Performance via Complexity
Need for Architectural Innovations

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
Outline

• Components of a basic computer
• Memory and caches
• Brief overview of pipelining, out of order execution, etc.
• Theme: Modern processors attain their high performance by paying in increased complexity
• The programmers, mainly, have to deal with the complexity and performance variability that results from it
Need for Architectural Innovation

• The computers didn’t become faster just relying on Moore’s law:
  • E.g., switching speeds increased at only a moderate rate

• So, to keep making clock speeds faster, architectural innovations were needed

(S. Borkar, Personal Communication, February 24, 2018)
So, let us review our schematic of a stored program computer to see where innovations were added.
The Stored-Program Architecture

• The processor includes a small number of registers
  • With dedicated paths to ALU (arithmetic-logic unit)
• In modern “RISC” processors since mid-1980s
• All ALU instructions operate on registers
• Only way to use memory is via:
  • Load Ri, x // copy content of memory location x to Ri
  • Store Ri, x // copy contents of Ri to memory location x

Before 1985, ALU instructions could include memory operands
Control Flow

• Instructions are fetched from memory sequentially
• Using addresses generated by the program counter (PC)
• After every instruction, the PC is incremented to point to the next instruction stored in memory
• Control instructions like branches and jumps can directly modify PC
Datapath - Schematic
Obstacles to Speed

• What are the possible obstacles to speed in this design?
  • Long chain of gate delays
  • “Floating point” computations
  • Slow.. I mean really S...l...o...w memory!!
  • Virtual memory and paging

• The theme for this module:
  • Overcoming these obstacles can lead to significant increase in complexity, and can make performance difficult to predict and control
Performance via Complexity

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
Obstacles to Speed

• What are the possible obstacles to speed in this design?
  • Long chain of gate delays
  • “Floating point” computations
  • Slow... I mean really S...l...o...w memory!
  • Virtual memory and paging

• The theme for this module:
  • Overcoming these obstacles can lead to significant increase in complexity, and can make performance difficult to predict and control
Latency vs. Throughput and Bandwidth

• Imagine you are putting a fire out
  • Only buckets, no hose
  • 100 seconds to walk with a bucket from water to fire (and 100 to walk back)
  • But if you form a bucket brigade
    • (Needs people and buckets)
  • You can deliver a bucket every 10 seconds
    • So, latency is 100 or 200 seconds, but throughput/bandwidth is 0.1 buckets per second... much better
• What’s more, you can increase bandwidth:
  • Just make more lines of bucket brigade
Reducing Clock Period – Pipelining
Pipelined Processor

• Allows us to reduce the clock period
  • Since long gate delay (critical paths) are reduced

• But assumes we can always pipeline instructions

• What can disturb a pipeline?
  • Hazards (may create “bubbles” in the pipeline)
  • Data hazard: instruction, which needs a result calculated by a previous instruction
  • Control hazard: branches and Jumps
Avoiding Pipeline Stalls

• Data forwarding:
  • In addition to storing the result in a register, forward it to the next instruction (store it in the pipelines buffer)

• Dynamic branch prediction:
  • Separate hardware units that track branch statistics, and predict which way a branch will go!
  • E.g., a loop: branch will go back in all cases, except the last
Impact of Branch Prediction on Programming

• Consider the following code

```c
for (unsigned c = 0; c < arraySize; ++c)
    { if (data[c] >= 128) sum += data[c]; }
```

• Assume data contains random numbers between 0..255, and the arraySize is 32k

• It was observed that sorting the data beforehand improves the performance five-fold

• Why?

• Potential answer: every “if” in the above is unpredictable, but with sorted data they are statistically predictable
  • (false, false, ... false, true, true, ... true)

(stackoverflow.com, n.d.)
Programming to Avoid Branch Misprediction

• When you have data dependent branches that are hard to predict:
  • See if you can convert them into non-branching code!
• Conditional move instructions help, and normally compilers should do the right thing, but sometimes compilers aren’t able to
• For example:
  • Sum += expression that evaluates to data[c] if > 128 or 0 otherwise;
  • Or, since there are only 255 possible values, pre-create a lookup table
  • Sum += table[data[c]];
Floating Point Operations

• A multiply and add is needed together in many situations
  • DAXPY: double-precision Alpha X Plus Y
  • for (i=0; i<N; i++) Y[i] = a*X[i] + Y[i];

