(Breaking) Caches
Part 3
Administrivia

• Exam has been graded, will be released for regrades shortly

Review Grades for **Midterm**

- **MINIMUM**: 0.0
- **MEDIAN**: 72.08
- **MAXIMUM**: 98.6667
- **MEAN**: 69.98
- **STD DEV**: 16.12
Components of a Computer
Processor-DRAM latency gap

1980 microprocessor executes ~one instruction in same time as DRAM access
2015 microprocessor executes ~1000 instructions in same time as DRAM access

*Slow DRAM access could have disastrous impact on CPU performance!*
Adding Cache to Computer

Processor

- Control
- Datapath
  - PC
  - Registers
  - Arithmetic & Logic Unit (ALU)

Memory

- Cache
  - Program
  - Bytes
  - Data

Input

Output

Processor-Memory Interface

I/O-Memory Interfaces
The Memory Hierarchy

- **Remote Secondary Storage (e.g., Internet)**: ~10 M cycles
- **Traditional Disk**: ~1 M cycles
- **Flash Disk**: ~100 cycles
- **Main Memory**: ~10 cycles
- **Cache**: On CPU
- **Primary Storage**: 1 cycle
- **Registers**: On CPU

**Faster Access**

**Higher Capacity**
Block Placement Schemes

- DM placement: mem block 12 in 8 block cache: only one cache block where mem block 12 can be found—(12 modulo 8) = 4
- SA placement: four sets x 2-ways (8 cache blocks), memory block 12 in set (12 mod 4) = 0; either element of the set
- FA placement: mem block 12 can appear in any cache blocks
Analyzing Caches with Code

- A conceptually simple set of nested for loops:
  - `int array[MAXLEN]`
  - `for(size = 1024; size <= MAXLEN; size = size << 1){`
    - `for(stride = 1; stride <= size; stride = stride << 1){`
      - // Some initialization to eliminate compulsory misses
      - // Repeat this loop enough to get good timing for computing AMAT
      - `for(i = 0; i < size; i += stride){`
        - `array[i] = array[i] + i`
      }
      - // Now add some timing information for how long the
      - // for loop takes
    }
  }
- This is **striding access**:
  - Rather than every element in the array this accesses every $i^{th}$ element
- This is designed to break caches:
  - So by seeing where and how the cache falls down, this can reveal the internal structure
Reminder: Cache Miss Types

- **Compulsory**: A miss that occurs on first reference
  - An infinitely large cache would still incur the miss

- **Capacity**: A miss that occurs because the cache isn't big enough
  - An infinitely large cache would not miss

- **Conflict**: A miss that occurs because the associativity doesn't allow the items to be stored simultaneously
  - A fully associative cache of the same size would not miss
Reminder:
Other Parameters

• Block or Line size:
  • The # of bytes in each entry

• Associativity:
  • The degree of flexibility in storing a particular address
    • Direct mapped: One location
    • N-way set associative: one of N possible locations
    • Fully associative: Any location

• AMAT: Average Memory Access Time
  • hit time + miss penalty * miss rate
Cache Failure Case: Capacity

- Up until the test exceeds the cache capacity...
  - Everything is fine!
- But once sizeof(array) > cache size:
  - Things break down and you start getting misses
- Which increases the loop time
  - $\text{AMAT} = \text{hit time} + \text{miss penalty} \times \text{miss rate}$
Cache Failure Case: Spacial Locality

- Spacial locality breaks down if only a *single* entry in each cache line is ever accessed
  - Since the rest of the cache line provides no benefit...
- So worst-case behavior occurs when each line is only accessed in one location
  - So when stride * sizeof(int) == block size
- Combined with where the capacity break occurs...
  - And you now know the line-size
Avoiding a Failure Case: Cache Associativity

• If your array is 2x the cache capacity...
  • But you are striding at >= 2*block size...

• You aren't using all the cache entries
  • So by definition, all your misses are no longer capacity misses but conflict misses!

• Reminder: Tag/Index/Offset...
  • The index specifies the possible locations for a set associative cache
  • So when do the accesses have different indexes?
    • That is when you have stopped having conflict misses
Analyzing Caches: Multiple Levels

• Each level is its own cache...
  • To test L2, you must be using references that break L1...

