Cache

Motivation

- Let’s consider a typical “high end” PC...
- I went to Alienware’s site[1] and pulled up their highest priced gaming system
- Here’s its specs:
  - AMD Ryzen™ Threadripper 2950X (16-Core, 40MB Cache, 4.4GHz Boost)
  - 64GB Quad Channel HyperX™ DDR4 XMP at 2933MHz
- Only $7999! Such a bargain!

[1] https://www.dell.com/en-us/shop/dell-desktop-computers/alienware-area-51-threadripper-edition-gaming-desktop/spd/alienware-area51-r7/dpcwCfxR7maxh

Explanation

- So what do those features mean?
  - The CPU is 4.4GHz
    - That means one clock tick every 227ps
    - Picosecond = one trillionth of a second
  - The memory is DDR4-2933MHz
    - That means ~23500 MB/sec = 23.5 GB/sec

Calculations

- I found that the average x86-64 instruction is 4 bytes long
- So: CPU requires $3.6 \times 10^9 \times 4$ bytes per second just to be supplied with enough instructions to be kept constantly busy
  - That's about 14.4GB/sec
- But that's not the whole story...

Calculations

- Many instructions read or write memory
- I found that on average about 40% of them do
  - For simplicity, let’s assume each access is 32 bits
  - Many will be 64 bits, so we’re being optimistic
- So we need another $0.4 \times 3.6 \times 10^9 \times 4$ bytes per second
- Our running total: 20.16GB/sec
- But that’s not the whole story...

Calculations

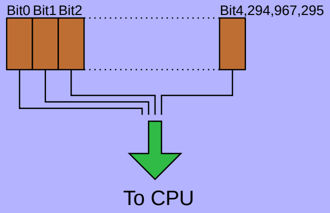
- Suppose we have a 16 core CPU
- Then we need to feed *sixteen* cpu's with 20.16GB/sec = 322GB/sec
  - Clearly, this outstrips our memory subsystem's speed (23.5GB/sec)
- What to do?

Channels

- One trick: When reading, fetch data from two (or four) memory modules simultaneously
  - Called "channels": One channel per module
- When CPU fetches data, read from *both* channels in parallel
- This doubles (or quadruples) throughput
- But that's not the whole story...

Latency

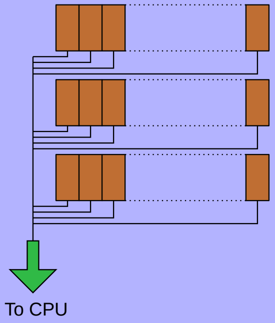
- We face a problem: Latency
- Imagine a 4GB memory module
- We *could* structure it internally like so:



- But that means 4 billion wires from memory to bus. Yuck!

Better

- Memory chips are internally designed as a grid structure:



- This means we have 64K × 64K cells
- That’s easier to manage than single row of 4 billion

Better Yet

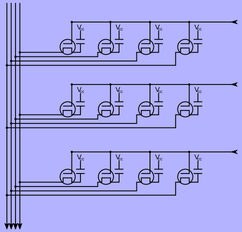
- We can arrange memory in several *banks*
  - Ex: For 4GB of memory: Have 8 banks of 512MB each
  - Each bank has 16K x 32K cells

Problem

- Selecting row/column/bank takes time
- For DDR-2900, about 55ns to initiate transfer
- That means the CPU ticks almost 200 times before data begins streaming in
- But that’s not the whole story...

Refresh

- DRAM gets incredible densities
  - Today, you can get a 64 *billion* byte module that’s just 8.2cm × 3cm (about 3“×1¼”)
  - Each DRAM bit is extremely simple: Just a capacitor and a transistor

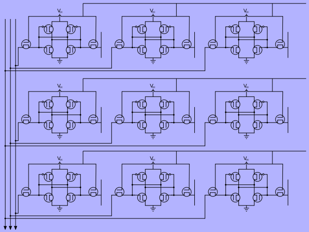


Problem

- Capacitors are inherently “leaky”
  - Charge dissipates over time
  - DRAM circuitry must periodically “refresh” memory cells
  - They’re unavailable for access when that’s happening
- A typical refresh interval might be once every 4μs
- A refresh might take around 50ns
- Remember: CPU ticks every 250ps, so 50ns is a significant amount of time
  - Memory controller can sometimes “delay” refreshes a bit if CPU wants to access a given region of memory

