Report on Assignment 2

Ayushi Agarwal
2018ANZ8503
ayushi.agarwal@cse.iitd.ac.in

1 Introduction

The goal of this study is to measure some of the microarchitecture parameters of the underlying processor on real machines like workstations, personal computers or server machines. Some of these parameters could be measuring the cache line size, the associativity of the cache, the total cache sizes, the cache hierarchy, the size of the Re-Order buffer, the memory level parallelism(MLP), number of physical registers, the division of ROB among the threads in hyper-threaded systems, etc. In this study, we focus our attention to the measurement of re-order buffer size, the degree of MLP and the number of speculative and non-speculative registers in the processor.
We have developed some micro-benchmarks to do the same and run them on different machines to analyze how the configurations change across the machines. While doing so we have made certain observations on how the benchmarking is affected by using cache flushes, hyper-threading in the system, turbo boost, etc and we have reported certain pitfalls in our codes obtained while exploration.

2 Method

In this section, we would discuss on how the benchmarks are created using simple assembly instructions and high level programming language like C++[1,2].

2.1 Measuring the size of the ROB

Reorder buffer, henceforth called ROB, is a crucial part of superscalar architectures which exploit instruction level parallelism by executing instructions out of program order instead of in-program order. However, the instruction fetch, decode and rename are done in program order. The reorder buffer maintains the program correctness since all the instructions commit from the ROB in program order. This makes sure that all precise exceptions were handled correctly. Since the instructions commit from ROB in-order, if there is a long latency instruction at the top of the ROB, it will block all the later instructions. Also, after the instruction commit is blocked, the ROB size would determine how many instructions can the processor execute ahead after the stall. This concept is used in measuring the size of the ROB.

We create a loop of a single memory load instruction (that is a load which always misses in the cache), combined with 'k' nops. Until the time that (k+1) is not equal to the size of the ROB, the memory loads (cache misses) can happen in parallel (because of memory-level parallelism). As soon as (k+1) becomes equal to the size of the ROB, the consequent memory loads get serialized.

- We use NOPS because they just fill the ROB and can be committed in parallel. They have no data dependencies. This number is not exactly equal to (k+1) as we will see in Section 3.
- Since we benchmark using C++ code, there are other instructions apart from the memory load that occupy the ROB. Hence we get a little lower value for k. We talk about this in more details in the Section 3.
- We will also talk about several pitfalls in detail in the Section 3.

2.2 Measuring the Memory-Level Parallelism

The Degree of Memory Level Parallelism supported by a system is depicted by the number of parallel memory accesses that a system can handle. Memory accesses are generated when there is cache miss in the system for a particular address in the memory. Hence, the number of parallel misses that can be handled by a system is limited by the line fill buffer size, also called the miss status handling registers present between L1 and L2 cache. We measure this by doing consecutive N memory loads followed by k nops as before when we were measuring the ROB size. When the (N+1)th load starts taking more time than the Nth load, that shows the MLP of the system because the (N+1)th memory access would now be serialized after the Nth access, hence, increasing the runtime.

2.3 Measuring the Speculative Registers

The superscalar architectures use register renaming to exploit instruction level parallelism that may have WAW

and WAR dependencies. The registers in the physical register files are used for this purpose. By renaming registers, these false dependencies are broken. Generally there are lesser registers than the reorder buffer because not all instructions use registers, hence if we change the *nop* instructions (that we used as filler while measuring the ROB size) with instructions that would use physical registers (unlike *nop* instructions), we can see a step increase in the execution time when the physical registers are exploited causing a stall.

So, we use a loop of one memory load instruction along with 'k' add instructions making sure that there are enough false dependencies between the operands. This will force the processor to keep renaming the registers till all registers are used and the execution is stalled until some instructions are committed from the ROB and registers are freed.

We have measured the size of both integer register files used for speculation as well as floating point register files, using the same fundamental. The register file holds both the speculation registers and the committed registers. Using our concept, we can only measure the speculative registers used for renaming. The difference from the values published by Intel would indicate the number of total architectural registers in the processor.