• Which is fine for capacity, but...
  • If L2 line size <= L1 line size, we can't reliably tell
    • Since the L1 cache provides the spacial locality
    • But generally most multi-level caches use the same line size, defined by the external memory interface
  • If the L2 associativity <= L1 associativity
    • The conflict misses will be removed in L1
Actual Test:
Raspberry Pi 3: L1 Cache hitting...

<table>
<thead>
<tr>
<th>Size (bytes):</th>
<th>32768</th>
<th>Stride (bytes):</th>
<th>4</th>
<th>read+write:</th>
<th>9 ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (bytes):</td>
<td>32768</td>
<td>Stride (bytes):</td>
<td>8</td>
<td>read+write:</td>
<td>7 ns</td>
</tr>
<tr>
<td>Size (bytes):</td>
<td>32768</td>
<td>Stride (bytes):</td>
<td>16</td>
<td>read+write:</td>
<td>8 ns</td>
</tr>
<tr>
<td>Size (bytes):</td>
<td>32768</td>
<td>Stride (bytes):</td>
<td>32</td>
<td>read+write:</td>
<td>9 ns</td>
</tr>
<tr>
<td>Size (bytes):</td>
<td>32768</td>
<td>Stride (bytes):</td>
<td>64</td>
<td>read+write:</td>
<td>9 ns</td>
</tr>
<tr>
<td>Size (bytes):</td>
<td>32768</td>
<td>Stride (bytes):</td>
<td>128</td>
<td>read+write:</td>
<td>7 ns</td>
</tr>
<tr>
<td>Size (bytes):</td>
<td>32768</td>
<td>Stride (bytes):</td>
<td>256</td>
<td>read+write:</td>
<td>9 ns</td>
</tr>
<tr>
<td>Size (bytes):</td>
<td>32768</td>
<td>Stride (bytes):</td>
<td>512</td>
<td>read+write:</td>
<td>9 ns</td>
</tr>
<tr>
<td>Size (bytes):</td>
<td>32768</td>
<td>Stride (bytes):</td>
<td>1024</td>
<td>read+write:</td>
<td>8 ns</td>
</tr>
<tr>
<td>Size (bytes):</td>
<td>32768</td>
<td>Stride (bytes):</td>
<td>2048</td>
<td>read+write:</td>
<td>9 ns</td>
</tr>
<tr>
<td>Size (bytes):</td>
<td>32768</td>
<td>Stride (bytes):</td>
<td>4096</td>
<td>read+write:</td>
<td>8 ns</td>
</tr>
<tr>
<td>Size (bytes):</td>
<td>32768</td>
<td>Stride (bytes):</td>
<td>8192</td>
<td>read+write:</td>
<td>8 ns</td>
</tr>
<tr>
<td>Size (bytes):</td>
<td>32768</td>
<td>Stride (bytes):</td>
<td>16384</td>
<td>read+write:</td>
<td>8 ns</td>
</tr>
<tr>
<td>Size (bytes):</td>
<td>32768</td>
<td>Stride (bytes):</td>
<td>32768</td>
<td>read+write:</td>
<td>9 ns</td>
</tr>
</tbody>
</table>
Actual Test:
Raspberry Pi: L1 missing...

- Size (bytes): 65536 Stride (bytes): 4 read+write: 7 ns
- Size (bytes): 65536 Stride (bytes): 8 read+write: 8 ns
- Size (bytes): 65536 Stride (bytes): 16 read+write: 9 ns
- Size (bytes): 65536 Stride (bytes): 32 read+write: 8 ns
- Size (bytes): 65536 Stride (bytes): 64 read+write: 8 ns
- Size (bytes): 65536 Stride (bytes): 128 read+write: 10 ns
- Size (bytes): 65536 Stride (bytes): 256 read+write: 10 ns
- Size (bytes): 65536 Stride (bytes): 512 read+write: 14 ns
- Size (bytes): 65536 Stride (bytes): 1024 read+write: 16 ns
- Size (bytes): 65536 Stride (bytes): 2048 read+write: 18 ns
- Size (bytes): 65536 Stride (bytes): 4096 read+write: 20 ns
- Size (bytes): 65536 Stride (bytes): 8192 read+write: 18 ns
- Size (bytes): 65536 Stride (bytes): 16384 read+write: 8 ns
- Size (bytes): 65536 Stride (bytes): 32768 read+write: 8 ns
- Size (bytes): 65536 Stride (bytes): 65536 read+write: 9 ns
So logic...

