Indian Institute of Technology, Delhi

Experiments on memory level parallelism

by

Anmol Panda, Ankit Bhardwaj, Shailja Pandey

A project under the supervision of

Dr. Sorav Bansal

in the

School of Information Technology

Department of Computer Science and Engineering

March 2018

Contents

Chapter 1

Memory Level Parallelism

When multiple memory accesses are to be served in parallel, the memory sub-system utilizes one L1 miss status handling register (MSHR) for each memory access. Consequently, we expect the maximum number of memory accesses that can be served in parallel to be limited by the number of L1 MSHRs. To prove the same, the following experiment was conducted.

1.1 Background

1.1.1 ROB

The size of the re-order buffer indicates the number of instructions that the CPU can execute out-of-order. Typically, when a memory access is launched, the CPU would move beyond (go out of order) the memory access instructions and execute the instructions after it, provided they are independent of the memory access that is being serviced. If any of these instructions include another memory access (independent of the first), that too can be serviced in parallel by allotting a new MSHR to it. Thus, we gain from memory level parallelism (MLP).

Let **R** be the size of the re-order buffer. Let **N** be the number of memory accesses, increased across multiple runs of the benchmark. The schedule of instructions is as follows:

N - 1 memory accesses + **K** NOPs + **1** memory access

The **K** NOPs are inserted between the **N-1**$^{\text{th}}$ and the **N**$^{\text{th}}$ memory access to iteratively separate the last access from the first set of memory accesses and push it outside the

ROB. At this juncture, the last access can no longer be serviced in parallel with the first **N - 1** and thus is served in sequence. On the other hand, when **N** exceeds the maximum available MLP of the system, the last memory access would be serviced in sequence, even when it is inside the ROB.

1.2 Hypothesis

1. With just two memory access (N=2), we expect a cliff when the 2nd mem access escapes the ROB. We call the region before the cliff zone 1 and after the cliff zone 2.

2. Let MLP = M. Let C be the cycles taken by two memory accesses in two consecutive ROBs i.e. two sequential memory accesses. When N = M + 1, the number of memory accesses is one more than the available MLP of the system. Also, when K is small, say K = 1 (just one NOP separates the last two memory accesses), all N accesses are inside the ROB but the last memory access can only be serviced in sequence as there is no MSHR available for it. Thus the N accesses take at least C cycles. By observing this trend, we can measure MLP of any given system.

1.3 Experiment

1.3.1 Setup

For the experiments, we used a server machine (Xeon5) with the following system specifications:

Name	Specification
CPU Model	Intel(R) Xeon(R) CPU E5-2640 v3
Micro-architecture	Haswell
Clock frequency	2.6 GHz
No. of CPUs	32
No. of sockets	2
No. of NUMA nodes	2
L1 cache	32 kb
L2 cache	256 kb
L3 cache	20 mb

For consistency and isolating memory latency from other variations, all frequency scaling and power optimization tools are disabled.

1.3.2 Benchmark

The benchmark is written in C. It was motivated from our experiments and the robsize.cc benchmark written by Henry Wong [1]. It uses RDTSC and RDTSCP instructions to measure cycles taken by the memory access. The RDTSCP instruction ensures that no instruction after itself will be executed before it. The CPUID ensures that no instruction before it can be executed after and vice-versa. Using these two instructions, we freeze the ROB at the boundaries of the memory accesses. We use PAPI to measure cache misses. The pseudo code of the benchmark is as follows:

CPUID
RDTSC
1^{st} memory access
2^{nd} memory access
•
•
$N - 1^{th}$ memory access
1^{st} NOP
2^{nd} NOP
•
•
K^{th} NOP
N^{th} memory access
RTSCP
CPUID

We vary the value of K from 1 to 210 (the ROB size of the Haswell architecture is 192). When the N^{th} memory access is pushed outside the ROB, we observe a cliff and the number of cycles doubles. The code above is run multiple times (50K or 500K times in different cases) to average the PAPI and RDTSC results. The CPUID and RDTSCP instructions guarantee that consecutive runs do not corrupt each other's readings.

[1] http://blog.stuffedcow.net/2013/05/measuring-rob-capacity/

1.4 Results

Figure 1.1 details the results of the benchmark.

