Page Size Duality Implication

Minali Upreti (16305R002) and Gaurav Kumar (163050078)

November 22, 2016

Contents

1 CONTEXT OF THE PROBLEM

When a process demands memory, the CPU allocates memory in chunks of 4K bytes (it's the default value of page size on many systems). These pages can be swapped to disk. Since the process address space are virtual, the CPU and the operating system monitor and manage things like which page belongs to which process, and where this page is stored. So, if a process needs more number of pages then the page table size of the process will be more, consequently more time will be needed to find the page in the memory i.e. more number of page table walks will be needed to map a virtual page to a physical page due to large page table of that process.

The solution to this is using Large Pages (2MB, 1GB pages) so that the number of entries in the page table of the process decreases by a huge number. With hugepages, finding an address is faster as few number of entries are needed in the TLB to provide memory coverage as compared to normal pages.

Not every workload benefits from huge-pages. Only some specific type of workloads show performance improvements using huge-pages. That is why sometimes using huge-pages can be dangerous, as it might sometimes lead to severely fragmented memory . Since the huge-pages are pinned to memory, they cannot be swapped out. Therefore main memory can fill up pretty quickly, and if you aren't swapping pages in and out, the perceived memory size will be much much smaller. So, pages of other applications running on the same system will be swapped in and out more frequently and thus such applications can experience severe latency issues.

2 GOALS

1. **Goal 1 :** To find the appropriate number of Huge-Pages that should be given to a process to take the performance benefits of Huge-Pages.

2. **Goal 2 :** To find the type of workloads that benefits from huge-pages and to find the huge page size setting in Guest and Host OS that will benefit the workload the most.

3. **Goal 3 :** To identify reasons (with respect to hardware counter events) so as to why a particular page size setting in Guest and Host OS gives high performance improvements for a particular type of workload.

3 DESIGN DETAILS

We are conducting this experiment to test various type of workloads under different page size settings (H:small, G:small), (H:large, G:small), (H:small, G:large), (H:large, G:large). During the execution of workloads we record various events like cache-references (LLC cache-references), cache-misses (LLC cache-misses), dTLB misses etc. We also record time taken to execute the workload.

1. **Number of HugePages :** We conducted all the experiments with same number of huge-pages. We kept this number large enough to accomodate all of the memory needed a memory intensive process in RAM using huge-pages. Since we are using allocating 2GB of memory for a memory intensive process. We kept the number of huge pages to be 1100.

2. **Page Size Settings :**

   (a) G: 4KB, H: 4KB

   (b) G: 2MB, H: 4KB

   (c) G: 4KB, H: 2MB

   (d) G: 2MB, H: 2MB

3. **Performance metrics :** Following performance metrics are recorded and analysis during the experiments. These metrics are recorded using the *Perf* tool:

   (a) **cache-references :** The LLC cache-references.

   (b) **dTLB-misses :** The data-TLB misses.

   (c) **Execution-time :** Time taken to perform the desired task in the process. This does not take into account the time taken to allocate memory needed by the process.

4. **Type of workloads:**We conducted experiments for following workloads:

   (a) **CPU Intensive Workload (Sysbench) :** This workload calculates the number of prime numbers in the given range. We have given the range of (0-20,000). The code outputs the total execution time. We test this workload for all the page size settings mentioned above.

   (b) **Memory Intensive Workload :** We have C code which allocates and initializes 2GB of memory. After initializing 2GB of memory with zeros, we read this memory with some percentage randomness. For eg: The first experiment that is done is purely a sequential code with %randomness = 0% which means that all the data in the memory of the process is read sequentially and written back with same values, similarly the second experiment that is conducted is code with %randomness = 20%. This means 20% of the data in the memory is read randomly and 80% of the data in the memory is read in sequential manner. We have carried out the experiment over the 0%, 20%, 40%, 60%, 80% and 100% random reads for each page size setting.

