Sean Penney and Paul Atkinson
May 16, 2014
CS 472 Lab 4

Memory Optimization

The memory operation paper talks about practices to improve memory optimization, both in general and in more specific circumstances.

The paper starts with why memory optimization is actually necessary. It is necessary because memory technology is progressing at a very slow pace when compared to how much CPUs are getting better. Thus, memory optimization is needed so our applications aren't bottlenecked by the memory rather then the processor. Moreover, there are a lot of easy practices you can develop that subtly help the run time of applications.

There are three main ways of optimizing code: rearranging, reducing and reusing. Rearranging is when you have variables that are accessed at the same time, you should have the variables declared next to each other. Reducing is reducing the amount of code to run, which obviously makes it more efficient. However, sometimes something that looks like less code in C is actually more lines of machine code, so it doesn't actually save time. In these circumstances, you want to compile the code and then analyse it to see if there are some ways of shorting the amount of code that needs to be run. The final R is reusing, which means reusing as much code as possible so the program doesn't have to load in more then it has to. This usually means making functions if you have to do the same thing more then once.

There are also three main ways of screwing up the cache: compulsory misses, capacity misses and conflict misses. Compulsory misses are the first miss that happens when the data is read for the first time. While these are inevitable, it is good practice to avoid these as possible. Capacity misses don't happen quite as much any more, but it is when the cache is too small to hold all the active data you want to hold. Sometimes you can hit capacity if you are accessing too much data in-between successive uses. Conflict misses happen when you have two things that are mapped to the same line in cache.

Software pre fetching and preloading can also help speed up memory operations. However, if you fetch too early, the data can take up unnecessary space or may be evicted from the memory before it is ever actually used. If the memory is fetched too late, then everything will be bottlenecked by it loading. Preloading is sort of the same as prefetching, however it isn't sure what is going to be needed so it guesses. Sometimes it guesses wrong and it ends up wasting time loading in the correct thing, but overall it usually saves more time then it loses.

What Every Programmer Should Know About Memory

This paper introduces starts by introducing why caching is important. Since mass storage and memory systems have improved slowly compared to the CPU and other component, the memory creates the bottleneck in a system.

Current hardware is discussed, focusing on CPUs and RAM types. A CPU is connected via a bus to the Northbridge which contains the memory controller. The implementation of the Northbridge effects the type of RAM chips that can be used (DRAM, Rambus, SDRAM, etc). The Southbridge communicates with PCI, PCI Express, SATA, USB buses, etc. Sometimes the Northbridge is connected to other controllers, which allows more memory since there is more total available bandwidth.

An Overview of RAM types are given, focusing on Static RAM and Dynamic RAM. SRAM is much more expensive, and use is CPU caches where the connections are small. The coast of the memory controller, motherboard, DRAM module, and DRAM chip is based on the number of address lines. Another memory technology is Synchronous DRAM, which works relative to a time source.

The paper next goes into more detail about the cache, and techniques for using it. All loads and stores have to go through the cache, so the connection between the CPU core and the cache is specialized and fast. One technique that helps with the efficiency of the cache is using separate code and data caches. The memory regions needed for code and data are different, which logically leads to different caches. Also, the decoding step is usually slow, so caching decoded instructions can speed up execution. These days more levels of cache have been added, since increasing the size of the level 1 cache led to much greater costs.

Caches can be implemented in different ways. A fully associative cache allows for each cache line to hold a copy of any memory location. Fully associative caches are most practical for smaller caches. A direct mapped cache is fast and easy to design. A direct mapped cache has a comparator, multiplexor, and some logic to select valid cache line content. A set-associative cache combines the best parts of the full associative and direct-mapped caches. One technique to speed up the cache is by prefetching the next cache line, so that when the next line is used it is halfway loaded.

One aspect of the cache is coherency, which should be transparent for the coder. If a cache line is modified, the effects should be the same as if only main memory was modified. This can be implemented using write-through cache or write-back cache. Write-through cache means that if the cache line is written to, the processor immediately writes the cache line to main memory. Write-back policy states that the processor does not write the cache line to main memory right away. The cache line is marked as dirty, and when the line is dropped in the future the data will then be written back. The write back is widely used, due to the greater efficiency.

The next section of the paper focuses on the virtual memory. The Memory management Unit of the CPU implements the virtual address space. With virtualization, the cost of cache

misses is higher than without virtualization.

NUMA (Non-uniform memory architecture), which at a basic level allows a processor to have local memory that is cheaper to access than local memory of other processors, has become more common. The OS has to be designed in a way that supports NUMA. The physical RAM assigned to a process' address space should be from local memory for max efficiency.

The next section discusses how programmers can take advantage of this information about memory in order to write more efficient programs. When a memory store operation reads a full cache line and then modifies the cached data without using it again immediately, performance takes a hit. Non-temporal write operations can alleviate these issues by writing the contents directly to memory. Steps a programmer can take to write efficient code are move the structure element which is most likely to be the critical word to the beginning of the structure. Elements should be accessed in the order in which they are defined in the structure if access order does not matter to the program. Other good programming practices include reducing the code footprint as much as possible, executing the code linearly, and aligning code when possible.

Issues that arise with multi-threaded applications includes concurrency, atomicity, and bandwidth. Concurrency means that when more than one thread shares the same date, coordination is needed.

The next section brings up benchmarking tools that allow a programmer to measure performance under different theoretical positions. Finally, the author discusses future technology that will improve memory efficiency.