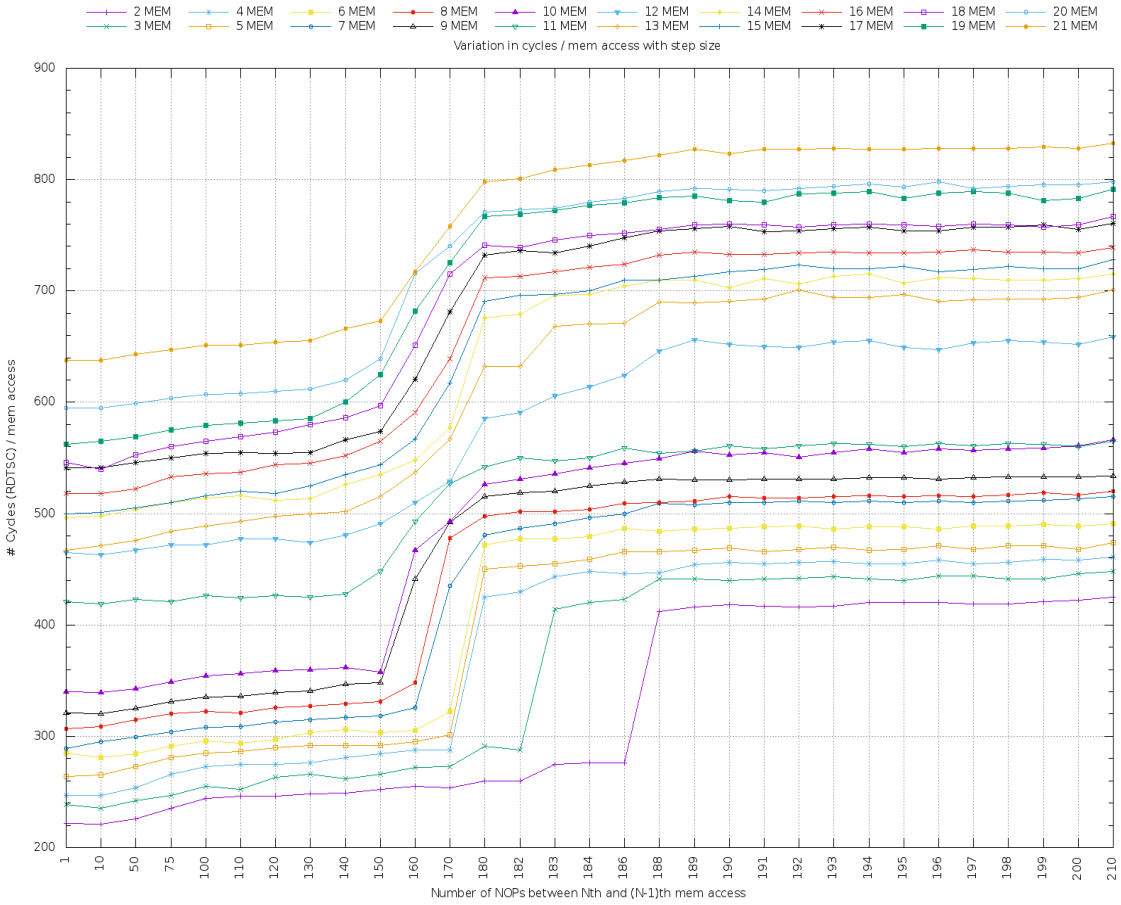

FIGURE 1.1: MLP of Xeon5 = 10.

1.5 Analysis

- Hypothesis #1 can be confirmed from the graph. The # cycles shoots up drastically between NOP = 186 and NOP = 188 (precisely at 187; ROB size = 192). Cycles C = 420 (approx.).
- From the graph, it can be seen that the number of cycles for N = 11 is approx 420 in zone 1. Based on hypothesis #2, the last memory access must have been in sequence and not in parallel with the rest. Hence, M = 10 i.e MLP = 10.
- There is a marked increase in cycles between N=2 and N=10. This could be the cost of parallelism.

Chapter 2

Effect of TLB misses on MLP

In the next three sections, we describe the effect of latency inducing factors like disabling of transparent huge pages (causing TLB misses), running the benchmark on the VM with disabling, and enabling huge pages on the host and the guest (4 configurations) and disabling NUMA (allowing the program to allocate mmory on the remote node).

Using transparent huge pages (2 MB) reduces the size of the page table, causing very few TLB misses. Disabling it reduces the page size to 4 KB, leading to a massive increase in the number of pages, subsequently increasing the number of TLB misses to almost 100%.

With transparent huge pages (THP) enabled, all our TLB accesses were hits. On checking total values we observe about 700 - 1000 TLB misses for 500000 iterations of the loop, regardless of memory accesses. Even with just one MEM access in the loop, this amounts to less than 1 TLB miss per 500 accesses (less than 0.20%).

2.1 Results with THP disabled

Figure 2.1 results were obtained with THP disabled on the Xeon5 server.

2.2 Analysis

1. We observe one TLB miss per memory access.

2. For one memory access, with Transparent Huge pages (THP) enabled, we get a memory access latency of approx. 203 cycles (err. of 4-5 cycles). This has been tallied with the ccbench benchmark as well. On disabling THP, single memory

