Christopher Martin, Jennifer Wolfe, Geoffrey Corey
May 31, 2013
CS 472 Lab 4

Part 1

meow meow meow

This paper explains the structure of memory subsystems in use on modern commodity hardware, illustrating why CPU caches were developed, how they work, and what programs should do to achieve optimal performance by utilizing them.Current technology allows for fast CPU's but slow memories. Slow RAM must be resolved with hardware designs like CPU cache and memory controllers. Commodity Hardware refers to low cost, simple computers. Static RAM is made from six transistors. State is maintained with power and no refreshing is required. Dynamic RAM is made from one transistor and one capacitor. DRAM has several design complications all based around the fact it uses a capacitor. The capacitor has to be refreshed due to current leaking out and after a read. Delays accrue due to charging and discharging the capacitor. DRAM is smaller and costs less to make compared to SRAM, which is why it is used for main memory.Important timing criteria are CAS Latency (CL), RAS-to-CAS delay (tRCD), RAS Precharge (tRP), Active to Precharge delay (tRAS) and Command Rate.In the 90's CPU clock speeds got faster than DRAM clock speeds. Code and data has temporal and spatial locality. So a block of memory will likely be reused again or a region will be linearly accessed. So temporarily storing a section of memory is a good way to improve upon having to continuously access slow main memory. This temporary memory is made of SRAM and is built into the CPU. It is called cache. Cache is typically 1/1000th of the size of the main memory. Modern CPU's have multiple caches named L1, L2, L3. When the cache does not hold the needed data it has to be fetched from RAM. Such a situation is called a cache miss and is very bad for performance. Virtual memory (VM) splits memory addresses into physical and logical. Memory addresses used in the CPU are logical and are converted into physical addresses by the Memory Management Unit (MMU). NUMA (Non Uniform Memory Access) hardware requires special care from the OS and the applications. The Linux Kernel provides support for the NUMA. Programmers can positively or negatively influence a programs performance in memory-related operations. Some other opportunities that a programmer has is with physical RAM access and L1 caches, OS functionality and L1 caches, etc. Cache and memory are some of the many tools available to help programmers understand performance characteristics. Scaling-up of the number of CPUs or cores is expected in the future.

The problem at hand is that memory is slow and cache is underutilized resulting in low performance. Cache is dived into lines of 32 or 64 bytes. Registers in the CPU are faster than L1 cache, L1 cache is faster than L2, etc. There are three types of cache misses: Compulsory, Capacity, Conflict. Code cache can be optimized by locality and size. Data cache can be optimized by prefetching, preloading, and anti aliasing memory. Compilers can do undesired tricks. Turn them off when trying to optimize cache yourself. Linearizing data at run time also helps to utilize cache. Avoiding aliasing by using local variables and minimizing global pointers. The gcc flag fstrict-aliasing can try to prevent aliasing based on data types. Programming language features can either help or hurt performance. C++ abstraction can lead to aliasing. The restrict keyword in 1999 ANSI/ISO C prevents aliasing with pointers.

Lyrics

Now this is the story all about how
My life got flipped, turned upside down
And I'd like to take a minute just sit right there
I'll tell you how I became the prince of a town called Bel-air

Part 2

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define KB 1024
#define MB 1024 * 1024

int main(int argc, char **argv)
{
        unsigned int meow = 256 * 1024 * 1024;
        static int arr[4 * 1024 * 1024];
        int lengthMod;
        unsigned int i;
        double timeTaken;
        clock_t start;
        int sizes[] = {1 * KB, 4 * KB, 8 * KB, 16 * KB, 32 * KB, 64 * KB,
                                        128 * KB, 256 * KB, 512 * KB, 1 * MB, 1.5 * MB,
                                        2 * MB, 2.5 * MB, 3 * MB, 3.5 * MB, 4 * MB};

        int results[sizeof(sizes)/sizeof(int)];
        int s;

        // for each size to test for ...         meow
        for (s = 0; s < sizeof(sizes)/sizeof(int); s++) {
                lengthMod = sizes[s] - 1;
                start = clock();
                for (i = 0; i < meow; i++) {
                        arr[(i * 16) & lengthMod] *= 10;
                        arr[(i * 16) & lengthMod] /= 10;
                }

                timeTaken = (double)(clock() - start)/CLOCKS_PER_SEC;
                printf("%d, %.8f \n", sizes[s] / 1024, timeTaken);
        }

        return 0;
}
```