

# Data Processing on Modern Hardware

Jana Giceva

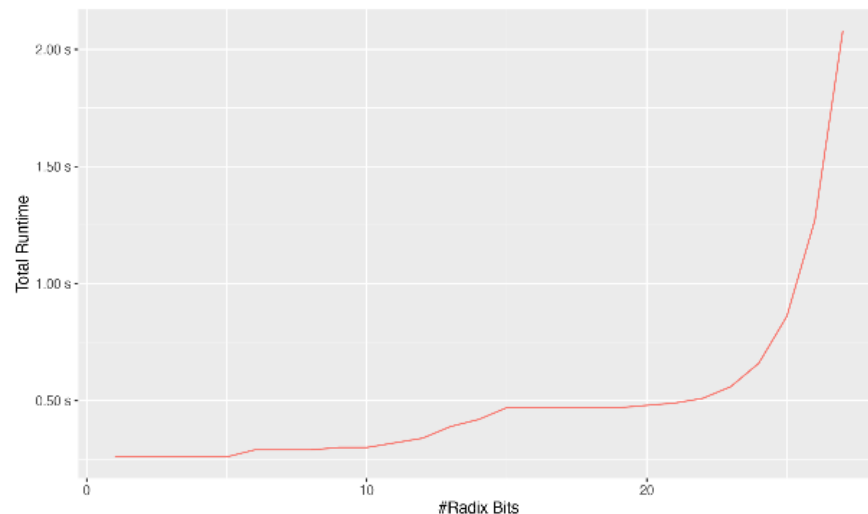
Lecture 5: Instruction execution



# Radix partitioning – sample exam question

- The plot shows the **total runtime** of a **naïve radix partitioning**.
- We vary the number of radix bits (x-axis)

Machine	
• 10 Cores (20 Threads)	
- L1d: 10x 2 <sup>15</sup> B	
- L1i: 10x 2 <sup>15</sup> B	
- L2: 10x 2 <sup>20</sup> B	
- L3: 2 <sup>23.3</sup> B	
• RAM: 64 GB	
• Pagesize: 4 KB	
• L1 TLB: 64 Entries	
• L2 TLB: 1536 Entries	

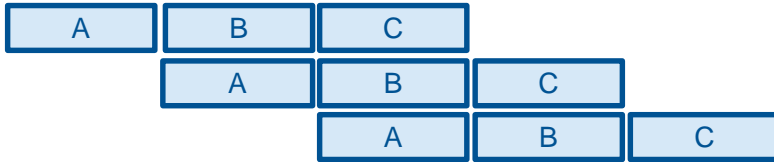


## ■ Questions:

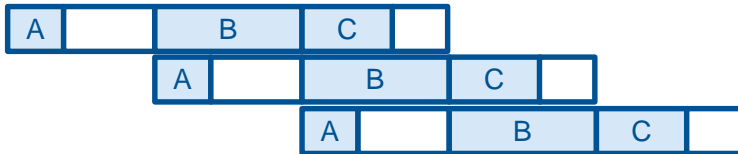
- What causes the step-wise decrease in performance?
- What is the idea of **software-managed buffers** and how can it help?
- How can we further optimize performance?

# Pipelining in CPUs

- Pipelining is a CPU implementation technique where multiple instructions are **overlapped in execution**
  - Break CPU instructions into smaller units and connect them in a pipe



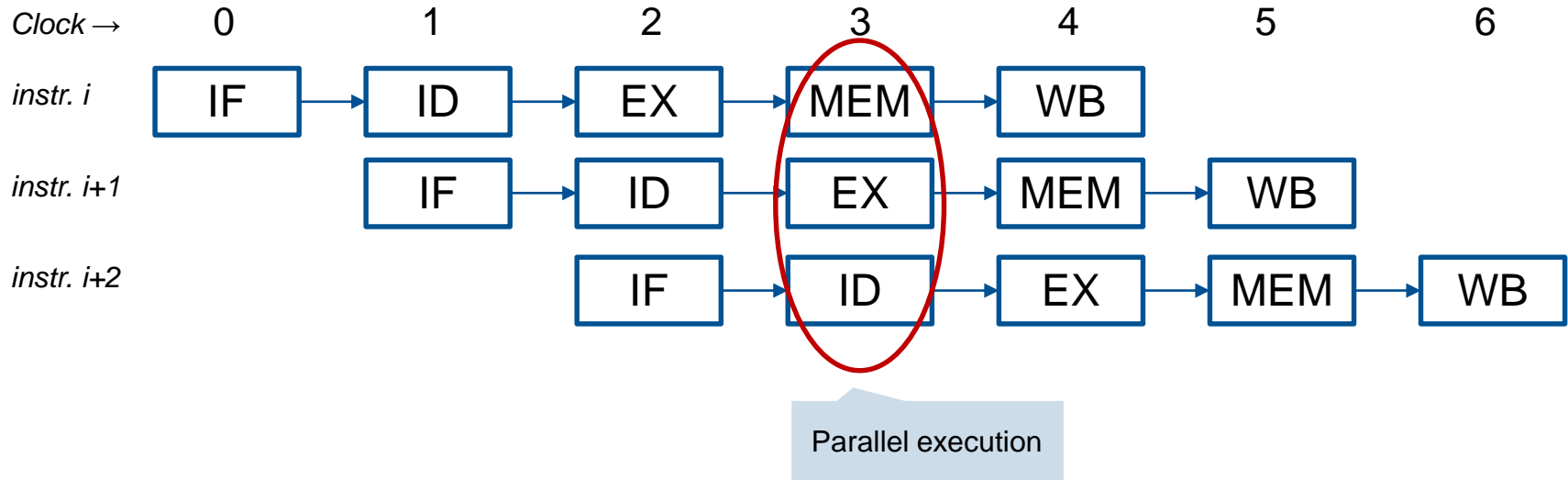
- Ideally, a k-stage pipeline improves the throughput performance by a factor of k.
- Slowest (sub-) instruction determines the clock frequency → danger of non-uniform stage delays



- Ideally, break instructions into k equi-length parts
- and reduce the number of cycles it takes to execute an instruction (i.e., the CPI).

# Pipelining in CPUs

- An example is the classical five-stage pipeline for RISC:
  - Every instruction can be implemented in, at most, 5 cycles with the following stages (clock cycles):
  - IF: Instruction Fetch, ID: Instruction Decode, EX: Execution, Mem: Memory Access, WB: Write-back



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

- **Structural hazard**

- Different pipeline stages needs the same **functional unit**
- (resource conflict: e.g., memory access ↔ instruction fetch)

- **Data hazard**

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

- **Control hazard**

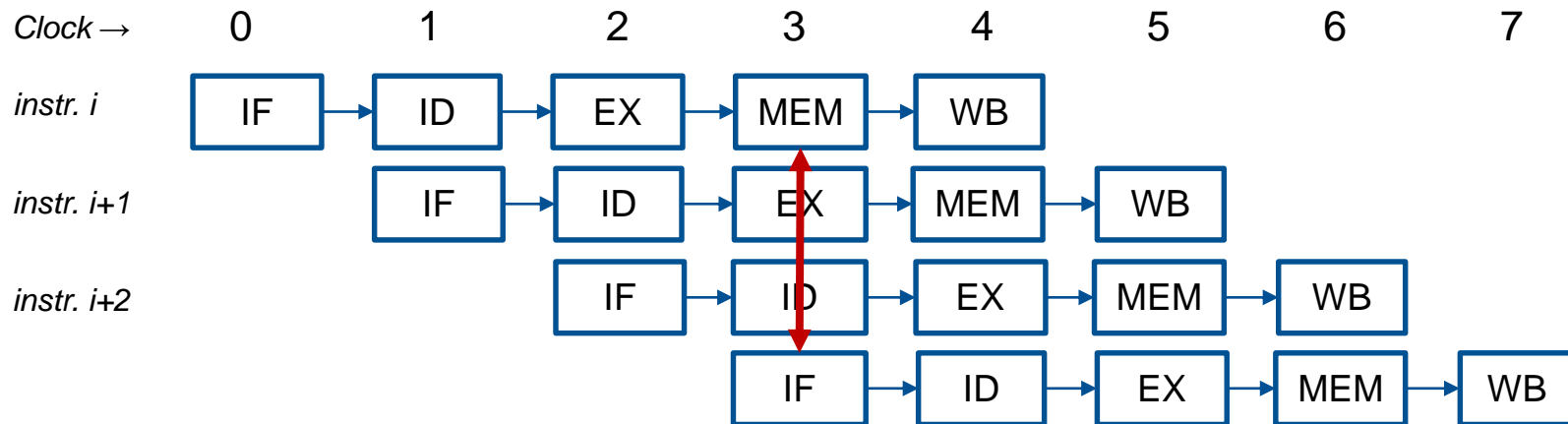
- Arises from branches or other instructions that modify the Program Counter (PC)
- (“data hazard on the PC register”)