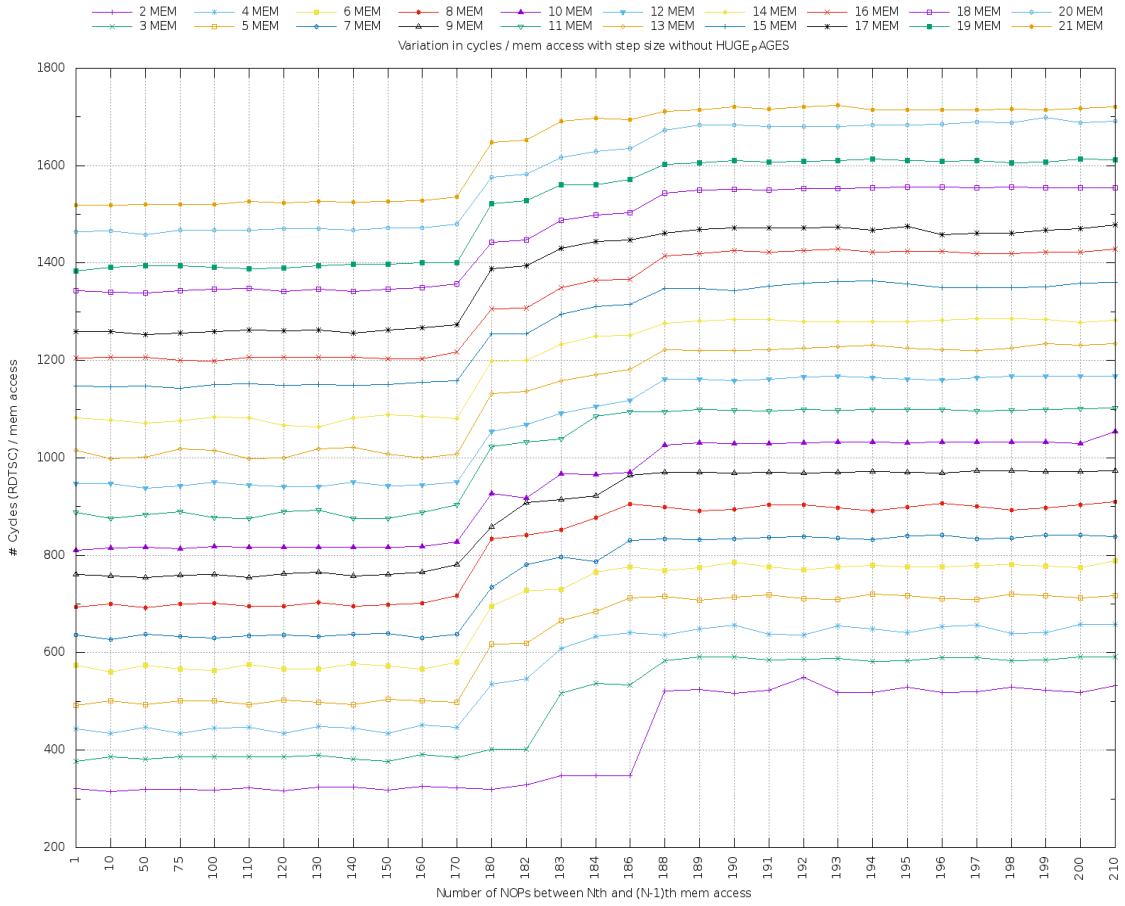

FIGURE 2.1: MLP of Xeon5 with THP diabled = 4 - 5.

access latency (NOT shown in graph) increases to around 260 cycles (err. 4-5 cycles), indicating a TLB miss latency of around 60 cycles.

3. For each subsequent memory access, we observe an increase of approx 60 cycles. This indicates that the TLB misses are being serviced in sequence and we do not get any TLB level parallelism.

4. As the memory accesses can be served in parallel (upto ten accesses) and the TLB misses can only be served in sequence, we expect an effect on the MLP of the system. It can be seen that the time taken for two sequential memory accesses in two separate ROBs (2 MEM in the graph above) is approx 520 cycles. This includes 200 cycles for each memory access and 60 cycles for each TLB miss. Based on our model, the first line on the graph that exceeds this latency with all its memory accesses inside the ROB gives the effective MLP of the system. This pattern can be observed between lines 5 MEM (approx 490 - 500 cycles) and 6 MEM (560 - 570 cycles).

5. Therefore, the effective MLP of the system is almost halved to between four - five (as opposed to ten with THP enabled). Thus, the TLB acts as the bottleneck, reducing the degree of MLP that can be harnessed.

6. Number of L2 misses are almost doubled (for 10 mem accesses we get 19 L2 misses, for 20 we get 38; they were equal to mem accesses when THP was enabled). This could be attributed to the page walk that must be executed when a TLB miss occurs.

2.3 Conclusion

1. TLB misses are serviced in sequence.

2. Each TLB mis adds approx. 60 cycles to a memory access

3. MLP is reduced to half i.e about from ten to 4-5.

Chapter 3

Effect of VM on MLP

In this experiment, we ran the benchmark on the VM to the effect of paging (disabling THP on guest and on host). We define a convention: H indicates host and G indicates guest. Enabling of transparent huge pages (THP) is indicated by 0 and disabling by 1. Thus, we get the following definition:

Convention	THP on host	THP on guest
H0G0	Enabled	Enabled
H0G1	Enabled	Disabled
H1G1	Disabled	Disabled
H1G0	Disabled	Enabled

TABLE 3.1: Convention for combinations of disabling THP on guest (VM) and host (Xeon5)

On running the first case, the results were found to be similar to running the benchmark directly on the host. However, a very high degree of noise was observed. On debugging the same (leading to the delay in reporting these results), the noise could be attributed to the migration of the VM's many processes across the different CPUs of the host machine.

To reduce the noise, we tried three different configurations with respect to to assignment of CPUs to the VM's processes (11 - 12 in total) on the host:

1. Allow the host kernel to allocate the processes arbitrarily, thus causing migration and resulting in noise in the benchmark results (see first graph)
2. Allocating all the processes to the same CPU (caused resource contention, noise was lower than first config. but still high)
3. Allocating a separate CPU (on same socket) to each process. This caused limited noise and gave decipherable results.

Consequently, the third configuration was chosen to run all four combinations. The conclusion from the above debugging is that the way the host manages the VM's processes has a significant bearing on the benchmark's performance (and thereby memory performance).

3.1 Results

3.1.1 Noisy results

Figure 3.1 was yielded through configuration 1 (allowing the VM's processes to migrate on the host CPUs) and combination H1G0. The less noisy variation (configuration 3) is included in the 5th section below.

