Data Processing on Modern Hardware

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

Summer 2014
Part III

Instruction Execution
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:

```
        0   1   2   3   4   5  clock →

  instr. i
  IF → ID → EX → MEM → WB

  instr. i + 1
  IF → ID → EX → MEM → WB

  instr. i + 2
  IF → ID → EX → MEM → WB
```
Pipelining in CPUs

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 (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 ↔ 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.
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.

![Diagram]

**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 ↔ performance trade-off.
### Data Hazards

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>R1, 0(R2)</td>
</tr>
<tr>
<td>DSUB</td>
<td>R4, R1, R5</td>
</tr>
<tr>
<td>AND</td>
<td>R6, R1, R7</td>
</tr>
<tr>
<td>OR</td>
<td>R8, R1, R9</td>
</tr>
<tr>
<td>XOR</td>
<td>R10, R1, R11</td>
</tr>
</tbody>
</table>

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

![Pipeline Diagram](image)

© Jens Teubner · Data Processing on Modern Hardware · Summer 2014
Resolution:

- **Forward** result data from instruction to instruction.
  - Could resolve hazard `LD ↔ AND` on previous slide (forward `R1` between cycles 3 and 4).
  - **Cannot** resolve hazard `LD ↔ 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.

\[
\text{for } (i = 1000; \ i > 0; \ i = i + 1) \\
x[i] = x[i] + s;
\]

naïve code

```
1:  L.D       F0, 0(R1)\hspace{2cm}1:  L.D       F0, 0(R1)
    ADD.D     F4, F0, F2\hspace{2cm}DADDUI  R1, R1, #-8
    S.D       F4, 0(R1)\hspace{2cm}ADD.D   F4, F0, F2
    DADDUI    R1, R1, #-8\hspace{2cm}ADD.D   F8, F6, F2
    BNE       R1, R2, l\hspace{2cm}ADD.D   F12, F10, F2
```

re-schedule

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

loop unrolling

```
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)
    S.D       F12, 16(R1)
    S.D       F16, 8(R1)
    BNE       R1, R2, l
```

source: Hennessy & Patterson, Chapter 2

© Jens Teubner · Data Processing on Modern Hardware · Summer 2014
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)
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: $PC \rightarrow \langle \text{predicted target, taken?} \rangle$.

<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 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**.

```c
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.

\(^3\)except to implement the loop
This was an example of **software predication**.

 العالي

**How about this query?**

```sql
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.
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);
        j += b;
        out[j] = i;
        j += b;
    }
    return j;
}

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**:

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

count = count1 + count2;
```
Conjunctive Predicates

In general, we have to handle multiple predicates:

\[
\text{SELECT } A_1, \ldots, A_n \\
\text{FROM } R \\
\text{WHERE } p_1 \text{ AND } p_2 \text{ AND } \ldots \text{ AND } p_k
\]

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

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

An alternative is the use of the logical &:

```c
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.

```c
for (unsigned int i = 0; i < num_tuples; i++)
{
    answer[j] = i;
    j += (p1 & p2 & ... & pk);
}
```
Fig. 1. Three implementations: Pentium.

The time per record is shown in microseconds on the vertical axis, measured against the probability that a test succeeds. The probability is controlled by setting an appropriate threshold for an element of the \( t \) array to be randomly set to 1. All functions in this graph have the same probability.

Our preliminary analysis of the three implementations is borne out by this graph. For low selectivities, the branching-and implementation does best by avoiding work, and the one branch that is frequently taken can be well predicted by the machine. For intermediate selectivities, the logical-and method does best. However, when the combined selectivity gets close to 0.5, the performance worsens. The no-branch algorithm is best for nonselective conditions; it does more “work” but does not suffer from branch misprediction.

Each of the three implementations is best in some range, and the performance differences are significant. On other ranges, each implementation is about twice as bad as optimal.

Thus, we will need to consider in more depth how to choose the “right” implementation for a given set of query parameters.

Looking at the performance numbers, one might wonder why we care about per-record processing times that are fractions of a microsecond. The reason we care is that this cost is multiplied by the number of records, which may be in the tens or hundreds of millions. When we don’t have an index, we have no choice but to perform a full scan of the whole table. Even when we’re scanning fewer records per query, the overall performance in queries-per-second is directly impacted by these performance numbers. In a dynamic query environment, for example, we might be aiming for video-rate screen refresh, and thus require the completion of 30 queries per second for each user. See Section 7 for another example.

From now on, when we show an implementation, we will omit the \( for \) loop, just showing the code inside the loop.

4. OPTIMIZING INNER LOOP BRANCHES FOR CONJUNCTIONS

Using standard database terminology, we will refer to a particular implementation of a query as a plan. We now formulate our optimization question:

Given a number \( k \), functions \( f_1 \) through \( f_k \), and a selectivity estimate \( p_{m}^{D_{1}, \cdots, D_{k}} \) for each \( f_m \), find the plan that minimizes the expected computation time.
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.

\[ j += \ldots \]

Performs memory write in **each** iteration.

**Notes:**

- Sometimes, \&\& is necessary to prevent null pointer dereferences: \( \textbf{if (p \&\& p->foo == 42)} \).
- Exact behavior is hardware-specific.
Experiments (Sun UltraSparc IIi)

Fig. 2. Three implementations: Sun.

for each one if we can derive estimates for the function cost and selectivity for the optimization algorithm. Since the loop code is small, we can probably tolerate thousands of such queries with a small expansion in the executable code size. However, for ad-hoc queries we need to be able to allow the functions to be specified at run-time. There are two complementary problems. First, executing a function call (and potentially dereferencing a function pointer as well) may be a significant performance overhead in a tight inner loop. Second, we don't know the selectivities and function costs until query time, and these statistics are important for the selection of the appropriate inner-loop plan. There are several potential solutions to this problem. We outline one below.

