• Function side effects
• Important keywords: `const` and `restrict`
• Hardware issues: `cache` memory, data `locality`, `pipelines`
• How to make your program run faster on modern hardware

Elias Rudberg, room P2440
elias.rudberg@it.uu.se
Optimization techniques

- Choose good **data structures**
- Reduce number of operations
- Use cheap operations – strength reduction
- Avoid too many small function calls – **inline**ing
- Improve **data locality**
- Use compiler **optimization flags**
- Help the compiler! Use **const** and **restrict**
Side effects in loops

char *str = "Hello World!";
int i;
for (i = 0; i < strlen(str); i++)
{
    if (str[i] == '!') str[i] = '?';
}

• What’s the complexity?
• Can you improve it?
• Can the compiler improve it?
• Does the compiler know the implementation of strlen?
The "const" keyword

• In ANSI C you add the keyword “const” to arguments that are read-only
  – The memory object will never be written to
• Can also be used for variables to make them into constants
• Helps the compiler in analysis
The von Neumann Model

- Fetches instructions from the memory referenced by the program counter (PC) and computes results based on the data the instruction specified.
- The Arithmetical-Logic Unit (ALU) does the actual work.
- In this simple model access times to memory are uniform.
• Any pointer of unknown origin can reference a value that is accessed through another variable
• Any pointer might be used as an array
  – Of unknown size
• Multiple “aliases” for the same memory location
• Makes compile-time optimization very hard
The "strict aliasing" rule

- Default mode in C99 and recent GCC
- Pointers of different types should not refer to the same memory
- Not a problem until you start being “clever”
- Significant compilation benefits
The "restrict" keyword

- restrict is another element of the C99 standard
- Available in many C/C++ compilers, including recent gcc (sometimes as __restrict)
- Within this context, any memory locations accessed by a restricted (pointer) variable will only be accessed through that pointer
- E.g. strcpy(char * restrict dest, char * restrict from)
Hardware considerations
Basic operation:

- Modern computers are of LOAD-STORE type
- These machines store operands in “registers”
- A “register”: small scratch memory very close to the arithmetical units
- Data must be loaded from memory into a register, operated upon, and then stored back
Registers

- Registers are a scarce resource
  - Store words (32/64 bit)
  - Special registers for floating-point, SIMD (128/256/512 bits!)
- The compiler tries to maximize the usage of the registers
  - Called register allocation
- If you run out of registers, you must temporarily store results back to memory and then retrieve them again
  - Register spill
  - Degrades performance
Registers in C

- There are two keywords that control register allocation from C
- The `register` keyword suggests (forces?) a variable to a register
  - Used for heavily accessed variables
  - Today, most compilers can figure this out themselves
  - You cannot take the address of something that is stored in a register
- The `volatile` keyword forces the results to be written back to memory
  - Used in low-level and parallel programming
Examples

// tell compiler that a should be stored in a register
register int a = 23;

// force b to be written back to memory
volatile int b = 43;
Basic types of machine instructions:

- Data movement between memory and registers: “move”
- Arithmetic and logical operations: add, multiply, bitwise ops, ...
- Conditions and jumps: “goto”, conditional goto
- Procedure calls: “call” and “return”
Execution in a CPU

- "Machine Code"
- "Data"

CPU
Architecture – too complex
CPU -- a simpler model
Five common steps

Fetch Instruction
Read Register
ALU
Memory Access
Write Register

Execution time = $N_{\text{instructions}} \times T_p$
How "long" is a CPU cycle?