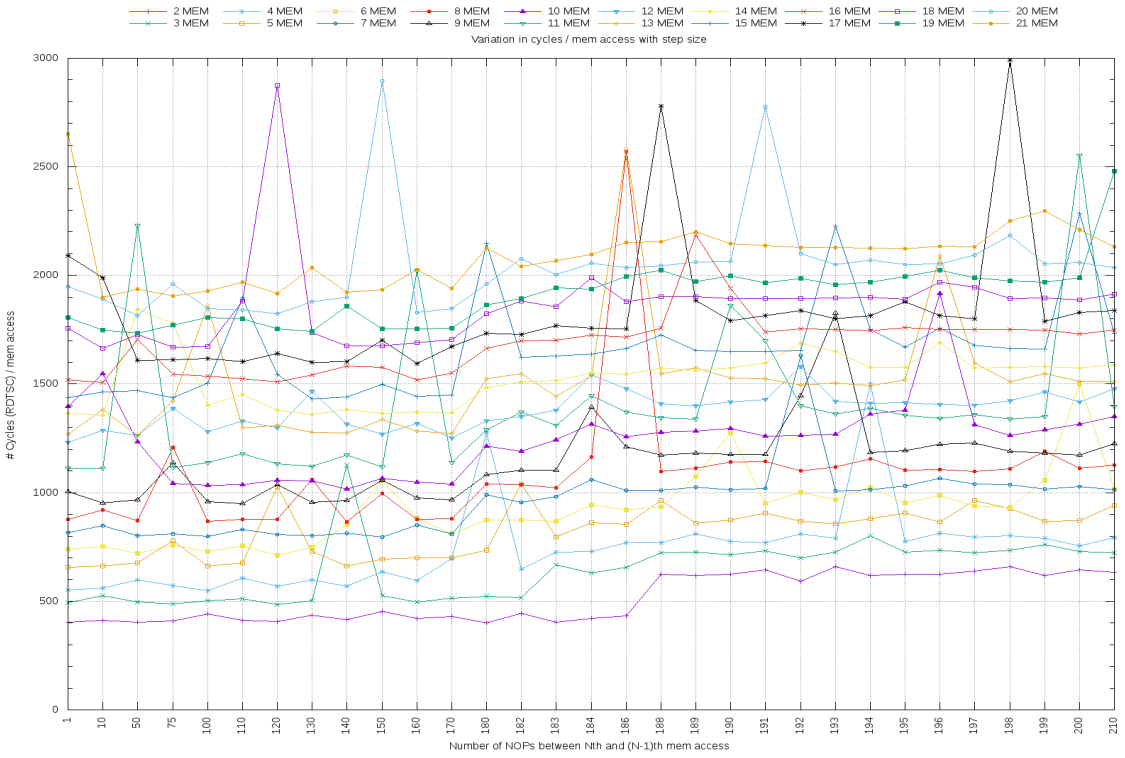

FIGURE 3.1: Very noisy results with configuration 1.

3.1.2 Combination H0G0

Figure 3.2 depicts the results with THP enabled on both the host and the guest.

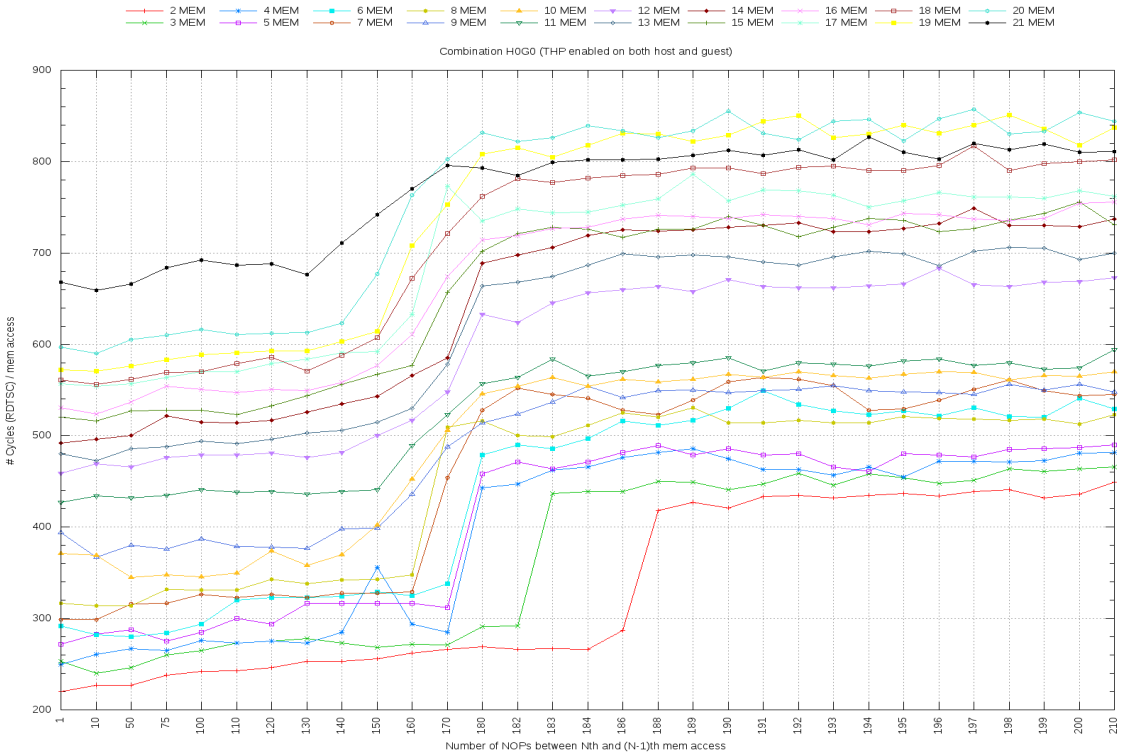

FIGURE 3.2: Combination H0G0 with configuration 3.

1. The results closely mimic those of running the benchmark directly on the host. There are intermittent bursts of noise but the data is reasonably stable.
2. MLP = 10
3. Time for 1 memory access = 200 cycles approx.

3.1.3 Combination H0G1

Figure 3.5 depicts the results with THP enabled on the host and disabled on the guest.

1. The data is stable. The overhead due to TLB misses on the host is sequential i.e the TLB offers no parallelism in handling misses
2. MLP = between 3 and 4 (two sequential accesses take approx 650 cycles. 4 parallel accesses take approx 630 and 5 parallel accesses take approx 690 cycles resp. By definition, this gives an MLP of 3 to 4)
3. 1 MEM access = 340 - 360 cycles

