 - Should not change **exception behavior**.

This scheme is called **predicted-untaken**.

→ Likewise: **predicted-taken** (but often less effective)

Branch Prediction

Modern CPUs try to **predict** the target of a branch and execute the target code **speculatively**.

- Prediction must happen **early** (ID stage too late).

Thus: **Branch Target Buffers (BTBs)**

- Lookup Table: PC \rightarrow \langle predicted target, taken? \rangle .

Lookup PC	Predicted PC	Taken?
:	:	:

- Consult Branch Target Buffer **parallel to instruction fetch**.
 - If entry for current PC can be found: follow prediction.
 - If not, create entry after branching.
- Inner workings of modern branch predictors are highly involved (and typically kept secret).

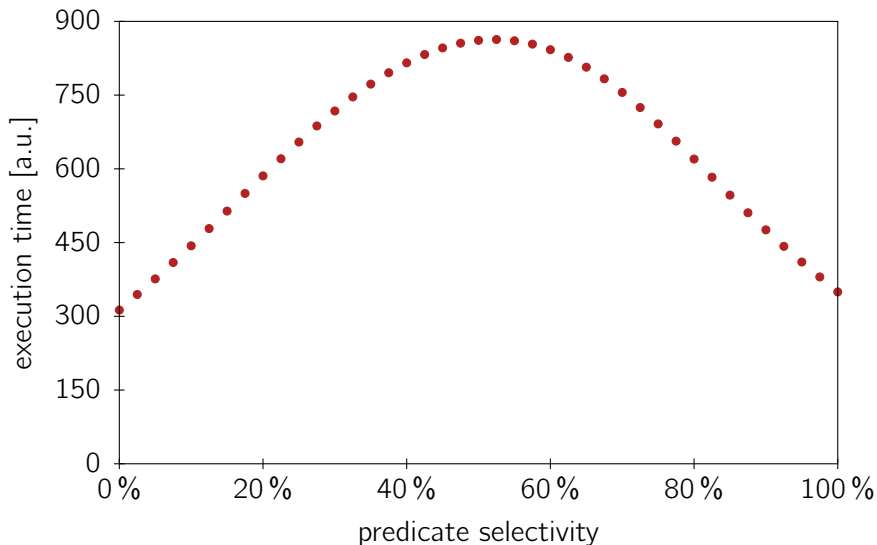
Selection queries are sensitive to branch prediction:

```
SELECT COUNT(*)  
  FROM lineitem  
 WHERE quantity < n
```

Or, written as C code:

```
for (unsigned int i=0; i<num_tuples; i++)  
  if (lineitem[i].quantity<n)  
    count++;
```

Selection Conditions (Intel Q6700)



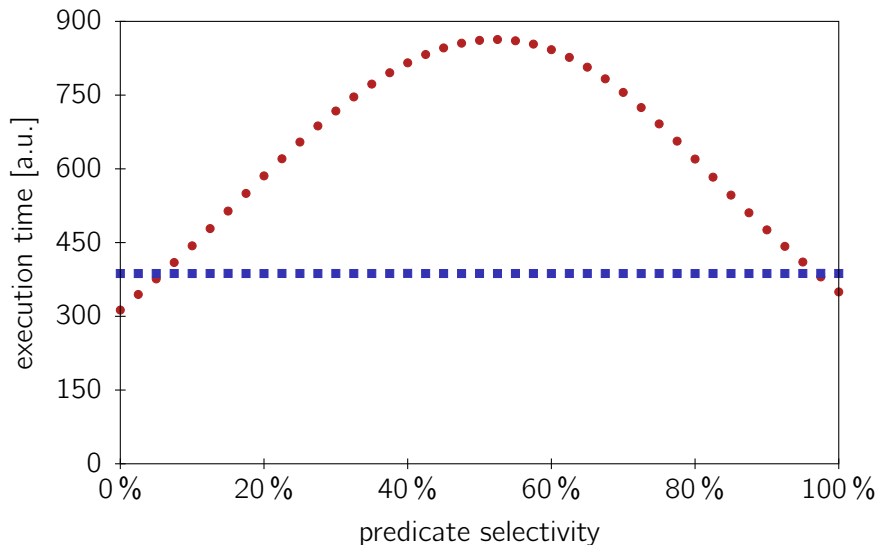
Predication: Turn **control flow** into **data flow**.

```
for (unsigned int i=0; i<num_tuples; i++)  
    count += (lineitem[i].quantity < n);
```

- This code does **not** use a branch any more.³
- The price we pay is a + operation for **every** iteration.
- Execution cost should now be **independent** of predicate selectivity.

