Austin Herring

CS 540 Homework 5

November 12, 2015

1. The included “commented-cache-hw-nc.c” has been commented with explanations of most of the code. For a more general description, see immediately below.

This program performs experiments to characterize the time it takes to perform memory accesses. The memory accesses are parameterized in two ways: one, the size of the working memory set being used (characterized by the "csize" variable in the code), and two, the size of the stride being used to access memory (characterized by the "stride" variable in the code), particularly a sequential array in memory. For different cache/set sizes from 1024 to 1048576, stride sizes from 1 to half the cache size are used. The entire array is traversed using this stride length, for a particular number of samples, with one read and one write at every location "visited" by the stride. The samples are taken until the total time for all of the samples is greater than a second, and this time is recorded. Additionally, the number of "steps" (i.e., iteration of samples) taken is recorded.

Next, the time *just* for executing the code of the loops for the samples, and not for accessing memory, is approximated. This is done by repeating the samples, for the same number of "steps" (or iterations), but instead of accessing memory for reads and writes, a temporary variable, which should (especially because of compiler hints) be kept in a register, is accessed, that is, read and written to, instead. Because the register accesses should be, essentially, "free," especially in comparison to the reads and writes, the total time taken for these samples is essentially the amount of time it takes just to execute the instructions for the loops.

In the end, the time taken to traverse the second set of loops, using the temporary register variable, is subtracted from the time taken for the strided memory accesses for that cache size, to give an approximation of the total amount of time spent on reads and writes alone. Then, the average time taken for a read and write is calculated by dividing by the total number of samples taken and elements

accessed. After this, the average time is printed out, and the program continues on to the next stride length and then, eventually, to the next cache size, until all combinations of stride lengths and memory sizes have been tried and their average read and write times reported.

2. The included “graph.png” file includes a graphical logarithmic plot of the output of the program. The data was collected under the following conditions:

- The code was compiled without optimizations using the GNU C Compiler, i.e., using the command “gcc cache-hw-nc.c”
- The data was collected on the tux cluster

A variety of interesting information can be gleaned from this surface plot. For one, of course, notice that when the stride is on the smaller side, before around the 10^5 mark, the average amount of time to read a single element increases drastically, generally, as the stride increases. This is because as the stride increases, the spatial locality of the references to memory are drastically decreasing. Normally, due to the fact that most programs have spatial locality, when memory is referenced, some surrounding number of bytes, a cache block, will all be brought into the cache. When locality is high, this means less references to main memory itself, because some of the data will have already been brought into the cache, and cache reads and writes are quicker than the same in memory. When the stride increases, however, less of the data will share cache blocks, and, eventually, for a large enough stride, each element will have to be read/written individually, once the stride is larger than the size of a cache block. Notice that, however, for some reason, after about the 10^5 mark, as the stride increases further, the read time actually starts to decrease. This must be due to some strange way that the program is accessing so few elements that there's greater variability and less read/write time overall perhaps, or something else strange is happening.

The plot also gives some insight into the way the working set size, and temporal locality, affects program performance in regards to average read/write time. In particular, note that as the working set

size increases, and therefore the amount of time between access of the same element increases, generally, the average read/write time increases as well. However, also note that it looks like there are three separate “levels,” or “ridges,” as they are called in the *Computer Systems* book, one from sizes greater than about 10^4 to about halfway between 10^5 and 10^6, one from there to about halfway between 10^6 and 10^7, and from from there out to the end of the plot, at 10^7.

Each of these levels are the cases where the entire array being accessed is able to fit within a certain cache or is forced back into main memory. For example, on the lowest ridge, the entire working set fits into the L1 cache. To explain more, while the first sample is being taken, the array will be brought into the L1 cache from main memory. For all subsequent samples, because all array elements were just recently accessed, then, the array will already be in the L1 cache, so all memory accesses will be incredibly cheap. Note that as long as the array fits in the cache, the average access time will stay approximately the same, creating the ridges. However, eventually, the working set size starts to get too large to all fit in the L1 cache, though some of it may still. Once the array completely no longer fits in L1, though, a new level is created. On this level, the entire working set can fit into the L2 cache, and the same reasoning as above, i.e., main memory no longer has to be accessed until the array gets too big to fit into the cache, explains why for these working set sizes the average read/write time remains relatively constant, though because the L2 cache is slower to access than L1, the ridge is “higher.” However, eventually, the size gets too large for the L2 cache, and, again as explained above, the read/write time slopes up to create a new level. Having examined the processor information for the tux cluster, there is no L3 cache, so this highest, slowest level must be for accessing main memory itself.

All of this explanation explains what can be learned from the working set size parameter. That is, as working set size increases, so does average read/write time, in general, but as long as the working set continues to fit in the same cache level, the size can be increased without a penalty in read/write time, but once the size exceeds that level of cache, the access time will jump fairly significantly, when the processor must rely on the next level of cache or memory.