Evaluating the performance of Linux memory mapping in Power8 system

Balamuruhan S
KVM on Power – FVT
Linux Technology Centre
IBM India Systems Development Lab
balamuruhans@in.ibm.com

Anshuman Khandual
Linux Kernel Team
Linux Technology Centre
IBM India Systems Development Lab
anshuman.khandual@in.ibm.com

***Abstract*—Purpose of this paper is to demonstrate impact in the performance of Linux memory mapping from application across various supported page sizes in Power8 system. This paper will characterize in detail and illustrates how various factors like Linux kernel memory management, Memory Management Unit (MMU) translation and Translation Look-aside Buffer (TLB) are involved in the performance felt from userspace through memory mapping.**

I. Introduction

To understand the performance of Linux memory mapping in Power8 systems we have to study the speed of memory buffer creation using `mmap()` system call, bandwidth felt by the application by faulting all the memory in sequence by touching the memory buffer created, faulting at various offset by randomly touching the memory, reclaiming the faulted memory under pressure. To evaluate the performance experiments are studied with different memory situations using required performance monitoring units (pmu) – when buddy is fresh without being fragmented (after reboot) and when buddy is fragmented (after some memory pressure tests) and with different memory pages - Hugepages, THP and normal.

II. Memory mapping

A. Need for virtual memory and memory mapping

With virtual memory, programs running on the system can allocate far more memory than is physically available. Indeed, even a single process can have a virtual address space larger than the system's physical memory. Virtual memory also serves *memory protection as e*very program uses a range of address called the *process address space* and prevent programs from interfering with other programs. These virtual addresses are mapped to physical memory by page tables, which are maintained by the kernel and consulted by the processor. The process address space consists of a number of memory segments - executable code, static data, heap, stack, shared libraries etc., we have the memory mapping segment where kernel maps contents of files directly to memory in process address space.

B. Creating memory mapping

It is also possible to create an anonymous memory mapping that does not correspond to any files, being used instead for program data. From user space we can ask for a mapping via the Linux `mmap()` system call. On success address of the new mapping is returned and speed of memory buffer creation is completely software dependent.

C. Memory mapping from OS perspective

In Power 8 the memory address translation takes place in 2 stages, the first stage maps the Effective Address (64 bit) to Virtual Address (80 bit) and second maps the Virtual Address (80 bit) to Real Address (64 bit). EA to VA translation is achieved using segment table stored in memory which is located in architectural Address Space Register(ASR) and maintained by Kernel. Power 8 architecture uses the MSB 36-bit Effective Segment ID (ESID) from EA as a key to search into a 4 K segment table. A unique segment table entry (STE) is extracted from the segment table and is used to translate the Effective address into a Virtual address. The speed of this translation is increased by caching it in a buffer known as Segment Look-aside Buffer (SLB).

VA to RA translation is achieved using Page Table stored in the memory which is located in architectural Storage Description Register 1(SDR1) and maintained by Kernel. MSB of 68 bits are used to search entry in Page Table for retrieving 52 bit Real Page Number and concatenated with 12 bit offset from Effective address to form 64 bit of Real Address. Similar to first stage a cache known as Translation Look-aside Buffer (TLB) is defined by Power 8 architecture for saving recently accessed page table entries. TLB entries are maintained by hardware.

III. Page fault

A. Page fault

Linux allocates memory to processes by dividing the physical memory into pages, and then mapping those physical pages to the virtual memory needed by a process. It does this in conjunction with the Memory Management Unit (MMU) in the CPU. Page fault is an interrupt raised by hardware when it couldn't access a memory page that is not actually mapped by MMU into virtual address space of process. When the address returned by `mmap()` is accessed for first time, page fault is triggered using interrupt delivery by hardware and interrupt handling is taken care by kernel. So the performance is both

software and hardware dependent which is measured and evaluated using associated parameters of pmu.

B. Page fault handling

Hardware MMU searches for the mapped page in TLB and if the entry is not available page fault is triggered and control is transferred to Kernel. Linux kernel handles the page fault and allocate pages to process, retrieve the pages from disk into memory, configure MMU by inserting the translation entry and asks the CPU to proceed by creating TLB entries. TLB entries are hardware cache speeds up the lookup process while MMU translation is active and present. TLB entry guarantees the translation even in the absence of MMU translation hence flushing of TLB entries are strictly taken care.

C. Page fault backed by page sizes

The physical memory gets divided into pages based on the page sizes supported by the architecture. Page sizes that are supported by Power8 are 64K, 16M and 16G and the page faults are inversely proportional to the page size.