³except to implement the loop

Predication (Intel Q6700)



This was an example of **software predication**.

 **How about this query?**

```
SELECT quantity
FROM lineitem
WHERE quantity <  $n$ 
```

Some CPUs also support **hardware predication**.

- *E.g.*, Intel Itanium2:

Execute **both** branches of an if-then-else and discard one result.

Experiments (AMD AthlonMP / Intel Itanium2)

```
int sel_lt_int_col_int_val(int n, int* res, int* in, int V) {
```

```
  for(int i=0, j=0; i<n; i++){
```

```
    /* branch version */
```

```
    if (src[i] < V)
```

```
      out[j++] = i;
```

```
    /* predicated version */
```

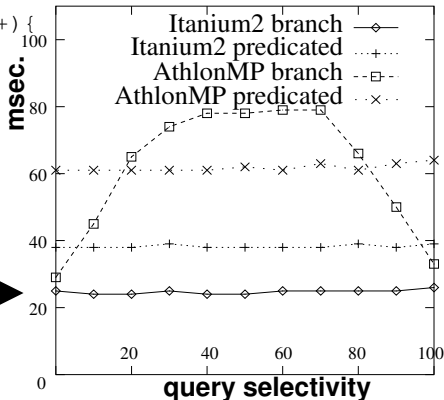
```
    bool b = (src[i] < V);
```

```
    out[j] = i;
```

```
    j += b;
```

```
  }  
  return j;
```

```
}
```



↗Boncz, Zukowski, Nes. MonetDB/X100: Hyper-Pipelining Query Execution. *CIDR 2005*.

Two Cursors

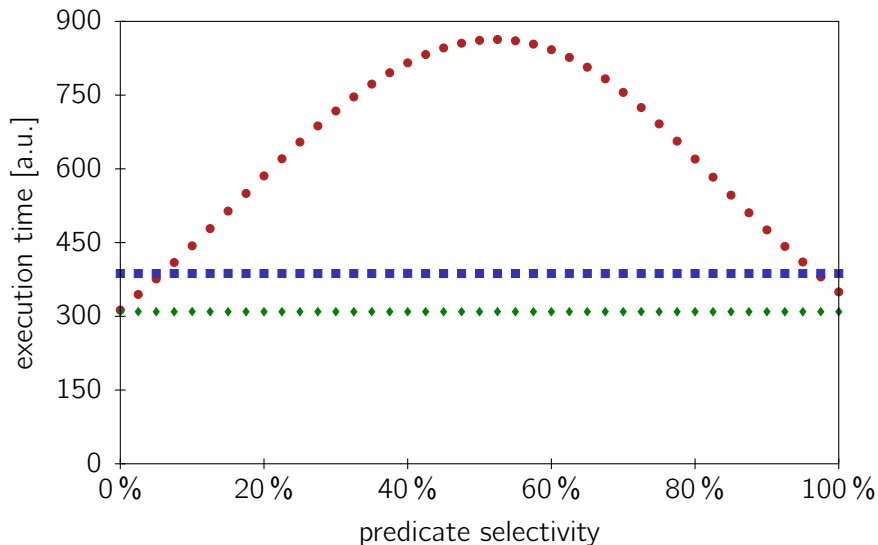
The `count += ...` still causes a **data hazard**.

- This limits the CPUs possibilities to execute instructions in parallel.

Some tasks can be rewritten to use **two cursors**:

```
for (unsigned int i=0; i<num_tuples / 2; i++) {  
    count1 += (data[i] < n);  
    count2 += (data[i + num_tuples / 2] < n);  
}  
count = count1 + count2;
```

Experiments (Intel Q6700)



Conjunctive Predicates

In general, we have to handle multiple predicates:

```
SELECT  $A_1, \dots, A_n$   
FROM  $R$   
WHERE  $p_1$  AND  $p_2$  AND ... AND  $p_k$ 
```

The standard C implementation uses `&&` for the conjunction:

```
for (unsigned int i=0; i<num_tuples; i++)  
    if ( $p_1$  &&  $p_2$  && ... &&  $p_k$ )  
        ...;
```

Conjunctive Predicates

The `&&` introduce even more branches. The use of `&&` is equivalent to:

```
for (unsigned int i=0; i<num_tuples; i++)  
    if ( $p_1$ )  
        if ( $p_2$ )  
             $\vdots$   
            if ( $p_k$ )  
                ...;
```

An alternative is the use of the logical `&`:

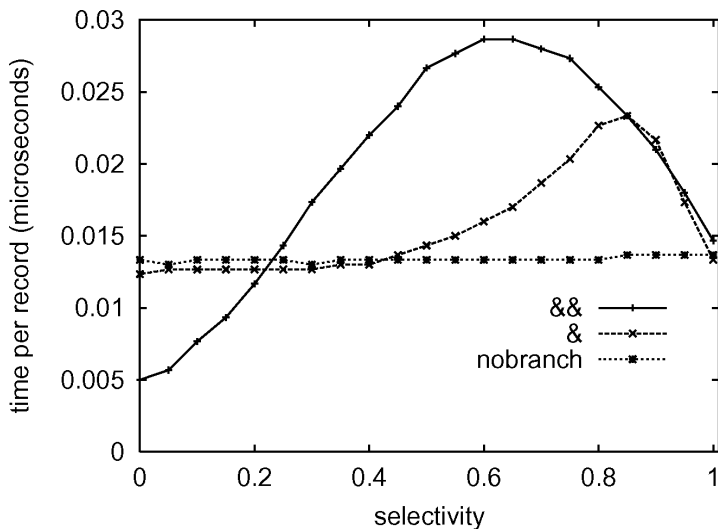
```
for (unsigned int i=0; i<num_tuples; i++)  
    if ( $p_1$  &  $p_2$  & ... &  $p_k$ )  
        ...;
```

Conjunctive Predicates

This allows us to express queries with conjunctive predicates without branches.

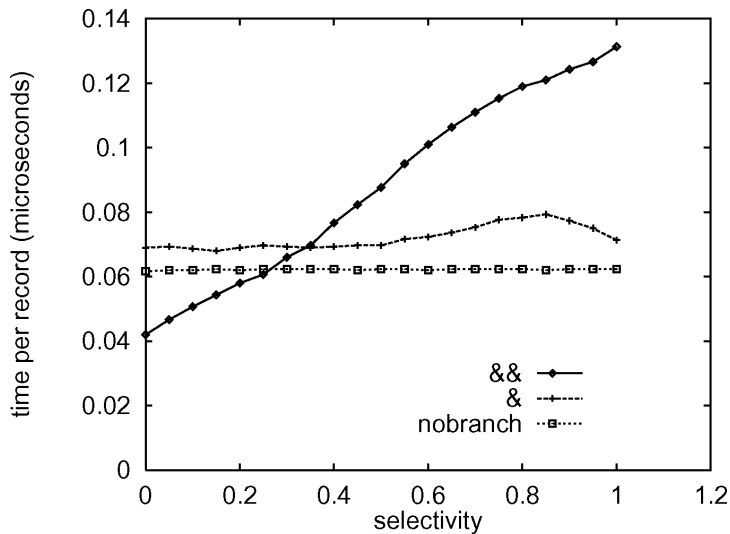
```
for (unsigned int i=0; i<num_tuples; i++)  
{  
    answer[j] = i;  
    j += ( $p_1$  &  $p_2$  & ... &  $p_k$ );  
}
```


Experiments (Intel Pentium III)



Ken Ross. Selection Conditions in Main Memory. *TODS 2004*.

Experiments (Sun UltraSparc Ili)



↗ Ken Ross. Selection Conditions in Main Memory. *TODS 2004*.

Cost Model

A query compiler could use a **cost model** to select between variants.

$p \ \&\& \ q$

When p is highly selective, this might amortize the double branch misprediction risk.

$p \ \& \ q$

Number of branches halved, but q is evaluated regardless of p 's outcome.

$j += \dots$

Performs memory write in **each** iteration.

Notes:

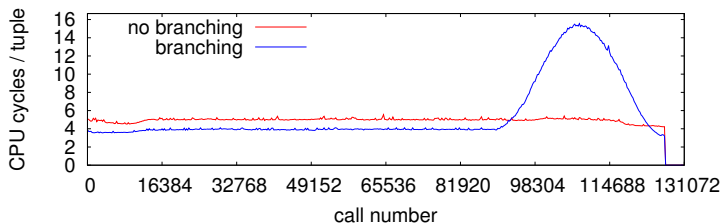
- Sometimes, $\&\&$ is necessary to prevent null pointer dereferences: `if (p && p->foo == 42).`
- Exact behavior is hardware-specific.

Cost Model

Unfortunately, predicting the cost of a variant might be **hard**.

→ Many parameters involved: characteristics of data, machine, workload, etc.

