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 [2]. 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. The result is not exactly what I expected, and I will try to explain that in the discussion section 3.4

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 hit rate of instruction cache. Additionally, all 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 automate the experiment, I also write code to measure cache size and associative sets. I reference the paper here [6]. 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. Due to the space limitation, I don't show the detail here. Further information should be found in [**?**].

3.3 Experiment Results

Figure [**?**] shows the timing result of TLB probing. The first part of the result looks quite reasonable, clear boundary and much stable than the second half after I have probed about 512 pages.

To verify the case, I use *perf* to collect the hardware data. The detail command and event will be disclosed in appendix **??**. It can also be found in **??**.

The collected result illustrates the behaviors: the thrashing of later curve is caused mainly by L2 cache miss. The reason is pages may be non-continuous in physical. I will discuss the in section 3.4.

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 exhibits the above behavior. The reason to blame for is the potential of non-continuous physical page. On my platform, there will be 512 cache sets on L2 cache, which means, if using 4KB page, 3 higher bit used to reference the cache sets will not be controllable by users. Operating system can even enforce the cache separation by not allocating physical pages with certain set index to users. The problem here is, although in theory we could probe more than $4096(8 - way512sets)$ pages before L2 cache misses, in practice only $512(8 - wayand6 - bitsets)$ can be ensured to get visited without miss.

An interesting observation is the chance for 8 pages having same high 3-bit index is rare. So we can adjust the walking algorithm, rather than fixing the stride and period of walking, we probe the good page sets, i.e. those uniformly distributed over the bits [15:12], this can be observed from timing result again. In such way, when we found enough pages, we can go on our walk by linking these pages together. I haven't got time to derive the actual expectation, but it should be much larger than the current turning point 512.

In my experiments, I didn't measure the size of huge page TLB. One reason is it is developing toward unified TLB, which measuring the small page TLB is enough. On the other hand, the whole thing required to measure huge TLB would be similar as above process, and much easier since L2 cache sets will all be embodied inside the continuous huge pages.

4. MEMORY ALLOCATOR

Memory allocator is a major interface that operating systems exposed to the users. On Linux, users request memory throught the system call *brk* or use the memory mapped file through *mmap*. This can also be unified by standard library interface *malloc*. The operating system, will ensure that if the requests are granted, then visiting these address would not cause protection errors and thus can be used by user applications. However, this does not mean operating system will allocate the physical page, say, setting up the virtual page to physical mapping immediately. Rather than fit user's request once and for all, operating systems may employ a lazy allocation strategy. In this configuration, only when a page fault occurs, will the operating system check the need to allocate physical memory. In this section, we verify two things. First is that whether operating system uses on-demand allocation, and second is when will operating system zero out the pages for security concerns.

4.1 Methodology

The strategy I used is quite simple. To determine whether operating systems allocate pages on demand, we just allocate a large memory area, and visit the performance we walk upon first time. It should be quite different than normal walking. To further distinguish what operating systems are doing, I will measure the time with and without allocating system call, here I used *mmap*, as well as the normal walk time.

To check when does operating system zero out the pages, if the allocation is not done at the system call time, then it's only possible for operating system either to 1.) choose a known zero page 2.) zero out page on demand. In order to exclude or include the first approach, I measured the time of allocation both when free memory is abundant and is scarce. I try to make sure that operating system will not have many free zero pages to borrow, then I could figure out whether there will be extra cost during allocation. One problem may still remain is that kernel can zero out pages when pages are unmapped. So I also measure the normalized time used to unmap a clean page and a dirty page.

4.2 Experiments

I measured several parts of the overhead:

- total overhead (tagged as "map-full"). Including mapping, memory walking and unmapping time.

- first time walk overhead. This will ensure the pages be allocated.

- unmapping overhead, with/without reset routine. Measuring this part is for the sake to explore if operating systems can do zero out at reclaim time.

- regular time walk.

?? These experiments are done under memory idle period, and memory pressure period. To compare the zero-out strategy, we also measured the allocation time when memory is scarse resource. In this scenario, the free memory I leave to operating system is less than 1.5GB, and the allocation unit I used is 1GB, to make sure few free pages are in the system, and it must zero out pages at the time of granting or revoking them to users.

From figure we can easily find out the allocation is on demand, since regular walking is much too cheap comparing with the first time memory walk after mapping a set of memory. And also the performance retains regardless of the memory pressure. According to that fact, it is only reasonable to zero out pages on allocation and deallocation time. But the measurement of unmapping time did not show much information, since the unmapping time is sufficiently large that we can not distinguish that if it actually does page zero out.

4.3 Discussion

One thing hard to illustrate is whether operating system will 'smartly' allocate some pages for small requests. For example, when user requests for only 1 page, then allocate the page immediately to avoid a page fault can be somewhat a good choice. But this period is too small and will hardly get measured correctly just by using wall clock. However, if the operating system can provide a page fault counter for each process, then it is still possible to measure such behavior. Due to both the time limit and space limit, I didn't implement this as part of the benchmark.