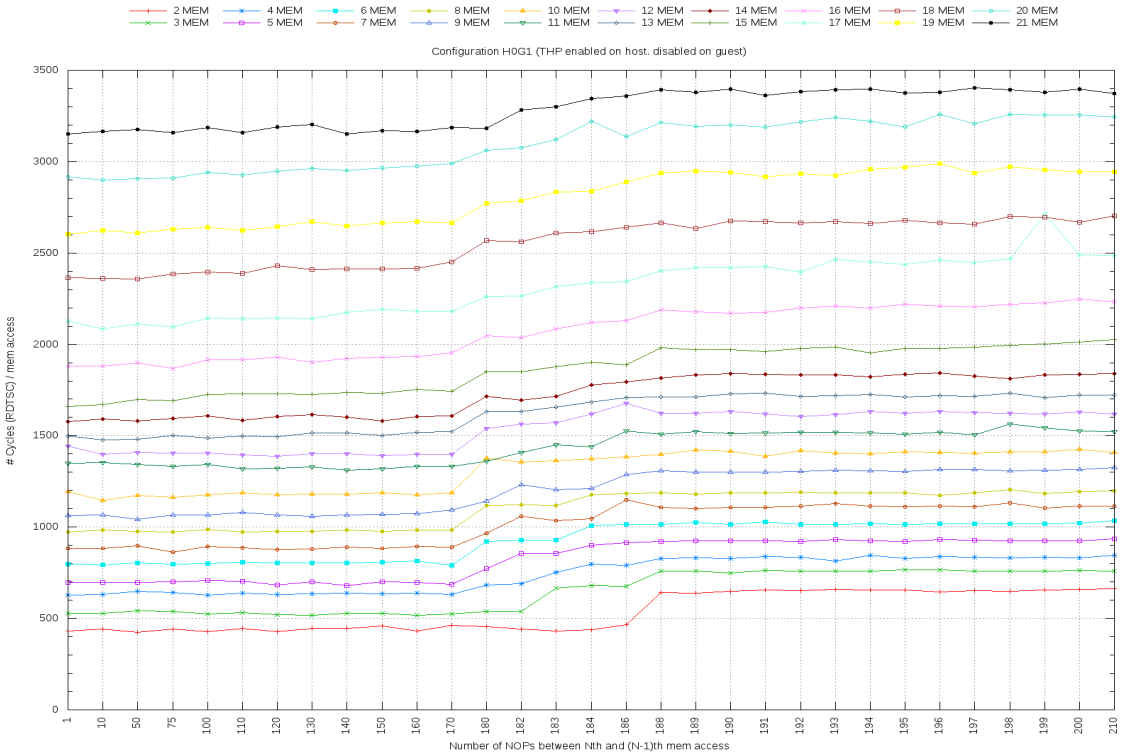

FIGURE 3.3: Combination H0G1 with configuration 3.

3.1.4 Combination H1G1

Figure 3.4 depicts the results with THP enabled on the host and disabled on the guest.

1. The data is relatively noisy. The overhead is sequential.
2. MLP = between 3 and 4, closer to 3 (2-MEM sequential take approx 800 cycles, 4-MEM parallel take 710 - 740 cycles while 5-MEM parallel take 800 - 850 cycles)
3. 1-MEM access = 420 - 450 cycles
4. 2-MEM SEQ = 800 cycles (780 - 810)

3.1.5 Combination H1G0

Figure 3.5 depicts the results with THP enabled on the host and disabled on the guest.

1. Data is relatively stable(compared to the configuration 1). TLB misses are serviced sequentially.
2. MLP = 3 (2-MEM SEQ = 600 cycles, 4-MEM PRL = 600 cycles approx, MLP = 4 - 1 = 3)

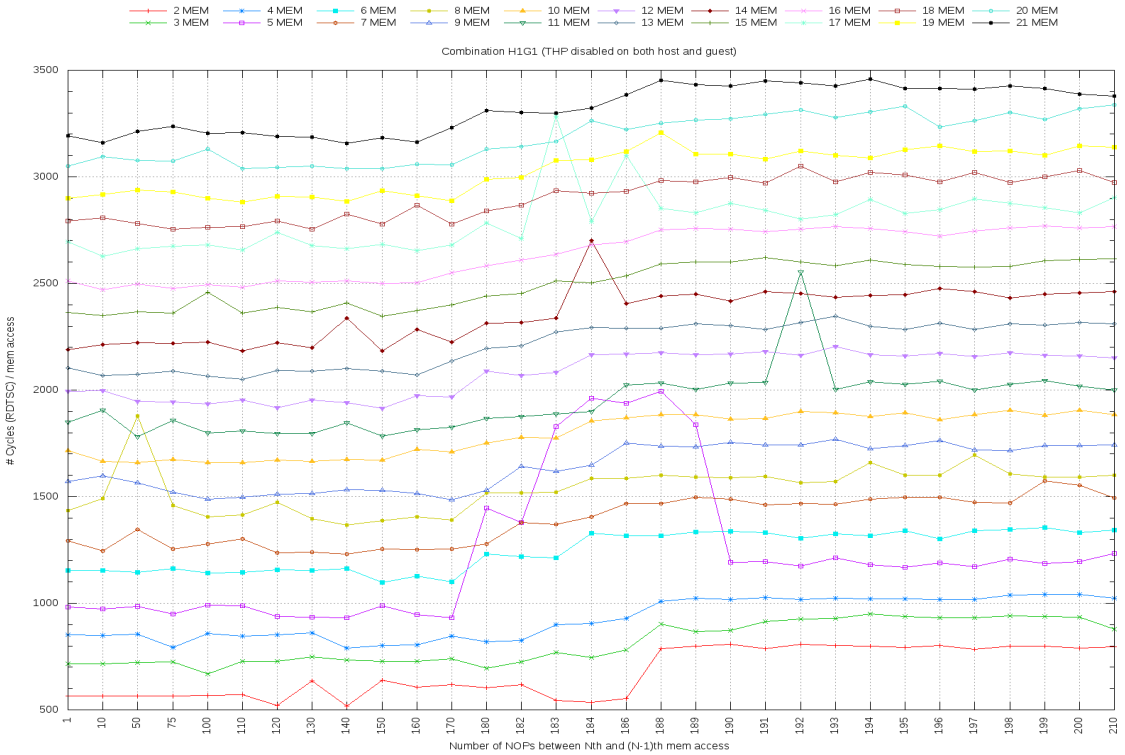

FIGURE 3.4: Combination H1G1 with configuration 3.