E.g., branching vs. no-branching in TPC-H Q12:



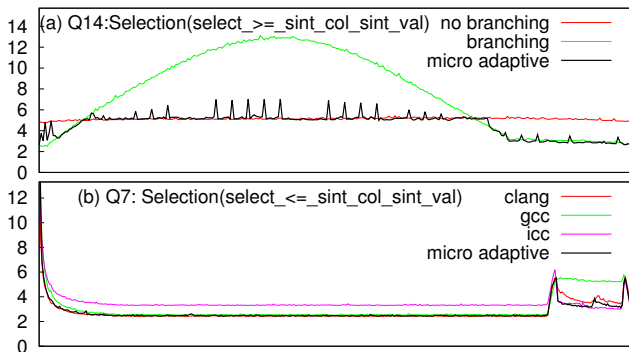
↗ Raducanu and Boncz. Micro Adaptivity in Vectorwise. *SIGMOD 2013*.

Idea:

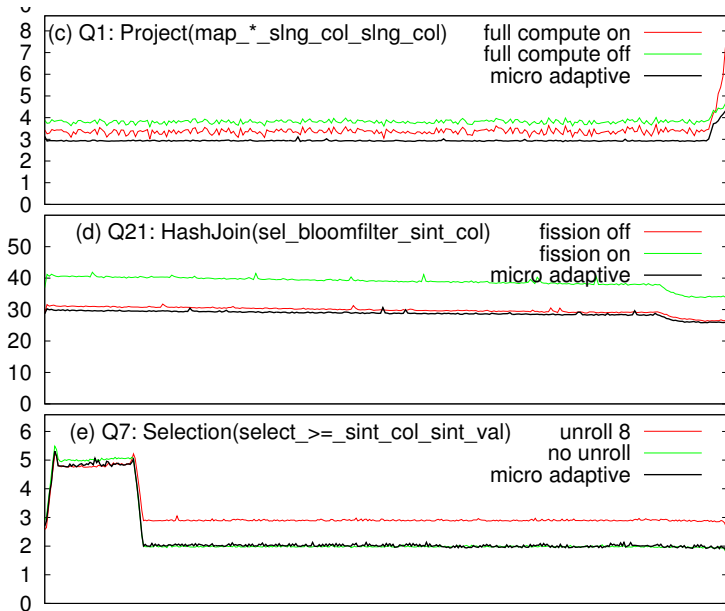
- Generate **variants** of primitive operators.
 - with/without branching
 - different compilers
 - operator parameters (hash table configuration, etc.)
- Try to **learn** cost model for each variant.
- **Exploit and explore:**
 - **Profile** every execution to refine the cost model.
 - Choose variant based on cost model (exploit), **but** with a small probability choose a **random variant** (explore).

Vector-At-A-Time Execution:

- Re-consider variant choice **for every n vectors**.
- **Adapt** to specifics of the particular query/operator.
- Even adjust to **varying characteristics** as the query progresses.




Experiments



Use Case: (De-)Compression

Compression can help overcome the **I/O bottleneck** of modern CPUs.

- disk ↔ memory
- memory ↔ cache (!)
- Column stores have high potential for compression.  **Why?**

But:

- (De-)compression has to be **fast**.
- 200–500 MB/s (LZRW1 and LZOP) won't help us much.
- Aim for **multi-gigabyte per second** decompression speeds.
- Maximum compression rate is **not** a goal.

Lightweight Compression Schemes

MonetDB/X100 implements lightweight compression schemes:

PFOR (Patched Frame-of-Reference)

small integer values that are positive offsets from a base value; one base value per (disk) block

PFOR-DELTA (PFOR on Deltas)

encode **differences** between subsequent items using PFOR

PDICT (Patched Dictionary Compression)

integer codes refer into an array for values (the dictionary)

All three compression schemes allow **exceptions**, values that are too far from the base value or not in the dictionary.

PFOR Compression

E.g., compress the digits of π using 3-bit PFOR compression.

header								3	1	} compressed data
4	1	5	⊥	2	6	5	3	5	⊥	
⊥	⊥	⊥	3	2						
										} exceptions
9		7		9		8		9		

Decompressed numbers: 31415926535897932

Decompression

During decompression, we have to consider all the exceptions:

```
for (i=j=0; i<n; i++)  
    if (code[i] !=  $\perp$ )  
        output[i] = DECODE (code[i]);  
    else  
        output[i] = exception[--j];
```

