Cache-TLB

Note: NONE of these work with aurora.cs (or any Xeon machine for that matter) due to the cache timings and latency deltas between runs varying widely compared to the i7 iLab machines (i.e. aurora.cs timed 6ns on L1 cache and deltas of 2ns even while in L1. The i7 machines averaged 2 ns with deltas of .02 when in L1). The assumption is that the clock speed and architecture differences of the Xeons heavily affects the cache latency (aurora being Westmere at less than 3ghz while the i7 iLabs are basically all Skylake at 3.4ghz+).

Software prefetching is disabled through compiling with -O0 so that the only prefetching done is HW prefetching.

1. Cache Line

To get the cache line, the programs starts from an assumed cache line size of 4 (1*sizeof(int)) and times repeated 32MB accesses. Specifically, a loop from k = [0, 32MB) is timed where a large array is accessed at element (k*i) & SIZE_OF_ARRAY and incremented. The average of this looping (test_time = delta_time/accesses) and the average of the repeats (test_time/repeats) is the time to access with the assumed cache line size. The cache line guess is incremented gradually in powers of 2. When a difference in latency of 1.6ns is seen, it is assumed that the cache line size has been found.

A difference of 1.6ns was used since that was the lowest latency delta observed across multiple runs on different iLab machines (plastic (i7 8700), ls (i7 7700), python (i7 6700)) when reaching the cache line. It is assumed that the first spike of 1.6ns is the cache line and any spikes after are from fetching multiple cache lines.

2. Cache Size

Similar to cache line, the cache size is measured through the average of the array access average. The main difference is in the cache line size being static at 64 bytes and the accesses being randomized. An array the same size as of the number of accesses is generated on every repeat of the access loop. The random array contains different strides depending on the cache size (guess) being tested (rands[r] = rand()%cache_size_guess). The access loop then traverses the large data array in [((rands[k] + sum)*cache_line) & cache_line_guess] steps where sum is the sum of the accesses so far from the large data array (to further randomize the access). This causes some overhead (about 2ns) so the overhead is subtracted from the timed value later.

Different cache size guesses are tested from 2KB to 64MB starting from L1 and ending at L3.

Calculating the caches heavily depends on the timings. Across three different CPUs (the ones mentioned in the cache line details), the lowest latency delta from L1 to L2 noticed was .2 ns. The current implementation uses this approach for L1 – if the delta is > .2ns, then L2 was reached. The limitation in this approach is the heavy variance in execution due to context switching and overall load at the time.

Approach 1: use latency delta between cache size guesses.

Approach 2: use the timing at the cache size guess. Above a certain boundary means we've gone past the cache size.

Both methods have the same limitations since they depend on being minimally affected by context switching and load.

L1 is found to be 32KB (the correct answer) when there isn't much variance but occasionally the delta will go below .2. Using the second approach, when going beyond the L1 boundary, the latency goes up to beyond 2.4ns. L1 timing uses the second approach since it is more consistent.

L2 relies on the second method since it is consistently below 6ns, so anything above that timing is considered out of L2.

L3 is problematic using both approaches. On an i7 7700 and i7 6700, the L3 cache latency spikes at the boundary of L3 (8MB) so using the first approach works well. Strangely, it does not go up when dealing with 12MB as the guess and then spikes again at 16MB. On an i7 8700, the L3 latency spikes *after* reaching the boundary (12MB) so using the same metric causes on overshoot on the guess. Using the second approach for the other two CPUs causes an undershoot of the guess. L3 currently uses the first approach since a consistent upper bound could not be found (22-24 depending on load, but the delta was always above 8ns).

Again, this does not work on aurora.cs due to the cache latency timings being so different from the i7 machines.

3. TLB

Note: This does not work.

For TLB the idea is to cause page faults – as many page faults as the size being tested (or guessed). Once the loop reaches that size, re-access all previous pages in an attempt to hit similar average latencies as the page faults on re-access.

4. Memory

Memory timing was simple. Randomly access the large array with variance beyond a page size so that the TLB and prefetcher do not contain the new address translation and data. The timing should be around 60-100ns depending on the load at the time.

Running the Program

The makefile compiles all the programs for this assignment.

To run: ./caches <type>

<type> is what we want to measure/find out. 0 = cache line, 1 = cache size, 2 = tlb, 3 = memory