D. Memory reclamation under pressure

When kernel doesn't have sufficient physical memory and under memory pressure, kernel starts to write pages to disk and use the newly freed pages to satisfy the current page-faults. Writing pages out to disk is a relatively slow process (compared to the speed of the CPU and the main memory), however it is a better option than just crashing or killing off processes. The process of writing pages out to disk to free the memory is called swapping-out. If later a page fault is raised because the page is on disk, in the swap area rather than in memory, then the kernel will read back in the page from the disk (very slow) to satisfy the page fault and insert the translation again. This is swapping-in.

IV. Experiment to study the performance

The performance of memory mapping from user space are affected considerably by page fault handling and memory reclamation. In this paper, experiments are designed to instrument pmu counters like page-faults, dTLB-load-misses, instructions, hw-cache-miss and hw-cache-reference during mapping and accessing memory from user space. These counters will be used for studying the performance of memory mapping backed by normal page, hugepage and Transparent hugepage different environment.

A. Page-faults:

As discussed earlier it is a type of trap raised by hardware when a memory page is mapped into virtual address space but not actually loaded into memory. We exclude pagefaults count from kernel space to retrieve value based on memory mapped from userspace.

B. dTLB-load-misses:

A data translation lookaside buffer (dTLB) is a cache that have virtual to physical address mapping for data and dtlb miss occurs if data is not found in this cache. We exclude counting dTLB-load-misses for kernel and retrieve only for userspace.

C. instructions:

Number of instructions executed for the program, we configured pmu to count instruction executed for userspace and kernel.

D. hw-cache-miss:

Cache miss is a state where the data requested for processing a application is not found in the cache memory. It causes execution delays by fetching the data from other cache levels or the main memory. Cache miss occurs within cache memory access modes and methods. For each new request, the processor searches the primary cache to find that data. If the data is not found it is considered a cache miss. Each cache miss slows down the overall process because after a cache miss, the CPU will look for a higher level cache, such as L1, L2, L3 and random access memory (RAM) for that data. Further, a new entry is created and copied in cache before it can be accessed by the processor.

E. hw-cache-reference:

It indicates the last level cache accesses and include prefetches and coherency messages

V. Environment to study the Performance

we use Normal pages (64K), Hugepage (16M), Transparent Hugepages (16M) in ideal evironment and in stressed environment to allocate 16MB of memory individually for studying the performance.

A. Ideal environment (no memory stress):

We consider the system after reboot, when buddy is fresh, not fragmented and no memory pressure as ideal environment. In ideal environment the pmu counters are instrumented and memory is accessed continuously and randomly. We use `memset()` for continuously accessing the memory and it involves overhead of both hardware and software considerably for performance. To access memory randomly, random offset with in the memory range is generated and accessed in iterative way, this will ensure to have more of software than hardware overhead.

B. Stressed environment (simulated memory stress):

To simulate the stressed environment, buddy is forced to get fragmented by allocation and demanding for the page in a separate pthread thereby creating memory pressure on the system. The demanding for a page is randomized by generating offset to access memory in random. In parallel the

experiment to allocate 16MB memory using HugeTLB, Transparent Hugepages and normal pages are performed to record the pmu counters in similar way as explained earlier. By this way, system will be forced to handle memory pressure and we study the performance of the Linux memory mapping under memory stress.

VI. Implementation and Configuration

A. Code design

In order to simulate the memory pressure, it is programmed to be threaded to have a thread allocating 95% of system's physical memory with normal page and release the control for the program to proceed but in background that thread touches the memory randomly with offset of normal pages. This will make sure there is always memory pressure felt by the kernel and in parallel program allocates 16MB of memory with respective memory pages supported and fault it by touching in iterative way (with loop to record software dependent) and with `memset()` (which involves both hardware and software). Instrumenting the counters of performance monitoring units in right places to retrieve the values and studied for analysis.

`mmap()` system call is used to allocate and map anonymous memory for hugepages and normal pages. For Transparent hugepages, we advise kernel to mark the memory allocated to promote for hugepages using `madvise()`.
echo madvise > /sys/kernel/mm/transparent_hugepage/enabled for configure the kernel to listen for `madvise()` call and MADV_HUGEPAGE advise from `madvise()` can be used only if kernel config CONFIG_TRANSPARENT_HUGEPAGE is enabled during kernel compilation. Transparent Huge Pages works only with private anonymous pages. The kernel will regularly scan the areas marked as huge page candidates to replace them with huge pages. The kernel will also allocate huge pages directly when the region is naturally aligned to the huge page size.

