Characterize the Cache Performance of a CPU Under Shared and Non-shared Conditions

Shanti Modi - A53305577 - shmodi@eng.ucsd.edu

Documentation:

Revisiting the aim - We have a machine whose cache characteristics, like levels and sizes are to be determined based on the read and read/write latencies. We also have to characterise the effect of multithreading nuances of the values.

In my program, I have declared a 2D Data structure called 'cache' of variable size. The size is iteratively selected starting from 1KB to 64MB . It is made up of an array of structures, each structure is an array of 16 4Byte Integers. These specifications are under the assumption that each cache line is 64 Bytes. So, each of my structures occupies an entire cache line, and I have as many structures as the number of cache-lines. The idea is to access the first element in each structure as that will determine if a new cache block has to be fetched or it's the subsequent elements in a cache block which is determined by the byte offset.

```
typedef struct {
        u_int32_t array[CACHE_LINE_ELEMENTS];
} cache_line;
cache_line *cache;
cache = (cache_line*)malloc(index_size*sizeof(cache_line));
```

I have run this programs for each of read and read/write scenario, with the threads ranging from 1 to 8 based on my laptop spects. It has 1 socket with QuadCore CPU and 2 HW MultiTreads are supported.

Latency Measurement:

For my data structure of N bytes, I'm accessing the first element of every cache line size and measuring the total latency for that read or read/write access. For eg, if my data structure size is 64KB, I'm measuring the time taken for reading those 1KB reads. I'm collecting multiple such time value samples by repeated iterations to eliminate any errors. The number of iterations can be changed at the precursor.

1. I'm using an instruction called **__rdtscp()** (instead of the regular __rdtsc(), an x86 instruction which reads the current value of the processors time stamp counter. **The RDTSCP instruction is not a serializing instruction, but it does wait until all previous instructions have executed and all previous loads are globally visible.** Here, a load is considered to become globally visible when the value to be loaded is determined.
   Source: https://www.felixcloutier.com/x86/rdtscp
   But it does not wait for previous stores to be globally visible, and subsequent instructions may begin execution before the read operation is performed. We can use a memory barrier when we measure r+w.

2. By only reading the first element which majorly contributes to the difference in latency, I'm eliminating the noise and fine tuning my values.
3. We are using random initialization of pointers to point to different cache lines as to **trick the prefetcher and used array chase operations**. I have employed two layers of randomization :
    a. Initialized pointers and randomly assigned values
    b. Swapped them randomly

```
for(int i = index_size-1; i >= 0; i--)
        swap(&trick1[i], &trick1[rand() % (i+1)]);
```

Compile time configuration:

I'm using the gcc compiler to compile my C program. As the parameters come, I'm using -o and the flag native which eliminates any optimizations that the compiler performs like loop unrolling, value predication etc.