- Hazards lead to **pipeline stalls** that decrease the IPC (instruction per cycle)

# Structural Hazards

A **structural hazard** will occur when a CPU cannot support all possible combinations of instructions simultaneously in overlapping execution (e.g., because of a special functional unit).

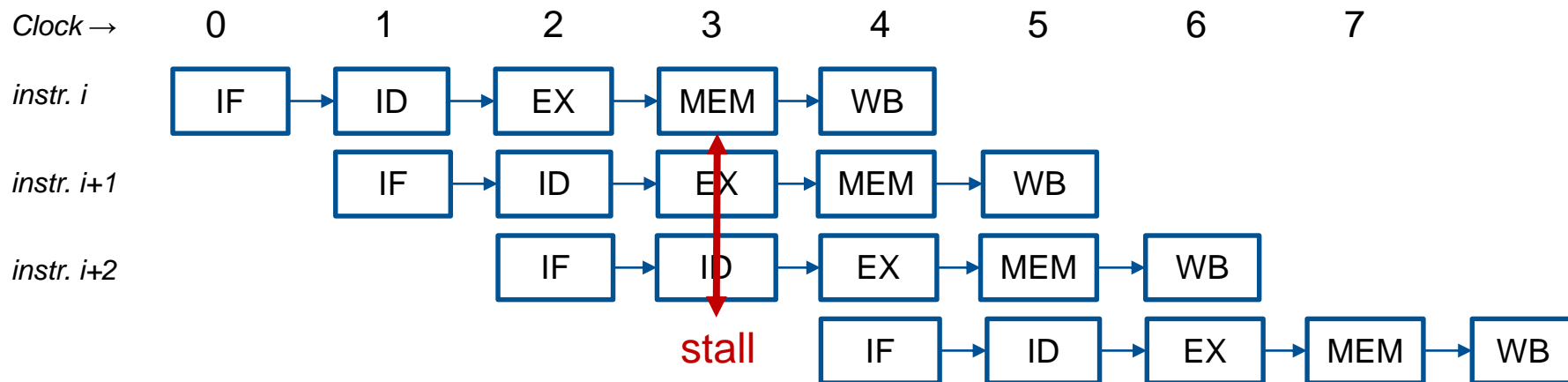
Hypothetically, if we assume that the CPU has only one memory access unit and *instruction fetch* and *memory access* are scheduled in the same cycle.



# Structural Hazards

A **structural hazard** will occur when a CPU cannot support all possible combinations of instructions simultaneously in overlapping execution (e.g., because of a special functional unit).

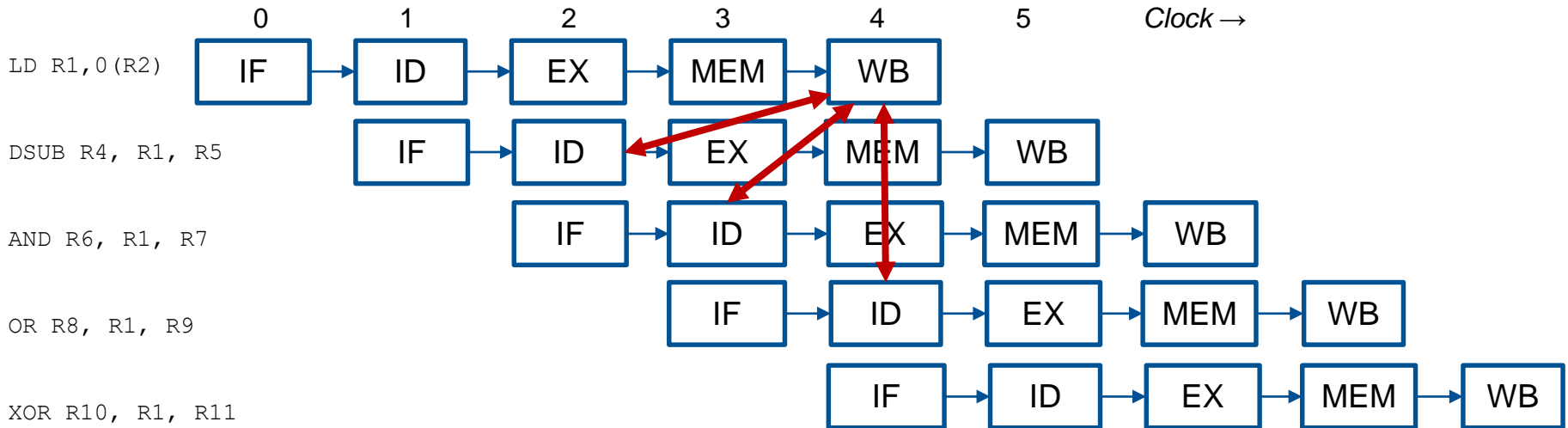
Hypothetically, if we assume that the CPU has only one memory access unit and *instruction fetch* and *memory access* are scheduled in the same cycle.



# 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 the LD instruction (recall that stage WB writes register results)
- Unless stalled, reading R1 will cause incorrect execution result.





## Resolution:

- **Forward** result data from instruction to instruction
  - **Can** resolve hazard LD ↔ AND on previous slide
  - **Cannot** resolve hazard LD ↔ SUB 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)
```

# Data Hazards

Tight loops are a good candidate to improve instruction scheduling

```
for (i=999; i>0; i=i-1)
    x[i] = x[i]+s;
```

```
l: fld      f0,0(x1)    // f0=array element
   fadd.d   f4,f0,f2    // add scalar in f2
   fsd      f4,0(x1)    // store result
   addi     x1,x1,-8     // decrement pointer
   bne      x1,x2,1      // branch x1!=x2
```

```
l: fld      f0,0(x1)
   stall
   fadd.d   f4,f0,f2
   stall
   stall
   fsd      f4,0(x1)
   addi     x1,x1,-8
   bne      x1,x2,1
```

no scheduling

```
l: fld      f0,0(x1)
   addi     x1,x1,-8
   fadd.d   f4,f0,f2
   stall
   stall
   fsd      f4,0(x1)
   bne      x1,x2,1
```

re-schedule

With rescheduling, we can reduce it from 8 to 7 clock cycles per element iteration.

# Data Hazards – loop unrolling

Tight loops are a good candidate to improve instruction scheduling

```
for (i=999; i>0; i=i-1)
    x[i] = x[i]+s;
```

```
1: fld      f0,0(x1)    // f0=array element
   fadd.d   f4,f0,f2    // add scalar in f2
   fsd      f4,0(x1)    // store result
   addi     x1,x1,-8    // decrement pointer
   bne      x1,x2,1     // branch x1!=x2
```