(c) **I/O Intensive Workload (File I/O) :** This is a script that copies any amount of memory from one file to another i.e. reads from disk and writes on to the disk again. We have copied 1.5GB of file using this script and recorded time under all page size settings mentioned above.

(d) **Redis :** REmote DIctionary Server. Redis maps keys to types of values. An important difference between Redis and other structured storage systems is that Redis supports not only strings, but also abstract data types like Lists of strings, Sets of strings (collections of non-repeating unsorted elements). We have created a map (of key-value pair) on redis server of size 500 MB and tried to access all of them. We recorded the time taken to access these values and did that for above mentioned page size settings.

4 IMPLEMENTATION DETAILS

We have conducted these experiments with Host OS as Ubuntu Desktop 14.04.4 LTS and Guest OS as Ubuntu Desktop 14.04.4 LTS. We have considered KVM hypervisor. To record the hardware events in guest and host we have used the *Perf* tool. *Perf* is a performance analysing tool available from available from Linux kernel version 2.6.31. This tool can measure a variety of hardware and software events during the execution of a process. We have measured cache-references and dTLB misses using this tool. For a particular page size setting, the workloads described above are run in the Guest OS and simultaneously the *Perf* tool is run in the Host OS which records the hardware performance counters for Guest OS as well as for Host OS. Given below is the dTLB information of the system on which the experiments were performed :

- 4KB : 4way 64 entries
- 2MB : 4way 32 entries

5 EXPERIMENTATION

5.1 Experiment 1:

Question: What is the appropriate number of Huge-Pages that should be given to a process to take the performance benefits of Huge-Pages ?

Strategy: We conduct this experiment with a memory-intensive process. Different settings for number of huge-pages are considered, these are: 200MB, 400MB, 600MB, 800MB, 1000MB, 1500MB, 2000MB. To see the effect of performance benefits of hugepages, it is convenient to give 2MB page size in Guest OS. So, we are using H:2MB, G:2MB page size setting in-order to carry out the experiment. *Figure 1* , *Figure 2* , *Figure 3* presents the dTLB-misses, cache-references and time taken to complete the program respectively under several settings for number of huge pages. According to the graph plotted the benefits of huge-pages can only be realized if the number of huge pages used is greater than at least 1500MB to see significant improvements. We have allocated 2000MB of memory to huge pages for conducting **Memory Intensive Workload** experiments.

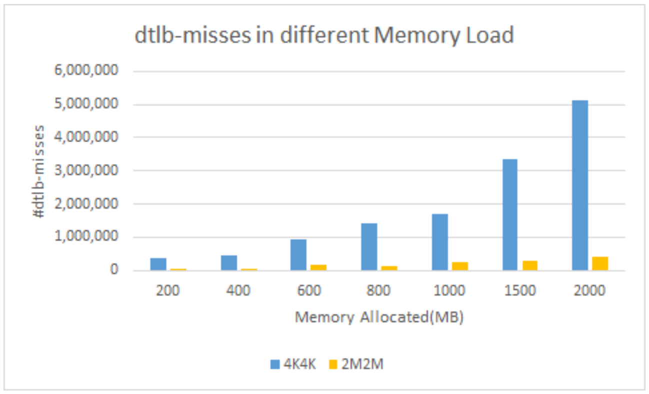

Figure 1: dTLB misses graph on using different number of huge pages for page size settings of (4K,4K) and (2M,2M)

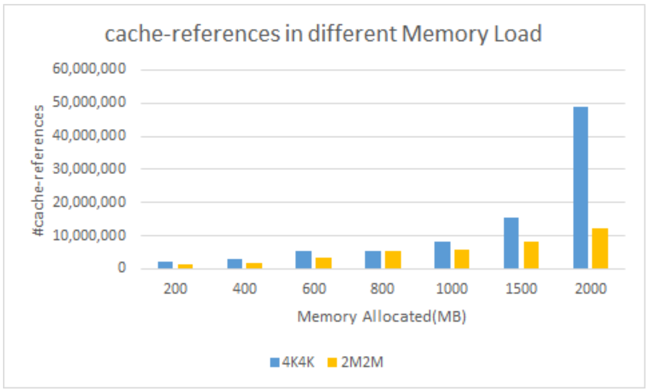

Figure 2: cache-references graph on using different number of huge pages for page size settings of (4K,4K) and (2M,2M)