- 32kB: no misses, 64kB misses
  - So it is a 32 kB cache
- On 64kB, a step at 64B
  - So it is *probably* a 64B line size...
- On 64kB, no misses when accessing 4 lines
  - So it is 4-way set associative
Actual Testing: Watching L2 Fail

<table>
<thead>
<tr>
<th>Size (bytes): 1048576</th>
<th>Stride (bytes): 4</th>
<th>read+write: 8 ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (bytes): 1048576</td>
<td>Stride (bytes): 8</td>
<td>read+write: 9 ns</td>
</tr>
<tr>
<td>Size (bytes): 1048576</td>
<td>Stride (bytes): 16</td>
<td>read+write: 12 ns</td>
</tr>
<tr>
<td>Size (bytes): 1048576</td>
<td>Stride (bytes): 32</td>
<td>read+write: 29 ns</td>
</tr>
<tr>
<td>Size (bytes): 1048576</td>
<td>Stride (bytes): 64</td>
<td>read+write: 61 ns</td>
</tr>
<tr>
<td>Size (bytes): 1048576</td>
<td>Stride (bytes): 128</td>
<td>read+write: 61 ns</td>
</tr>
<tr>
<td>Size (bytes): 1048576</td>
<td>Stride (bytes): 256</td>
<td>read+write: 61 ns</td>
</tr>
<tr>
<td>Size (bytes): 1048576</td>
<td>Stride (bytes): 512</td>
<td>read+write: 117 ns</td>
</tr>
<tr>
<td>Size (bytes): 1048576</td>
<td>Stride (bytes): 1024</td>
<td>read+write: 117 ns</td>
</tr>
<tr>
<td>Size (bytes): 1048576</td>
<td>Stride (bytes): 2048</td>
<td>read+write: 124 ns</td>
</tr>
<tr>
<td>Size (bytes): 1048576</td>
<td>Stride (bytes): 4096</td>
<td>read+write: 140 ns</td>
</tr>
<tr>
<td>Size (bytes): 1048576</td>
<td>Stride (bytes): 8192</td>
<td>read+write: 61 ns</td>
</tr>
<tr>
<td>Size (bytes): 1048576</td>
<td>Stride (bytes): 16384</td>
<td>read+write: 24 ns</td>
</tr>
<tr>
<td>Size (bytes): 1048576</td>
<td>Stride (bytes): 32768</td>
<td>read+write: 24 ns</td>
</tr>
<tr>
<td>Size (bytes): 1048576</td>
<td>Stride (bytes): 65536</td>
<td>read+write: 21 ns</td>
</tr>
<tr>
<td>Size (bytes): 1048576</td>
<td>Stride (bytes): 131072</td>
<td>read+write: 18 ns</td>
</tr>
<tr>
<td>Size (bytes): 1048576</td>
<td>Stride (bytes): 262144</td>
<td>read+write: 8 ns</td>
</tr>
<tr>
<td>Size (bytes): 1048576</td>
<td>Stride (bytes): 524288</td>
<td>read+write: 7 ns</td>
</tr>
<tr>
<td>Size (bytes): 1048576</td>
<td>Stride (bytes): 1048576</td>
<td>read+write: 8 ns</td>
</tr>
</tbody>
</table>
So on L2

- 512kB L2 cache...
- Again, 64B line size
  - And it is clearer this time
- **Looks like** 64-way Associative
  - But that’s weird, there could be other things going on here... Which is why this is no longer homework!
Can Already See How Caches “Fall Off a Cliff”

• For a memory-bound task...
  • Fit in L1 cache: AMAT 7ns
  • Fit in L2 cache: AMAT ~20ns
  • Exceed L2 cache: AMAT 100+ns

• Performance drops by an order of magnitude when you exceed the capabilities of the cache even by not that much!
Complications...

• We don’t ask you to do this as homework anymore:
  • There are a lot of additional complications on modern processors

• Memory fetching/prefetching...
  • L2 cache is starting to hit memory at a stride of 64, but...
  • Performance keeps getting worse until the stride is larger
  • Memory is probably transferring 256B at a time to L1 as well as L2...