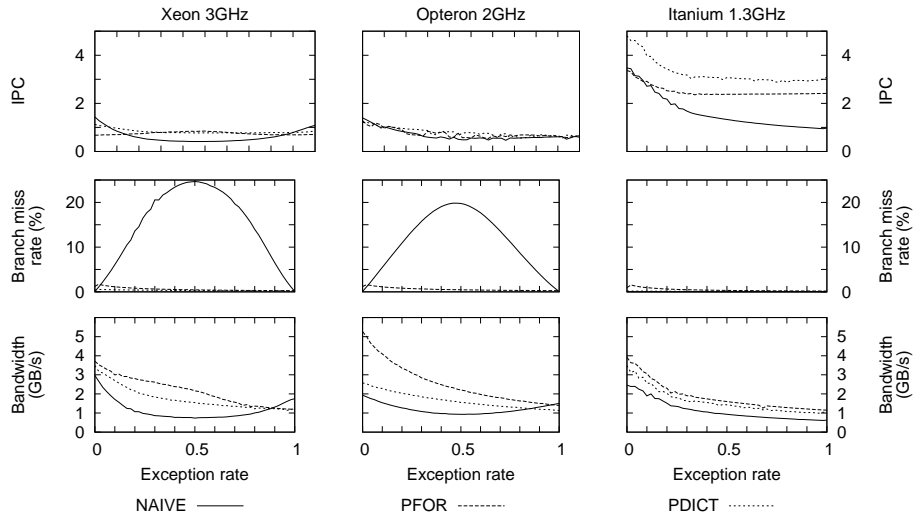
For PFOR, DECODE is a simple addition:

```
#define DECODE(a) ((a) + base_value)
```



The **branch** in the above code may bear a high **misprediction risk**.

Misprediction Cost



Source: M. Żukowski. Balancing Vectorized Query Execution with Bandwidth-Optimized Storage. PhD Thesis, University of Amsterdam. Sept. 2009

Avoiding the Misprediction Cost

Like with predication, we can avoid the high misprediction cost if we're willing to invest some unnecessary work.

Run decompression in **two phases**:

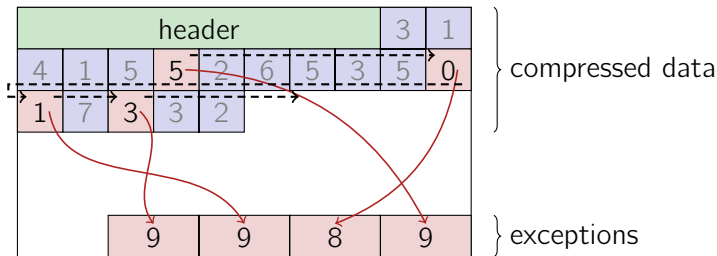
- 1 **Decompress** all regular fields, but don't care about exceptions.
- 2 Work in all the exceptions and **patch** the result.

```
/* ignore exceptions during decompression */  
for (i=0; i<n; i++)  
    output[i] = DECODE (code[i]);  
  
/* patch the result */  
foreach exception  
    patch corresponding output item ;
```

Patching the Output

🖋️ We **don't** want to use a branch to find all exception targets!

Thus: interpret values in “exception holes” as **linked list**:



→ Can now traverse exception holes and patch in exception values.

Patching the Output

The resulting decompression routine is branch-free:

```
/* ignore exceptions during decompression */  
for (i=0; i<n; i++)  
    output[i] = DECODE (code[i]);  
  
/* patch the result (traverse linked list) */  
j=0;  
for (cur=first_exception; cur<n; cur=next) {  
    next = cur + code[cur] + 1;  
    output[cur] = exception[--j];  
}
```

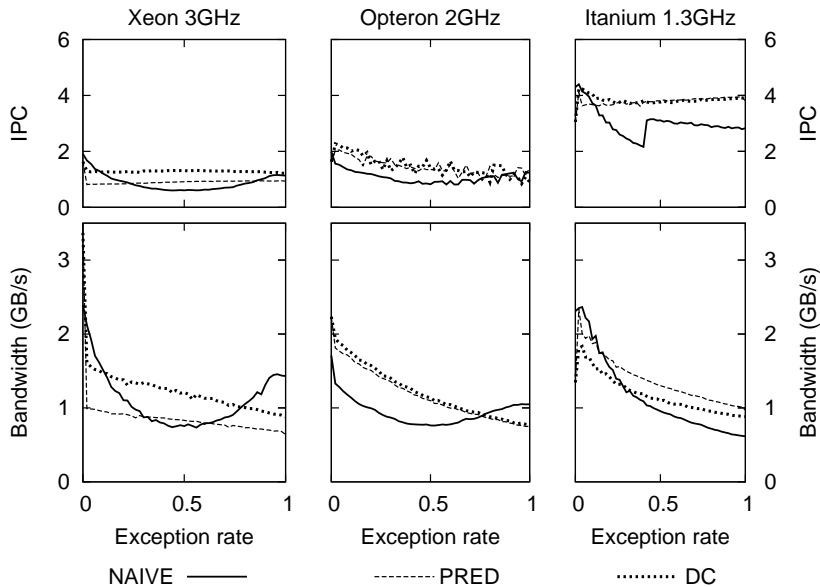
→ See slide 124 for experimental data on two-loop decompression.