Solution

- We'd like to address problems of DRAM memories
- One possibility: Use SRAM
- Drawback: More expensive (physically larger)
  - So we can't get as much capacity in given physical space

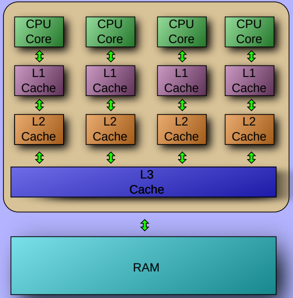


Cache

- Def: (dictionary.com): “A hiding place...for ammunition, food, treasures, etc.”
- A CPU cache is similar
  - Small capacity chunk of memory that’s fast
- If we arrange for cache to (nearly) always have the data we need: We can run at (nearly) full CPU speed

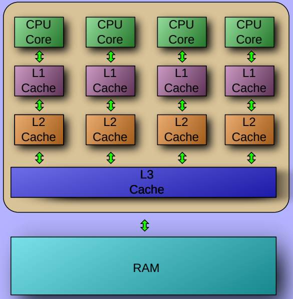
Hierarchy

- Often, caches are arranged in a hierarchy
  - Level 1 (L1), Level 2 (L2), Level 3 (L3)
  - As caches get larger, they usually get slower



Hierarchy

- Example: Each core has its own L1, L2 cache
- L3 cache is shared by all cores



Structure

- Some architectures have L1 $\subseteq$ L2 $\subseteq$ L3
- Others make them disjoint
  - If data is in L1, it's not in L2
  - And if data is in L2, it's not in L3

Example

- Intel Skylake architecture
  - L1 = 32KB, 4 cycle latency
  - L2 = 256KB, 12 cycles
  - L3 = 2MB per core, 44 cycles

Challenge

- Need to try to keep cache hit rate high
- Modern CPU's manage cache in the background
  - So we don't explicitly interact with it
  - But: Must understand how it works so we can best use it

Design

- Recall: There's a startup cost for pulling data from memory
- So: We'd like to spread that cost out as much as possible
- Thus, fundamental quantum for cache is not bytes or words
- Called a *line*
- Ex: On Skylake: Line size = 64 bytes

Line

- Ex if we ask for data at address 1,000: Cache will pull in data from addresses 960...1023
  - 960 = 15*64; 1024=16*64
- Idea: If we ask for data at address x, we'll probably ask for data at address x+1 soon

Note

- Many modern CPU's maintain separate caches for instructions & data
- Rationale: Code & data usually widely separated in RAM
- So we don't use same cache for both
- Ex: Skylake:
  - L1 = 8KB/8KB split
  - L2, L3 = *unified*

Question

- Where does data get placed in cache?
- Ex: Suppose cache is 32KB (typical size today)
  - 32KB = 512 lines in cache
- Layout of one cache entry: Several fields
  - Valid bit (is there valid data in this slot?)
  - Dirty bit (is cache same as main-memory copy?)
  - Tag bits: Identify which line of memory this holds
  - Data: The line itself
- At bootup or context switch, all “Valid” bits get set to zero
  - Also known as cache flush

Direct Mapped

- Suppose we’re accessing address A
- Let L = A mod 64
- Store data to cache line L, set tag=A
    - Remember: A line is smallest quantum for cache
    - We can’t load “half a line” to the cache
- This is simple & easy
- So what’s the drawback?

Drawback

- Conflicts: If two addresses map to same line, they'll evict each other from cache
- If cache is 8KB (512 lines), *any* two addresses 8KB apart *must* conflict
- Power-of-two array sizes are very common in computer programs, so this problem appears frequently

Example

- Copy data between two arrays:

```
unsigned A[2048];
unsigned B[2048];
for(i=0;i<2048;++i){
    A[i] = B[i];
}
```

- Consider what happens here...