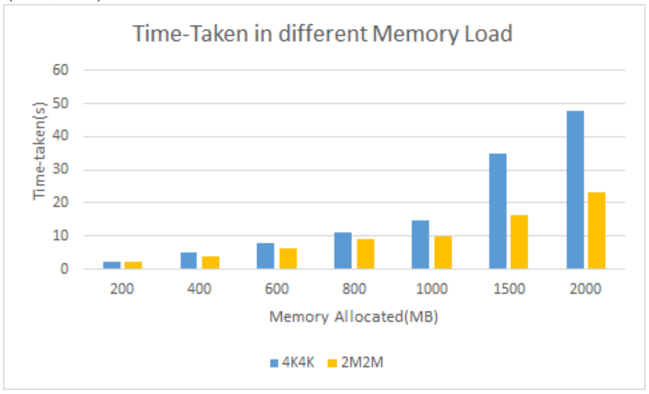

Figure 3: Time of execution graph on using different number of huge pages for page size settings of (4K,4K) and (2M,2M)

Figure 4: Performance Improvement with number of Huge Pages

5.2 Experiment 2:

Question: It is known that huge-pages improve the performance in some type of workload. What are these workload types ? And which huge page size setting will benefit the application most ? If a particular workload is performing good in one page size setting and not in other then why ?

Strategy: These questions are answered for all workloads in isolated conditions i.e. No other VM is running in the KVM and also no other application is running in Guest OS and Host OS. To answer these questions we collected the data for the four page size settings as mentioned in section 3. For the **Memory Intensive Workload**, we have considered 4 different page size settings and for each setting we have varied %randomness in the program.