B. System Configuration:

Machine: Power 8 Habanero
Distribution: Ubuntu 1704 (Zesty Zapus)
Linux Kernel: 4.14.0-rc2-00048-gdc972a67cc54 (upstream)
Gcc: 6.3.0 20170406

C. Source code link:

https://github.com/balamuruhans/linux_mm_experiment

D. Experimental Test code:

normal_memory_analysis_continuous.c
normal_memory_analysis_random.c
pressure_memory_analysis_continuous.c
pressure_memory_analysis_random.c
event.c, event.h, reg.h, utils.h are taken from Linux kernel source at linux/tools/testing/selftests/powerpc/pmu.
event_apis.h have wrapper functions to make use of pmu.

VII. Analysis of Experimentation

From above experimentation, we can observe that page-faults are inversely proportional to the page sizes. Normal pages with size 64K requires 256 faults whereas Hugepage or Transparent Hugepage with 16M page size requires only 1 fault to access the 16M memory irrespective of continuous or random access and in ideal or memory pressure environment. As dTLB load miss also depends on the page size, we can see the values are exactly same as page-faults that is 256 dTLB load misses for 64K pages and 1 miss for 16M Hugepage or Transparent Hugepage.

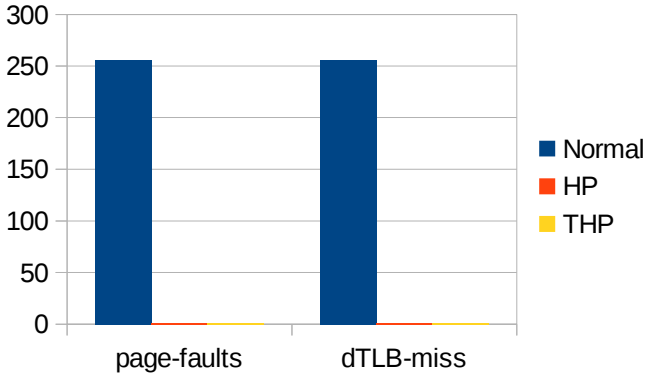
Hardware cache reference and Hardware cache miss which contributes considerably for memory performance have recorded in results for our analysis. Hardware cache references for ideal environment and continuous access for all the page sizes are almost same (< 120). Under memory pressure with continuous access THP have performed with less cache references (< 100) which is better among normal and Hugepage (> 120). Upon random accessing the memory irrespective of ideal or under pressure HP and THP have given almost same (< 6000) better performance than normal pages (> 12000).

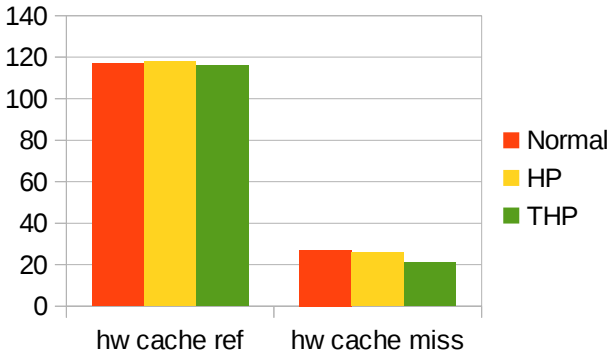
Under ideal environment with hardware cache-miss, THP have performed better than Hugepage and Hugepage have performed better than normal pages. Under memory pressure upon continuous access normal page performed better than THP and THP performed better than HP. Where as under memory pressure with random access of memory THP have performed better than HP and HP better than normal page with reduced hardware cache misses.

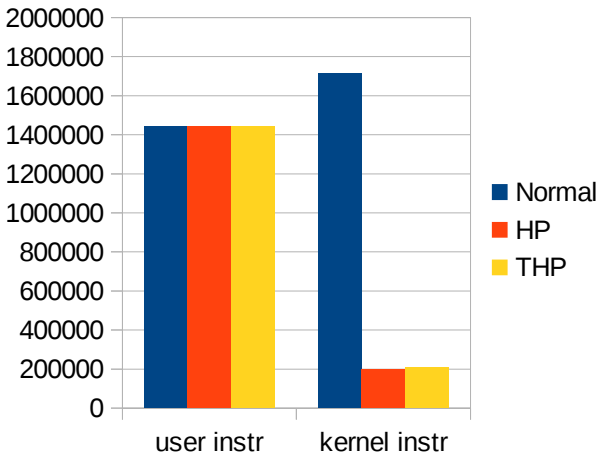
Instructions executed for userspace and kernel are recorded by the experimentation which contribute for the performance notably. When we compare instructions executed for userspace in ideal environment continuous access, ideal environment random access, memory pressure environment continuous access and memory pressure environment random access remains same for all the page sizes. Even-though the values remains to be same irrespective of page-size, in continuous access we touch all the allocated memory so the value is proportionally more (> 1400000), where as in random access we touch only certain memory randomly and the value is relatively low (< 30000). Now when we analyze the instructions executed for kernel, the values are varying inversely with page size. The kernel instructions executed for Normal page with 64K is high (> 1600000) in irrespective of environment and access. Hugepage and Transparent hugepage records relative low (200000) than normal pages, but Hugepage shows more performance than THP among them by executing less instructions even under memory pressure environment.