3. 1-MEM access = 330 cycles approx

4. 2-MEM SEQ = 600 cycles

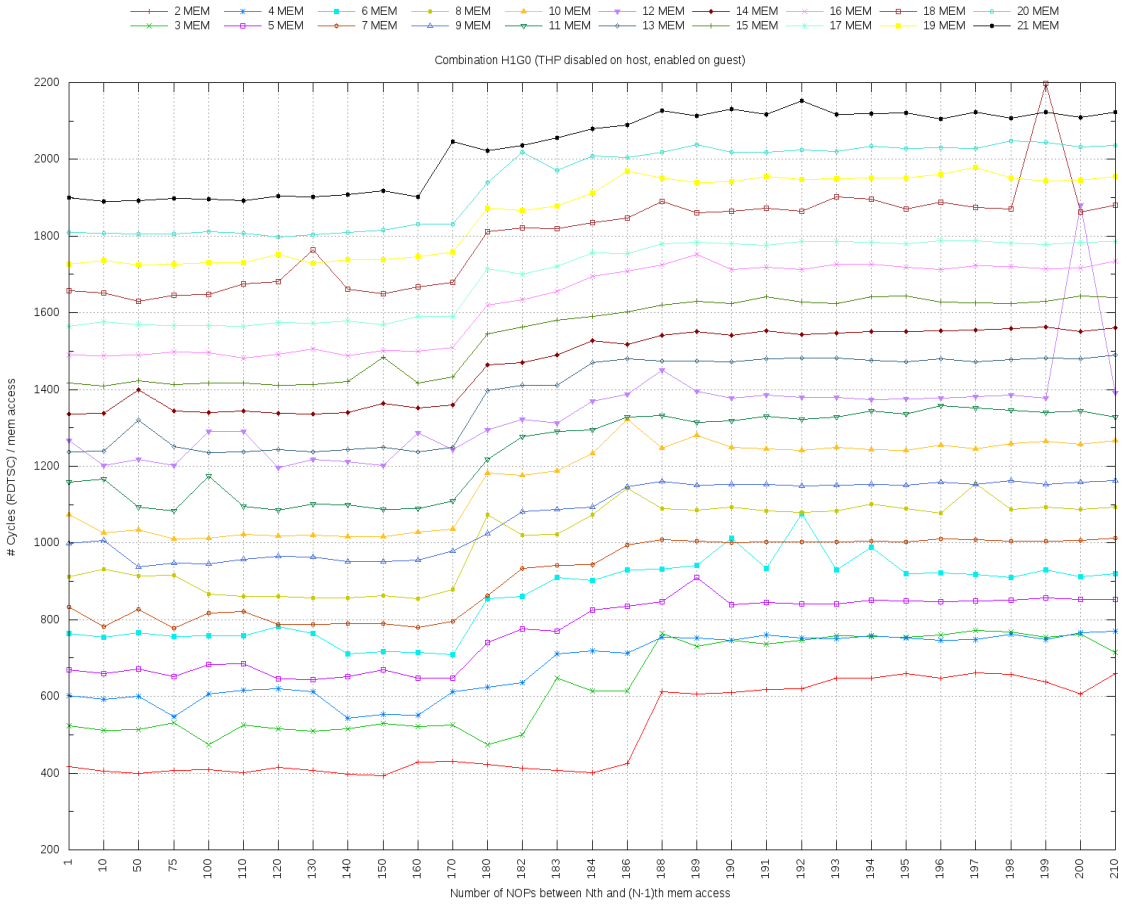

FIGURE 3.5: Combination H1G0 with configuration 3.

3.2 Conclusion

1. A TLB miss on the host (H0G1) is costlier than a TLB miss on the guest (H1G0).

2. Multiple runs across time give different results. This indicates an external/miscellaneous overhead that is not related to memory (as it is not multiplicative and does not add up on each access but 'shifts' the graph up by the same quantity). To reduce this, the VM (in some cases the server also) was rebooted for each of the above graphs.

3. Disabling THP on the guest i.e VM adds a slightly greater (30-40 cycles more) overhead than disabling on the host i.e. Xeon.

4. **Miscellaneous overhead** - If we observe the figures above, there is a slight mismatch. For eg., in the case of H1G0, 1-MEM takes about 330 cycles but 2-MEM SEQ take 600 cycles (Ideally it should be at least double i.e. 660 or above). Similarly, the figures are 420-450 and 800 resp. for H1G1 and 340-360 and 650 resp. for H0G1. This indicates that there is some overhead which is unrelated to memory and affects all the runs regardless of the number of MEM accesses

(each run has a different number of MEM accesses). This leads us to the following equation:

$$T(n) = n * M + C$$

where T(n) is the time taken by n sequential memory accesses, M is the time taken by one memory access and C is the constant overhead. Back-of-the-envelope calculations give C a value of about 40-50 cycles. I am writing a script to extract the values for up to 12 sequential memory accesses (as opposed to parallel in this case) and feed the data to an SMT solver, which can solve the above equation and give us more precise results.

Chapter 4

Effect of NUMA on MLP

Disabling NUMA makes the CPU treat the entire memory as a unified system, thus allocating some of its data on the remote node (adding the QPI access latency to each memory access). We cannot guarantee that the entire array is on the remote node or even what part of it is.

4.1 Results with NUMA disabled

Figure 4.1 results were obtained with NUMA disabled on the Xeon5 server:

4.2 Analysis

1. The QPI bus adds a latency of about 60-70 cycles to 1-MEM
2. However, as the QPI bus offers parallelism, the overhead is not cumulative for multiple parallel accesses. For 2-MEM we observe about 100 cycles of overhead (it increases from 220 to 320 cycles) and for 10 memory accesses we get about 110 cycles overhead (from 340 to 450 cycles).
3. 2-MEM SEQ i.e. two sequential memory access across two consecutive ROBs (the cliff region of the 2-MEM line) takes 540 cycles (200 for each memory access and about 60 - 70 for each QPI transfer).
4. Consequently, the effect on 1-MEM is proportionally greater than for multiple memory accesses. This divergence in the effect on 1-MEM compared to the effect on multiple parallel accesses leads to an increase in MLP.