5. HUGE PAGES

Huge pages are an optimization with the demand of high performance. On latest Intel x64-64 platform, the page size can be even 1GB. This provides good chance for those applications suffer from virtual memory abstraction to tune their performance, like the databases, or the host mode virtual machine monitors. The benefits of huge pages have been long mentioned, but viewing from the virtual memory system, there could be very time consuming to prepare these huge pages, since huge page means continous in physical, and it will be very difficult to find out large chunks of physical continous spaces if the system is at busy time. In this section, I try to measure the cost to use huge pages.

5.1 Methodology

The experiments will contain two part, the first is the time to prepare huge pages. This would be carried out after a busy period to observe the effect. I choose to measure this after each Interval of section 6. Then I try to illustrate the different timing behavior when using large pages and small pages, by allocating the same amount of memory, and do several kinds of walking on that.

Based on the experiments in section 4, it's obvious that if we just reference a few bytes after requesting a huge page, the cost would be larger than using a set of small pages. But this is meaningless in real applications. So I use both sequential walking, and random access to test the visiting time. Also, to justify the possible impact of allocation, I also make walking happen several times: in real applications, it is not strange to walk through a data area for several times, and the first time page faults can be amortized in later walk.

5.2 Experiments

Figure **??** shows the cost of using small and huge pages from 2MB memory occupation to 128MB. The stride we used to walk on the pages are from 4KB to 2MB, with multiple iterations in each walk. In the figures, huge pages performs bad when iteration is small, and stride is relative large. With the iteration increases, its average performance is more improved.

Random walking is more interesting. Small pages still perform better at the beginning, but normalized difference is smaller. The result again illustrates the benefit of on demand allocation: sequential walking will force all pages being brought into the systems, and that would may not be a must.

5.3 Discussion

Huge pages can be useful. But strange enough, in Linux normal users can get huge page through *mmap* interface using anonymous mapping, without causing protection problem. However, the similar access through *shmget* interface will not work, unless super user privilege is granted. I didn't see difference between this two approaches, hopefully it is not a security hole.

6. SHARED CODE PAGE

Shared code pages is an important optimization to the virtual memory. Previously I have shown the benefits from demand allocation as an optimization, now I will go on exploring the shared library, or shared object. Shared code usually appear as shared library or dynamic linking library in the system. With shared library, code size are largely reduced.

Moreover, the shared library is usually place independent and can be dynamically loaded by the applications. This enables great flexibility in design. For instance, apache [1] uses dynamic libraries to implement its modules, which can be loaded on demand and built separately. In this section I compare the typical applications' code size by linked them statically and dynamically. Performance is nearly the same for no matter static or dynamic linked program. I will discuss this in 7.

6.1 Methodology

It's quite straightforward to measure the code size. I compiled LLVM [5] and Apache [1] to see the code size difference for shared library version and statically linked version.

I also tried to use an apache server to exam the performance issue of dynamic linked library. But finally I didn't, it is because the *fork* system call will make parent-children sharing the same code region, then the difference in code size will be small. So I didn't do the experiment about this part.

6.2 Experiments

The code size from LLVM is shown in Table **??**. Obviously the statically linked programs are much larger in code size than the counterpart in dynamically linked program. Increasing in code size is usually not acceptable, so dynamic library naturally comes to our sight.

6.3 Discussion

Originally I would like to compile firefox [4] to see the code size change. But after some attempts I found they do not support static linking in their build scripts so I moved to LLVM.

Another fact is measuring performance difference between static programs and dynamic problems are quite difficult. The system call *fork* provides a chance to share the read only code. Thus although programs are larger in size, it may not occupy rally that much memory. So I didn't continue my experiments on this.

Although shared objects are extensively used in current systems, it is not correct to say that static library is no longer required. Some problems raised only in shared objects, like library dependency, and the binary compatibility [3]. Also, in scenario requires extreme good performance, like high performance scientific computing, static library would be their first choice. The last thing is based on COW and shared memory, it is still possible to enable runtime sharing. *fork* will just copy itself, and its code page can be shared then.

7. CONCLUSION

In this work I mainly finished four parts of experiments, to explore the details of virtual memory subsystem. Total code lines are quite small, 2000 lines C and inline assembly in total with comment. The code, the experiment raw data, and also the tex source, could be downloaded from [**?**]. The results and conclusions from each part basically confirms my knowledge about how things are working. There is still some parts that can not be explained well, and I will seek to find the answer in the future.

8. ACKNOWLEDGEMENT

of out of order execution in measuring TLB size, and it does help to get the final result.

9. REFERENCES

[1] Apache Web Server. http://httpd.apache.org.
[2] C10K problem. http://www.kegel.com/c10k.html.
[3] Dll hell problem.
    http://en.wikipedia.org/wiki/DLL_Hell.
[4] Firefox. www.mozilla.org/en-US/firefox/.
[5] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.
[6] 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.