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

Ranganathan Ramkumar
A53299500

How does your program measure latency?

First, we initialize a linked list based on the input size given by the user.

All the graphs and figures in this report are based on "random access" to our data structure unless mentioned otherwise. When a user desires random access to be carried out, the initialized linked list calls the *randomize_ll* function which jumbles up the order in which the list is linked. For sequential access, each node points to the next node in the list whereas for random access, the pointer for each node could be pointing to any of the nodes in the list. There are two basic operations we carry out to calculate latency - read and read-modify-write. For read, the linked list is accessed a number of times(count) and we read the pointer of the node in the linked list (helps us to know where to go next). The latency is calculated as the difference in time between the start and end of this operation divided by the total number of times the linked list was accessed. These operations give us a relative latency (the presence of loop overhead means this is not actual cache latency).
For read-modify-write, I have used the following method to avoid the compiler from misrepresenting latency through optimization of my inner loops. Figure 1 shows what a read-modify-write operation might look like.

```
    x = nextnode_rmw->data;
    x = x ^ 0x7FFFFEEF;
    nextnode_rmw->data = x;
    nextnode_rmw = nextnode_rmw->nextptr;
    rmw_count++;
    loop_rmw_count--;
}
```

Figure 1 : Read-modify-write

What options or compile time configuration does your program take and what do those options or compile time configurations do?

Through command line options, the user can select the number of threads, the size of the data structure to be used, the access mode (random or sequential) and the stride length (in case of sequential access). The "write" option helps the user switch to read-modify-write mode (default is read)

Using “-help” at the command line while executing shows the user all available options. Figure 2 has all available command line options.

```
(base) RanganathansMBP:project ranga.ramkumar$ ./cacheperf --help
Usage: ./cacheperf <options>
Options:
        -t:<threads>
        -s/-r (seq or random access)
        -size:<array_size> (in KB)
        -stride:<stride length> (only for use with sequential)
        -write (For changing to read-modify-write mode, default is read
```

Figure 2: Command line options and usage

What limitations does your program have? (e.g. uses of system dependent functions, can only scale to a maximum of N threads).

Latency measurement is reliable at lower number of threads but might suffer as we increase the number of threads. Since memory has to be allocated for each thread, this program could become resource intensive as we increase the number of threads beyond 16. (Memory is allocated on the heap using malloc and the OS might be forced to kill the program if this becomes full)

The program was run on a machine that uses a 1.4Ghz quad-core Intel i5 processor. Figure 3 contains details about cache sizes and organization. The L1 cache size is 32KB for the data cache and instruction cache respectively. The L2 cache size is 256KB and the L3 is 6MB. [L1 and L2 caches are for each core whereas L3 is shared]

```
(base) RanganathansMBP:project ranga.ramkumar$ sysctl -a | grep cache
kern.flush_cache_on_write: 0
kern.kernelcacheuuid: A641BA08-26A6-343E-A9E4-A8C6E9640315
kern.namecache_disabled: 0
vm.vm_page_filecache_min: 383399
vfs.generic.nfs.client.access_cache_timeout: 60
vfs.generic.nfs.client.readlink_nocache: 0
vfs.generic.nfs.server.reqcache_size: 64
net.inet.ip.rtmaxcache: 128
net.inet.tcp.clear_tfocache: 0
net.inet.tcp.init_rtt_from_cache: 1
net.inet6.ip6.rtmaxcache: 128
hw.cacheconfig: 8 2 2 8 0 0 0 0 0 0
hw.cachesize: 8589934592 32768 262144 6291456 0 0 0 0 0 0
hw.cachelinesize: 64
hw.l1icachesize: 32768
hw.l1dcachesize: 32768
hw.l2cachesize: 262144
hw.l3cachesize: 6291456
machdep.cpu.cache.linesize: 64
machdep.cpu.cache.L2_associativity: 4
machdep.cpu.cache.size: 256
security.mac.asp.cache_entry_count: 1095
security.mac.asp.cache_allocation_count: 4389
security.mac.asp.cache_release_count: 3294
(base) RanganathansMBP:project ranga.ramkumar$ s
```

Figure 3: Cache hierarchy for the machine being used.