VIII. Results

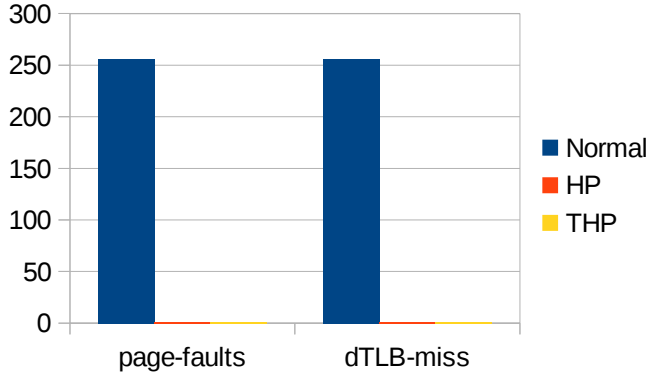
A. *Ideal environment continuous memory access*

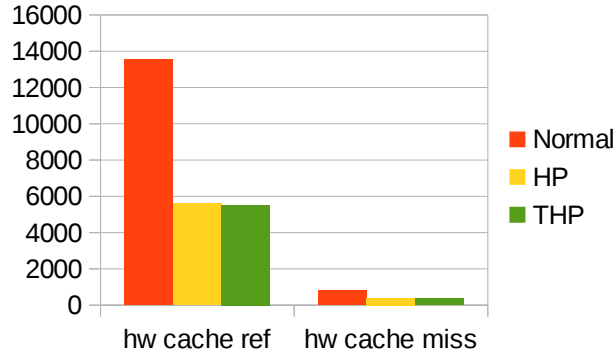


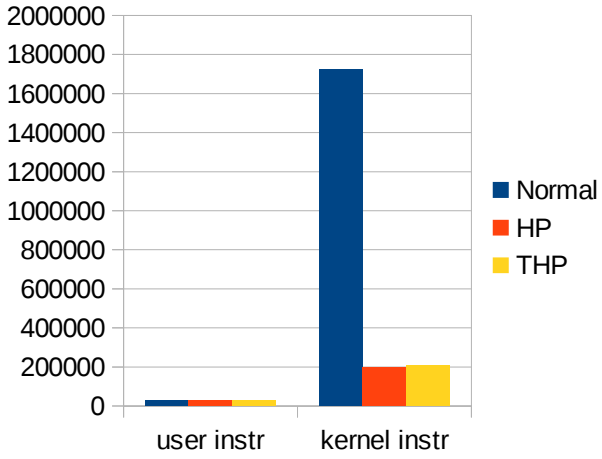




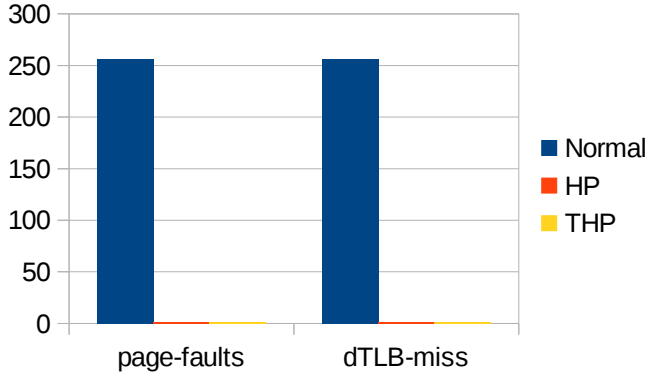
B. *Ideal environment random memory access*