3-bit-PFOR-compressed representation of the digits of e ?

$e = 2.718\ 281\ 828\ 459\ 045\ 235\ 360\ 287\ 471\ 352\ 662\ 497\ 757\ 247\ 093\ 699\ 959\ 574\ 966\ 967\ 627\ 724\ 076\ 630\ 353\ 547\ 594\ 571\ 382\ 178\ 525\ \dots$

PFOR Compression Speed

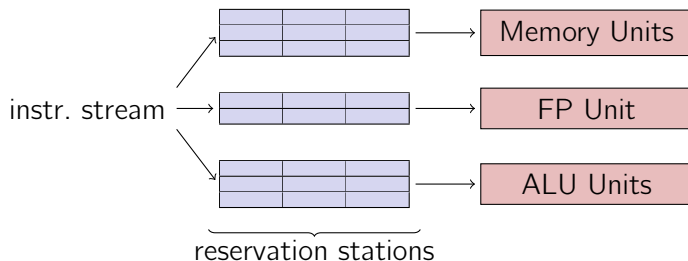


Source: M. Żukowski. Balancing Vectorized Query Execution with Bandwidth-Optimized Storage. PhD Thesis, U Amsterdam. 2009.

- The actual execution of instructions is handled in individual **functional units**
 - *e.g.*, load/store unit, ALU, floating point unit.
 - Often, some units are replicated.
- Chance to execute **multiple instructions** at the same time.
- Intel's Nehalem, for instance, can process **up to 4 instructions** at the same time.
 - IPC can be as high as 4.
- Such CPUs are called **superscalar CPUs**.

Dynamic Scheduling

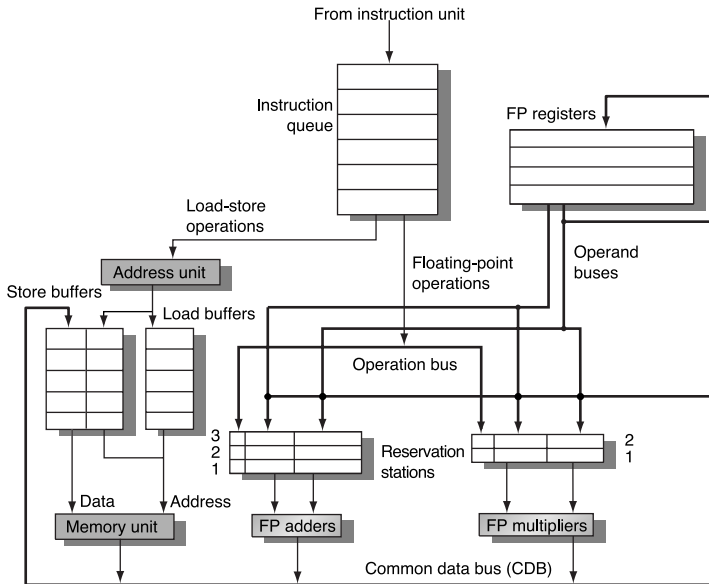
Higher IPCs are achieved with help of **dynamic scheduling**.



- Instructions are **dispatched** to **reservation stations**.
- They are **executed** as soon as all hazards are cleared.
- **Register renaming** helps to reduce data hazards.

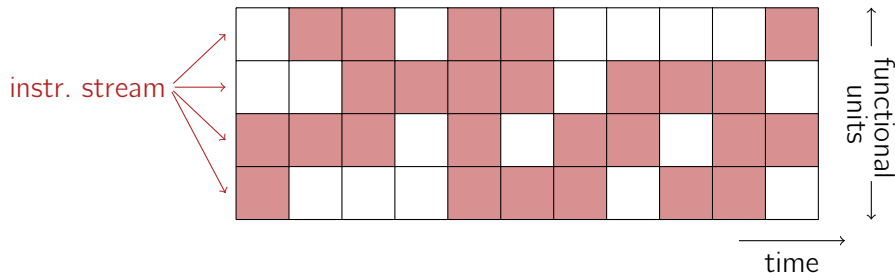
This technique is also known as **Tomasulo's algorithm**.

