Measruing the Linux Virtual Memory Subsystem

Yan Zhai
Department of Computer Science and Technology, University of Wisconsin Madison
{zhaiyan920}@gmail.com

ABSTRACT

Virtual memory is one of the most important subsystems inside modern operating systems. Although it is transparent to users, understanding the virtual memory can help to build better applications, especially in performance improvement. In this paper, I come to four issues of virtual memory: TLB, User space allocator, huge pages, and optimization of code sharing. I use performance measurement to explore deeply how these parts are working underlying our benchmarks. Most of the experiments run as I expect, except the TLB part. I will explain further in the paper about the methods, the results, and the conclustions on each separate parts. Moreover, I will try to illustrate why TLB measurement does not work well.

1. INTRODUCTION

The virtual memory subsystem has become an indispensable constitution in modern operating systems. With the proper hardware support in paging and segmentation, operating systems build their own mechanisms to protection and abstractions to users. This greatly simplifies the price to write correct code, and also make the operating system more reliable to the user faults.

However, writing correct code is not equal to writing good code. The applications may not perform well without knowing underlying operating systems and hardware. For example, designing a good web server will often require good knowledge of how to maximize the usage of memory and avoid long latency from disks [1]. Thus, it would be quite attractive to reveal what is beneath the beautiful illusory operating systems provide with you.

The virtual memory subsystem is quite huge, we mainly focus on several topics:

- TLB(Translation Lookaside Buffer), the 'cache' of the virtual memory. It will be necessary to know how the TLB are working, how large it is. We try to measure the TLB size through a set of experimentations, in order to know more deep about this small buffer.
- Huge Pages. To avoid TLB missing, huge pages can serve quite good for this purpose. But while it has many benefits to use huge pages, what is the cost? In this paper, I try to illustrate the cost of huge pages with the performance cost in page preparation, allocation aspects.
- Memory Allocator, like *malloc*, *mmap*. The only thing users can see is these allocators will allocate the virtual pages when invoked. However, when can user really uses this page? What is the allocation policy of the physical pages? These are all interesting questions to answer.
- Optimization of Virtual Memory System. There are many optimization for the virtual memory, like better page replacement algorithm, prefetching, and so on. In this paper, I explored the benefits from the object sharing.

To flexibly employ all kinds of measurement strategies, we choose Linux as our major experiment environment. Four major experiments, and several minor ones are carried out toward above topics, and most of them run well, matching my understanding to the architecture and system. One thing that causes trouble is the TLB. The measurement of TLB is not quite accurate, and I will explain the reasons with the gathered timing and hardware events data.

The paper is organized in following way: section 2 will introduce the environment, including time function I used, and some configuration details. section 3 is measuring different level TLB sizes. section 4 describes how we measure the memory allocator. section 5 will test the huge page overhead, section 6 illustrates the benefits from shared objects.

2. EXPERIMENTAL ENVIRONMENT

2.1 Timer in Linux

To best measure the times in experiments, I still decide to use system call *gettimeofday*. The major idea behind this is to enlarge the experiment scale, and ammortize the overhead. There are some high resolution things like *Rdtsc* instruction on Intel's X86/64 platform. But it's hard to use, and its behavior varies on different platforms as I tested. Though *gettimeofday* only supports measurement at ms level, we could see later, it is enough for our experiments.

2.2 Hardware and Software Environment

The machine I used for testing is a x86-64 machine. The processor is Sandy Bridge family, Intel-i5 2500K 3.3GHZ. This processor has two level private cache, and a last level shared cache. Both L1 data cache and instruction cache are 32KB in capacity, 8 way set associative, with 64 byte cache line. L2 cache is 256KB, also 8 way set associative. The last level cache is 6MB. Ram size is 16GB. The operating system I chose Fedora 17, with the linux kernel version 3.3.4. Compiler version is gcc-4.7.0. I also used a performance tool

called *perf*, which operates on hardware performance counter related interfaces.

3. MEASURING TLB SIZE

TLB is one greatest optimization that makes virtual memory to work fast. Without TLB, each memory reference will have to do more less than twice memory reference, for actual physical pages associated with, and for the real address CPU wants to visit. In this section, I try to measure the TLB size on my machine.

3.1 Methodology

To measure the size of TLB, I choose to observe the timing difference on referencing memory. If the TLB hits, then the reference should be faster than TLB misses cases. The goal could be achieved if we carefully construct the memory visiting sequence, then we can find out the thrashing behavior in timing when come to the point TLB begins to miss. Specifically, if not mentioned, TLB later all mentions to data TLB.