• There may be a "victim cache"
  • A small fully associative cache that holds the last few evicted cache lines:
    So although L2 is only 16 way according to ARM’s documentation, still are getting good performance
    on the 32 way and 64 way test:
  • Bet on a 64-entry victim cache

• And this is on a simple modern processor
  • “Only” 4 cores, 2-issue in-order superscalar
More on Victim Caches...

• Observation: Conflict misses are a pain, but...
  • Perhaps a little associativity can help without having to be a fully associative cache

• In addition to the main cache...
  • Have a very small (16-64 entry) **fully associative** "victim" cache

• Whenever we evict a cache entry
  • Don't just get rid of it, put it in the victim cache

• Now on cache misses...
  • Check the victim cache first, if it is in the victim cache you can just reload it from there
Another Pathological Example...

- int j, k, array[256*256];
  for (k = 0; k < 255; k++){
    for (j = 0; j < 256; j++){
      array[256*j] += array[256*j + k + 1]
    }
  }

- This has a nasty pathology...
  - It experiences no spacial locality of note: both array reads and the array write are a stride of 256 entries
  - And it also generates a huge number of capacity misses
But a minor tweak...

- int j, k, array[256*256];
  for (j = 0; j < 256; j++){
    for (k = 0; k < 255; k++){
      array[256*j] += array[256*j + k + 1]
    }
  }

- And now it runs vastly better as it changes from stride 256 to stride 1 and stride 0...
  - Stride 0 == best case temporal locality
  - Stride 1 == best case spacial locality
Blocking-out Data

- Very common motif
  - for (int i = 0; i < len_A; i++) {
    for (int j = 0; i < len_B; j++) {
      fn(A[i], B[j])
    }
  }
- "Do something for every pair of elements"
- If B fits in the cache, we’re good
- But if B doesn’t:
  - The inner iteration is going to be dominated by capacity misses as B has to keep being reloaded
    - So there is no more temporal locality for B[j]
  - But the fetches of A are still going to be fine because a lot of temporal locality for A[i]
- And its going to be very good up until the moment things break down
  - Caches performance doesn’t tend to degrade gracefully: Instead you get step-functions
Implications...

- If one array is a lot bigger than the other...
  - It should be the outer one in the loop: It doesn't generally matter if the outer one doesn't fit
    - Since there is tons of temporal locality for the outer array that the cache will take advantage of

- But if both don't fit, you need to be better
  - for i = 0; i < len_A; i += BLOCK {
    for j = 0; j < len_B; j += 1 {
      for k =0; k < BLOCK; k++ {
        if (k + i) < len(A) {
          fn(A[i*block+k],B[j])
        }
      }
    }
  }

- Now have a lot more temporal locality for the entries of both A and B, if BLOCK is set correctly
Another Cache: Branch Predictor

- In our simple pipeline, we assume branches-not-taken
  - Always start fetching the next instruction
- If a branch or jump is taken...
  - Then we have to kill the non-taken instructions so they don't cause side effects
- But both branches and jumps are PC relative...
  - So if we can quickly look at the instruction and decide 'eh, probably taken/not' we can compute the new location for the PC if we can guess right
    - Which for \texttt{jal} we always can, but branches we need to guess
- Idea: \textit{branches} have temporal locality!
  - Loops: \texttt{for} (\texttt{x} = 0; \texttt{x} < \texttt{n}...)
  - Rare conditionals: \texttt{if} (\texttt{err}) ...
A Simple Branch Predictor

- Have an N entry, direct-mapped memory
  - EG, a 1024x1b memory

- If fetched instruction is a branch...
  - Check if the bit for pc[12:2] is set in this special memory during IF...
    - If so, set next PC to PC + branch offset fetched (in ID probably, if not IF)
    - Set bit in pipeline to say "branch predicted-taken"

- When actually evaluating branch in EX...
  - Set pc[12:2] in the branch predictor to branch taken/not-taken status
  - If branch taken but predicted not-taken
    - Kill untaken instructions
  - If branch not taken but predicted taken
    - Kill predicted instructions
Where to do this?

