CSE 520 Computer Architecture – Spring 2019
Ravi Bhushan ASU#1214347783

Task 1: Memory Hierarchy Performance Measurement

Method used:
Two tools were used to study the Memory Performance Measurement:
1- clock()
2-perf

clock()

Usage:
While performing the measurements, clock was used to get time just before and just after the memory read/write operation. The difference between this time gave the time required to access all the elements of the array. This divided by array size gave the time required to access a single element of the array.

Limitations of clock:
1- clock is not very precise
2- It cannot measure instances less than 1 milliseconds

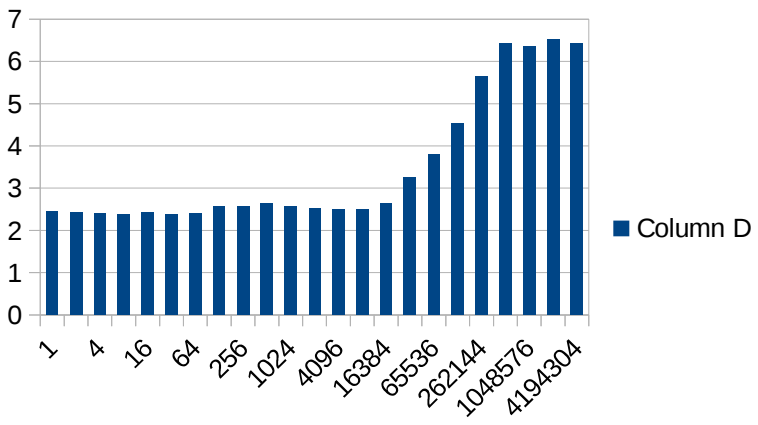
How those limitations were overcome and other code optimizations:
1- Code to access each array element was run for a particular number of times depending on the size of array. For example, for array sizes less than 16384, the operation to fetch each elements of the array was run 100000 times to get a reliable reading. For array sizes between 16384 and 2097152, the operation to fetch each elements of the array was run 100 times. For array sizes greater than 2097152, the operation to fetch each elements of the array was run only 10 times. This reduced the test time and also optimized the readings.
2 - CPU affinity was set so that the code runs on a single CPU
3- For linear access, experiments were performed in the strides of 1,10 and 20. This gave an optimal picture of how memory access was affected by increasing the stride of access.
4- Each array element was equal to the line size of L1 cache(64 bytes). This was done so that each element occupy one line size.
5- Items were prefetched in the cache during read operations, so that only the read time from cache was recorded.
6- The iteration time of for loop used to iterate through the array was very negligible. So, it was also included in the calculations and it did not make much of a difference in the readings.

Graphs:

Linear Write with Stride 1

x-axis : Time in nanoseconds

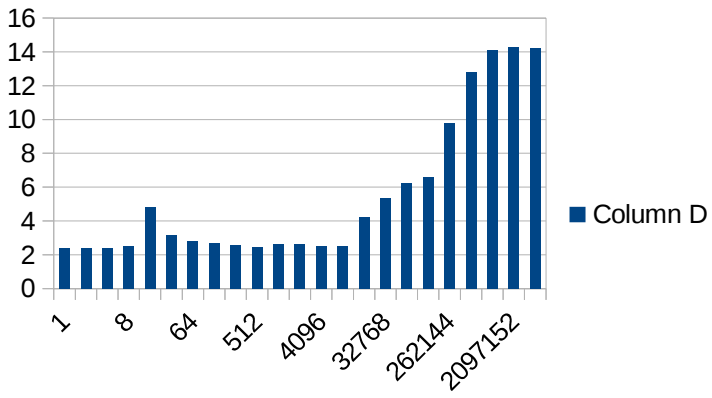
y-axis : Array size



Linear Write with stride 10

x-axis : Time in nanoseconds

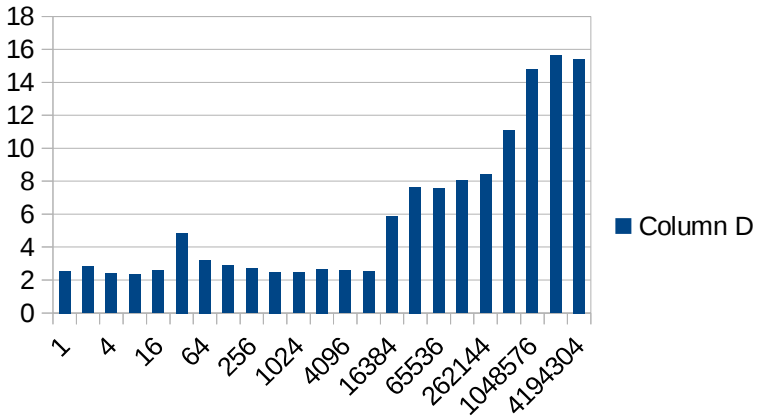
y-axis : Array size



Linear Write with stride 20

x-axis : Time in nanoseconds

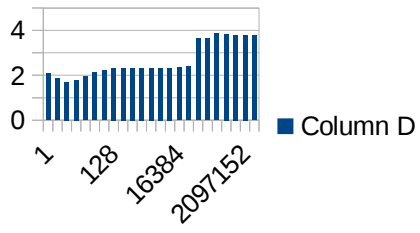
y-axis : Array size



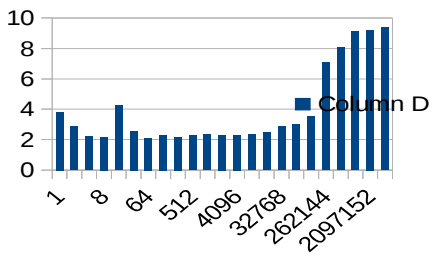
As you can see, the jump at 16384 and 262144 denote the cache boundaries.

Other graphs are below:

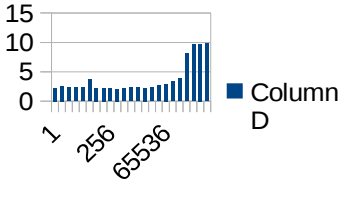
Linear read with stride 1



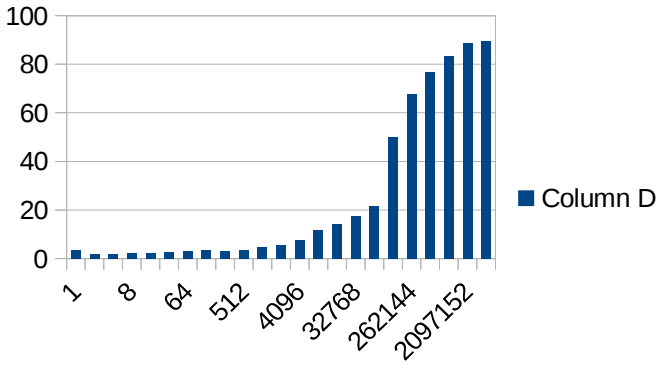
Linear read with stride 10



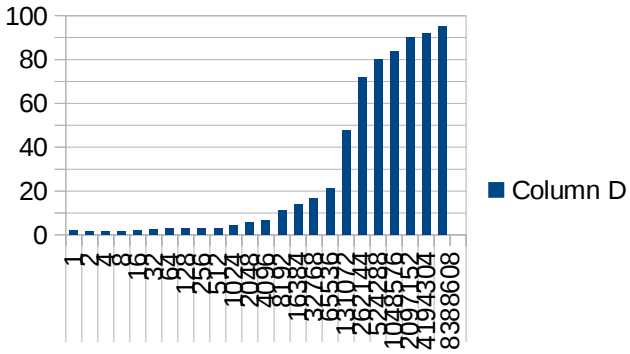
Linear read with stride 20



Random Write



Random Read



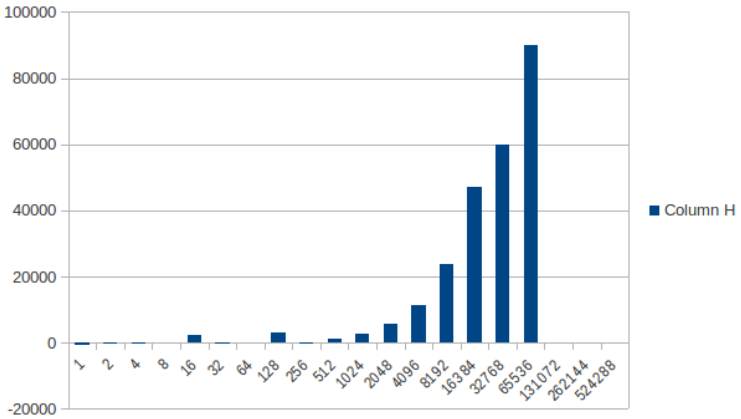
perf

We used two files to measure memory performance using perf.
One file code_performance_measurement_perf_gem5.c, was not having the cache fetch code.
Another file memory_Performance_measurement_perf_gem5.c was having the cache fetch code.