```
1: fld      f0,0(x1)
   fadd.d   f4,f0,f2
   fsd      f4,0(x1)
   fld      f6,-8(x1)
   fadd.d   f8,f6,f2
   fsd      f8,-8(x1)
   fld      f10,-16(x1)
   fadd.d   f12,f10,f2
   fsd      f12,-16(x1)
   fld      f14,-24(x1)
   fadd.d   f16,f14,f2
   fsd      f16,-24(x1)
   addi     x1,x1,-32
   bne      x1,x2,1
```

Unrolled loop will run in  
26 cycles:

- fld has 1 stall
  - fadd.d has 2 stalls
  - 14 issue instructions
- 6.5 cycles per element

Loop unrolling

Loop unrolling  
w/ scheduling

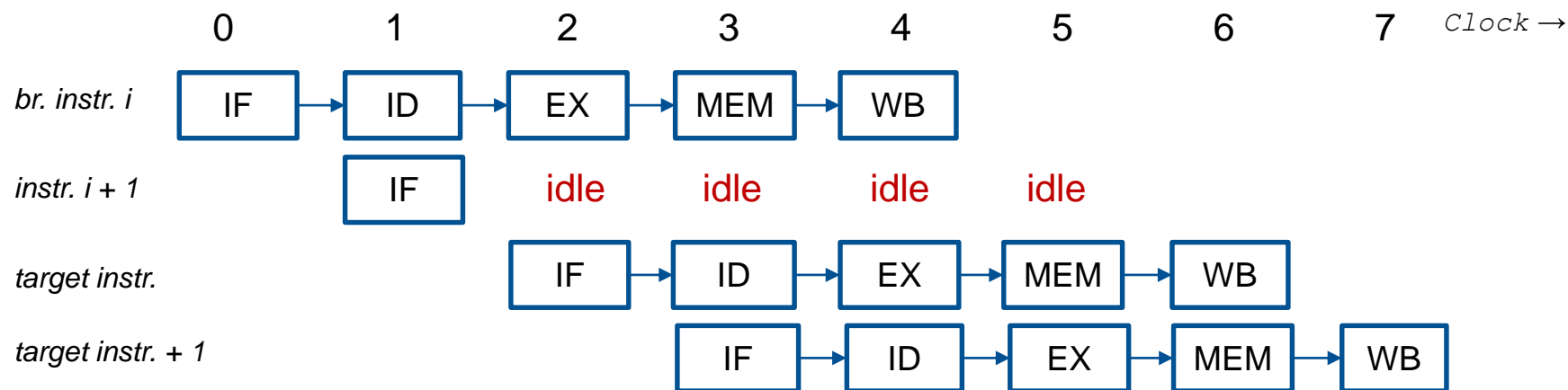
```
1: fld      f0,0(x1)
   fld      f6,-8(x1)
   fld      f10,-16(x1)
   fld      f14,-24(x1)
   fadd.d   f4,f0,f2
   fadd.d   f8,f6,f2
   fadd.d   f12,f10,f2
   fadd.d   f16,f14,f2
   fsd      f4,0(x1)
   fsd      f8,-8(x1)
   fsd      f12,-16(x1)
   fsd      f16,-24(x1)
   addi     x1,x1,-32
   bne      x1,x2,1
```

With scheduling, we can  
reduce to 14 instructions  
Or 3.5 cycles per element

# Control hazards

**Control hazards** are often more severe than data hazards.

- Most simple implementation: **flush pipeline, redo instruction, fetch**



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

# Branch prediction

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

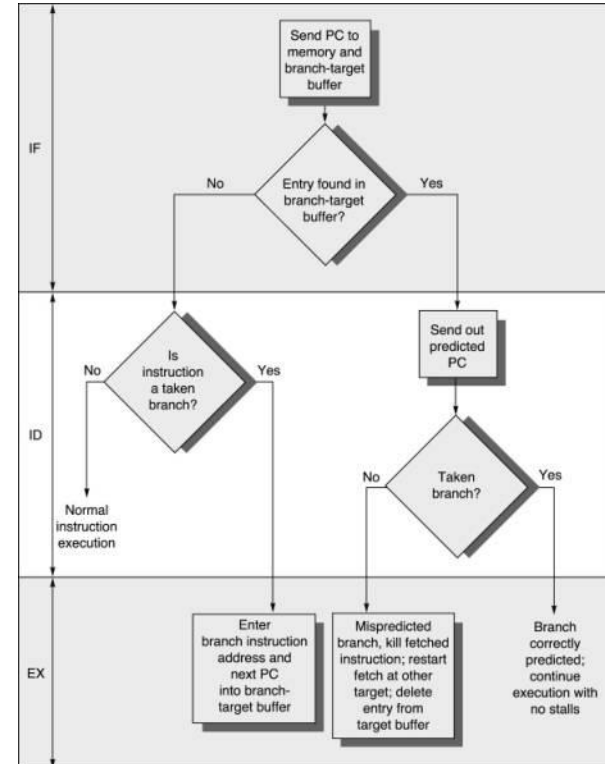
- Prediction must happen **early** (ID stage is too late).

Thus, **Branch Target Buffers (BTBs)** or a Branch Target Cache

- Lookup Table: PC  $\rightarrow$  (predicted target, taken?)

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 Conditions

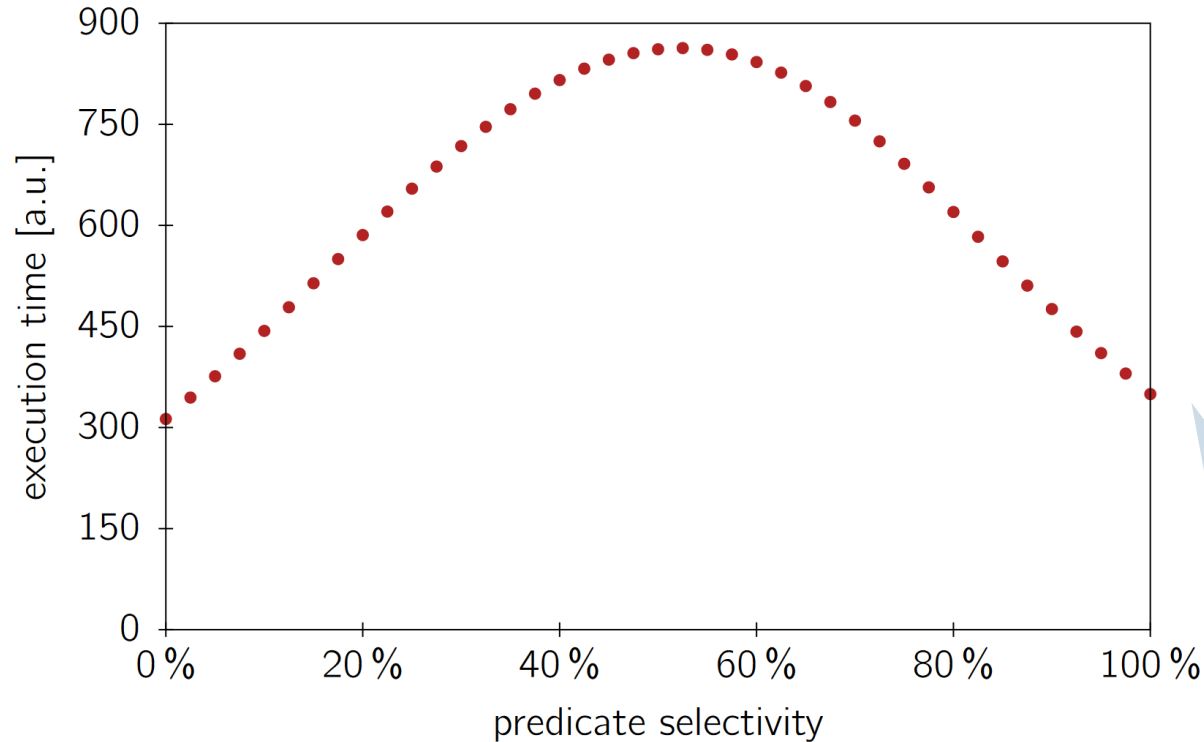
**Selection queries** are sensitive to branch prediction:

```
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++;  
end for
```

# Selection Conditions (Intel Q6700)



The performance of the query is dependent on the selectivity of the predicate (and how predictable it is for the hardware speculator).

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

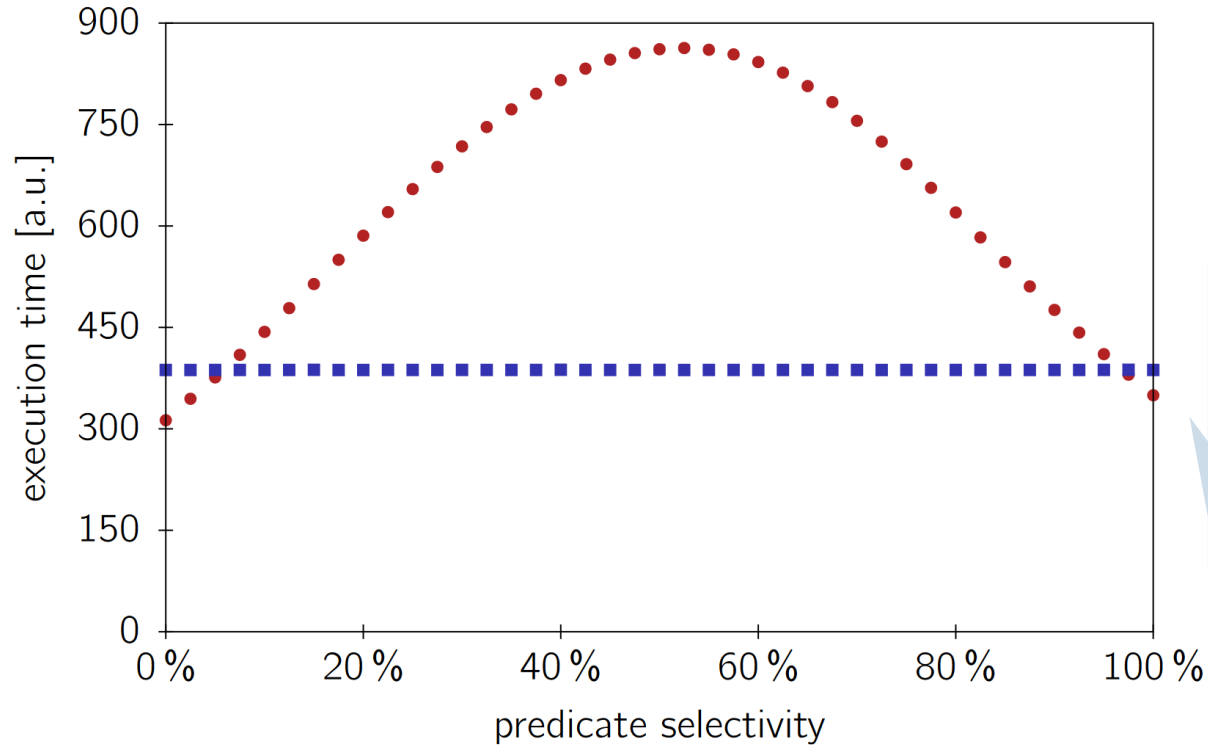
```
for (unsigned int i=0; i < num_tuples; i++){  
    if (lineitem[i].quantity < n)  
        count++;  
}
```



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

- This code does **not** use a branch any more (except to implement the loop).
- The price we pay is an + operation for **every** iteration
- Execution cost should now be **independent** of predicate selectivity.





The performance of the query is now **independent** on the predicate selectivity.

Faster overall, slower at the extreme ends.

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

**How about this query?**

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

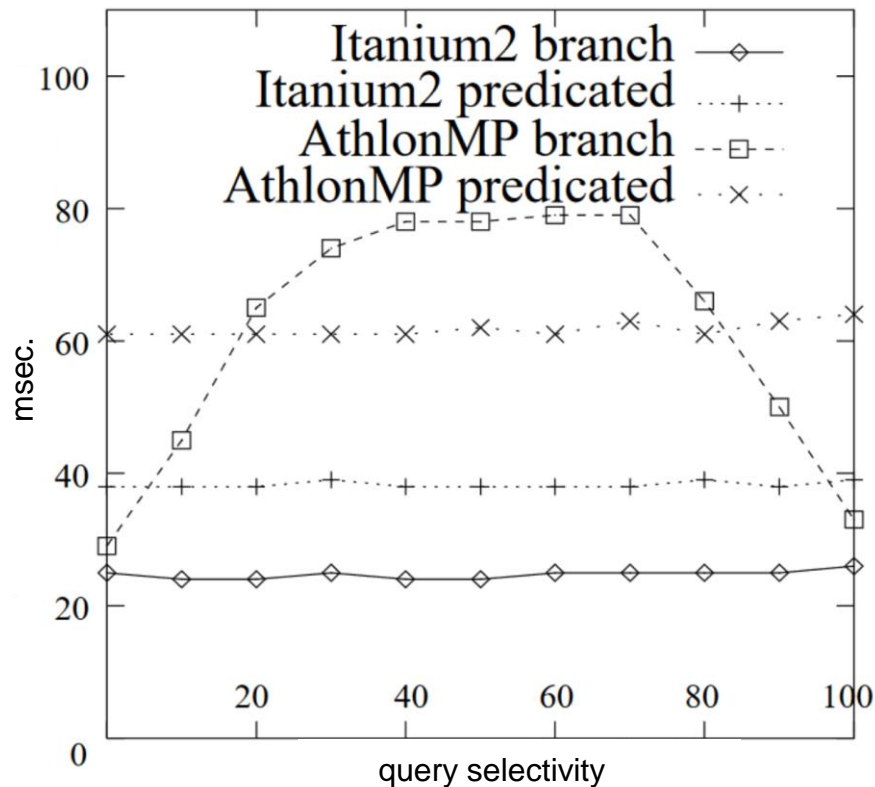
Some CPUs also support hardware predication.

- *E.g.*, Intel Itanium 2
  - 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;
    }
}
```



# Two cursors

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

- This limits the CPUs possibilities to execute instructions in parallel

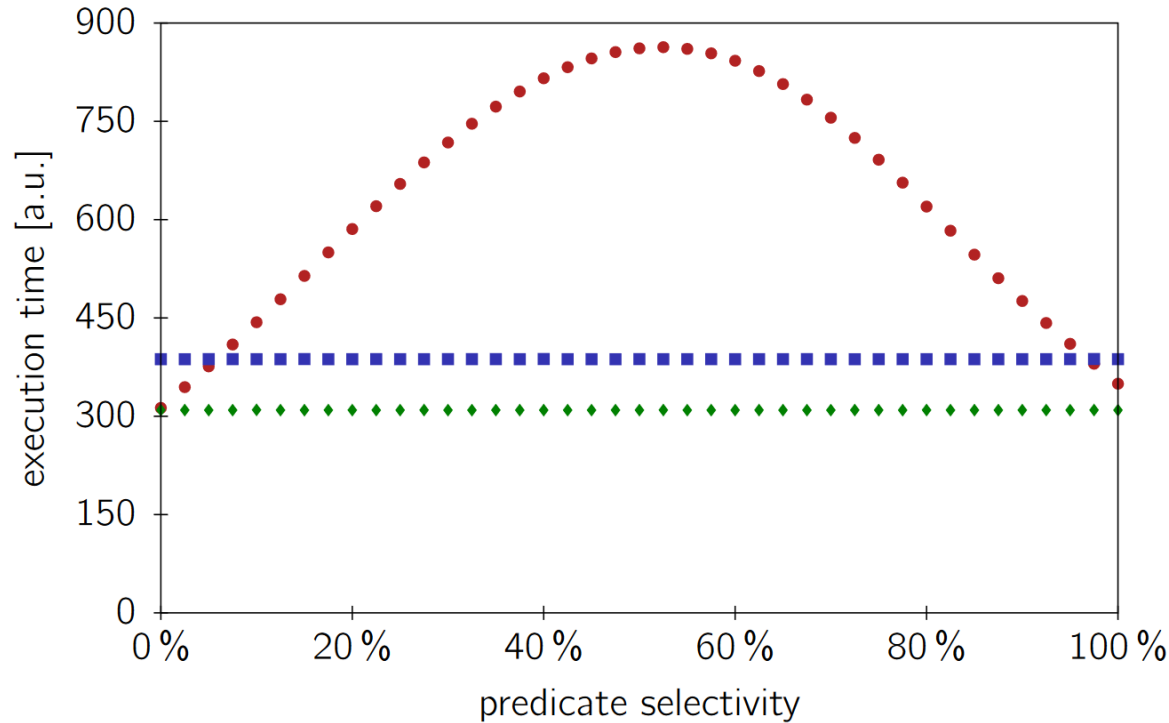
Some tasks can be rewritten to use **two cursors**:

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



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

# Two cursors (experiments)



Two cursors achieves even better overall performance.

