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
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
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
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, ... 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

PC (Program Counter)

Move to the next location

CPU

Register set

Data Memory

Register name

Data

Instruction
Latency to Memory

• Data processing involves transfers between data memory and processor registers

• DRAM: large, inexpensive, 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
<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>Component</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”)
Memory Access Challenges
Performance Impact of Caches

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
Recall: 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

• Size of cache

• Multiple levels of cache

• Performance impact of caches
  • Designing programs for good sequential performance
Cache: Parameters and Metrics

• Cache line: spatial locality
  • 64 bytes or 128 bytes, consecutive memory addresses
  • Make it too short: what happens?
  • Make it too long: what happens?

• Hit ratio
  • Fraction of times a data item is found in cache
  • Best possible: 1
  • May be higher than that given by reuse ratio because of spatial locality
Cache: Parameters and Metrics

• Cache line: spatial locality
  • 64 bytes or 128 bytes, consecutive memory addresses
  • Make it too short: what happens? Too many cache misses, ..
  • Make it too long: what happens? Potential waste if not that much spatial reuse

• Hit ratio
  • Fraction of times a data item is found in cache
  • Best possible: 1
  • May be higher than that given by reuse ratio because of spatial locality
Cache Mapping and Associativity

• Say you have 64 KB of cache, 64 byte cache lines
  • So, 1024 cache blocks in all

• On a read miss, you have to fetch a cache line from memory
  • (Or from next level cache)
  • Where in the cache should it go?
    • I.e., which of the 1024 blocks should be used to store it?
    • A fixed place based on the address: direct-mapped cache
    • Anywhere: fully-associative cache
      • Expensive (hardware and time!)

Solution: Set associative cache
Set Associative Cache

Program’s memory: a sequence of cache lines

Where in the cache can the shaded one go
(if it needs to be brought into the cache)?

Direct mapped

Set associative: 4 sets of 2 cache lines each

Fully associative
Cache Replacement Policy

• Which cache line to evict?
• For direct mapped cache, there is only one candidate
• For associative caches, multiple choices exist, so a policy is needed
  • Random
  • LRU: least recently used
• What do you think will be a good policy?
• But no significant impact on performance across applications is observed...
Categorizing Cache Misses

• Compulsory misses:
  • “Cold” cache accesses, i.e., first time you access it

• Capacity misses:
  • There is no space in cache

• Conflict misses:
  • There is space, but it is not in the right set
  • The idea of a “working set”
    • A set of addresses that the program accesses over a time window

• Question: Which misses cannot happen with fully associative cache? Conflict misses
Example

for (i = 0; i < n; i += M)  // M is the cache-line size in words
A[i] = B[i] + C[i];

• What happens with:
  • Direct map cache?
  • Fully associative cache?
  • 4-way associative cache?

• How many cache misses?

• What if the loop repeats?
  • I.e., there is an outer loop
  • Working set: the data in A, B, and C
Another Example

for (i = 0; i < n; i++). sum = sum + A[i];

• How many cache misses?
• What kind of misses are those?
• Can we reduce the cost of those misses? How?
• Answers:
  • One for every cache line... N/k, where k is the size of a cache line in words
    • So, 64 byte cache line, single precision words of 4 bytes: k = 16
• These are all compulsory misses
  • First time we access the data
Imagine a sequential program running using a large array, A
Values in the indirection array are all between 0...size-1
How long should the program take, if each addition is a ns?
What is the performance difference you expect, depending on the value of size?

```c
for (i = 0; i < repetitions/size; i++)
    for (count1 = 0; count1 < size; count1++)
```
Other Architectural Innovations
Prefetch

• Look for pattern in memory accesses:
  • That you are accessing successive memory locations in A
  • If I make the hardware detect this pattern, what can I do?

• Two techniques:
  • Bring multiple cache lines in on a miss
  • Prefetch even more lines... asynchronous load instructions
Out of Order Execution

• Keep many instructions “in flight”
• If there are data dependencies, buffer the instruction
• Important techniques: Register renaming, scoreboard, etc., to satisfy dependencies and ensure correctness
• Also, try to issue multiple instructions per cycle!
Superscalar Processors

• Can we execute multiple instructions per cycle?
  • FMAD was a single instruction

• Idea: Fetch multiple instructions in each cycle...
  • And execute as many of them in parallel as is possible without violating data dependencies
  • Need more hardware on chip
  • More registers for buffering results
  • Multiple load units, etc.

• IPC: instruction-level parallelism
  • How many can you execute in each cycle?
  • Typically 2-3 is the max
Impact on programming

• Usually, we should let the compiler handle this
• But its useful to know whether our program is exploiting the superscalar processors well
  • IPC : instructions per cycle --- this is a useful metric to track
• Try to write code such that consecutive instructions can be executed concurrently
  • Avoid dependences.. But these may be unavoidable
  • ILP (Instruction level parallelism)
• Avoid pipeline bubbles
Performance and Complexity

Additional Issues: Paging, Compilers, ...

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
Multiprogramming and Virtual Memory

• A computer runs many programs “simultaneously”
  • It gives each program a time slice of some milliseconds (e.g., 10ms)
• How can it keep their memories from mixing up?
• Answer: virtual memory
  • Each program thinks it has the whole processor to itself, and can address memory from 0...X gigabytes
  • This “virtual” address is mapped to a physical address by the hardware and operating system via “page tables”
  • Each page has (say) 4 kilobytes of data... It may or may not be in physical memory
  • If it is not in memory, it is kept on disk (in “swap” space)
  • The mapping itself is stored in memory!
    • But most commonly/recently used entries of this map are stored in hardware: TLB or Translation Lookaside Buffer
Load Instruction

(Morris, 1998)
What Is a Good Page Size?

• In most OSs it is set at 4 KB
• That’s fine for multiprogrammed machines
• For a dedicated parallel computer?
  • HugePages option: 2 MB pages, 4 MB pages
  • On BlueWaters, simply load the 2 MB page module before compilation and execution
  • Loading a module redirects mallocs to use larger pages
Compilers

• Capabilities
• Black-box nature
• Optimization levels: -O1, -O2, -O3...
  • Higher level of optimization is better for performance, but occasionally (rarely) may compile wrong!
  • Lower levels make debugging easier
• Flags
  • More specialized, nuanced, control over what kinds of optimizations the compiler may try
  • Debug information retained
Bottom Line?

• The speed increase has come at the cost of complexity
• This leads to high performance variability that programmers have to deal with
• It takes a lot to write an efficient program!
References