Analysis

The following figures show us how latency changes as we change the number of threads and the size of the data structure.

a. *Read Latency for a single thread with different sizes*

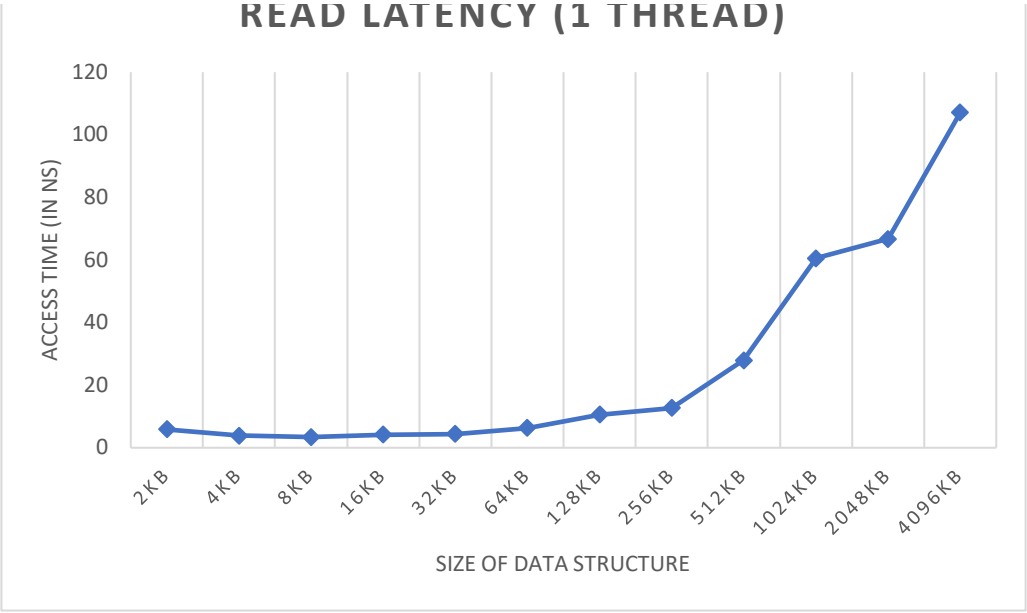

Figure 4: Single thread Read

Figure 4 shows how the relative latency of a read access changes as the size of the data structure changes. The L1 cache has a size of 32KB but both L1 and L2 are "on-core" so the latency increases but only by a small amount as we move from 32KB to 64KB. However, ***once we cross the 256KB mark*** (capacity of L2 cache) the latency jumps up by a large number since the program is now forced to access the L3 cache which is a shared cache common to all the cores.