# Conjunctive predicates

Usually, 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 (p1)  
        if (p2)  
            ⋮  
            if (pk)  
                ...;  
}
```

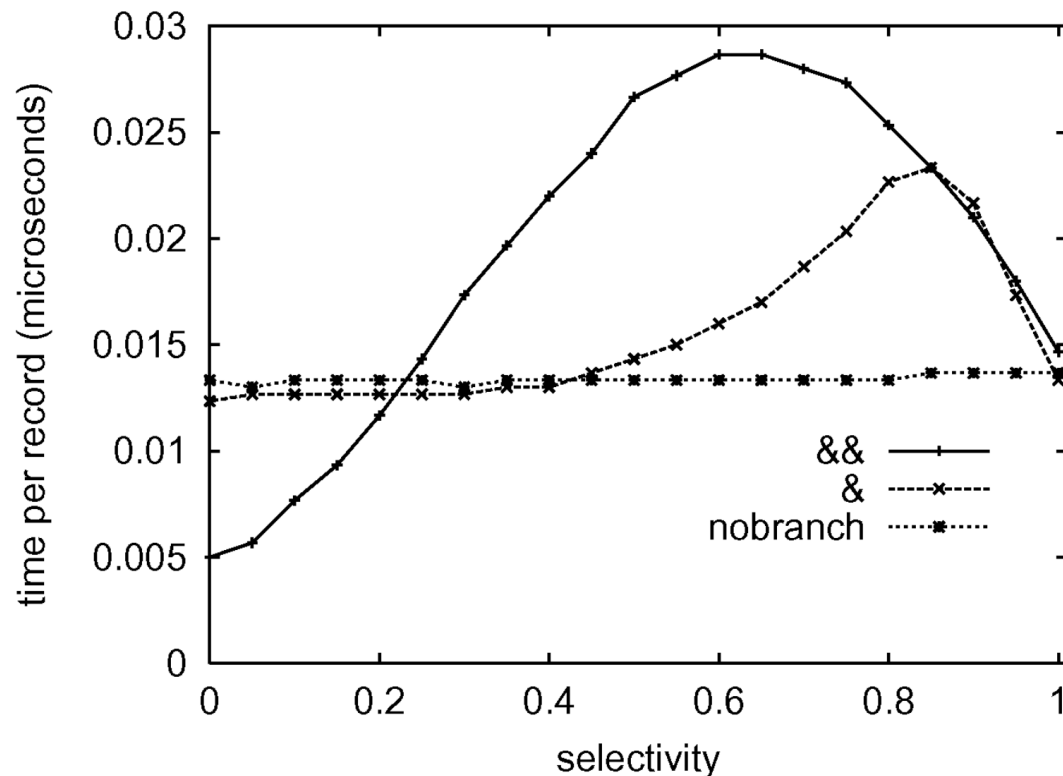
An alternative is the use of the logical `&`:

```
for (unsigned int i=0; i<num_tuples; i++){  
    if (p1 & p2 & ... & pk)  
        ...;  
}
```



```
for (unsigned int i=0; i<num_tuples; i++){  
    answer[j]=i;  
    j+= (p1 & p2 & ... & pk) ;  
}
```

# Conjunctive Predicates



Intel Pentium III

1. && is very good when  $p_1$  is very selective.
2. & reduces to only one branch.
3. No-branch gives predictable performance at the expense of doing extra work.



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 mis-prediction risk
- `p & q` : number of branches halved, but `q` is evaluated regardless of `p`'s outcome
- `j +=` : performs memory write in **each** iteration.

## Notes:

- Sometimes, `&&` is necessary to prevent null pointer dereferences

