Determining a Cache Hit/Miss over RDMA

A NetCAT Replication

Emerson Ford   Calvin Lee

CS 6465 - Fall 2019

NetCAT Overview

Claim

Using RDMA over Infiniband, a remote host can measure if a remote memory access is served from LLC or from DRAM on a target host with DDIO enabled.

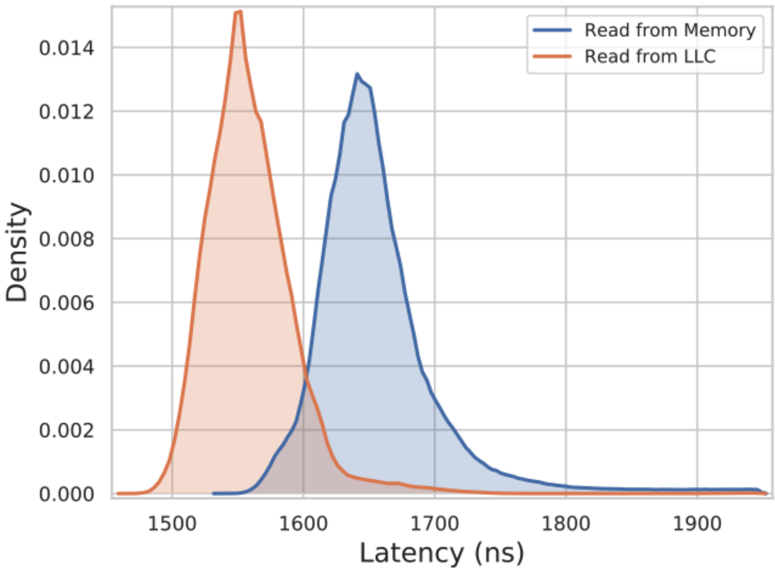
Impact

Enables cache-timing based attacks (such as PRIME+PROBE) over the network which then enables attacks like SSH keystroke timing attacks.

Key Replication Questions

1. Is it actually possible to measure cache hit/cache hit on a remote memory access?
2. If so, can we replicate their method of building a remote eviction set?

Key Graph to Replicate



Accomplishments

- *Probably* able to measure if a remote memory access is served from LLC or DRAM.
- Didn't get much farther as we struggled to get consistent results.
- Project became far more learning based than result based.
- Learned quite a lot about RDMA, Infiniband, DDIO, caches, CPU scaling, timing, etc.

RDMA Overview

1. Server and client both register memory to be used for RDMA.

Reads

2. Client specifies a remote address and fires off ‘READ’ verb.
3. Client NIC communicates with remote NIC to read remote memory address (no CPU involvement).
4. Client NIC places remote memory contents into client’s registered memory.

Writes

2. Client alters local registered memory.
3. Client specifies a remote address and fires off ‘WRITE’ verb.
4. Client NIC communicates with remote NIC to write local memory contents at remote address (no CPU involvement).

Other Key Facts

DDIO

- Reads can be served from LLC or DRAM. If served from DRAM, the memory is **not** loaded into LLC.
- Writes will load memory into the LLC if not already present.
- DDIO is "restricted to 10% of the last-level cache".

Infiniband

- DRAM access and LLC access for an Infiniband NIC should take longer than a CPU's access due to PCIe communication?
- Infiniband RDMA reads (on `apt080` and `apt083`) take 1900ns on average with 50ns standard deviation.

Test Hardware

1. Apt Cluster `r320`: 1 x Xeon E5-2450 processor (8 cores, 2.1Ghz), 16GB Memory (4 x 2GB RDIMMs, 1.6Ghz), 1 x Mellanox MX354A Dual port FDR CX3 adapter w/1 x QSA adapter
2. Apt Cluster `c6220`: 2 x Xeon E5-2650v2 processors (8 cores each, 2.6Ghz), 64GB Memory (8 x 8GB DDR-3 RDIMMs, 1.86Ghz), 1 x Mellanox FDR CX3 Single port mezz card
3. Notchpeak Cluster `notch010`: 2 x Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz, 186GB Memory, EDR Infiniband

Timing Code

Let x be a remote address.