Example: Dynamic Scheduling in MIPS



Instruction-Level Parallelism

Usually, not all units can be kept busy with a single instruction stream:

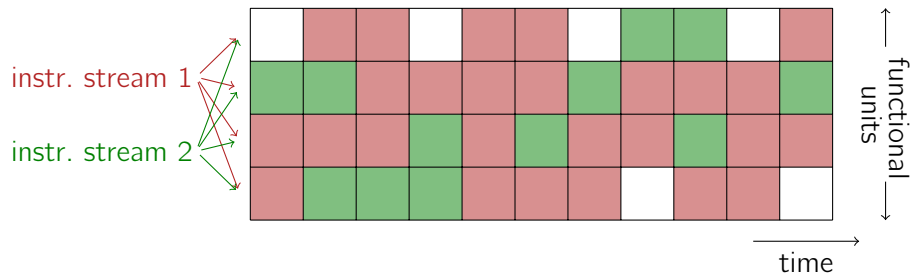


Reasons:

- data hazards, cache miss stalls, ...

Thread-Level Parallelism

Idea: Use the spare slots for an **independent instruction stream**.



This technique is called **simultaneous multithreading**.⁴

Surprisingly few changes are required to implement it.

- Tomasulo's algorithm requires **virtual registers** anyway.
- Need separate fetch units for both streams.

⁴Intel uses the term "hyperthreading."

Threads **share** most of their resources:

- **caches** (all levels),
- branch prediction functionality (to some extent).

This may have **negative effects**...

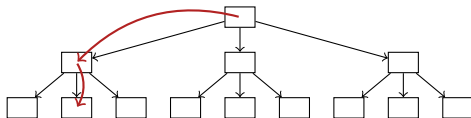
- threads that **pollute** each other's caches

...but also **positive effects**.

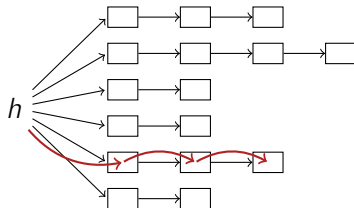
- threads that **cooperatively** use the cache?

Use Cases

Tree-based indexes:



Hash-based indexes:

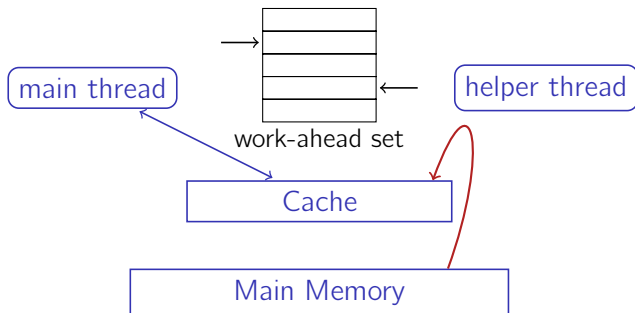


Both cases depend on hard-to-predict **pointer chasing**.

Helper Threads

Idea:

- Next to the main processing thread run a **helper thread**.
- Purpose of the helper thread is to **prefetch** data.
- Helper thread **works ahead** of the main thread.



- Main thread populates **work-ahead set** with pointers to prefetch.

Consider the traversal of a tree-structured index:

```
1 foreach input item do  
2   |   read root node, prefetch level 1 ;  
3   |   read node on tree level 1, prefetch level 2 ;  
4   |   read node on tree level 2, prefetch level 3 ;  
   |   :  
   └──
```

Helper thread will not have enough time to prefetch.

Thus: Process input in groups.

```
1 foreach group  $g$  of input items do  
2   foreach item in  $g$  do  
3     └ read root node, prefetch level 1 ;  
4   foreach item in  $g$  do  
5     └ read node on tree level 1, prefetch level 2 ;  
6   foreach item in  $g$  do  
7     └ read node on tree level 2, prefetch level 3 ;  
   ⋮
```


Data may now have arrived in caches by the time we reach next level.