Process

- CPU fetches B[0]
  - Brings in whole line.
  - Suppose it ends up in cache slot (line) 10.
- CPU writes to A[0]. This requires bringing in that line.
  - It will end up in slot 10, so it evicts data from B
- Then CPU fetches B[1]
  - This results in a *cache miss*, so the line is brought in to slot 10
- CPU writes to A[1].
  - Another miss!
- And so on

Fully Associative

- Suppose we're accessing address A
- See if line is in cache: Check every tag entry for match with A
- If none: Scan cache for free line
  - Free = valid bit = 0
  - If no free line: Select *least recently used* line
  - Load data to that place, set tag=A
- This is the most flexible option
- So what's the drawback?

Drawback

- It’s too difficult to check all 512 lines
- Every line would need to be checked in parallel
  - And: Must happen within one access operation
  - Remember: 4 cycles per access!

N-Way Associative

- Compromise between the other two strategies
- Suppose we have an 8-way set associative cache
- Each line could go in one of 8 possible places
- Ex: 8KB cache, 64 byte lines, 512 lines total
- Suppose we're accessing memory address A
- If we pull in line L, it can map to (L mod 512), (L+64), (L+128), (L+192), ...
  - Why 64 here?
  - 64 = numberOfLines ÷ lineSize
  - 64 = 512 ÷ 64
- This is the scheme most modern CPU's use

Example

- Intel Skylake:
  - L1 = 8 way associative
  - L2 = 4 way associative
  - L3 = 16 way associative

Line State

- Suppose two CPU cores accessing same area of RAM
- We'd have two copies of that line in two L1 caches
- This results in data inconsistency. What's the "true" value of that memory area?
- CPU's resolve this by using *cache protocol*

MESI

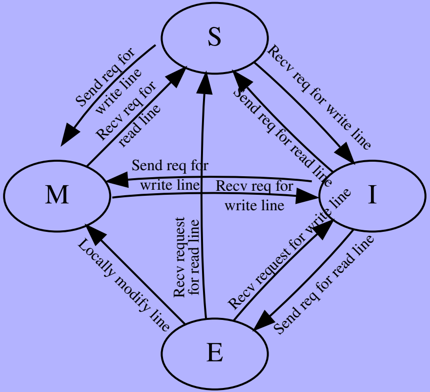
- Common protocol: MESI
- Each cache line can be in one of four states:
  - Modified
  - Exclusive
  - Shared
  - Invalid

Definition

- Invalid = This cache line doesn’t contain useful data
- Shared = This line is also in some other core’s cache, and it’s the same in both of them
- Exclusive = This core’s cache is the only one that has this data, but it’s the same as in main memory
- Modified = This core’s cache is the only one that has this data, and it’s been modified from main memory

Transition

- Transition diagram:



Result

- Worst case: Two cores are manipulating variables that are in same line

```
int A[10];
int B[10];
int C[10];
int D[10];
void core1(){
    for(int i=0;i<10;++i)
        B[i] = A[i]+1;
}
void core2(){
    for(int i=0;i<10;++i)
        D[i] = C[i]+1;
}
```

- Consider what happens...

Result

- A, B, C, D are in same line
- Core 1 loads line for A into its cache, marks as Exclusive
- Core 1 can write to B immediately
- Suppose core 2 starts executing

Result

- Needs line. Broadcasts message asking for it
- Core 1 puts contents of line on bus, moves it to Shared
- Core 2 loads contents from bus, marks as Shared
- Core 2 writes to D[0].
  - Broadcasts message on bus to take ownership
  - Marks its copy of line as Modified
  - Core 1 marks its copy as Invalid

Result

- Now, core 1 wants to read A[1] and write B[1]
  - Broadcasts message on bus asking for it
  - Core 2 moves from Modified $\rightarrow$ Shared
  - Core 1 moves from Invalid $\rightarrow$ Shared
- Etc.

