Data Processing on Modern Hardware

Jana Giceva

Lecture 5: Instruction execution
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).
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

<table>
<thead>
<tr>
<th>Clock</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>instr. i</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instr. i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instr. i+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Parallel execution
The effectiveness of pipelining is hindered by hazards

- **Structural hazard**
  - Different pipeline stages need 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)
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.

![Diagram showing structural hazards in a CPU pipeline](image-url)
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.

![Diagram of Structural Hazards](image)

- Clock → 0, 1, 2, 3, 4, 5, 6, 7
- instr. i: IF → ID → EX → MEM → WB
- instr. i+1: IF → ID → EX → MEM → WB
- instr. i+2: IF → ID → EX → MEM → WB

**stall**
Data Hazards

- 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.

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

Resolution:

- **Forward** result data from instruction to instruction
  - Can resolve hazard LD $\leftrightarrow$ AND on previous slide
  - **Cannot** resolve hazard LD $\leftrightarrow$ 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

\[
\begin{align*}
\text{SD } & R1, 0(R2) \\
\text{LD } & R3, 0(R4)
\end{align*}
\]
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,l      // branch x1!=x2
```

```
no scheduling
```

```
re-schedule
```

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

Src: Hennessy and Patterson, Chapter 3: ILP and Its Exploitation.
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,l   // branch x1!=x2
```

Unrolled loop will run in 26 cycles:
- `fld` has 1 stall
- `fadd.d` has 2 stalls
- 14 issue instructions
6.5 cycles per element

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

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

Src: Hennessy and Patterson, Chapter 3: ILP and Its Exploitation.
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**.
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 → (predicted target, taken?)

<table>
<thead>
<tr>
<th>Lookup PC</th>
<th>Predicted PC</th>
<th>Taken?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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:

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

Or written as C code:

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

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?

```sql
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)

```c
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;
    }
}
```

Src: Boncz, Zukowski, Nes. MonetDB/X100: Hyper-Pipelineing 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**:

```cpp
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);
}
```

```cpp
count=count1+count2;
```
Two cursors achieves even better overall performance.
Conjunctive predicates

Usually, we have to handle multiple predicates:

```sql
SELECT A_1, ..., A_n
FROM R
WHERE p_1 AND p_2 AND ... AND p_k
```

The standard C implementation uses && for the conjunction:

```c
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:

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

An alternative is the use of the logical &:

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

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.

Src: Ken Ross. Selection Conditions in Main Memory. TODS 2004
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

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

- Exact behavior is hardware-specific.
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:

Src: Raducanu and Boncz. Micro-Adaptivity in Vectorwise. SIGMOD 2013
Micro Adaptivity

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.

Src: Raducanu and Boncz. Micro-Adaptivity in Vectorwise. SIGMOD 2013
Micro Adaptivity (experiments)

(c) Q1: Project(map_ * slng_col slng_col)  
- full compute on  
- full compute off  
- micro adaptive

(d) Q21: HashJoin(sel_bloomfilter_sint_col)  
- fission off  
- fission on  
- micro adaptive

(e) Q7: Selection(select_>=_sint_col_sint_val)  
- unroll 8  
- no unroll  
- micro adaptive
Improving IPC

- 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**.
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:

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

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

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)

Src: Zhou, Cieslewicz, Ross, Shah. Improving Database Performance on Simultaneous Multithreading Processors. VLDB 2005
Problems

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)

Src: Zhou, Cieslewicz, Ross, Shah. Improving Database Performance on Simultaneous Multithreading Processors. VLDB 2005
Cache miss distribution

Src: Zhou, Cieslewicz, Ross, Shah. Improving Database Performance on Simultaneous Multithreading Processors. VLDB 2005
Various papers cross-referenced in the slides

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

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