```
if (p && p->foo == 42)
```

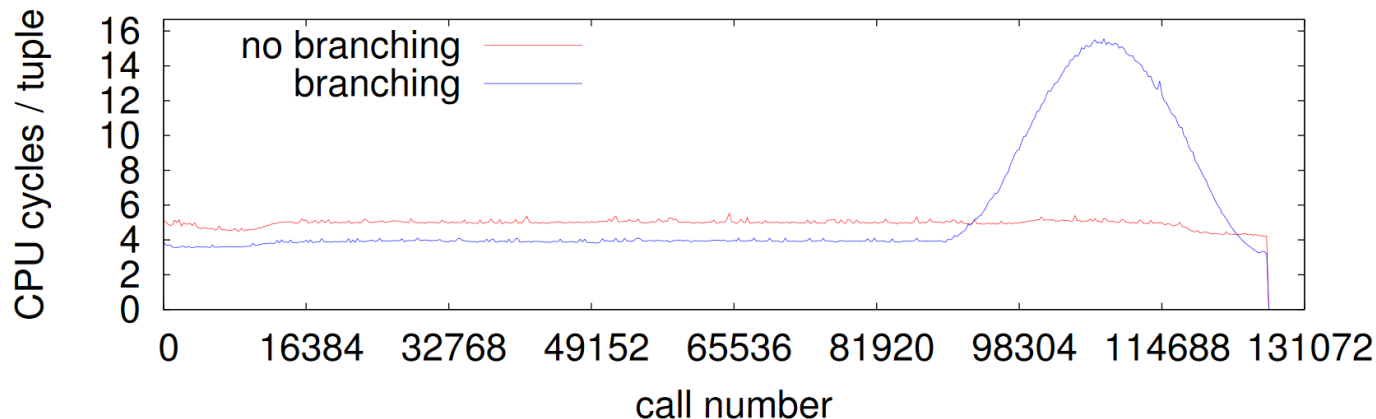
- Exact behavior is hardware-specific.

# 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:



Idea:

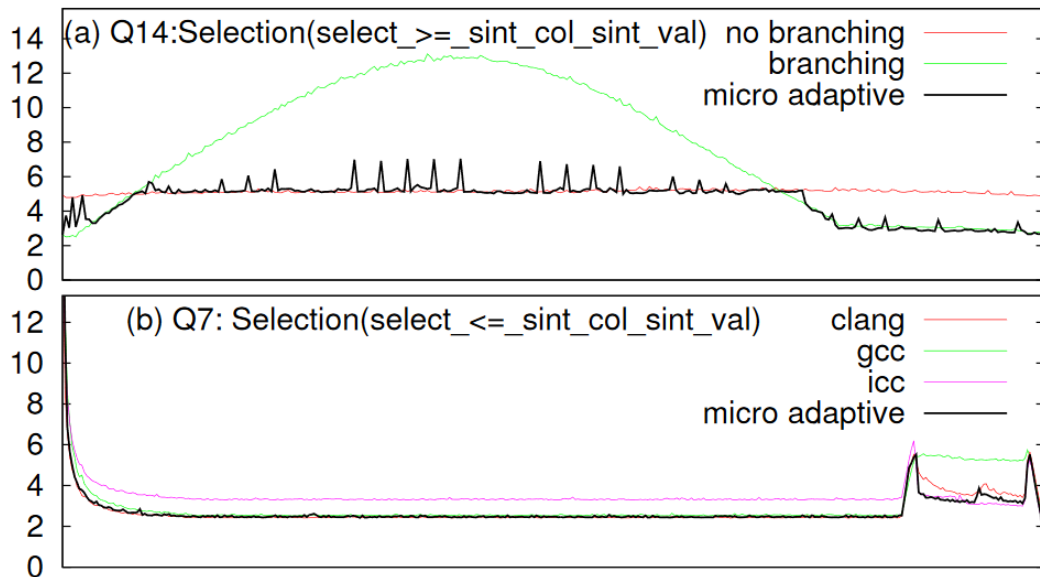
- Generate **variants** of primitive operators
  - With/without branching
  - Different compilers
  - Operator parameters (hash table configurations, etc.)
- Try to **learn** cost model for each variant.
- **Exploit and explore:**
  - **Profile** every execution to refine the cost model
  - Choose **variant**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

Offline training is not suitable for this problem → real-time learning for multi-armed bandit (MAB) problems.

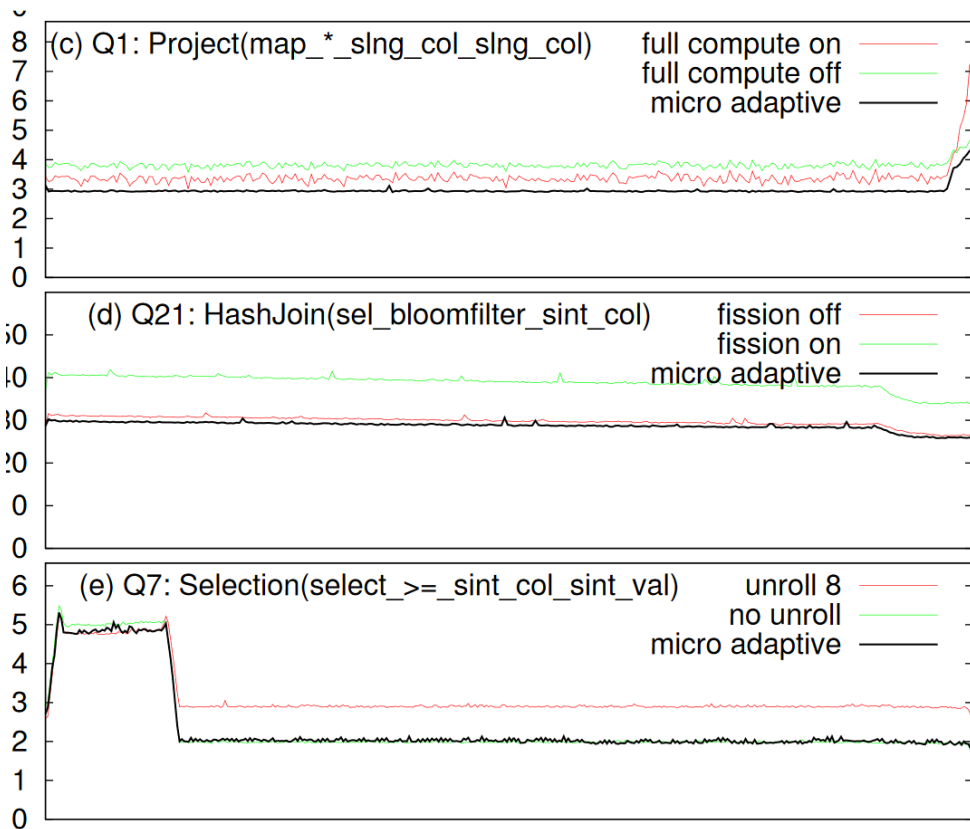
# Micro Adaptivity

## Vector-at-a-time execution:

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



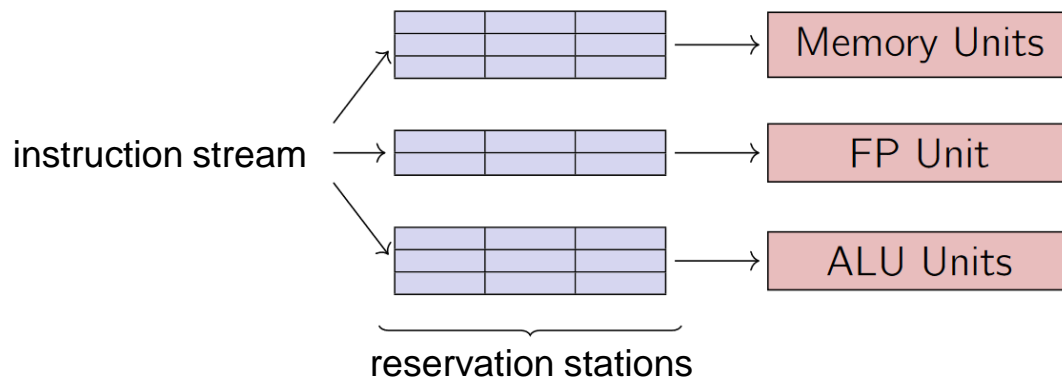
# Micro Adaptivity (experiments)



- The actual execution of instructions is handled in individual **functional units**
  - E.g., load/store unit, ALU, floating point unit, etc.
  - Often, some **units are replicated**.
- Chance to execute **multiple instructions** at the same time.
- Modern CPUs, 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

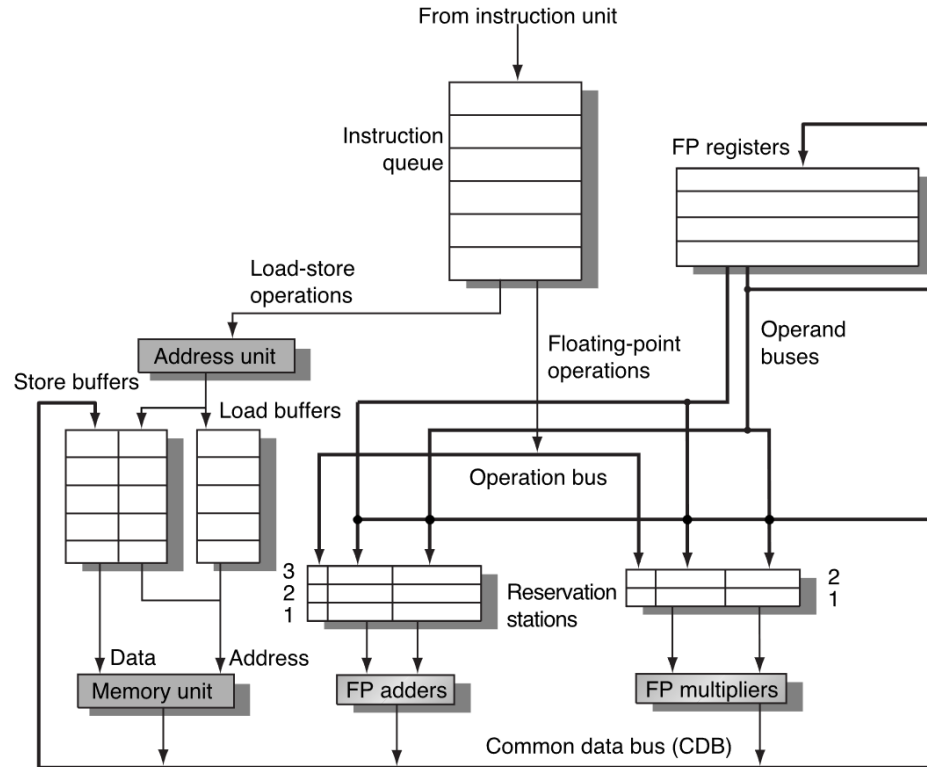
Higher IPCs are achieved with help of dynamic scheduling



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

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

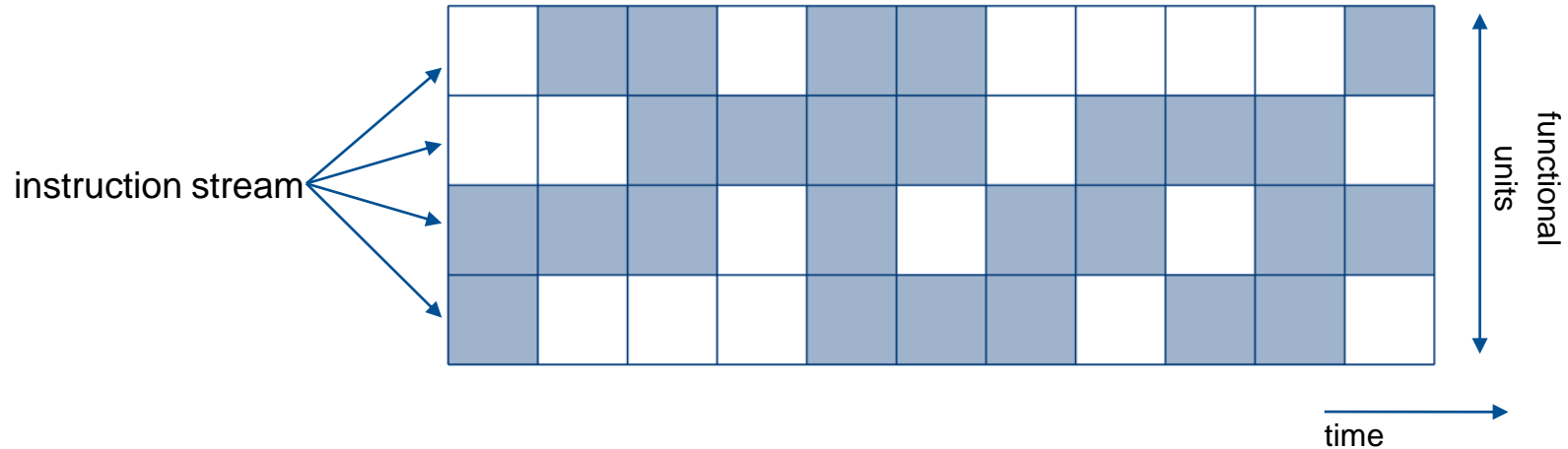