1. Read x (cache miss)
2. Write to x (pull into cache)
3. Read x (cache hit)

```
start_cycle_count = start_tsc(); // lfence -> rdtsc -> lfence

rc = ibv_post_send(res->qp, &sr, &bad_wr);
if (rc)
    fprintf(stderr, "failed to post SR\n");
do {
    poll_result = ibv_poll_cq(res->cq, 1, &wc);
} while (poll_result == 0);

end_cycle_count = stop_tsc(); // rdtscp -> lfence
```

start_tsc/stop_tsc code taken from Google's Highway Hash Git repo.

Read-Write-Read Methods

1. Read-write-read single address with a `clflush` between each iteration
2. Sequential reads with strides to (hopefully) overcome any prefetchers

   (64 byte msgs, columns count $= 4$, row count $= 524288$, $\sim$134 MB total)
3. Random access

Data Note

Graphs, unless noted, filter out data points where the diff was negative or the diff was $>=$ 99 percentile. Graphs generated with `ggplot2` in `R`.

clflush Method

Client read→write→reads then syncs with server to call clflush.

Client Code

```
for (i = 0; i < iters; ++i) {
    if (read_write_read(&res, start_addr, cycles_to_usec)) { ... }

    if (sock_sync_data(res.sock, 1, "A", &temp_char)) { ... }

    if (sock_sync_data(res.sock, 1, "B", &temp_char)) { ... }
}
```

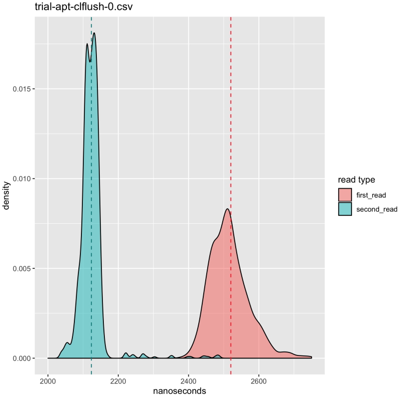
Server Code

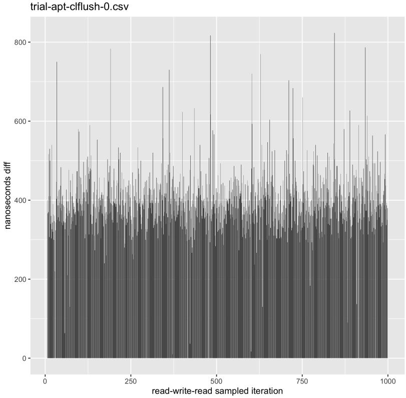
```
for (i = 0; i < iters; ++i) {
    if (sock_sync_data(res.sock, 1, "A", &temp_char)) { ... }

    _mm_clflush(res.buf);
    _mm_mfence();

    if (sock_sync_data(res.sock, 1, "B", &temp_char)) { ... }
}
```