Timing

```
#define N 1000
int A[N][N], B[N][N], C[N][N];
int main(int argc, char* argv[]){
    for(int i=0;i<N;++i){
        for(int j=0;j<N;++j){
            A[i][j] = rand() & 0xff;
            B[i][j] = rand() & 0xff;
        }
    }
    clock_t st,et;  struct tms x;
    st = times(&x);
    for(int i=0;i<N;++i){
        for(int j=0;j<N;++j){
            int total=0;
            for(int k=0;k<N;++k)
                total += A[i][k] * B[k][j];
            C[i][j]=total;
        }
    }
    et = times(&x);
    //prevent optimizing it out
    cout << "C=" << C[rand()&0xff][rand()&0xff] << "\n";
    cout << et-st << "\n";
    return 0;
}
```

Result

- When I run it, I get a time of 1.03 sec
- But suppose we change this line: total += A[i][k] * B[k][j]; to: total += A[i][k] * B[j][k];
  - Obviously, this wouldn't be correct matrix multiplication
  - But ignore that for now...
- The time goes down to 0.54 seconds!
- Why?

Locality

- C stores arrays in *row major order*
- So what happens when we have the line total += A[i][k] * B[k][j]
- Suppose k=0
  - Load cache with line containing A[i][0]...A[i][15]
  - Load cache with line containing B[0][j]...B[0][j+15]
  - Do multiply, store result
- So far, so good

Locality

- What happens on next iteration?
  - CPU loads A[i][1] into a register
    - Fetches from cache. Very fast.
  - CPU loads B[1][j] into a register
    - Not in cache!
    - A miss...Slow
- And this happens for every single array element

Locality

- But consider what happens if the statement is: total += A[i][k] * B[j][k];
- Again, let k=0
  - Load cache with line containing A[i][0]...A[i][15]
  - Load cache with line containing B[j][0]...B[j][15]
  - Do multiply, store result

Locality

- What happens on next iteration?
  - CPU loads A[i][1] into a register
    - Fetches from cache. Very fast.
  - CPU loads B[j][1] into a register
    - In the cache: Fast
- When do we get cache misses?

Misses

- Every time k crosses over a value of 16
  - 16, 32, 48, 64
- Note: Some CPU’s have hardware prefetchers
  - If memory access pattern is regular: It starts pulling in consecutive lines
  - So we might not have any misses at all!

Prefetcher

- Prefetcher looks for access patterns and anticipates them
- Ex: Going linearly forward or backward in an array
- How to best structure accesses?

Compiler

- We can help the compiler out:
  - Avoid globals (they're stored far from locals in RAM)
  - Simple control flow (try not to nest if's, switch, loops, etc.)
  - const can help
- CPU also provides 'prefetch' instruction to allow you to specify addresses you'll need soon

Write Caching

- Caches also handle writes
  - When you write to cache line, CPU collects all writes and issues them in background
  - Called “write-back” strategy

Sources

- Intel Optimization manual
- http://www.7-cpu.com/cpu/Skylake.html
- http://valgrind.org/docs/manual/cg-manual.html
- https://gist.github.com/jboner/2841832
- DDR3 vs. DDR4: Four Generations, Raw Bandwidth by the Numbers. http://www.corsair.com/en-eu/blog/2015/september/ddr3_vs_ddr4_generational
- Rajinder Gill. The Ins and Outs of Memory Addressing. https://www.anandtech.com/show/3851/everything-you-always-wanted-to-know-about-sdram-memory-but-were-afraid-to-ask/2
- DocMemory. http://www.simmtester.com/page/news/showpubnews.asp?num=168
- Thomas Schwarz, S.J. COEN 180. http://www.cse.scu.edu/~tschwarz/coen180/LN/DRAM.html
- Thomas Schwarz, S.J. COEN 180. http://www.cse.scu.edu/~tschwarz/COEN180_06/LN/sram.html
- Micron Corp. https://www.micron.com/products/datasheets/%7B3D323C4D-6BC7-4193-908D-E99AD746AA4E%7D?page=30#topic=concept_03583372F4E04674BE846190D4B79C89
- Intel Corp. Intel 64 and IA-32 Architectures Optimization Reference Manual.

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org