• Special hardware units that can do the two together
  • And, of course, it is pipelined

• When there are enough such operations in sequence, the pipeline is full, and you get two floating point ops per cycle

• Machines support a FMAD instruction (saves instruction space)
References

Memory Access Challenges
Introduction to Caches

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
Components of a Stored-Program Computer

- **Instruction Memory**
  - Instruction

- **CPU**
  - PC (Program Counter)
  - Move to the next location
  - Register name
  - Data

- **Data Memory**
  - Register set
Latency to Memory

• Data processing involves transfers between data memory and processor registers

• DRAM: large, inexpensive, non-volatile memory
  • Latency: ~50ns
    • Comparatively slow improvement over time: 80 -> 30 ns
  • A single core clock is 2 GHz: it beats twice in a nanosecond!
    • Can perform upward of 4 ALU operations/cycle
  • Modern processors have tens of cores on a single chip

• Take away:
  • Memory is significantly slower than the processor
Bandwidth Can Be Increased

• More pins can be added to chips
• 3D stacking of memory can increase bandwidth further
  • Need methods that translate latency problems to bandwidth problems
• Solution: concurrency
• Issues:
  • Data dependencies
Cache Hierarchies and Performance

- Cache is fast memory, typically on chip
  - DRAM is off-chip
- It has to be small to be fast
- It is also more expensive than DRAM on per-byte basis
- Idea: bring frequently accessed data in the cache
Why and How Does a Cache Help?

• *Temporal and spatial locality*
  • Programs tend to access the same and/or nearby data repeatedly

• Spatial locality and cache lines
  • When you miss, you bring not just the word that CPU asked for, but a bunch of surrounding bytes
  • Take advantage of the high bandwidth
  • This “bunch” is a cache line
  • Cache lines may be 32-128 bytes in length
Cache Hierarchies and Performance
## Some Typical Speeds/Times Worth Knowing

<table>
<thead>
<tr>
<th>Latency</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modern processor</td>
<td></td>
</tr>
<tr>
<td>L1 cache</td>
<td></td>
</tr>
<tr>
<td>L2-L3 cache</td>
<td></td>
</tr>
<tr>
<td>DRAM</td>
<td></td>
</tr>
<tr>
<td>Solid state drive</td>
<td></td>
</tr>
<tr>
<td>Hard drive</td>
<td></td>
</tr>
<tr>
<td>Network: Cluster</td>
<td></td>
</tr>
<tr>
<td>Network: Ethernet</td>
<td></td>
</tr>
<tr>
<td>Network: World-wide-web</td>
<td></td>
</tr>
</tbody>
</table>
Some Typical Speeds/Times Worth Knowing

<table>
<thead>
<tr>
<th></th>
<th>Latency</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modern processor</td>
<td>0.25 ns</td>
<td></td>
</tr>
<tr>
<td>L1 cache</td>
<td>several ns</td>
<td></td>
</tr>
<tr>
<td>L2-L3 cache</td>
<td>10s ns</td>
<td></td>
</tr>
<tr>
<td>DRAM</td>
<td>30-70 ns</td>
<td>10-20GB/s</td>
</tr>
<tr>
<td>Solid state drive</td>
<td>0.1ms</td>
<td>200-1500 MB/s</td>
</tr>
<tr>
<td>Hard drive</td>
<td>5-10 ms</td>
<td>200MB/s</td>
</tr>
<tr>
<td>Network: Cluster</td>
<td>1-10 µs</td>
<td>1-10GB/s</td>
</tr>
<tr>
<td>Network: Ethernet</td>
<td>100 µs</td>
<td>1GB/s</td>
</tr>
<tr>
<td>Network: World-wide-web</td>
<td>10s of ms</td>
<td>10Mb/s (note b vs. B)</td>
</tr>
</tbody>
</table>
Architecture Trends: Pipelining

• Architecture over 2-3 decades was driven by the need to make clock cycle faster
  • Pipelining developed as an essential technique early on
  • Each instruction execution is pipelined:
    • Fetch, decode, execute, stages at least
    • In addition, floating point operations, which take longer to calculate, have their own separate pipeline
  • So, no surprise: L1 cache accesses in Nehalem are pipelined
    • Even though it takes 4 cycles to get the result, you can keep issuing a new load every cycle, and you wouldn’t notice a difference (almost) if they are all found in L1 cache (i.e., are “hits”)