# Example: Dynamic scheduling in MIPS





# Instruction-level parallelism (ILP)

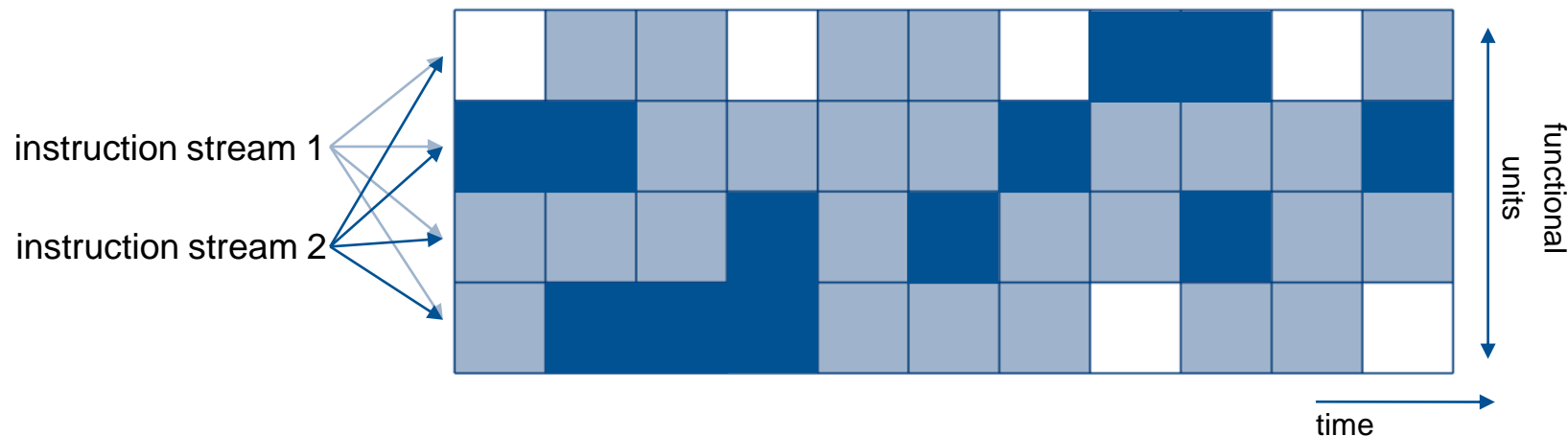
- Usually, not all units can be kept busy with a single instruction stream:
  - due to data hazards, cache misses, etc.



# Thread-level parallelism

**Idea:** use the spare slots, for an **independent instruction stream**

- This technique is called **simultaneous multithreading (hyper-threading by Intel)**



- Surprisingly few changes are required to implement it
- Tomasulo's algorithm requires **virtual registers** anyway
- Need separate fetch units for both streams

# Resource sharing

These SMT (hyper-threads) share most of their resources:

- Caches (all levels)
- Branch prediction functionality (to some extent).

This may have **negative effects**:

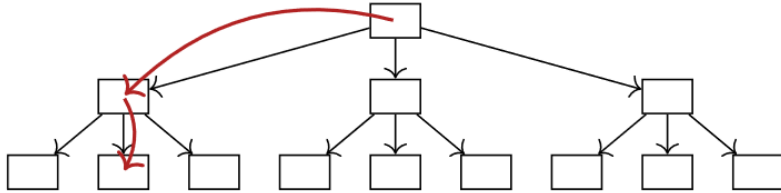
- Threads can **pollute** each other's caches

But also **positive effects**:

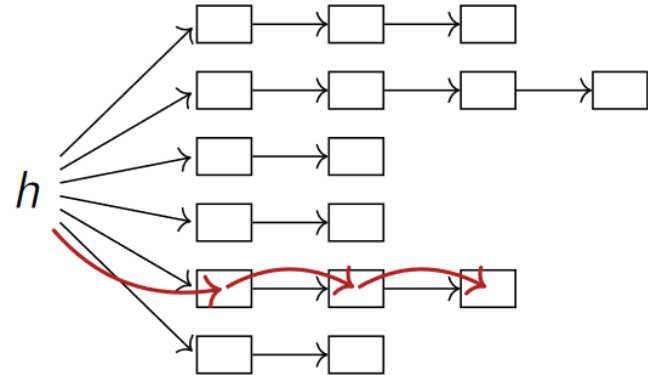
- Threads can **cooperatively** use the caches.

# Use cases

Tree-based indexes:



Hash-based indexes:



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

# Helper threads

## Issue with software pre-fetching!

### Idea:

- Next to the main processing thread, run a **helper thread**.
- They communicate with a circular array of work-ahead set of addresses.
- Purpose of the helper thread is the **pre-fetch** data.
- Helper thread **works ahead** of the main thread.

# Main thread

Consider the traversal of a tree-structured index:

```
foreach input item do  
  read root node; prefetch level 1;  
  read node on tree level 1; prefetch level 2;  
  read node on tree level 2; prefetch level 3;  
  ...  
end for
```

Helper thread will not have enough time to pre-fetch.

# Main thread

Recall, group-based prefetching. We can apply that technique here.

```
foreach group g of input items do  
  foreach item in g do  
    read root node; prefetch level 1;  
  end for  
  foreach item in g do  
    read node on tree level 1; prefetch level 2;  
  end for  
  foreach item in g do  
    read node on tree level 2; prefetch level 3;  
  end for  
  ...  
end for
```

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

# Helper thread

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

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

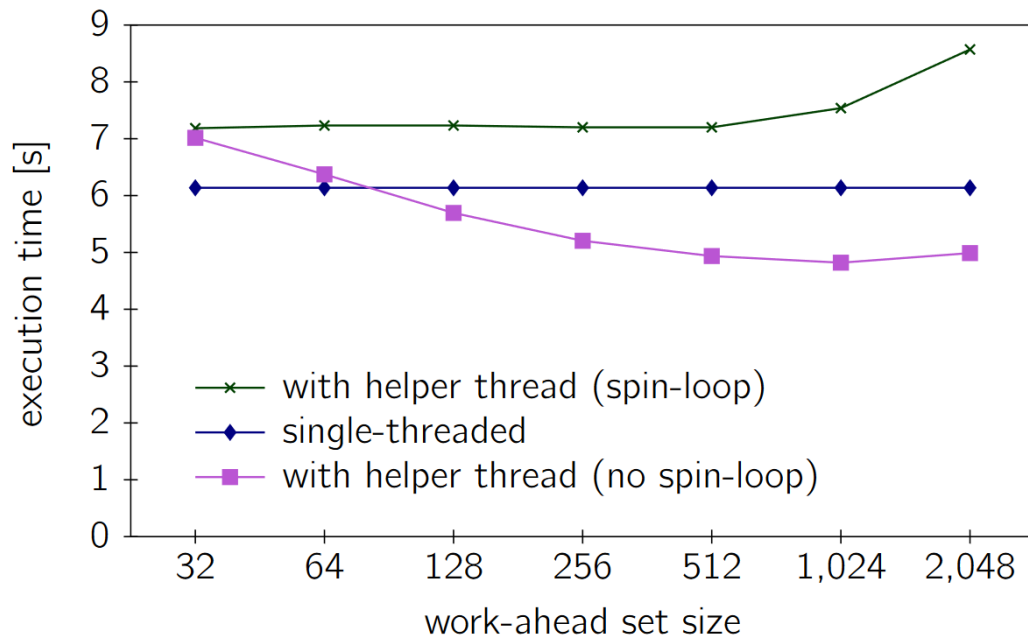
- Purpose: load data into caches, the value of temp is not important

## Technique:

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



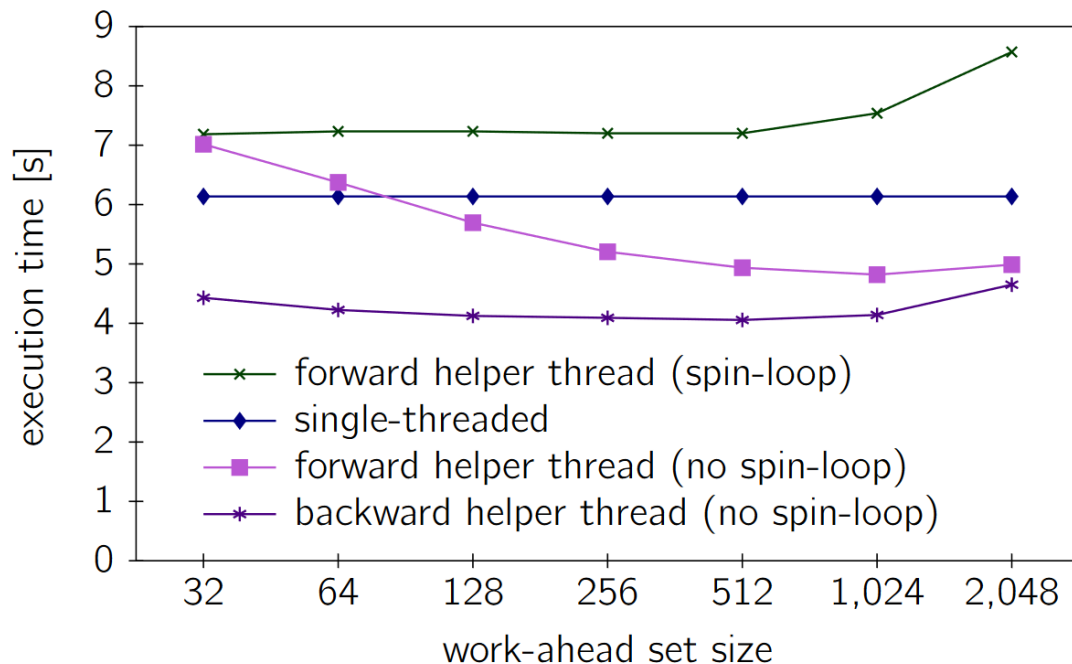
# Helper thread (experiment, tree-based index)



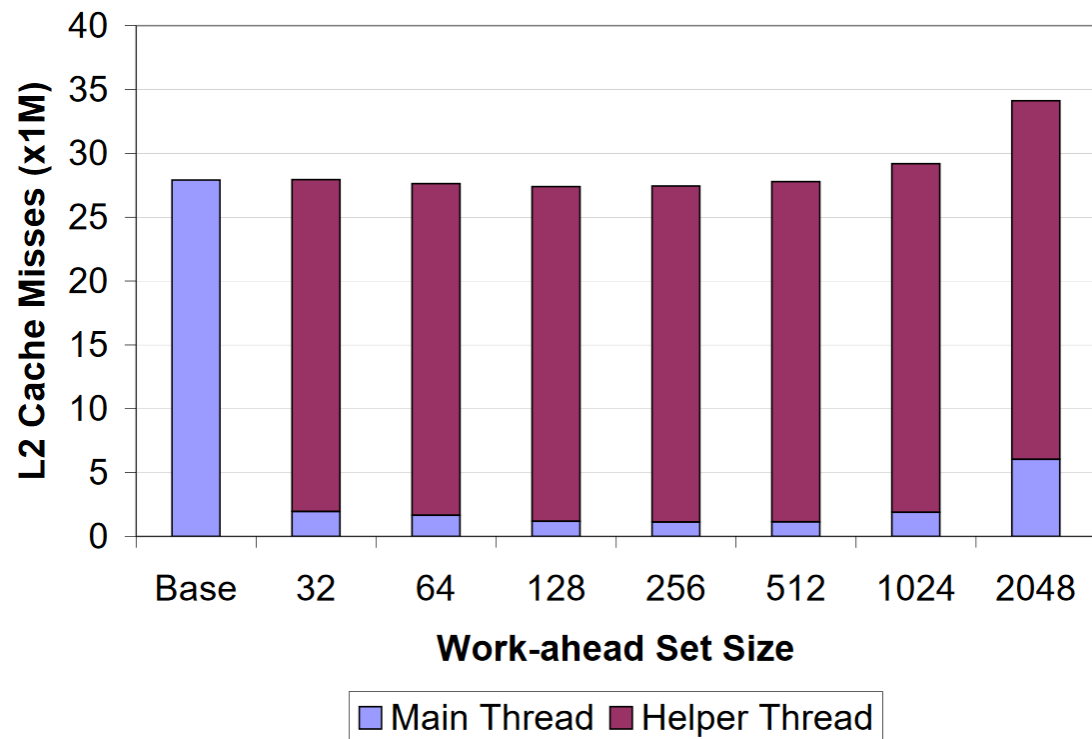
There is a high chance that both threads access the **same cache line at the same time**.

- 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
- Let helper thread work **backward**.

# Helper thread (experiment, tree-based index)



# Cache miss distribution



# References

- Various papers cross-referenced in the slides
  - Boncz, Zukowski, Nes. *MonetDB/X100: Hyper-Pipelineing Query Execution*. CIDR 2005
  - Ken Ross. *Selection Conditions in Main Memory*. TODS 2004
  - Raducanu and Boncz. *Micro-Adaptivity in Vectorwise*. SIGMOD 2013
  - Zhou, Cieslewicz, Ross, Shah. *Improving Database Performance on Simultaneous Multithreading Processors*. VLDB 2005
  
- Lecture: *Data Processing on Modern Hardware* by Prof. Jens Teubner (TU Dortmund, past ETH)
  
- Book: *Computer Architecture: A Quantitative Approach* by Hennessy and Patterson
  - Chapter 3 and Appendix C