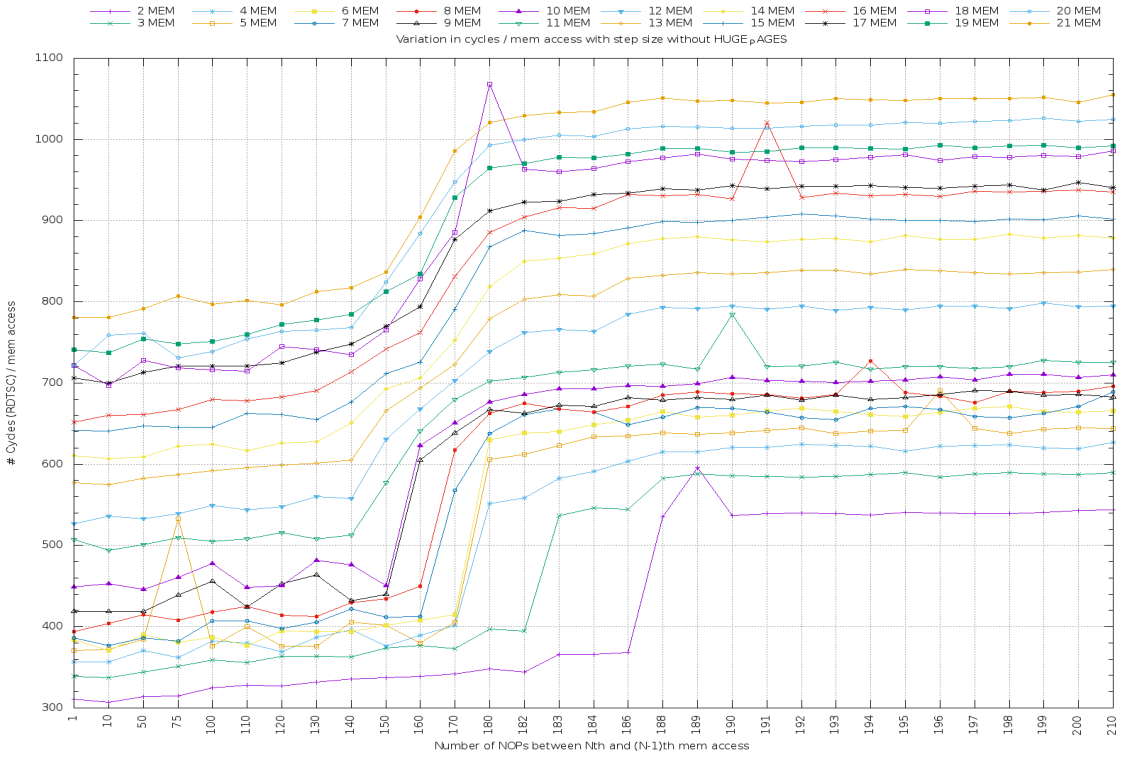

FIGURE 4.1: MLP of Xeon5 with NUMA disabled = 11.

5. By definition, the number of memory accesses that take the same time to execute in parallel as 2-MEM SEQ is equal to MLP +1. That number, as per the graph above, is 12. Thus, effective MLP = 11.

4.3 Conclusion

1. QPI bus offers parallelism. It affects one access proportionally more than multiple parallel accesses.

2. Thus, **effective MLP** increases to 11.

The main conclusion from the experiments so far is that while all bottlenecks (TLB, TLB-VM and QPI) add latency to memory accesses, their effect on 1-MEM and multiple parallel accesses can be different. While TLB misses are served only in sequence, it adds latency and reduces MLP. On the other hand, any device that offers some parallelism (like the QPI) will add latency but can also increase effective MLP by disproportionately impacting 1-MEM in comparison to multiple parallel accesses.

We are conducting a few more tests to isolate the constant overhead of VM on memory accesses, regardless of page size. We will report the same

Chapter 5

Varying the size of the array

5.1 Experiment

The size of the array is varied from 1024 bytes (1 KB) to 1 GB, incremented iteratively in powers of 2. We have plotted the graphs for 1, 2, 3 ... 11 memory accesses. For each run, we run the loop 50000 times to average the RDTSC and PAPI data and the benchmark is run 10 times for each memory access to reduce noise in L1, L2 and L3 hit regions.

5.2 Results

Figure 5.1 results were obtained by varying the size of the memory array being accessed. The accesses are random.

5.3 Analysis

1. The results are stable. We average them across 10 iterations for each value of N, where N is the number of memory access.
2. The L1 cache takes approx 4 cycles for one access (although for one L1 access, rdtsc is reporting only 1-2 cycles, but for two L1 hits, it takes 8 cycles, for three it is 12 cycles).
3. L1 hits are serviced in sequence i.e. we get no L1 level parallelism.