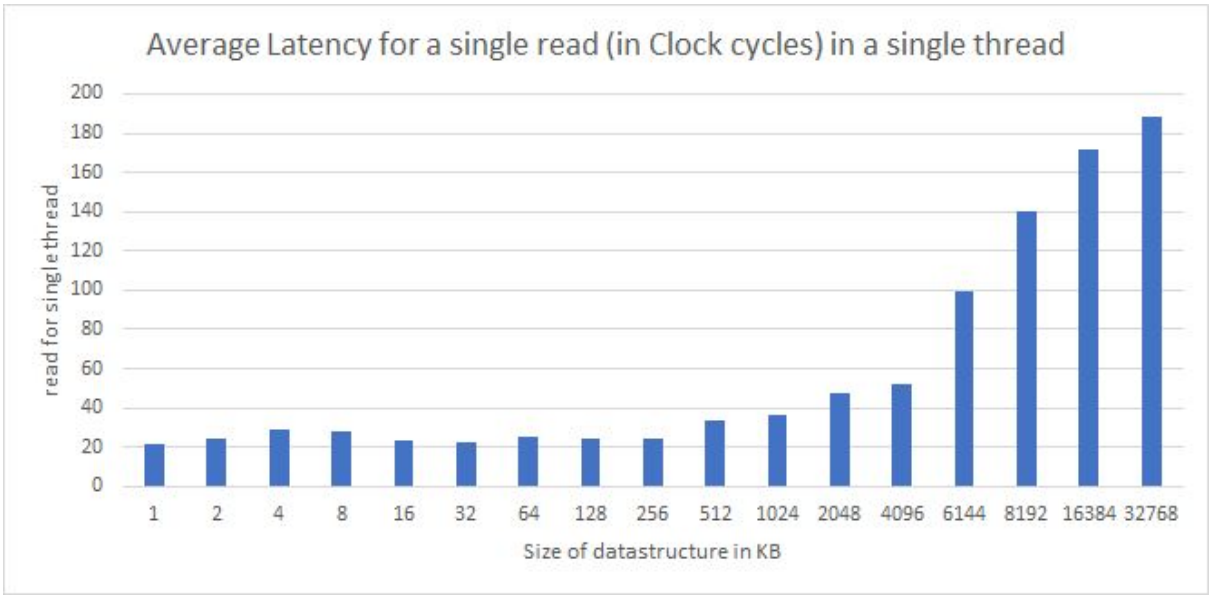
Compilation command : gcc -pthread -march=native -o <object_file> <program_file>

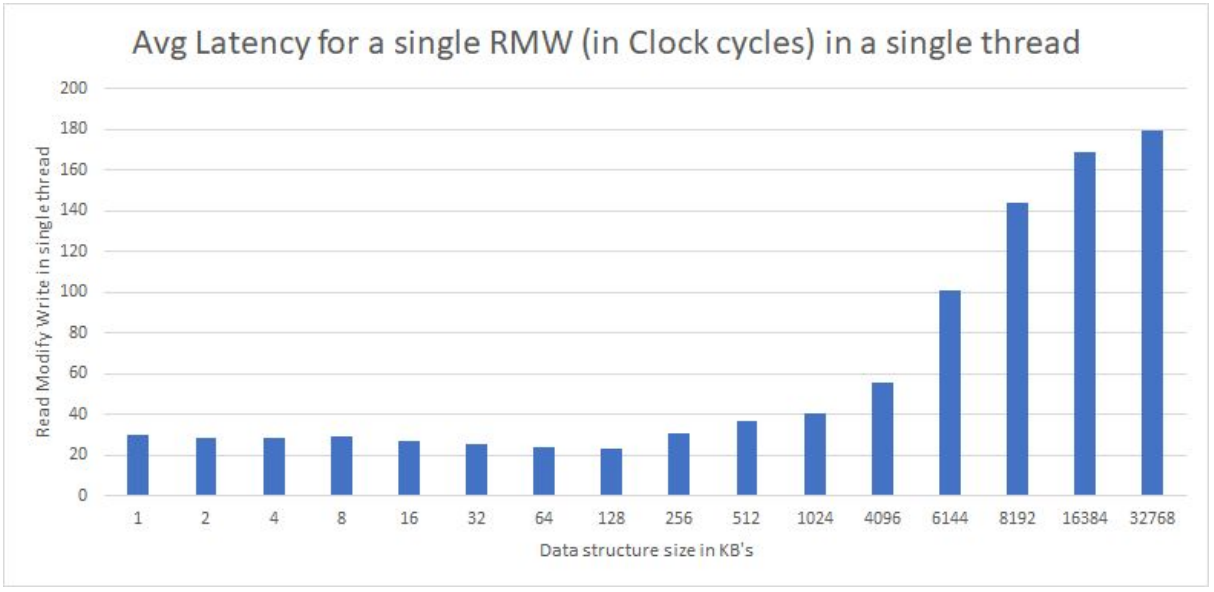
I have also declared the address that I have to read or read/write as **volatile** which tells the compiler that this variable value may change anytime and to leave out any optimization related to it.