- **If we could, do it in IF**
  - Now on correct predictions we will always be right

- **If we can't, do it in ID**
  - First non-taken instruction will be fetched regardless, so we need more complex control logic in determining which to kill

- **This does complicate the pipeline a fair bit, but worth it!**
  - If we can predict in IF in the 5 stage pipeline:
    - Correct predicted branches -> **no stalls**
    - Incorrect prediction -> 2 stalls for killed instructions
  - If we can predict in ID:
    - Correct predicted taken branch -> 1 stall
    - Correct predicted not-taken branch -> 0 stalls
    - Incorrect predicted taken branch -> 2 stalls
    - Incorrect predicted not-taken branch -> 1 stall
A related cache: return target location...

- Observation:
  - On RISC-V, you call a function with `jal` or `jalr` with the return set to `ra`
  - And you return with a `jalr` writing to `x0` with the source register as `ra`

- So let’s maintain a small stack in hardware...
  - Whenever we see `jal` or `jalr` writing to `ra`:
    We write PC+4 into the stack
  - Whenever we see `jalr` reading from `ra` and dest as `x0`:
    We predict the top of this stack as the next PC, and pop this stack

- Result: We should *always* correctly predict a function return address...
  - Works as long as we don't exceed the stack depth: once we hit that we will start getting misses
And A Final Related Cache: Branch Target Buffer

- Function calls using jal we will never mispredict on RISC-V
  - Since they are all PC relative we can do the add in the decode where we change our PC prediction
- But so much today is object-oriented programming which uses jalr:
  C++ and Java object calls are equivalent to calling pointers to functions
  - foo.bar() is implemented as something like this:
    lw t0 0(a0) # Get the pointer to the "virtual function table" in the object
    lw t0 8(t0) # Get the pointer to the function to actually call
    jalr ra t0  # Do a JALR to call bar(),
      # with the object foo as the first implicit argument
- So cache this as well:
  - On a jalr which writes to ra rather than x0.
    Look in a small cache for the address to predict to based on current PC
  - When evaluating the jump, set the value in this cache to the address used
- It is the x86 equivalent of this cache that is part of one of the Spectre vulnerabilities
Caches and Multiprocessors...

• These days practically every computer is a multiprocessor
  • Since that is the only way we know how to increase computation by throwing more silicon at the problem
  • But we can't make single-processor performance \textit{worse} in this process
    • So these processors \textit{must} have significant caches
  • And because the L1 caches are integrated into the pipeline, to prevent structural hazards each processor must have its \textit{own} caches

• What happens if multiple processors are accessing the same piece of memory?
Multiple Processor Reading Memory?

- No problem!
  - Each processor just caches the data independently
- There is *no* issue with multiple processors reading the same thing
  - Their own caches have a unique copy...
    But the values should always be the same
Multiple Processors Writing?

- We need **coherency**: Writes from one processor *must* be reflected in memory that other processors read after some short period of time
- There have been processors made without this, but it is impossible to program these
- Goal is to guarantee the following property:
  - If processor A writes to memory location L, *within time* $T$, all other processors will see the updated data
Idea: Broadcasting Messages...

• We need a way for the processors to communicate
  • So we have some sort of fabric
    • It could be a shared bus
    • It could be something looking more like a packet-switched computer network

• Each processor (or more precisely its cache) can send and receive messages
  • Requests are "broadcast", a single sender can send a message to anybody...
  • Replies may or may not be broadcast: Can go to everyone or could go to a specific recipient
Broadcasting Writes...

- Processor $A$ wants to write to physical location $L$ for the first time...
  - First do a write-allocate (if you don't already have a copy)...
  - It then sets the *dirty* bit on the cache
  - And *broadcasts* a message that "I am writing to $L"

- All other processors which receive the message
  - Do I have address $L$ cached?
    - If no: Do nothing
    - If yes: *invalidate* the entry in the cache
      - *Snooping* on requests:
        - Term comes from when all processors shared the same memory bus to communicate with the memory
Broadcasting Reads

• If there is a miss in the upper level of the cache in the processor...
  • Broadcast a read request: "Hey, does anyone have location $L$?"

• If nobody has written to this location...
  • The memory controller/common cache just does a fetch and returns it:
    Just like any other cache miss

• If a processor has this location with the dirty bit set...
  • It goes "Hey, I have this"
  • It **flushes** the entry (writing the value to memory) and clears the dirty bit
  • It then says the new value to the requesting processor
Why Does This Work?