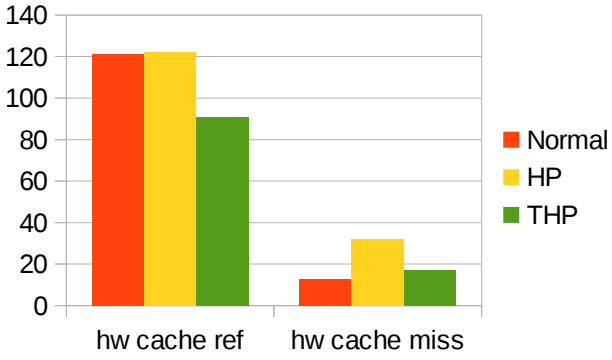


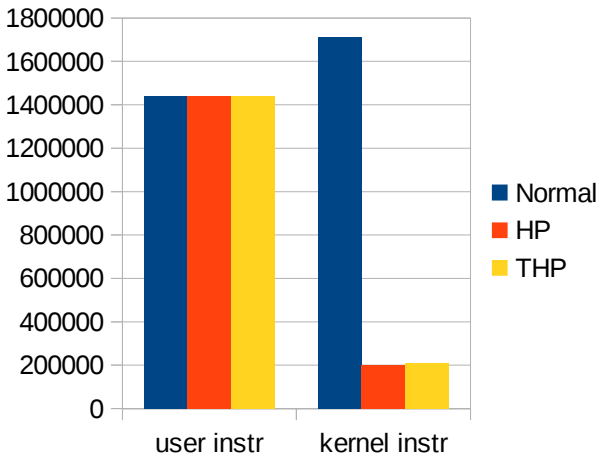




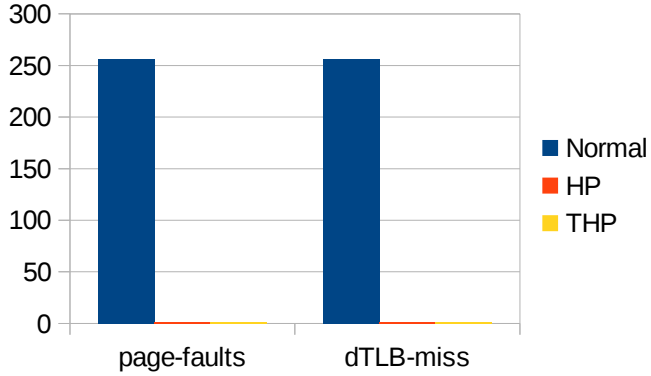
C. Pressure environment continuous access

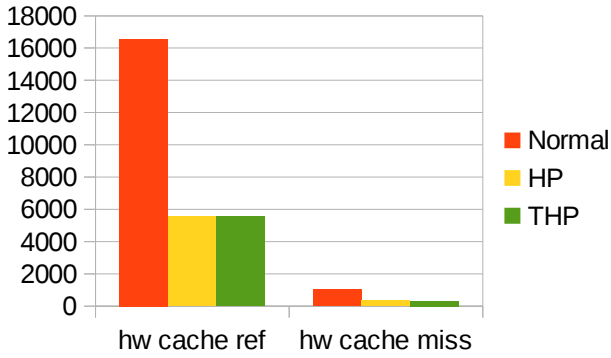


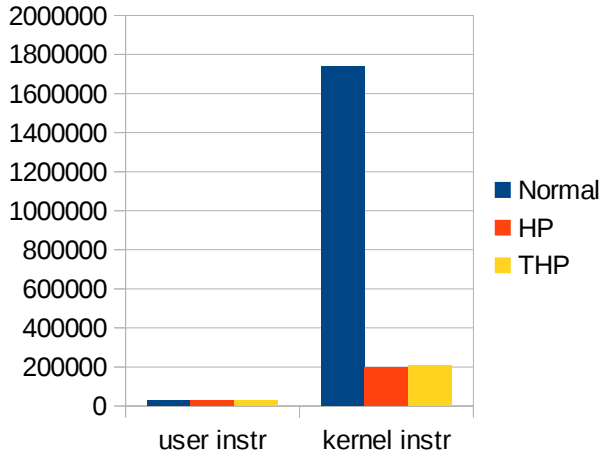




D. Pressure environment random access







IX. Conclusion

With the experimentation on Power 8 machine to evaluate the performance of memory mapping and accessing it under different environment, we can understand that Hugepage and Transparent Hugepage perform better in all the environment. To conclude we should consider to use HP or THP based on the requirement and behavior of applications to use high memory aligned to Hugepage size for better performance.

X. Future work

As a future work we have understand the behavior of memory management unit with kernel to learn the Power architectural design and implementation, to study the Linux kernel memory subsystem components and experiment to test it whether the algorithm works as per the design and implementation.

XI. References

[1] C.Ray Peng, Thomas A . Petersen and Ron Clark, The PowerPC Architecture: 64-Bit Power with 32-Bit Compatibility.

[2] Wikipedia – Page-fault, Memory Management Unit

[3] Manpages – mmap(), memset(), madvise(), perf_event_open()

[4] https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Memory/virtual.html

[5] http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/