b. *Read-modify-write Latency for a single thread across different sizes*

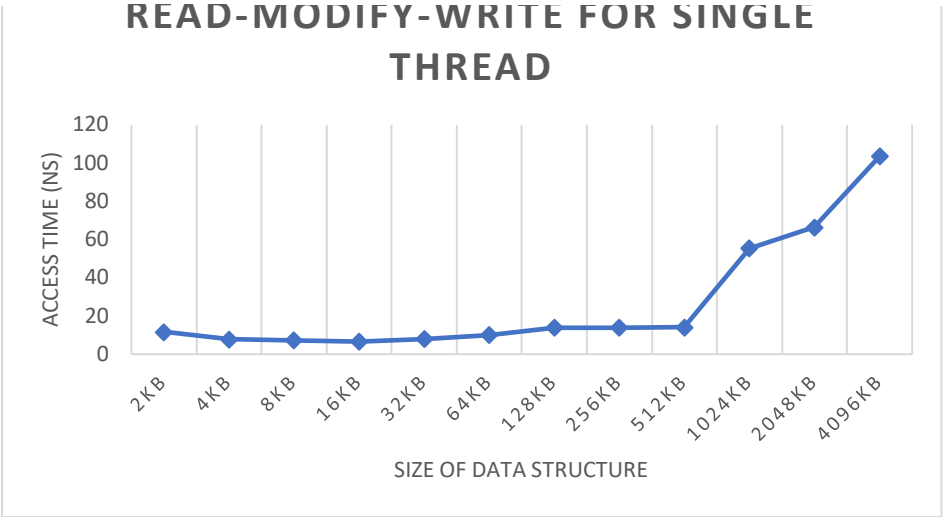

Figure 5: Read-Modify-Write for single thread

As expected, the read-modify-write accesses follow the same trend as the read accesses. The only difference is that the latency numbers are higher than for a read operation. (Makes sense that a read-modify-write will take longer than for a plain read operation)

c. *Read Latency for multiple threads*

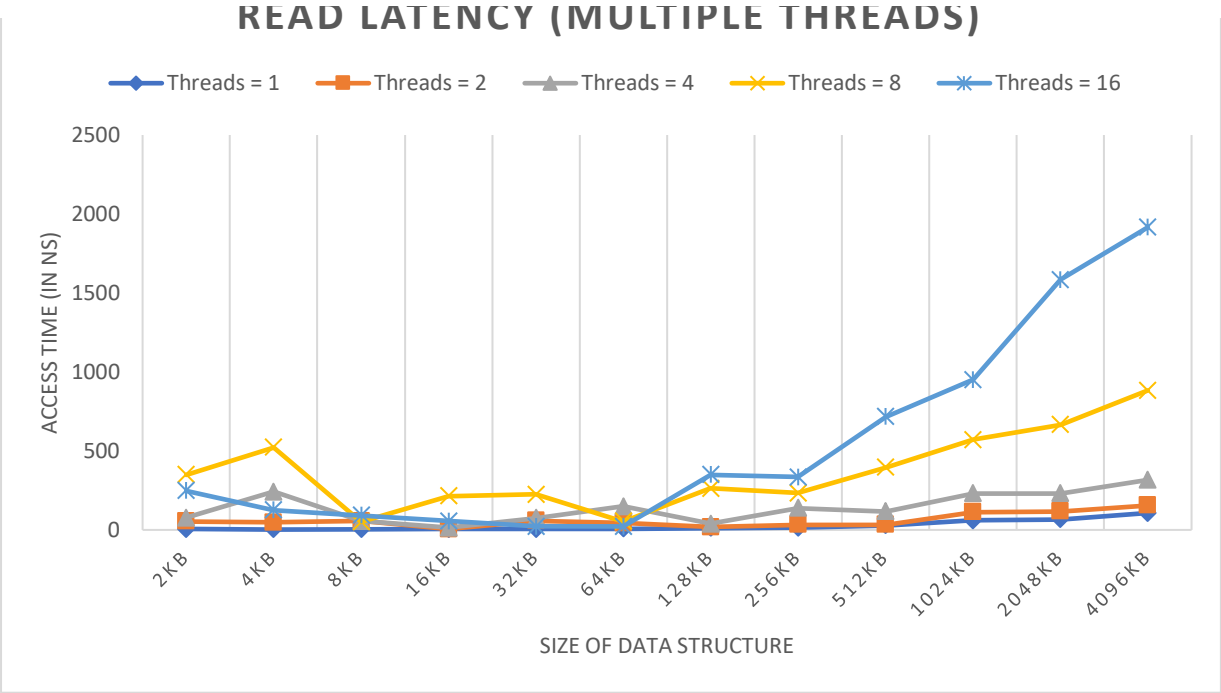

Figure 6: Read Latency when multiple threads are used

Since memory is allocated for each thread, the caches should be filled sooner. We see that the latency numbers are higher than we saw for the single thread case, but relative latency is approximately the same between different threads. The latency for higher thread count cases (8 and 16) start increasing as we move into the 512KB region. We know that the L3 cache will be accessed for such large sizes. As we increase the number of threads, the chances of L3 cache blocks getting filled and subsequently evicted by each thread also increases. For the high thread count cases, it makes sense that this process happens more frequently, resulting in an increase in access time.