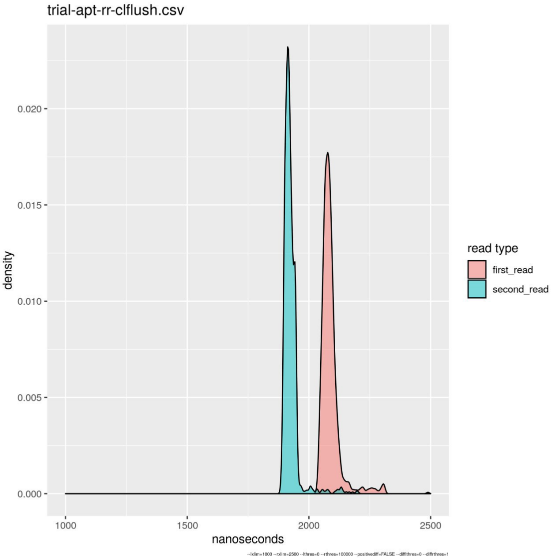
clflush Method





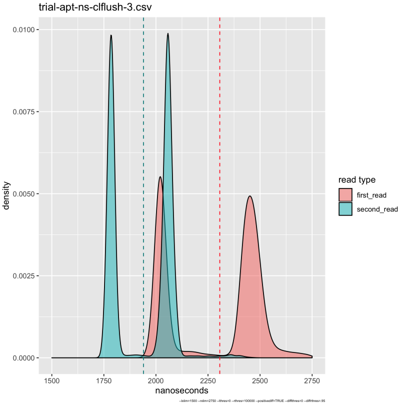
clflush Method

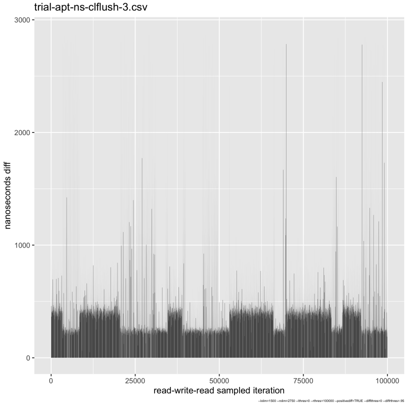
Calvin tried read→read only... :(



`clflush` Method

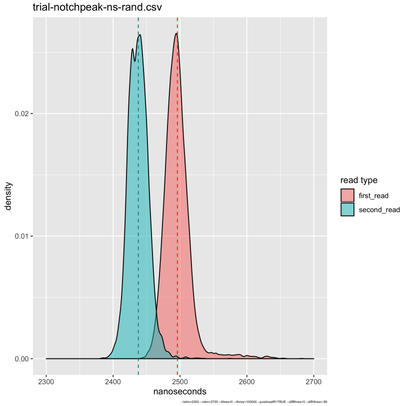
And these happened a few times with read→write→read?

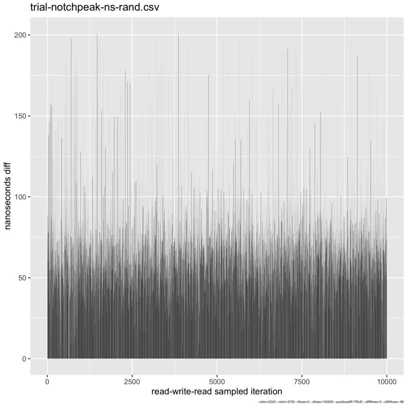




Random Access Method

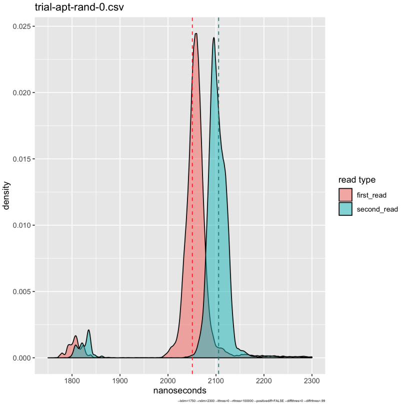
Tended to produce the cleanest data.

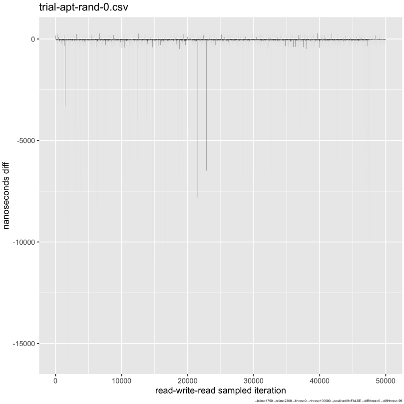




Random Access Method

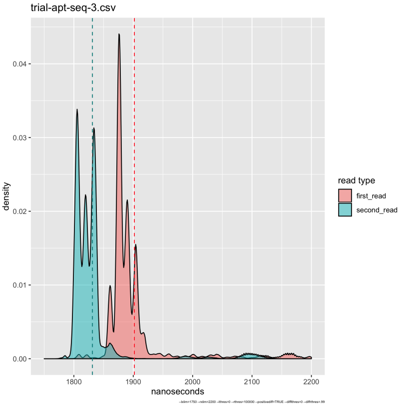
Sometimes the results came out flipped?