Helper Thread

Helper thread accesses addresses listed in work-ahead set, *e.g.*,

```
temp += *((int *) p);
```

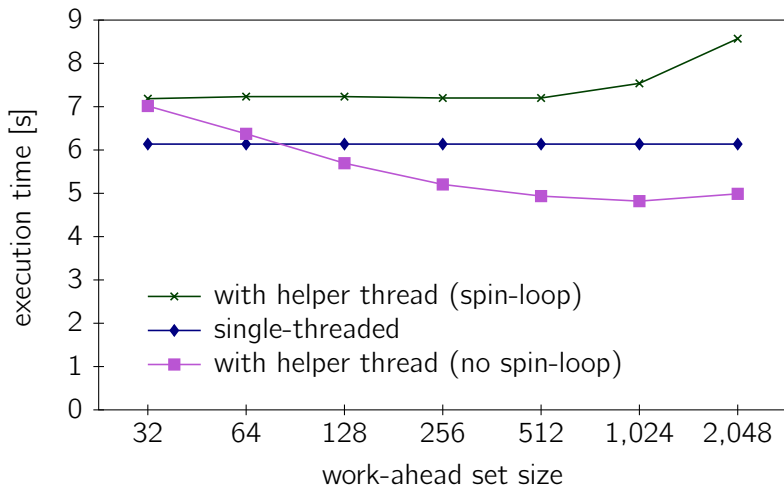
- Purpose: load data into caches

 **Why not use prefetchxx assembly instructions?**

- Only **read** data; do **not** affect semantics of main thread.
- Use a **ring buffer** for work-ahead set.
- **Spin-lock** if helper thread is too fast.

 **Which thread is going to be the faster one?**

Experiments (Tree-Structured Index)



↗ Zhou, Cieslewicz, Ross, Shah. Improving Database Performance on Simultaneous Multithreading Processors. *VLDB 2005*.

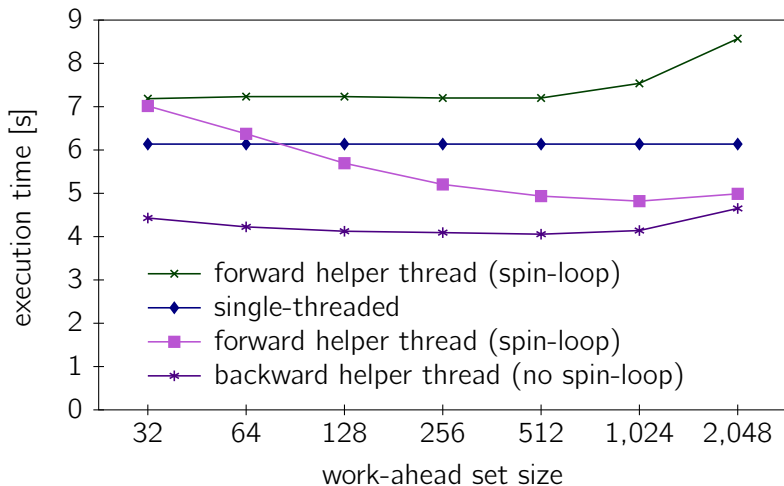
There's a high chance that both threads access the **same cache line at the same time**.

- Must ensure **in-order** processing.
- CPU will raise a **Memory Order Machine Clear (MOMC)** event when it detects parallel access.
 - Pipelines flushed to guarantee in-order processing.
 - MOMC events cause a high penalty.
- Effect is worst when the helper thread spins to wait for new data.

Thus:

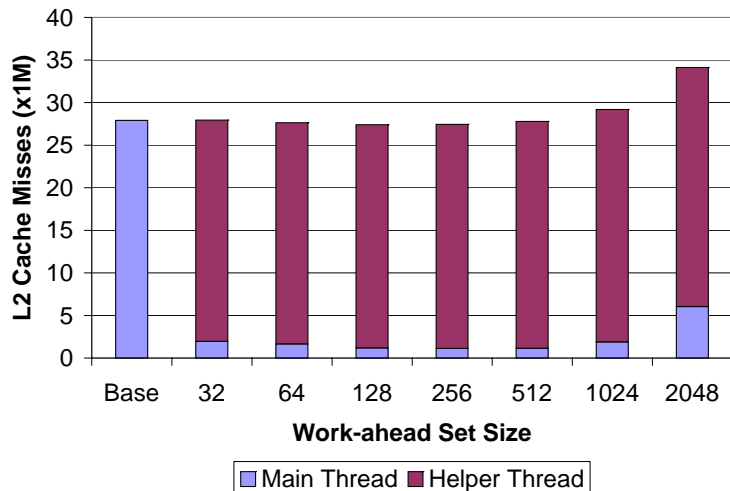
- Let helper thread work **backward**.

Experiments (Tree-Structured Index)



↗ Zhou, Cieslewicz, Ross, Shah. Improving Database Performance on Simultaneous Multithreading Processors. *VLDB 2005*.

Cache Miss Distribution



Source: Zhou *et al.* Improving Database Performance on Simultaneous Multithreading Processors. VLDB 2005.