d. Read-Modify-Write Latency for multiple threads

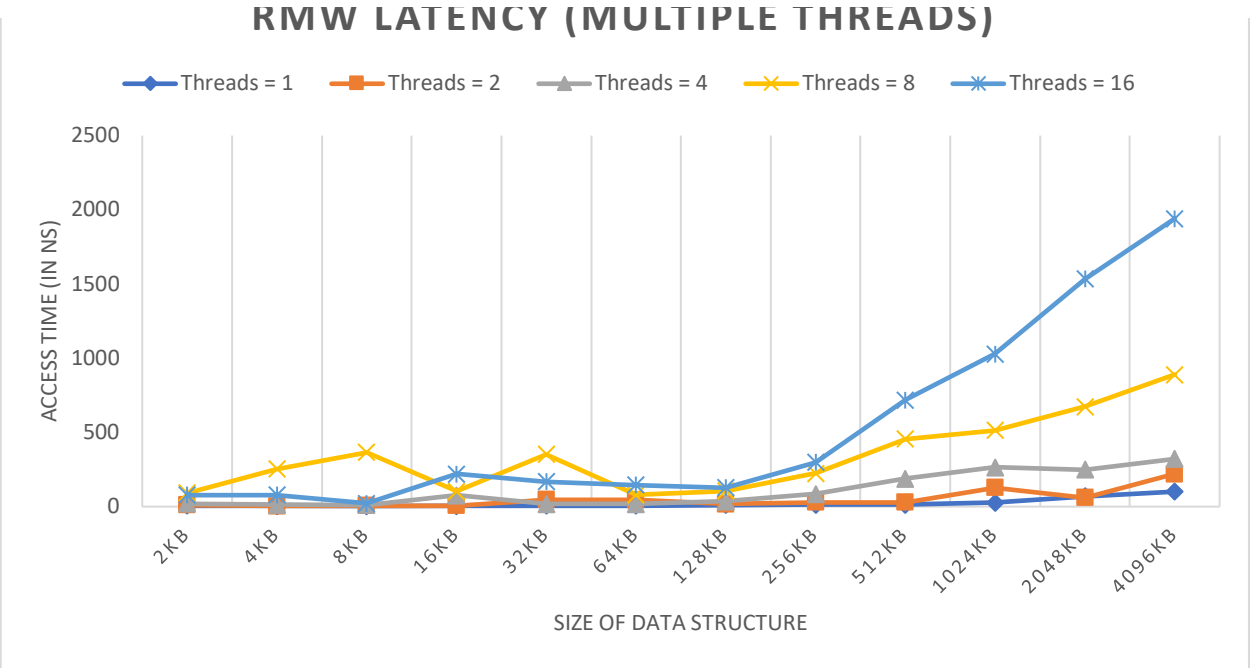

Figure 7: Read-Modify-Write for multiple threads

As expected, the RMW graph shows a similar trend to that observed in the read case. Latency values are slightly higher than plain reads as writes are a costlier operation as compared to reads. The difference between L1 and L2 caches can also be seen in this graph (for threads=16 case).

Both graphs seem to have a similar degree of stability.

Although write latencies are higher than read latencies, the difference is not large. Therefore, we can conclude that the cache is probably a write-back cache and not a write-through cache.
The machine indicates that it has L1-32KB, L2-256KB and L3-8MB which corresponds to the graphs and jumps in latency as we move to a lower level memory. We can validate the correctness of our code by observing how the latency changes at these points. Using this method and observing the graphs, we can be sure our code is correct as the jumps in latency are usually at the 32KB or 256KB points.

EXTRA CREDITS

The following section analyzes the change in latency between random accesses and sequential strided accesses. For this experiment, the stride length was assumed to be 2.