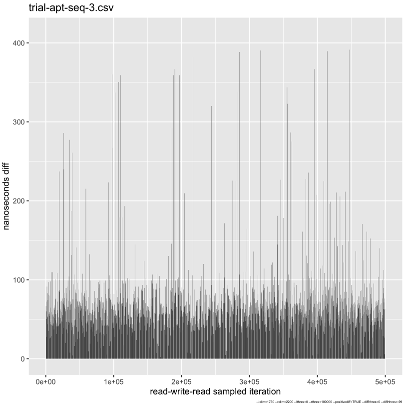




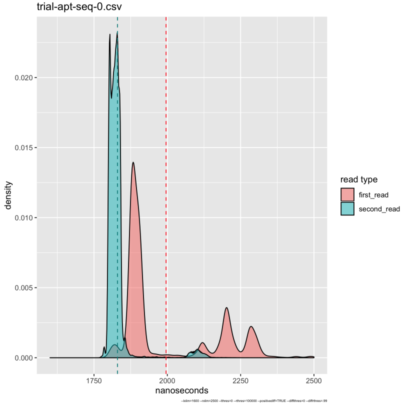
Sequential Access Method

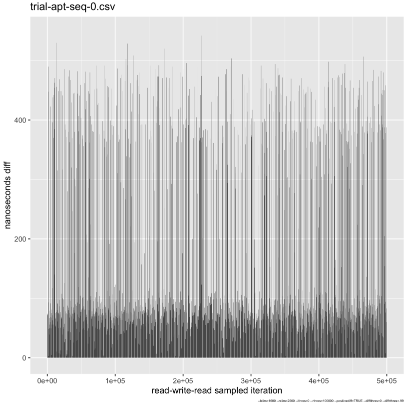
Tended to consistently produce the results we wanted but the data was noisier.





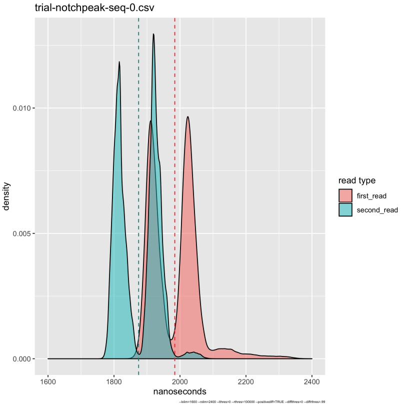
Sequential Access Method

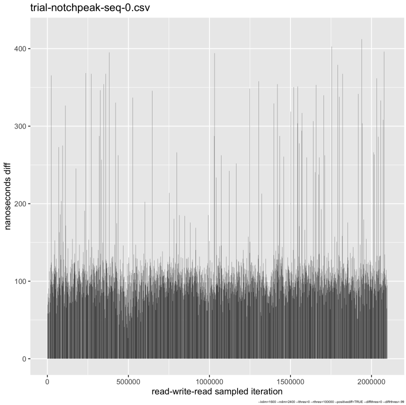




Sequential Access Method

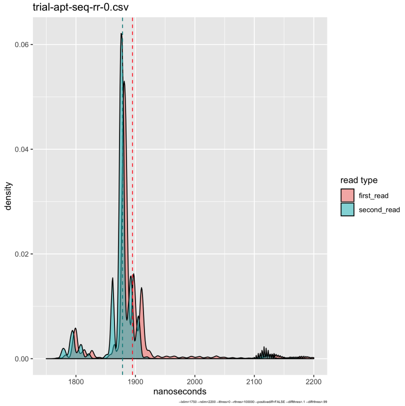
Sequential access also observed a similar “diff stepdown” as `clflush`.

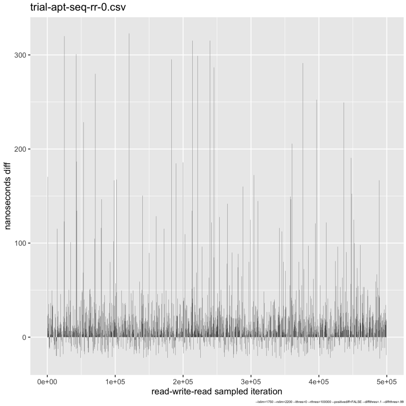




Sequential Access Method

Sequential access produces expected behavior for read→read.





What Might Causes These Graphs?

- NUMA
- Prefetchers (the NIC might have a prefetcher?)
- CPU frequency scaling
- CPU power management
- CPU Affinity
- Hyperthreading
- Saturated Infiniband fabrics seem to increase mean latencies and variance.
- Potential Infiniband pathologies?
- Potential DDIO pathologies (perhaps with `clflush` interactions, no documentation)?
- RDMA-enabled nodes are likely to be network-traffic intensive (more cache evictions)

Problems Moving Forward

- The "diff stepdown" destroys any statistical predictions on a cache miss/miss.
- Noisy data is harder to predict on
- Many of distributions are not normal, so we cannot use common regression tools.
- Couldn't figure out a baseline for the ns difference between DRAM and LLC access
- Mapping addresses to sets was far more challenging than expected
- Haven't check if the compiler does anything weird

Conclusion

- *Probably* can predict cache misses/hits
- Lots of problems/pathologies you need to work through first

https://github.com/emersonford/NetCAT-Replication