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
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.
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.

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

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 ↔ 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

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

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

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;

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

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

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

l: 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)  
addi x1,x1,-32  
bne x1,x2,1

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:

```
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
```
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

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

```c
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?

```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 selLtIntColIntVal(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
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**:

```c
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:

```sql
SELECT A₁, ..., Aₙ
FROM R
WHERE p₁ AND p₂ AND ... AND pₖ
```

The standard C implementation uses && for the conjunction:

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

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

```c
for (unsigned int i=0; i<num_tuples; i++){
    if (p_1)
        if (p_2)
            \vdots
            if(p_k)
            \ldots;
}
```

An alternative is the use of the logical &:

```c
for (unsigned int i=0; i<num_tuples; i++){
    if (p_1 & p_2 & \ldots & p_k)
        \ldots;
}
```
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
- 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:

![Graph showing CPU cycles per tuple vs. call number for branching and no-branching](image-url)

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**)
Vector-at-a-time execution:

- Re-consider variant choice for every \( n \) vectors.
- Adapt to specifics of the particular query/operator.
- Also adjust to \textit{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
Data dependency for OoO – loop fission

```c
size_t sel_bloomfilter_sint_col (size_t n, size_t* res, char* bitmap, sint* keys){
    size_t i, ret = 0;
    for (i = 0; i < n; i++) {
        slng gv = bf_hash(keys[i]);
        res[ret] = i;
        ret += bf_get(bitmap, gv); // loop data dependency and cache miss
    }
    return ret;
}
```

```c
size_t sel_bloomfilter_sint_col (size_t n, size_t* res, char* bitmap, sint* keys){
    size_t i, ret = 0;
    for (i = 0; i < n; i++) {
        slng hv = bf_hash(keys[i]);
        tmp[i] = bf_get(bitmap, hv); // cache miss
    }
    for (i = 0; i < n; i++) {
        res[ret] = i;
        ret += tmp[i];
    }
    return ret;
}
```
In the previous example, the loop fission variant:

- When it sustains a cache miss for `bf_get()`, due to its data-independence, the CPU can continue executing the next loop iteration(s), leveraging the large OoO execution capabilities of modern CPU processors (>100 instructions).
- This way the CPU can get multiple (up to 5 on IvyBridge) loop iterations in execution at any time, leading to 5 concurrent outstanding cache misses, maximizing memory bandwidth utilization.

In contrast, the non-fission variant:

- Each iteration waits on each other due to the loop-iteration dependency → less concurrent cache misses and therefore lower memory footprint.
Instruction-level parallelism (ILP)

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

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

Hash-based indexes:
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.
Consider the traversal of a tree-structured index:

```c
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;
    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.,

```c
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
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

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

- Various papers cross-referenced in the slides
  - Ken Ross. *Selection Conditions in Main Memory*. TODS 2004
  - 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