Figure 8 shows how randomly accessing the data structure requires more time than a sequential access. For smaller sizes, the entire data structure can fit into the L1 or L2 cache and the mode of accessing doesn't really matter since the entire linked list is already inside the cache. For larger sizes, a part of the data structure will reside in the L3 shared cache which will require a greater latency as compared to an access from L1 or L2. Sequential accesses enable the compiler to perform prefetching. The required cache block is therefore kept ready inside L1 or L2. That is why we see that the access time in the sequential case remains the same as what you'd expect for L1 or L2 even when we go to larger sizes.

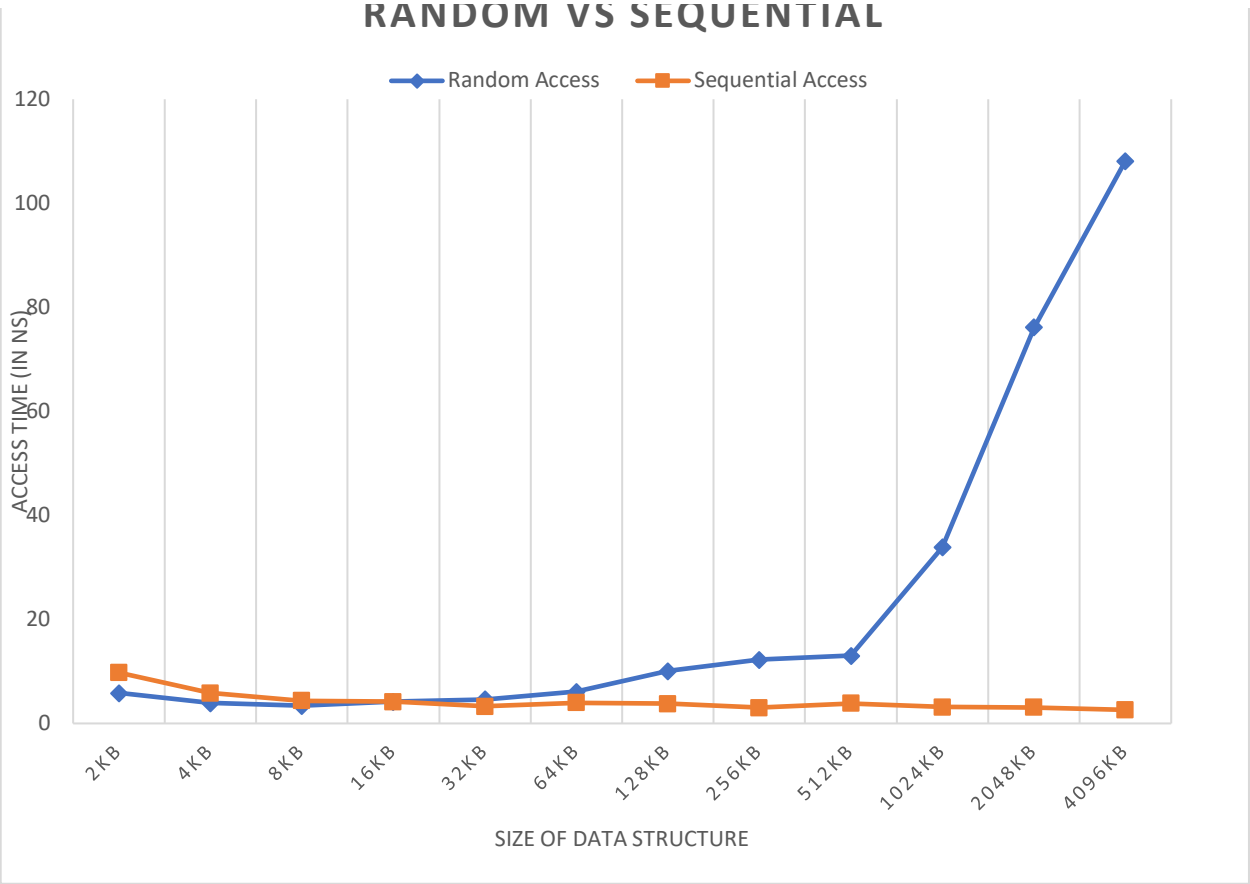

Figure 8 : Read access latency for Random vs Sequential accesses