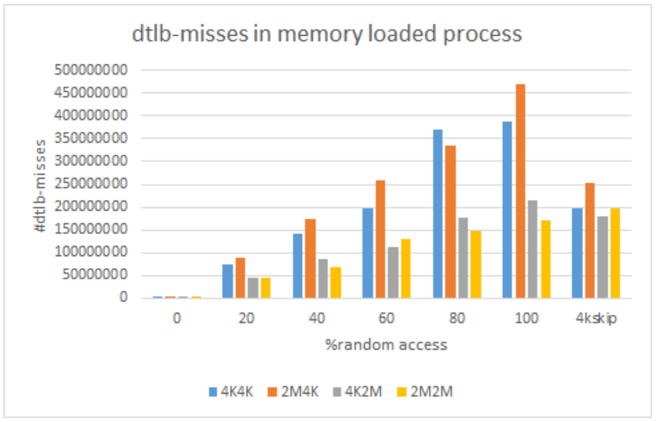

Figure 5: Guest dTLB misses graph for various page size settings

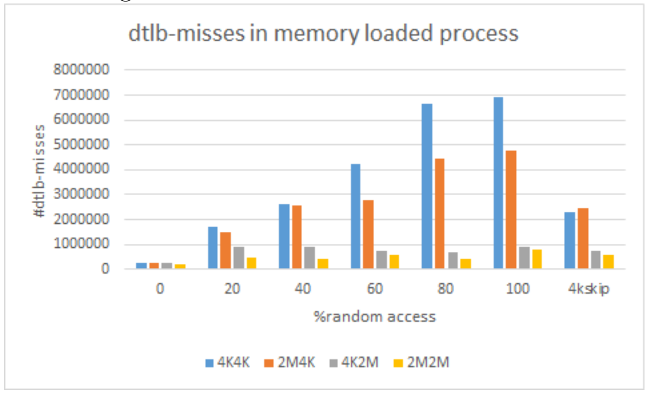

Figure 6: Host dTLB misses graph for various page size settings

Figure 7: dTLB misses for Guest OS and Host OS for a Memory Intensive Process

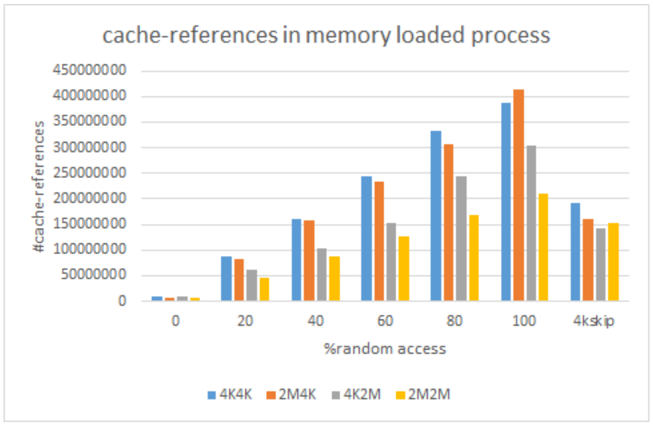

Figure 8: Guest cache-references graph for various page size settings

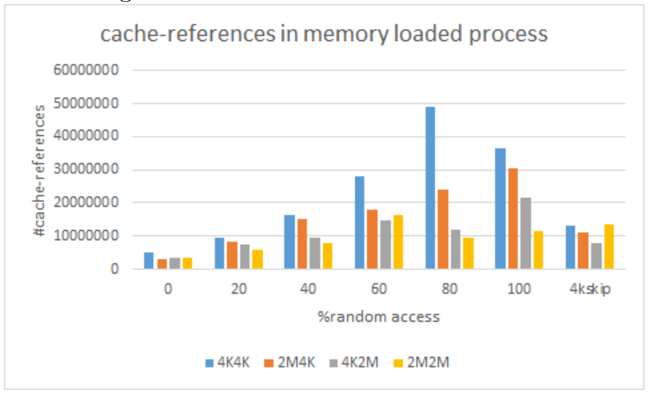

Figure 9: Host cache-references graph for various page size settings

Figure 10: cache-references for Guest OS and Host OS for a Memory Intensive Process

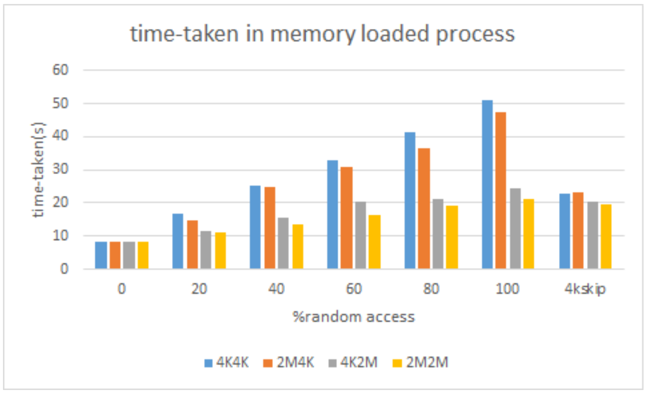

Figure 11: Time of execution for a Memory Intensive Process

Figure 7 shows the dTLB misses graph for various page size settings when %randomness is varied along the x-axis. From the *Figure 7* , *Figure 10* , *Figure 11* it can be concluded that the workload shows best results with H:2MB, G:2MB page size setting. Also the large percentage improvements are observed when Guest OS is run with huge-pages than when Host OS is run with huge-pages i.e. The order in which the setting can be arranged which benefit the workload from least to most is the following :

(H:4KB, G:4KB) <(H:2MB, G:4KB) <(H:4KB, G:2MB) <(H:2MB, G:2MB)

Also, with the increase in %randomness in the application the figures for improvements are also getting better. A **memory intensive application** performs the best when it is 100% random.