3 Results and Analysis of Code

In this report, the results are presented on Intel's Kaby Lake processor machines for all the subsections. The reported numbers in Intel documents are : ROB = 224 entries, MLP = 10, Integer PRF size = 180 (Physical Register File) and Floating-point Registers = 168.

We have also presented some pitfalls in Section 3.1 that we had while doing the exploration along with the code and corresponding graphs. All the Execution Times reported here are in cycles. We have used the RDTSC counter to count execution cycles.

3.1 Code Crux

We have used arrays in C++ to execute the memory load instructions. We have to make sure that the array element that is accessed every time is always a cache miss so that it takes long latency to commit in the ROB. This can be done in many ways in C++. In this section, we will talk about some of the code cruxes that we found out while doing our exploration on calculating the ROB size.

- **Fixed Stride Accesses:** Memory loads are done by accessing various array elements. If we use fixed strides between consecutive loads then, on a real machine the processor speculates the stride and prefetching is done to bring many lines from the memory together into the cache. Since adding bandwidth is cheaper than latency, bringing lots of data together from the memory is cheaper than bringing small chunks again and again. This increases the number of subsequent cache hits. But it hinders with our ROB calculation as we want all loads to be cache misses. In our experiments, we saw that the execution time of our code with fixed stride memory access is an increasing function of the #nops inserted. So, fixed stride access cannot be done.

- **Random Stride Access:** For random accesses to the array elements we have to create random indices. This can be done in the following ways:
  - We can use rand() function in C++. This is a complex function and calling it inside the loop before memory load inserts function call instructions between the consecutive loads. This makes it difficult to quantify the #nops obtained. It can also have unpredictable affects on the execution time.
  - So, we have generated random array indices using the Time-Stamp Counter in the hardware. The counter keeps increasing (counts the number of cycles) as the machine is reset. The RDTSC instruction reads the TSC and stores it in EDX:EAX. We have generated random indices by masking the MSBs according to the size of the array, as below:
    $\_\_asm\_\_\_volatile("rdtsc" : "=a"(k), "=d"(hi))$;
    $< index >= k \ \& \ 0x0000FFFF$: This is when array size is 100000
    The number of extra instructions generated in the assembly by this method can be quantified. By looking at the assembly code, this adds around 5 to 6 instructions between loads.

- **Using Non-Temporal Loads:** To ensure cache misses, we explored non-temporal loads as well. Non-temporal accesses are done to bypass the cache to avoid cache pollution. However, non-temporal accesses are mostly done in the cases of Write-Combining memory as opposed to the default Write-Back memory system. The non-temporal loads also face some prefetching into line-fill buffers hence we have avoided using non-temporal loads.

- **Using Static Array:** If we use a static array, we are limited by the size allocated to it in the memory as it is allocated from the stack at compile time. If the size of the array is less than the number of loop iterations, then we have to keep flushing the cache lines that we load from memory when we read a particular array element. This is because the random values generated as above, can repeat with high probability and lead to potential cache hits.
  - When we add *CLFLUSH* in the inner loop, there is a cost to it. It adds around 5-6 extra instructions in the inner loop. So the effective

number of *NOPS* needed now to see a step increase, reduces by an approximate factor.
  - The *CLFLUSH* instruction is critical when hyper-threading is enabled because it tries to flush the cache line from all the caches of all the cores. So it might extend the execution time on hyper-threaded systems.

- **Using Dynamic Array:** If we use dynamic array, it is allocated from the heap at run-time. We have used *malloc* to initialize a big array with size equal to the number of outer loop iterations. This will ensure that there are very less probable cache hits and the probability of same random indices being generated is fairly less. So, there is no need of cache flushing (though there still is a possibility of some cache hits). Also we introduce randomization by doing array shuffling while initialization to avoid cache hits.