3.1.1 Complications

But correctly measuring TLB size, is not an easy task due to following complications:

- Hardware cache can interference heavily during the visiting process. To correctly measure the event, one needs to distinguish the cache miss event and TLB miss event. Unfortunately, these two events are usually comparable in missing penalty, and make things complex. Even worse is that caches are usually physical associated, and the addresses we can provide at userspace are virtual addresses. For the set associative caches, if their associative sets number are larger than the number of cache lines per page, then we can not fully control the cache. Due to the space limitation, if not necessary, I will omit derivations of the non-experimental conclusions.

- Hardware can have mechanism that ruins the assumptions about sequential programming model, like out-of-order instruction retiring, multiple processing units, and hardware prefetching and so on.

- Modern CPU can have multiple level of TLB. On my platform, there is two L1 data TLB for different page sizes, one L1 instruction TLB, and one shared L2 TLB. We need to let level 1 TLB to fail before we can fail level 2 TLB.

- Difficulties in generating the correct benchmark. The overhead in language constructs, operating systems interactions, and the compiler's aggressive optimizations can all become obstacles to obtain the correct results.

3.1.2 Strategy

To solve the above complications, I carefully construct my sequence to walk on memory. First thing is to design a pattern to maximize cache hit. One observation that helps is: level N TLB (N=1,2) usually has less entries than the total cache line number in level N cache, but its total size is larger than the corresponding cache. Due to this fact, we can force level N cache to hit, while level N TLB to miss.

This goal can be achieved by visiting exactly one cache line inside each page. We first allocate sufficient pages, and gradually increase the number of pages to walk on. The phase change point in timing would be approximation size of TLB size. To make cache hit, the stride we use to walk on pages is sum of page size and cache line size. This ensures we visit a different set of cache lines in the next page and will maximize the cache utilization. When it goes off the page boundary, then just rewind to the start of next page and keep on this procedure.

This strategy will work for both L1 and L2 TLB. Actually, by maximize the cache utilization, at the point of L2-TLB miss, we will observe different timing behaviors. Let's define C_i to be level i cache hit, T_i to be level i TLB hit, and C_m, T_m to be cache and TLB miss correspondingly. Then we should observe C_1T_1, C_1T_2, C_1T_m, or C_1T_1, C_1T_2, C_2T_2, C_2T_m. There should not have C_2T_1, or C_mT_2 behavior, so the timing curve would be monotonically non-decreasing as we enlarge the walking page size.

Since memory references are tiny things to measure, we repeat the memory walk many times and measure the average.

3.2 Implementation

The implementation is quite tricky. Firstly I manually unrolled innermost loops of all walking routines, and ensuring they have the least number of instructions. This can avoid loop overhead to small loops as well as improve the instruction cache to hit. Moreover, all the operands are aligned to same size to avoid size extending. Before walking, the memory should be warmed several times and evict out dirty data. The last thing then come with the compiler optimization. In one hand, we can rely on the optimization to reduce the uncessary memory visit and computation, rather than manually coding assembly (actually I did this for some very small walk kernels), but on the other hand we should add some fake "use" to avoid our code being optimzed out. Also, inline optimization should not be used abusively. Large chunk of inlined code can hurt the instruction cache, which is unnecessary overhead.

To automatically the experiment, I also write code to measure cache size and associative sets. I reference the paper here [2]. To avoid the problem of physical not continuous, I resort to huge pages, and then things become a lot more easier, just capacity probe suffies to find out how many cache lines and how many associative sets are in each cache.

3.3 Experiment Results

3.4 Discussion

The first time I used register-base-scale addressing to implement the walk kernel. However, this is not quite good choice. The reason is even CPU miss on a TLB visit, it can still issue following instructions because there are no data dependency, and the overhead is ammortized so that phase change is not clear, even if I have observed high TLB miss rate and cache hit rate.

The next method I tried is using linked list. In each cache line I encode the next address to visit and make it a linked list. In this way, the phase change become apparent when I increase the probing page count by factor 2. However, if I increase probing count 1 by 1, then the normalized time for one memory reference grows linearly before a big phase

change. This does not quite make sense since their TLB hit and cache hit rate is nearly the same. I also checked branch prediction misses, instruction cache/TLB, and all looked normal. The only abnormality is that there are many bubbles in pipeline. This is caused by linked list style visiting, where data dependencies are heavy for each memory instruction. Then I doubt it might be overhead of other instructions, but it is not the case since fixing total memory reference numbers still behave like this. In that case, the walk for smaller page number should have more iterations, leaving more intervals between the kernel loop of walk, thus should introduce more extra instructions. However, it goes in reverse direction. This is still not addressed yet and I am doing finer granularity performance analysis on that.

4. REFERENCES

[1] C10K problem. http://www.kegel.com/c10k.html.

[2] K. Yotov, K. Pingali, and P. Stodghill. Automatic measurement of memory hierarchy parameters. In *ACM SIGMETRICS Performance Evaluation Review*, volume 33, pages 181–192. ACM, 2005.