The time taken by the first file to run was subtracted from the time taken by the second file to run. This gave the time taken to access all the elements of the array. This time was divided by array size to get the time taken to access single element of the array.

Perf logs were stored in an output file for analysis later and plotting of graphs.
Graphs plotted using perf were erronous.

Below is a Random Read graph



Task 1: Gem5 Simulator

The logic to measure the memory access time for a Gem5 simulator is same as that of perf.
One file code_performance_measurement_perf_gem5.c, was not having the cache fetch code.
Another file memory_Performance_measurement_perf_gem5.c was having the cache fetch code.

The time taken by the first file to run was subtracted from the time taken by the second file to run. This gave the time taken to access all the elements of the array. This time was divided by array size to get the time taken to access single element of the array.

Gem5 gives us the following output in the console.

command line: ./build/X86/gem5.opt --outdir=./tests ./configs/example/se.py --cpu-clock=1GHz --cpu-type=DerivO3CPU --caches --l1i_size=32kB --l1i_assoc=2 --l1d_size=64kB --l1d_assoc=4 --l2cache --l2_size=256kB --l2_assoc=16 -c code_performance '--options=2 1 2048'

Global frequency set at 1000000000000 ticks per second
warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (512 Mbytes)
0: system.remote_gdb: listening for remote gdb on port 7003
**** REAL SIMULATION ****
info: Entering event queue @ 0. Starting simulation...
warn: ignoring syscall access(...)
warn: ignoring syscall access(...)
warn: ignoring syscall access(...)
warn: ignoring syscall mprotect(...)
warn: ignoring syscall mprotect(...)
warn: ignoring syscall mprotect(...)
warn: ignoring syscall mprotect(...)
Exiting @ tick 609501000 because exiting with last active thread context

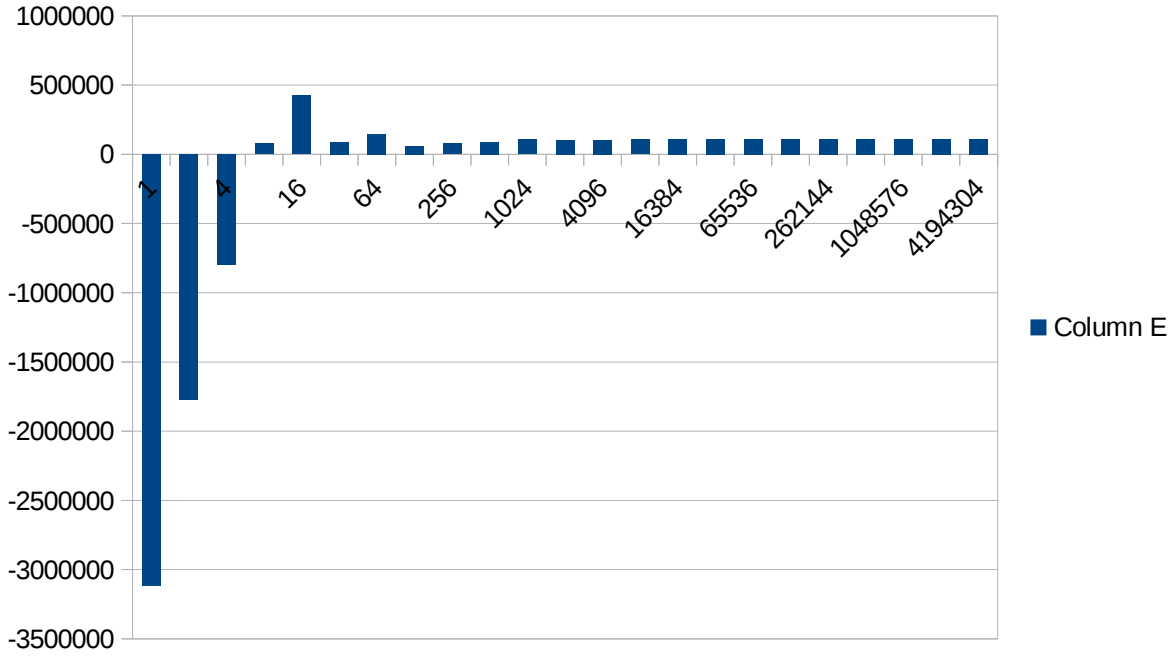
.....
.....