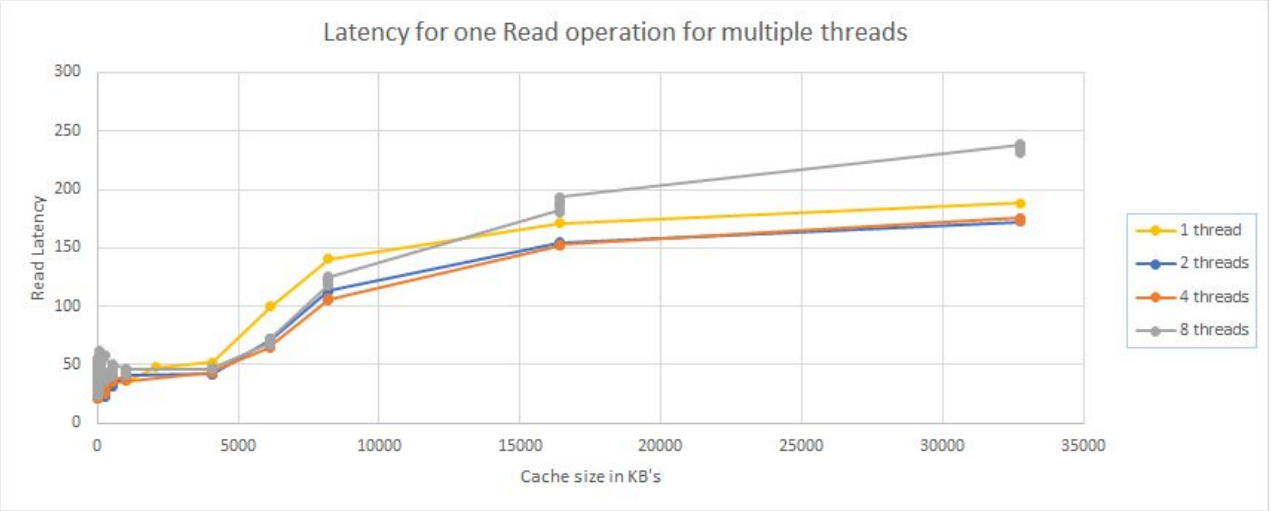
System dependent functions:

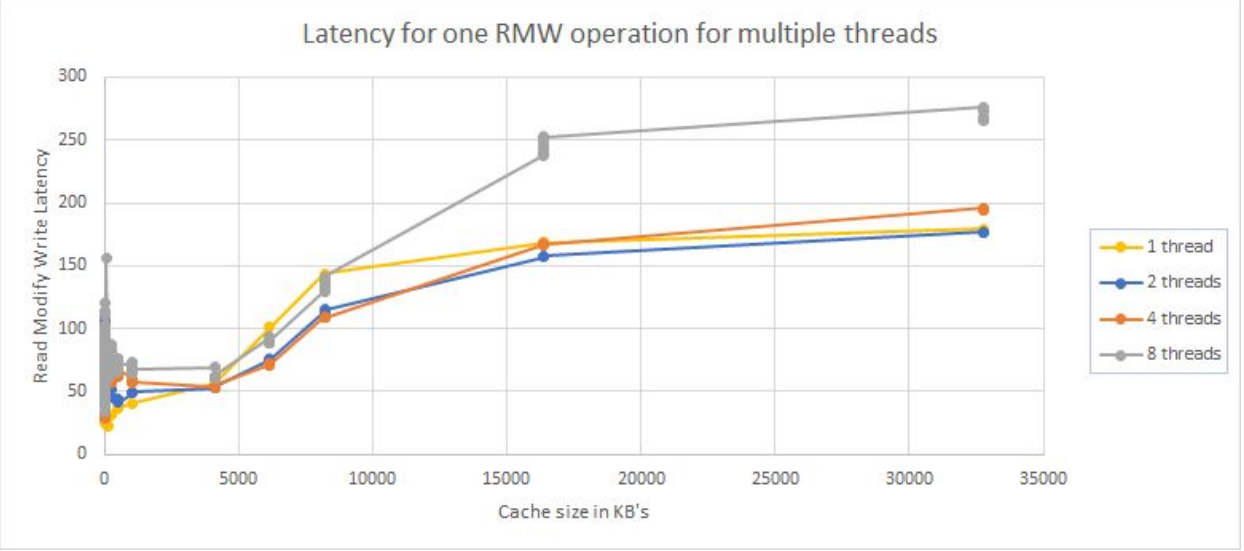
1. **Threads**: I have run this programs for each of read and read/write scenario, with the threads ranging from **1 to 8** based on my laptop specifications. It has 1 socket with QuadCore CPU and 2 HW MultiTreads are supported. The program asks the user to give the number of threads we want to characterize for and hence it's a runtime configuration.
2. **Levels of Caches :** Since I'm not hardcoding any number of levels, we can observe the pattern of the curves and determine not only the cache size but also the number of levels. The only thing is to watch out the size of the data structures that I'm considering. We need a max Cache size to be able to stride through the ranges of cache sizes to consider.
3. **Cache flush function** : As a precaution, everytime I fetch the contents into cache inorder to measure access latency, I'm flushing it out in every iteration so as to not mess with the next latency measurements. Since I'm using intel x86, I'm using a function called __mm_clflush() which eliminates the variable from all level of caches. More at https://www.felixcloutier.com/x86/clflush

Graphs:


Average Latency for a single read (in Clock cycles) in a single thread
read for single thread
200
180
160
140
120
100
80
60
40
20
0
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
6144
8192
16384
32768
Size of datastructure in KB


Avg Latency for a single RMW (in Clock cycles) in a single thread
Read Modify Write in single thread
200
180
160
140
120
100
80
60
40
20
0
1
2
4
8
16
32
64
128
256
512
1024
4096
6144
8192
16384
32768
Data structure size in KB's





Analysis on graphs:

Read-only on a single thread:

The idea of the program is to observe points where there is a change in slope. As expected, as data structure size increases, though we are reading a single 4Byte Integer, the latency increases as the data structure will not fit in the cash anymore.

For eg: if our L1 cache size is 256KB and we are considering a datastructure of 512KB, 256KB of it is accessible in our L1 cache and the rest of it from L2 cache. Hence we see a change in slope at this point.

Read-modify-write on a single thread:

The same above argument holds good for the RMW operations as well. I have followed the example given in the project document on what constitutes an atomic Read-Modify-Write operation.

As expected the RMW operation latency values are higher than the corresponding read only latencies. If we observe the trends in both the graphs **together,** we can tell that the slope changes at

256KB, 1MB, 6MB.

Hence we can tell that **there are 3 levels of cache** in the machine the data was collected. Now, it is pretty obvious to categorize each of them as higher or lower level caches based on their latency values.

Latency of 256KB < Latency of 1MB < Latency of 6MB

Hence,

L1$ size is 256KB
L2$ size is 1MB
L3$ size is 6MB

There are two ways where we can cross check if our values are correct :

1. Lscpu command on the terminal :

```
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              8
On-line CPU(s) list: 0-7
Thread(s) per core:  2
Core(s) per socket:  4
Socket(s):           1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               142
Model name:          Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz
Stepping:            12
CPU MHz:             2112.000
CPU max MHz:         2112.0000
BogoMIPS:            4224.00
Virtualization:      VT-x
Hypervisor vendor:   Windows Subsystem for Linux
Virtualization type: container
```

By looking up the CPU specifications, I could verify that the values I obtained are indeed correct.