- **Register Hints to compiler:** One very crucial point that we have observed from this experiment is that there should be no other memory loads/stores in the loop except for what was intended. So, we have used *register* datatype in C++ to give the compiler a hint that the variables are supposed to be in registers and not memory location. We have significantly reduced the memory accesses inside the loop using register variables. However, this poses some limitations because the memory loads are always stored in %rax register. So the compiler optimizes the statements like : register j = a[0]. So, we put an add statement after the load for the compiler to know that it is a valid operation.

- **Disable Hyper-Threading:** We have disabled Hyper-threading[3] in the machine put to test because, in hyper-threaded machines, the ROBs are shared and divided equally among the threads. So, if we run our code on hyper-threaded cores, it is difficult to reason that the ROB size that we infer from our experiments is because of our benchmark and not some other process running on the system. So, we benchmark our code on a single CPU. However, this makes the benchmarking highly susceptible to other workloads running on the machine. They can widely affect execution time by occupying parts of ROB.

  Even though, we didn't observe much difference in the results obtained on single CPU vs. multi-CPUs, we are able to quantify our numbers with better clarity. We also observed that on hyper-threaded machines, the OS keeps switching the running benchmark between CPUs. We suspect a little increase in execution time because of this.

- **Disable Compiler Optimization:** We have disabled the compiler optimization by using -O0 while compilation to avoid any compiler interference with the assembly code.

- **Disable Turbo Mode:** We have also tested our benchmarks by disabling Intel's turbo-boost[3]. During our exploration, we tried using the perf[4] tool, which is a performance monitoring tool for linux. We used the following command :

  $perf\ \ stat\ \ -\ e\ \ resource\_stalls.all\ \ <\ executable>$

  This command gives a statistics on all the resource stalls happening in the hardware while running a benchmark. What we observed was that our benchmarks were always running on higher frequencies than expected and the frequency kept varying with different runs. We suspected the reason to be Intel's turbo boost. However, when we repeat our ROB experiment with turbo boost off, we don't see a significant difference in the execution time. We report our results in Section 3.2.

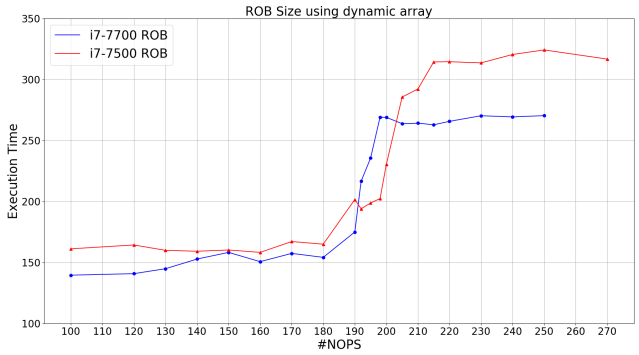
3.2 Size of Reorder buffer

As discussed in Section 3.1, we have explored the usage of both static and dynamic arrays in our experiment to calculate the size of reorder buffer in the machine. Figure 1 (a) shows the ROB size measurements obtained on two machines : i7-7700 and i7-7500. Both are based on Intel's Kaby Lake architecture. The reported ROB size if 224 entries. On i7-7500, we have obtained a step increase in execution time at 198 entries and on i7-7700, we obtain the step increase in execution time at 190 entries. The memory load instructions are generally broken into 6 μops[2] and there are some extra instructions between the loads that are introduced due to C++ language intrinsic. We noted that there are about 19 extra instructions in the assembly apart from memory load and k nops. Hence, we want to emphasize that the effective ROB size on i7-7500 would be around 224. We have not been able to reason why i7-7700 shows an even lower number of *nops* to stall the ROB. Figure 1 (b) shows that for this benchmark, there was no significant affect when we turned off turbo-boost. Figure 1 (c) shows that the step increase in execution time when using static arrays with cache flushing is obtained at 195 entries of *nops* on i7-7500. This is because of the extra instructions added by *clflush*.