One interesting observation that can be inferred from the figure is that there is no change in the data that is collected for sequential access of data (0% randomness). The graph becomes flat when the **memory intensive application** is run with 0% randomness i.e. when the data is read sequentially, the page size setting didnot matter because when there is sequential read, most of the data needed by the processor is present in the L1-cache or L2-cache, so there are less number of TLB lookups. This is evident from the number of LLC cache references in *Figure 10* .

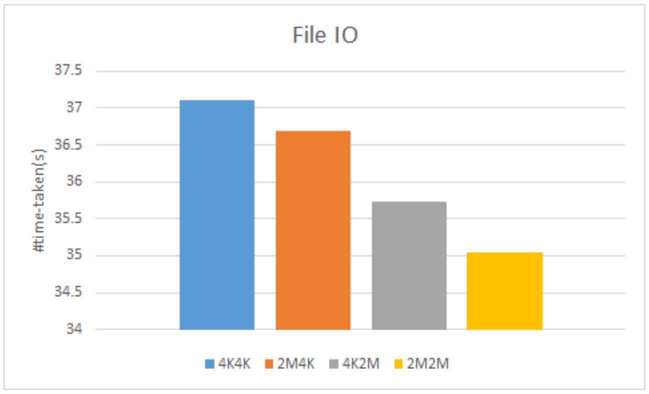

Figure 12: Time of execution for a I/O Intensive Process under all page size settings

In **I/O Intensive workloads** operations like reading from the memory and writing into the memory is done by using a stream. Stream can occupy one or two pages in the memory. Because of this the number of pages occupied in memory is very less in number while dealing with large files also. Since such a process is not using large amount of RAM, we do not expect so see performance improvements with huge pages in such workloads. This is evident from the *Figure 12* . There is negligible improvements in time taken (nearly 1-2 seconds) to execute the program when it is run with different page size settings.

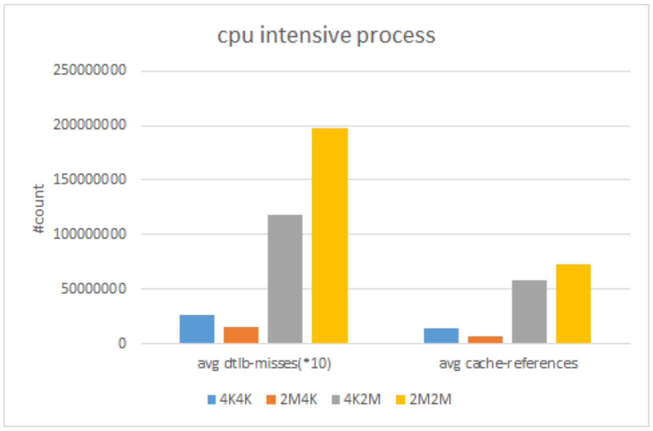

Figure 13: Guest dTLB misses and cache-references graph for CPU Intensive Workload for various page size settings

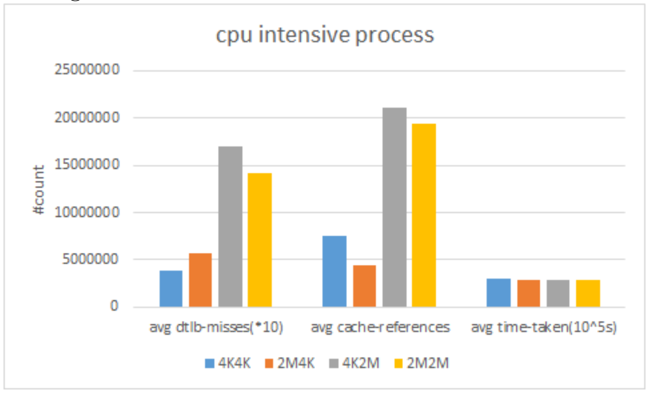

Figure 14: Host dTLB-misses, cache-references and time of execution graph for CPU Intensive Workload for various page size settings

Figure 15: Data for Guest OS and Host OS for a CPU Intensive Process for various page size settings

From the *Figure 15* it can be concluded that in **CPU Intensive Workloads** the number of pages used by the process in the memory is very less in number. So, in this case also we did not expect performance benefits from hugepages. Same is evident from the results.