When responding to an ad-hoc query, we still may have time to perform the optimization described above, compile a new version of the loop, with the appropriate combination of &&s and &s, and link it into the running code. Systems such as Tempo [Consel and Noel 1996; Noel et al. 1998] allow such run-time compilation. Run-time code specialization of this sort would be beneficial only if the optimization time plus the compilation time are smaller than the improvement in the running-time of the resulting plan. As we saw in Sections 4.4 and 4.5, the optimization time is relatively small. The code to be compiled is also relatively small. For scans of large tables, such an approach may indeed pay off.

Run-time code specialization is different from self-modifying code. With self-modification, a program changes its own byte-code during its execution. While such a technique might actually present the most efficient solution to our code specialization problem, code modification is generally considered to be a bad idea. Such code is not reentrant, sharable, able to reside in ROM; it leads to cache coherency problems; it isn't easy to understand and it is architecture dependent.

6.3 Internal Parallelism

The results for the experiment of Section 3.2 on a Sun UltraSparc are given in Figure 2. Unlike the Pentium, as the selectivity approaches 1, the performance of the && plan continues to worsen. The reason for this behavior is that the Sun can execute multiple instructions at a time. For the & algorithm and the ACM Transactions on Database Systems, Vol. 29, No. 1, March 2004.

© Jens Teubner · Data Processing on Modern Hardware · Summer 2014
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.
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.
E.g., compress the digits of $\pi$ using 3-bit PFOR compression.

<table>
<thead>
<tr>
<th>header</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>⊥</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>⊥</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>⊥</td>
<td>5</td>
<td>⊥</td>
</tr>
<tr>
<td>⊥</td>
<td>⊥</td>
<td>3</td>
</tr>
<tr>
<td>⊥</td>
<td>⊥</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

Decompressed numbers: 31415926535897932
Decompression

During decompression, we have to consider all the exceptions:

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

For PFOR, `DECODE` is a simple addition:

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

The **branch** in the above code may bear a high **misprediction risk**.
Figure 6.4: Decompression bandwidth, branch miss rate and instructions-per-cycle depending on the exception rate demonstrates most clearly on Pentium4 how NAIVE decompression throughput rapidly deteriorates as the exception rate gets nearer to 50%. The cause are branch mispredictions on the if-then-else test for an exception, that becomes impossible to predict. In the graph on top, we see that the IPC takes a nosedive to 0.5 at that point, showing that branch mispredictions are severely penalized by the 31 stage pipeline of Pentium4.

To avoid this problem, we propose the following alternative “patch” approach:

```c
int Decompress<ANY>( int n, int bitwidth, ANY *__restrict__ output, void *__restrict__ input, ANY *__restrict__ exception, int *next_exception) {
    int next, code[n], cur = *next_exception;
    UNPACK[bitwidth](code, input, n); /* bit-unpack the values */
    /* LOOP1: decode regardless of exceptions */
    ...
}
```

We collected IPC, cache misses, and branch misprediction statistics using CPU event counters on all test platforms.

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.

```c
/* ignore exceptions during decompression */
for (i = 0; i < n; i++)
    output[i] = DECODE(code[i]);

/* patch the result */
foreach exception
    patch corresponding output item;
```
We *don’t* want to use a branch to find all exception targets!

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

\[
\begin{array}{ccccccccc}
4 & 1 & 5 & 5 & 2 & 6 & 5 & 3 & 5 \\
1 & 7 & 3 & 3 & 2 & & & & \\
\end{array}
\]

\[
\begin{array}{cc}
3 & 1 \\
3 & 2 \\
0 & & & & & & & & & \\
\end{array}
\]

\[
\begin{array}{c}
9 \\
9 \\
8 \\
9 \\
\end{array}
\]

\[
\begin{array}{c}
\text{compressed data} \\
\text{exceptions} \\
\end{array}
\]

→ Can now traverse exception holes and patch in exception values.
The resulting decompression routine is branch-free:

```c
/* 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 120 for experimental data on two-loop decompression.
3-bit-PFOR-compressed representation of the digits of $e$?

$$e = 2.71828182845904523536028747135266249757247093699959574966967627724076630353547594571382178525\ldots$$
6.2.5 Compression

Previous database compression work mainly focuses on decompression performance, and views compression as a one-time investment that is amortized by repeated use of the compressed data. This is caused by the low throughput of compression, often an order of magnitude slower than decompression (see Figure 6.2), such that compression bandwidth is clearly lower than I/O write.

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**.
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

From instruction unit

Instruction queue

Load-store operations

FP registers

Operand buses

Floating-point operations

Operation bus

Reservation stations

FP adders

FP multipliers

Common data bus (CDB)

Memory unit

Data

Address

Load buffers

Store buffers

Address unit


© Jens Teubner · Data Processing on Modern Hardware · Summer 2014
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.

![Diagram showing two instruction streams](image)

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.

\(^4\)Intel uses the term “hyperthreading.”
Resource Sharing

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**.
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`
   - read root node, prefetch level 1;
3. `foreach` item in $g` `do`
4. `foreach` item in $g` `do`
   - read node on tree level 1, prefetch level 2;
6. `foreach` item in $g` `do`
7. `foreach` item in $g` `do`
   - 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** `prefetch` 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)

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.
Cache Miss Distribution