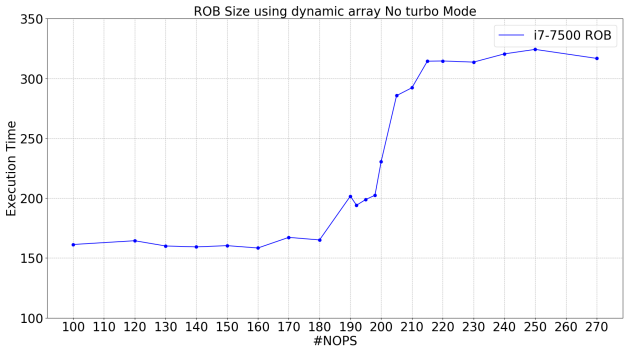
3.3 Available Memory-Level Parallelism

We have setup the experiment for measuring the degree of memory-level parallelism in a machine as described in Section 2.2.

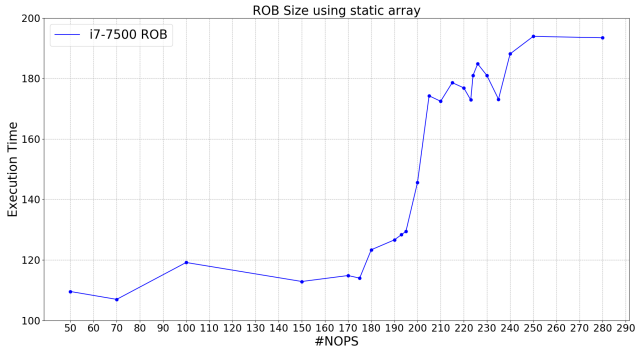
To create N consecutive memory loads we have repeated the same load operation that we have used for calculating the ROB. The inner loop now has N



(a) Re-Order Buffer Size using Dynamic Array loads



(b) Re-order Buffer Size using Dynamic Array with Turbo Boost OFF



(c) Re-order Buffer Size using Static Array loads

Figure 1: Re-order Buffer Sizes for 2 Kaby Lake Architectures using different array implementations

loads and *k*nops. These N loads should all be independent of each other otherwise it would not exploit memory parallelism. We tried some pointer-chasing methods like $while(1)\ r = array[r]$; or $j = array[array[index]]$;(this translates 2 memory loads but they have data dependency) to do consecutive loads, but this creates a data dependency between them. The second load cannot commit till the data of the first load is received from the memory. Figure 2 shows the degree of Memory-Level Parallelism available in Kaby Lake. For a given value of N (number of consecutive memory loads), we see a step increase in execution cycles when (N+*k*) is equal to the ROB size. Also, when N becomes equal to the degree to memory level parallelism available, we should see a step increase in the execution time for the *N+1*th memory load because that would be serialized after the *N*th load. From our measurements as shown in Figure 2, we infer that since the gap between the curves for N=9 and N=10 is significantly higher than the gap

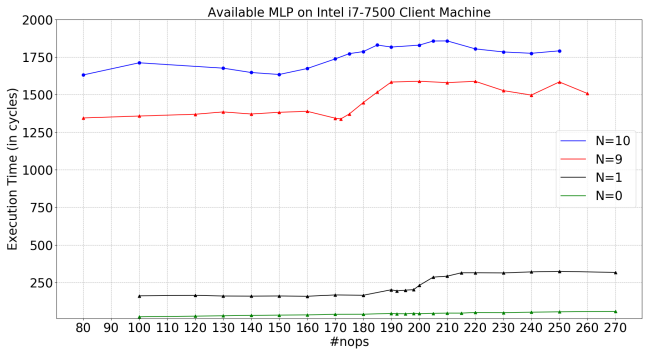


Figure 2: Degree of Memory-Level Parallelism in Intel's Kaby Lake Architecture i7-7500

between N=0 and N=1, hence the degree of MLP on our target machine is 10. In Section 2.2, we had established that the degree of MLP is limited by the number of Line-Fill buffers in the architecture. Intel has reported LFB =10 in Kaby Lake architecture which is in line with our results.