2. Using the cache flush command mentioned above, we can flush a value out, read it and document the value. Unlike ARM, Intel has a ready to use cache flush command.

Reading more from graphs:

Analyzing the presence of multiple threads should tell us if the caches are **internal** to the core. Though, more or less, the graphs fall in the same region, there slopes and the way the difference in the slopes differ various cache regions should give us many details. For eg: Comparing the difference in the slope from the previous region, we can tell that this lower level cachce is inclusive or exclusive. But since we don't have two exact cache hierarchy machine, one with inclusive lower level cache and one with exclusive lower level cache, we will not be able to conclude the exclusive nature of the cache.

Looking at the slope differences for different number of threads in a particular region, we can tell the cache traffic. Since we have two threads for one physical processor, the cache traffic can be from the same core or different core. It depends on how the scheduler is scheduling the threads when we create two threads using the pthread function. If it creates two threads on the same physical processor, then the conclusions will be different than the otherwise case.

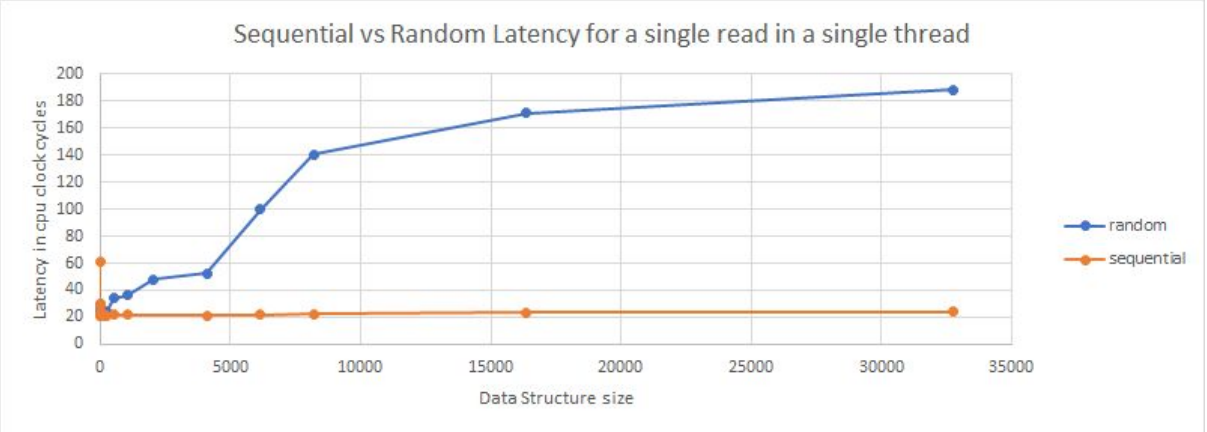
Read vs RMW graphs:

When it comes to stability, **Read only graph looks more stable**. We have two explanations for this. For single thread - this is because, for RMW, we are using a rsctp which does not really ensure previous **store commit**. Hence, the RMW is not really sequential everytime. The next load can overlap with the present store. We can use a memory barrier to make it stable but as discussed it takes exhaustive amount of time and could not proceed with it. For multiple threads, since they use the same higher caches, **shared cache traffic** increases, invalidation protocols will be sent across all nodes and hence will not give us stable results.

```
addr = &cache[trick1[i]].array[0];
start = __rdtscp(&junk);
junk = *addr ;
junk = junk & 0x7FFFFFFF;  //for RMW
*addr = junk; //for RMW
end = __rdtscp(&junk);
```

Extra credit:

As discussed, to trick the prefetcher, I used pointers, used random funciton in c and swapped them arbitrarily. If I don't do any of those tricks and just access through the first element of every cacheline of my data structure, following is the data obtained.



As expected, prefetcher brings in the data in the next memory and hence the latency will always be around L1$ for any amount of data.

Note : Different runs of my program gives different data but almost in the same range. There are few exceptions sometimes. If you run the program and not get the exact data, please retry.