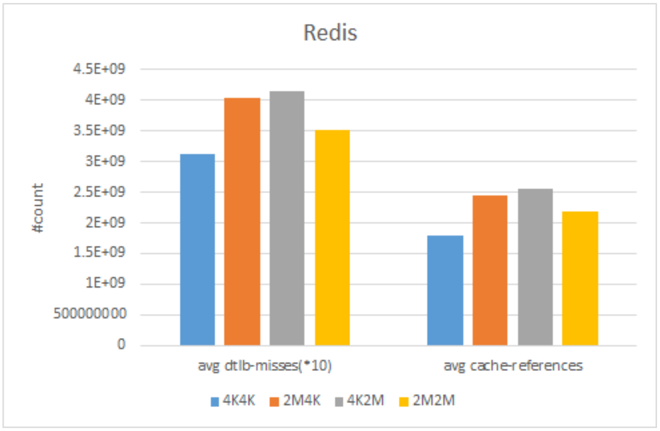

Figure 16: Guest dTLB misses and cache-references graph for Redis various page size settings

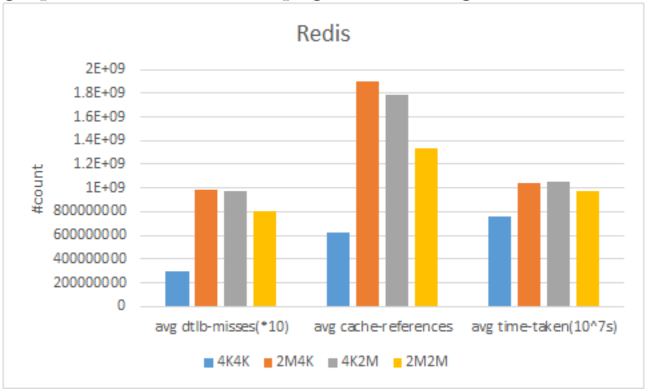

Figure 17: Host dTLB-misses, cache-references and time of execution graph for Redis various page size settings

Figure 18: Data for Guest OS and Host OS for a Redis Process for various page size settings

From the *Figure 18* it can be concluded that for workloads like **Redis** , the performance can deteriorate with huge pages. **Redis** incurs a big performance loss because it uses fork call in-order to persist on disk. In-order to make a copy of the data in the disk at least in every 2 seconds (source : wiki) so that it can recover from abrupt system failures. To do this, fork is called and two processes with shared huge pages are created. Few event loops can cause copy-on-write of the whole memory trigerring a heavy number of huge pages allotment to the process. This results in big memory usage, and thus increasing the latency of the system.

6 INFERENCES

Following inferences can be made after performing the above experiments :

1. It is necessary to give appropriate number of huge pages to the process inorder to experience performance improvements due to huge pages.

2. Workloads that require large amount of memory very often benefit from huge pages. For example : A large chunk is unable to fit into single 4KB page (as it spans over thousands of 4KB pages). On the other hand if huge pages are used only few number of pages will be used, potentially improving performance. But this is not always true, as we have seen in the case of Redis.

3. In isolated conditions, i.e. when there is no other VM that is running over KVM and there are no other applications running in Host OS and Guest OS, (G:2MB, H:2MB) gives best results out of all page size settings.

4. Performance improvements due to huge-pages can be seen in memory intensive workloads. On the other hand it has neutral effects on cpu intensive and I/O intensive workloads, since very less amount of memory in RAM is used. Whereas the performance deteriorates in some workloads (eg: Redis) , with huge pages even though they occupy large section of memory like Redis.