1982: 5MHz clock
1 cycle \(\leftrightarrow\) 200ns

2002: 3GHz clock
1 cycle \(\leftrightarrow\) 0.3ns

2012: 3GHz clock
1 cycle \(\leftrightarrow\) 0.3ns
Instruction stages

- Instructions need to be fetched from memory and then decoded
- To know which parts of the CPU logic that is to be used
- Many instructions read/write memory from/to registers
- Every instruction is an encoded binary number
- Arithmetic work can be done once data is stored in the registers
- The different stages are performed using logic from different parts of the chip
Instructions can often be divided into several independent parts (stages). Example:

1. Instruction fetch (IF)
2. Instruction decode/register fetch (ID)
3. Execution (EX)
4. Memory access (MEM)
5. Write-back (WB)
Instruction stages, example
Observations

- All steps are performed in succession
- Steps are waiting for work most of the time
- Logic is idling
- The CPI (cycles per instruction) is at least 5
- Solution: use **pipelining**
- Start a new instruction each cycle
- Each clock cycle becomes a pipe stage
- Maximize use of logic
Pipelines

- Pipelining allows several instruction steps to execute simultaneously
- In this example, up to 5 x faster execution
- Requires that instructions are independent
Pipelines

Execution time = $N_{\text{instructions}} \times T_s$
Pipeline speedup

- For a 5-stage pipeline we can execute one instruction per cycle if the pipe is full.
- In general a pipe of length n will give a speedup of n if we ignore startup cost.
- Same technique used in fabrication industry:
  - Cars, mobile phones etc.
Problems with pipelines

- Works fine if we can start a new instruction every clock cycle
- Instructions must be independent
- If the pipeline detects a hazard its stalls, pipeline bubble
- Branch hazards
- A branch can make instructions halfway through the pipe unnecessary
- The pipe needs to be flushed
Avoiding branches

• Branching can be expensive

• Don’t repeat checking the same condition

```c
for (i = 0; i < 42; i++)
  if (a) dosomething(i);

if (a)
  for (i = 0; i < 42; i++)
    dosomething(i);
```
How to get more independent instructions?

• Loop unrolling – rewrite loop to do more work in each loop iteration

• Loop fusion – turn two (or more) loops into one
Today: ~10-20 stages and 4-6 pipes

- + Shorter cycletime (more MHz)
- + ILP (parallel pipelines)
- - Branch delay even more expensive
- - Even harder to find "enough" independent instructions.
Instruction scheduling

- Compilers know which resources are available and how long instructions take
- Compilers schedule the instructions
- Tries to minimize pipeline bubbles
- Keep all resources busy (increase parallelism)
- Sometimes we will have bubbles due to hazards
- Branch prediction
- Out-of-order execution
Multiple pipelines

**Execution time** = \( N_{\text{instructions}} \times \frac{T_s}{N_{\text{Pipelines}}} \)

Problems of the Pipeline remain!
Model of CPU with several pipelines
Modern memory: ~150 CPU cycles access time

- + Shorter cycletime (more MHz)
- - Memory access even more expensive.

- Memory access is slow.
Implications for pipelines

- Waiting for data stalls the pipeline
- We will need a lot of registers to hide this latency
- Solution: cache memories
- A cache memory is a smaller SRAM (faster) memory which acts as a temporary storage to hide the main memory latencies
Data locality

- We can predict what instructions and data a program will use based on its history.
- Temporal locality, recently accessed items are likely to be accessed in the near future.
- Spatial locality, items whose addresses are near one another tend to be referenced close together in time.
Caches

- Several levels of cache with varying latency
- L1 (approx. 10-100 kB) 1-5 cycles
- L2 (256 kB- ) 10-20 cycles
- L3 (> 2 MB), slower
- Caches organized into "cache lines", approx. 128 bytes each
Example of costs associated with cache/memory access

<table>
<thead>
<tr>
<th>To Where</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>&lt;= 1</td>
</tr>
<tr>
<td>L1d</td>
<td>~3</td>
</tr>
<tr>
<td>L2</td>
<td>~14</td>
</tr>
<tr>
<td>Main Memory</td>
<td>~240</td>
</tr>
</tbody>
</table>

(Numbers listed by Intel for a Pentium M)
Cache optimization

- An example of “hardware dependent optimization”
- Blocking:
  Reorder data accesses to improve data locality
- Modern prefetchers are great, can help.
Cache optimization – how?

Try to increase the chances that the needed memory locations are already in cache:

- Nearby memory accesses in time should be to nearby locations in memory
Cache

CPU

address

address

Cache

hit

data (a word)

Memory
Quick summary – important hardware-related concepts

- Pipelining --> independent instructions are good
- Cache memory --> data locality is good