command line: ./build/X86/gem5.opt --outdir=./tests ./configs/example/se.py --cpu-clock=1GHz --cpu-type=DerivO3CPU --caches --l1i_size=32kB --l1i_assoc=2 --l1d_size=64kB --l1d_assoc=4 --l2cache --l2_size=256kB --l2_assoc=16 -c memory_performance '--options=2 1 2048'

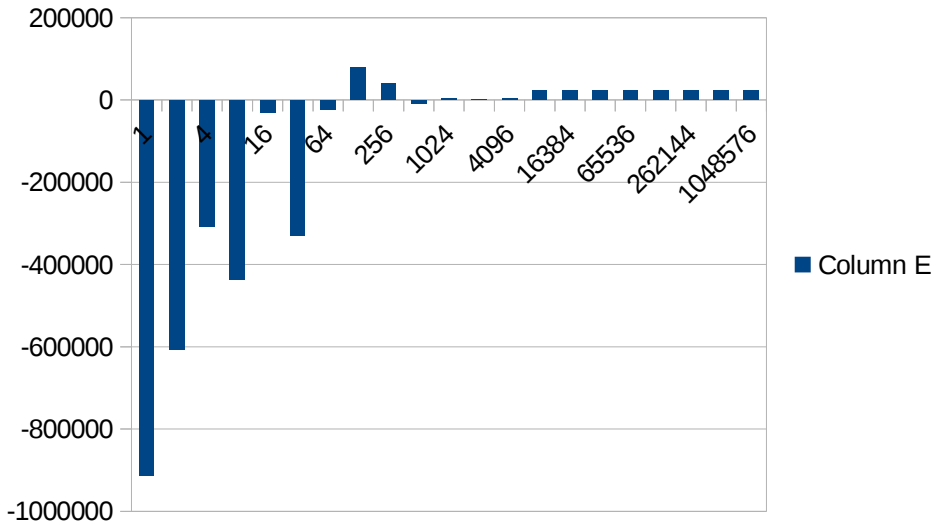
Global frequency set at 1000000000000 ticks per second
warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (512 Mbytes)
0: system.remote_gdb: listening for remote gdb on port 7001
**** REAL SIMULATION ****
info: Entering event queue @ 0. Starting simulation...
warn: ignoring syscall access(...)
warn: ignoring syscall access(...)
warn: ignoring syscall access(...)
warn: ignoring syscall mprotect(...)
warn: ignoring syscall mprotect(...)
warn: ignoring syscall mprotect(...)
warn: ignoring syscall mprotect(...)

Exiting @ tick 610465000 because exiting with last active thread context

Note that the first log is of code_performance and second log is of memory_performance. When we subtract the second tick(610465000) from the first(609501000) we will get the number of ticks required to access the array of size 2048. This tick can be divided by the Global frequency and then size of array to get the access time of a single element.
However, the graphs did not come as expected.
Graphs:
Write Linear



Read Linear



(a) Is this average access time represents the latency or throughput of memory hierarchy? Why? How did you mitigate the overhead in measurement, or how did you improve the accuracy of your measurement?

Yes, the average access time represents the latency of memory hierarchy. This is because with increasing array size, the number of cache misses increases and the data gets fetched from slower memory. But, with array of lesser size, they can be accommodated in faster caches.
Accuracy improvements have been mentioned earlier.
(b) What makes the access time different for linear access pattern and random access pattern? Is there difference for read and write? Why?
Linear pattern stores the elements consecutively while random pattern stores it randomly and so not in same cache.
Yes, read and write will be different as read is reading from cache while write is writing to cache.