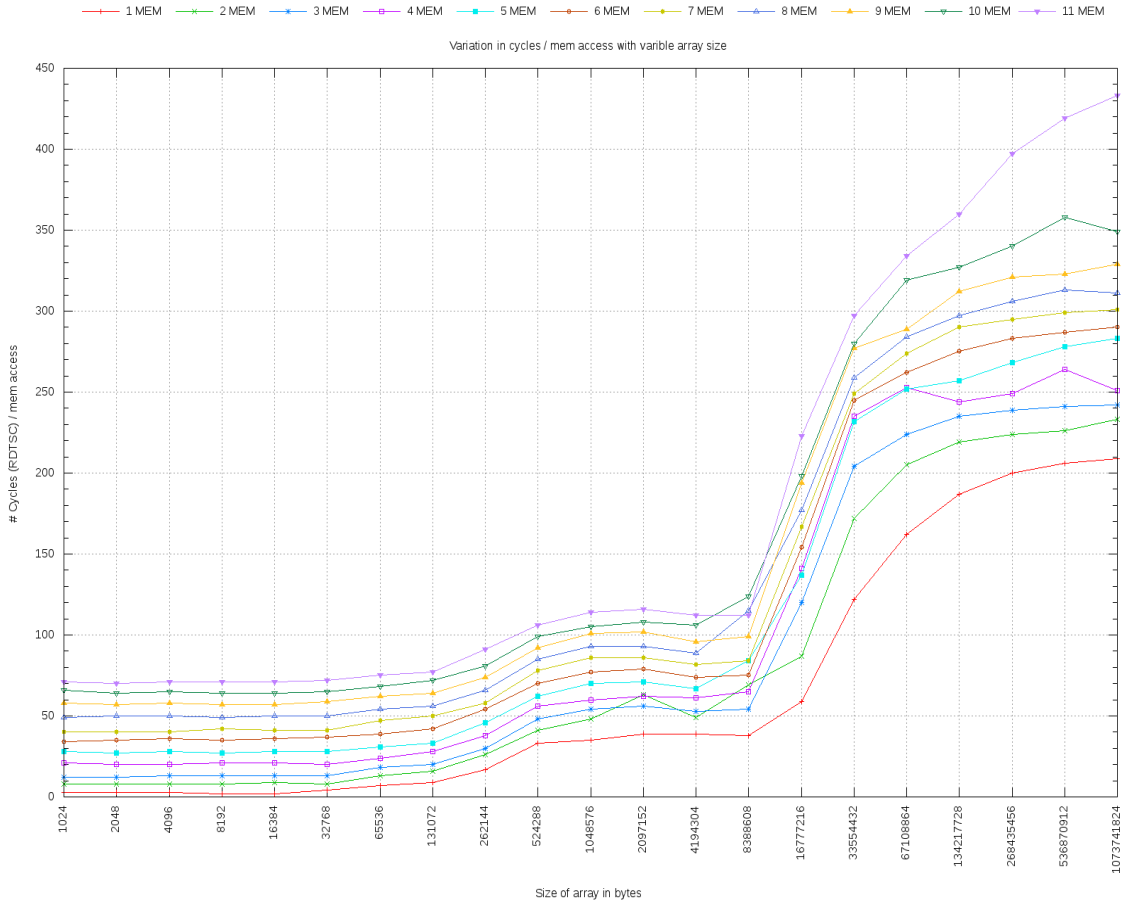

FIGURE 5.1: Varying the size of the array shows change in the relative performance of 11 accesses vis-a-vis 1 access

4. The L1 cache is 32K. For 64K array size, PAPI reports half of the accesses to be L1 hits and remaining are L2 hits (L1 misses). Thus, for 2 accesses we get one L1 and one L2 hit, for four we get two each and for 10 we get 5 each.

5. For 128K array, we observe that one-fourth of the accesses are L1 hits and rest are L2 hits i.e for 4 accesses we get one L1 and three L2 hits, for 8 we observe two L1 hits and six L2 hits.

6. This indicates that the L1 cache (32 K in size) is almost fully utilized by the program by allocating half of the 64K array on L1 and the remaining half on L2 and by allocating one-fourth of the 128K array on L1 and the remaining three-fourths on L2. Moreover, the random nature of the accesses ensures that both halves (in the case of 64K array) get equal number of accesses and the we experience little or no cache fragmentation. Also, the associativity of the cache could have reduce the number of L1 hits if multiple accesses mapped to the same set (as can be expected for strided access) but randomness reduces that factor to negligible levels, as is indicated by the data presented above.

7. For L2 cache (256K), we haven't achieved isolation in this experiment. For 256K array, we observe at least 1-2 L1 hits for 11 accesses and at least 1-2 L2 misses. This corrupts the data as L1 hits reduce latency and L3 accesses increase it greatly. This is the reason why no L2 hit region is noticeable in the graph although a slight increase can be seen after 32K. The first region has L1 hits, the second plateau being L3 hits and the final plateau has memory accesses.

8. However, for L2, we do get near full utilization like L1. For eg., for 512K array we observe 5 L2 hits for 10 accesses, 4 L2 hits for 8 and 2 L2 hits for 4. Thus, half the array is in L2 and the remaining half in L3.

9. In the L1 hit region, 10 accesses we observe an access latency of approx 65 cycles (err. 2-4 cycles). Assuming 4 cycles for one access, 10 accesses take more than 16 times to finish compared to one L1 hit. The ratios are 2, 3, 5, 7, 8.5, 10, 12.5, 14.5, 16 and 17.5 for 2, 3, 4, 5, 6, 7, 8, 9, 10 and 11 L1 hits respectively, indicating that L1 performance deteriorates as it takes multiple consecutive accesses (The ratio should have been at most 11 for 11 access, 10 for 10 accesses and so on, but we get higher ratios).

10. For, memory on the other hand, we observe that up to 10 accesses the latency increases to at most 1.7 times that of one access (1-MEM takes 200 cycles approx, 10-MEM take 340) and for 11-MEM we get a factor of 2 - 2.1 owing to memory level parallelism.

5.4 Conclusion

1. L1 cache is serviced sequentially. The performance is worse than sequential as 11 L1 hits take 16 times more than 1 L1 hit.

2. L1, L2 and L3 caches are fully utilized. Random access overcomes fragmentation due to set associativity.

3. Memory offers parallelism of up to 10 parallel accesses. MLP reduces when latency inducing factors like TLB misses, cross-QPI accesses and virtualization.

Chapter 6

Summary

The following table summarizes the results of our experiments.

Cases	Effective MLP	One MEM cycles	Time for MLP + 1 number of accesses	Overhead
No restrictions	10	200	420 cycles	Nil
Disabling huge pages	4 - 5	260	490-500 / 560-570 [1]	60 cycles
VM (H0G0)	10	200	425	Nil with some noise
VM (H0G1)	3 - 4	340 - 360	630 / 690	140 - 160 cycles
VM (H1G1)	2 - 3 (almost 3)	420 - 450	710-740 / 800-850	220 - 250 cycles
VM (H1G0)	3	330	600	100 - 130 cycles

Table 6.1: Summary