3.4 Speculative Integer Registers

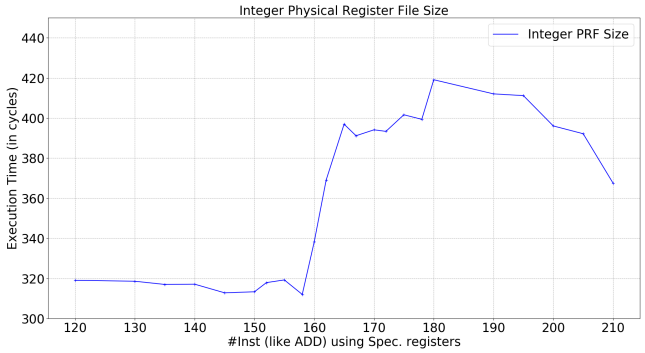
For the measurement of speculative integer registers, we have used static array and *clflush* in the inner loop. But it doesn't matter here, because we are calculating the number of physical registers available for speculation purposes. Since, this number is usually less than the ROB size, the bottleneck is the register renaming step and not the ROB stall. A few extra instructions in the ROB, doesn't affect our results. As mentioned in Section 2.3, the crucial point to keep in check here is that the ADD instruction used should create enough false dependencies so that the processor keeps renaming the registers to exploit instruction parallelism through speculation.

We have reported our results for Intel i7-7500 machine. The size of the Integer Physical register file reported by Intel is 180.
Figure 3 (a) shows that the number of speculative integer registers found from our exploration are 165 registers so the number of non-speculative registers are around 15.

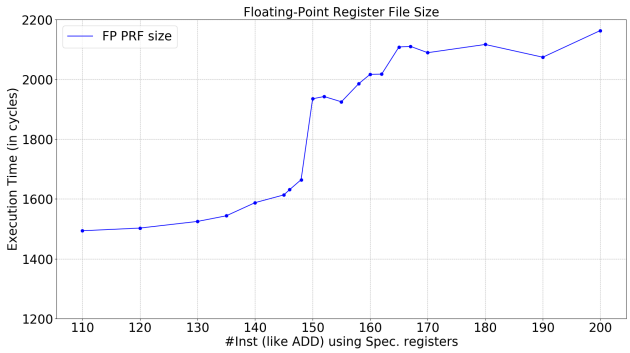
3.5 Speculative Floating-point Registers

Using the concepts described in Section 2.3 and 3.4, we have calculated the size of the floating point register file on Intel i7-7500 machine. The ADD instruction here is on floating point data so uses XMM, YMM, and other floating point registers in the system.

Figure 3 (b) shows that the number of speculative floating point registers. It is close to 148 entries as seen in the curve. So, the non-speculative portion of the register file has around 20 entries which is a little higher given a total of 16 architectural registers for XMM/YMM. The figure also shows an increase in the total execution cycles that the floating point operations take when compared to integer operations, which is as expected.



(a) Speculative Portion of Integer Physical Register File



(b) Speculative Portion of Floating-Point Register File

Figure 3: Measuring the speculative portions of the Physical Register Files in Intel's Kaby Lake i7-7500 Architecture

4 Conclusions

This report gives a summary of how to explore some of the microarchitecture parameters by using high level programming languages. It also mentions some of our own explorations that we did to reach the results which include some of the coding cruxes and Linux benchmarking characteristics. We observed that the execution time varies a lot when we execute the same benchmark again and again. We predict that this is because of other processes running on the target machine that also use the hardware resources. This report was to some extent able to benchmark microarchitecture parameters. The code base could be extended to calculate other parameters like cache sizes, associativity, etc.

5 References

1. Measuring Reorder Buffer Capacity
2. P4+BACUS for developing high-performance software switches
3. Benchmarking on Linux
4. PERF Tool Wiki page