Drake Bridgewater & Ryan Phillips
May 16, 2014
CS 472 LAB 4

Summary of What Every Programmer Should Know About Memory article

As the documented suggested I skipped to section 6 as other assignments need our effort too. The content is actually really interesting and I we will have to read it all at a later time, but it is intended to advise software developers "on how to write code which performs well in the various situations" [1, p. 2]

A CPU without a cache is rare as caches can cover up most of the cost of random access this is due to the advancements we have made in optimization of un-cached read and write. As a programmer performance is essential in this world of "I want now." Therefore as Drepper states "changes affected the level 1 cache... will likely yield the best results" [1, p. 49] This can be achieved by aligning code and data and improving locality. This is easier said then done as optimization of L1i is done at the instruction level meaning that unless the programmer writes assembly then you are relaying on the compiler, but as a programmer you still can "indirectly determine the L1i use by guiding the compiler to create better code." [1, p. 55] Some ways of creating better code as far as speed is to when a function is called a single time to ensure it is executed inline; when using gcc you can add the always_inline function attribute at compilation time to instruct the compiler to move functions in with the code. This can become a problem the function is called multiple times as it will create many more instructions. For the next level of cache L2 and higher you want your code to dynamically adjust itself to the cache line size.

Another way to hide latency, which is implemented on many of today's processors, is the use of pipelining, out-of-order execution, and prefetching. Prefetching can be triggered by certain events (hardware prefetching) or explicitly requested by the program (software prefetching). [1, p. 61] With software prefetching the programs don't have to change, but ensuring the access patterns don't happen across page boundaries is key.

Multi-threading can increase overall speed, but there is concurrency optimization, atomicity optimization, and bandwidth considerations that need to be considered. Concurrency optimization is optimizing the code to prevent other threads from accessing cache lines in 'E' (exclusive) state. This can be done by grouping like access levels such as read-only and read-write commands.

The set of provided atomic operations varies throughout the different processor architectures. For x86 and x86-64, the same instructions can be used in atomic or non-atomic mode. So at runtime, if the situation allows it, you can bypass atomic lock for libraries (that would otherwise be threadsafe), allowing you to avoid the performance costs of the atomic operations.

Bandwidth can be better utilized by careful placement of threads on available cores. This is something that isn't done in an informed way by the scheduler, so it's up to the programmer to define the thread affinity. The goal is avoid excess reads/writes to memory by having cache sharing cores access the same data.

Summary of Memory Optimization article

The article begins with the following problem: CPU speeds are improving at a much faster rate than memory speeds. The burden of trying to avoid this blatant memory bottleneck then falls on the cache.

Next, the cache is briefly explained by way of it's component parts, the possible mappings, the hierarchy, and some typical specs. There are 3 types of cache misses: compulsory, capacity, and conflict. But there are 3 techniques that can be used to help avoid these: rearrange, reduce and reuse.

How can you make sure your code is optimized to use cache efficiently? Well there is a pretty extensive list of practices/techniques that will help. Encapsulation and the object oriented programming pattern will both typically hurt cache performance, but monolithic function on the other hand, will help. Another import factor is code size. If you can rewrite portions of your code in asm, there is a good chance that your version will be smaller than the compiled version. You should also avoid inlining, unrolling, and large macros. Additionally, compressing data can help and you can also pad your data to ensure that it aligns to cache lines.

Prefetching and preloading are two powerful techniques that, when used properly, will enable your code to better utilize the cache. The goal of prefetching is to get the 'timing' right, so that data is available exactly when needed (not too early or too late). Preloading is the process of loading multiple pieces of data into the cache and then processing it, rather than alternating load, process, load process. Loops will typically do the latter rather than the former if you aren't careful.

It's also important to keep in mind structures and the layout of your code. If two variables are commonly used together, well then store them in a struct; and you can even improve on that by placing the fields/variables right after one another to ensure that they will be stored contiguously in memory. Another related technique is hot/cold splitting. Also, if a struct contains many variables, note the sizes used, use padding if necessary, or use a decreasing size structure, from largest variable types at the top, to smallest variables types at the bottom.

When using tree data structures, there are a lot of options for how your can rearrange ndoes and reduce the size. The proper order for storing the data that constitutes a tree really depends on how you are typically going to be accessing that data - i.e. will you more commonly do a breadth-first or depth-first traversal? One of the best things you can do is to linearize your data at runtime - it gives you the best possible spatial locality and it makes data easily prefetchable.

Aliasing is multiple references to the same storage location and it causes all sorts of optimization problems via poisoning the data cache, negatively affecting instruction scheduling, etc. Better languages and compilers can help, but there are things you can do as a programmer, too. For example, when you are unrolling, if you consume all inputs before producing outputs, you can reduce refetches.

Another pretty general problem is that higher levels of abstraction nave a negative effect on optimization. This is true for C++ (And I assume this applies to many other languages, too).

Some additional general tips for avoiding aliasing include: minimizing use of globals, pointers, references (by passing small variables by value); use local variables as much as possible; and don't take the address of variables. You can also use fstrict aliasing in gcc to help you avoid certain aliasing issues.

How will you know if your optimizations are effective? One of the best things you can do is study the generated code, and this will help you build intuition about how to write code that produces simpler output (when compiled to asm). You can also profile your cache utilization using a number of different commercial products, but a good place to start is to just use gcc with the p flag and mcount.

References

[1] Ulrich Drepper. What every programmer should know about memory. November 2007.

[2] Christer Ericson. Memory optimization. March 2003.