• If processor A wants to write to a non-dirty line...
  • If the element is in the cache...
    • It will write, and all other processors invalidate: If they then want to read that location they will have to broadcast that request
  • If not, it performs a read first...
    • So if reads are correct, it is going to be correct from there

• If processor B wants to read...
  • If the element is in its cache...
    • It is correct, because if someone else wrote to that location it would be invalid already
  • If the element is not in its cache...
    • It will get the correct value from either another processor or the right location in memory
    • And that other processor will now know it can't write because the dirty bit got cleared
CPU-0 reads byte at 0xdeadbeef...

Upper Level Cache & Memory...

CPU-0: I want to read 0xdeadbeef0
Cache Controller: value of 0xdeadbeef0 is 0xf00dd00dca11c003
CPU-1 reads byte at 0xdeadbeef...

Upper Level Cache & Memory...

CPU-1: I want to read 0xdeadbee0
Cache Controller: value of 0xdeadbee0 is 0xf00dd00dca11c003

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
<th>V</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>deadbee0</td>
<td>f00dd00dca11c003</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
<th>V</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>deadbee0</td>
<td>f00dd00dca11c003</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

CPU-0

CPU-1
CPU-1 writes data at 0xdeadbee0...

Upper Level Cache & Memory...

CPU-1: I want to start writing to 0xdeadbee0

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
<th>V</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>deadbee0</td>
<td>f00dd00dca11c003</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>deadbee0</td>
<td>cafef00dda016666</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

CPU-0

CPU-1
CPU-0 reads byte at 0xdeadbeef...

Upper Level Cache & Memory...

CPU-0: I want to read 0xdeadbee0
CPU-1: value of 0xdeadbee0 is 0xc0ffed00da016666
So Enter a New Miss Type: **Coherence**

- A coherence miss occurs when two processes want to access the same data
  - Coherence misses are caused only by *writes*, not *reads*
    - The write will invalidate all *other* caches
- It means there is an **anti-pattern** that can cause performance artifacts on multiprocessors
  - Multiple processes reading to the same memory? Sure!
  - But if one starts writing to that memory...
    - The *other* processor will start getting coherency misses
    - But such misses also only go up to the shared cache:
      Why multiprocessors, when possible, use a shared cache between all processors
- Reasonably easy to avoid *with proper program structure*
Oh, and *Incoherence misses*

- What about when we have multiple processes running on the same processor
  - A modern x86 creates two "virtual" processors which share resources ("Hyperthreading"/"Symmetric Multithreading")
  - After all, if that 6 issue super-scalar really has a CPI of ~1, why not run two different programs at the same time

- If those two processes have the same working set...
  - Great!

- If those two processes have different working sets...
  - This effectively acts like reducing the cache size with the corresponding increase in the miss rate
Virtual Memory Paging As A Cache...

- How virtual memory works we will cover later...
- But for now, its easy to model as a basic cache...
- Your program is given the illusion of as much RAM as it wants...
  - But this breaks things unless it doesn’t really want all of it!
- Idea: Virtual memory can copy "pages" between RAM and disk
  - The main memory thus acts as a cache for the disk...
Virtual Memory’s “Cache” Properties

- **Associativity**: Effectively fully associative
- **Replacement policy**: Approximate LRU
- **Block size**: 4kB or more
  - Some argue it should now be 64kB or 256kB these days
- **Hit time**...
  - Call it 0
- **Miss penalty**...
  - Latency to get a block from disk: 1ms or so for an SSD...
    Or put in clock terms, a 1 GHz clock -> 1,000,000 clock cycles!
  - Or if you have a spinning disk: 10ms or so...
    So **10,000,000 clock cycles**!
Implications

• As long as you don’t really use full capacity, Virtual Memory is great...
  • Basically as long as your working set fits in physical memory, virtual memory is great at handling a little extra...

• But as soon as your working set exceeds physical memory, your performance falls of a cliff
  • The system starts **thrashing**: Repeatedly needing to copy data to and from disk...
  • Similar to **thrashing the cache** when your working set exceeds cache capacity (but the cliff here is much steeper!)
  • You have almost certainly experienced it: Suddenly your computer becomes super, **super** slow