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. 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 in some case.

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 of 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 will the page be actually allocated? 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, I choose Linux as my 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 L2 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 I 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 decide to use the system call *gettimeofday*. The major idea behind this is that even if I have accurate enough timer, it's still easy to get disturbed by system noises. To avoid these noises, it's best to magnify the running scale, and also should run multiple times. In this way, *gettimeofday* is already enough to measure the system behavior. 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. In my experiments, this interface works quite well.

2.2 Hardware and Software Environment

The machine I used for testing is a x86-64 machine. The processor is Intel-i5 2500K 3.3GHZ, family of Sandy Bridge. 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* [6], which operates on hardware performance counter related interfaces, to verify some of my result and conclusions.

3. MEASURING TLB SIZE

TLB is one greatest optimization that makes virtual memory to work faster. Without TLB, each memory reference will have to do no less than two 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 chose 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

Correctly measuring TLB size is not an easy task due to following complications:

- Hardware cache can interference heavily during the walking process. To be accurate, 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. And this in fact causes great trouble when I came to measure the L2 TLB size.
- 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 miss before we can measure 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 should 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 [9]. To avoid the problem of physically 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 won't show the detail here. Further information could be found in [7].

3.3 Experiment Results

Figure 1 shows the timing result of TLB probing. The first half of the result looks quite reasonable, clear boundary and more 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 *perf* event combined with umask I used are: r1008,dTLB,r01D1,r02D1,r04D1,r20D1. They can be found in [5], section about Sandy Bridge i5-2xxx serial.

From the figure 3 and figure 2, I can confirm before the page number grows to 512, things are correct, and the L1

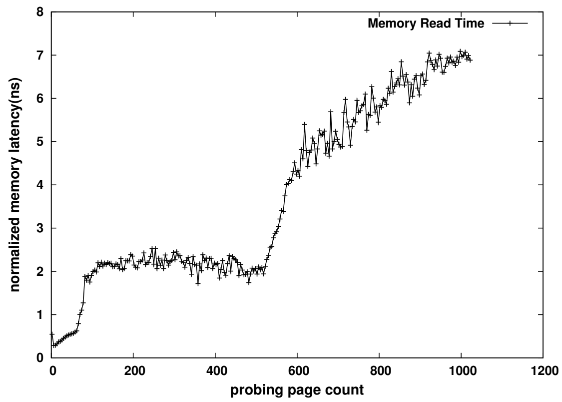

Figure 1: Memory walk time on different number of pages. The stride size I use is 4160 byte, the sum of page size and cache line size, so the memory reference will happen on each page exactly once.

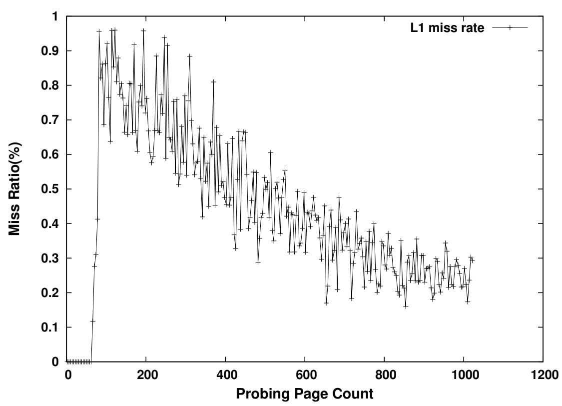

Figure 2: L1 TLB miss rate during memory walk, generated by *perf*. The L2 TLB data, however, nearly have no missing all the time no matter how much I used the memory.

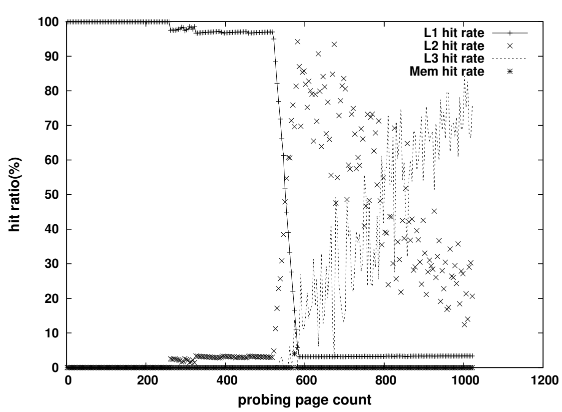

Figure 3: Multiple level of caches hit ratio. The hit rate for each level is exclusive, i.e. if L2 cache is hit, then L1, L3 miss is not hit. Memory hit simply means memory reference, and this seldom happens

TLB size would be around 64 entries. However, after I reached about 512 pages, it becomes quite unstable. The reason can be found mainly in figure 3, the L2 cache begins to miss heavily. The reason is pages could be non-continuous in physical. I will discuss a possible workaround to this problem in section 3.4.

Another annoying fact is that *perf* didn't show me the reasonable L2 TLB miss number. But from the change of L1 TLB miss rate, which indicates L1 TLB miss and L2 TLB hit, it's almost reasonable to assume the L2 TLB is also missing. The reason is quite simple: with my walking sequence, if L2 TLB is not missing, then L1 TLB either keeps missing on every access, or don't miss at all. So I don't even need a hardware counter to confirm the correctness, as long as the L1 TLB data is not also faulty.

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 amortized 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 was linked list. Within each cache line I encode the next address to visit and make it a linked list. In this way, the phase change becomes more outstanding when I increase the probing page count by factor 2. However, if I increase probing count by adding 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-way 512 sets) pages before L2 cache misses, in practice only 512 (8-way and 6-bit sets) can ensure visiting without miss.

But on the other hand, assume operating systems are not restricting users' control on cache lines, then an interesting observation may help workaround this problem. That is, the chances for 8 pages having same high 3-bit index are rare. So we can change the walking algorithm to:

```
probe(npage, high_bit, low_bit)
  page_set[low_bit]
  again:
    i <- rand() % low_bit;
    page <- get_random_page
    if test_no_l2_miss(page_set[i], page)
      page_set[i] += page;
    if pages count in page_set < npage
      goto again or exit and report failure
  memory_walk(page_set)
```

Testing whether can add a new page is to test whether walking on same cache line in that set can cause L2 cache miss. When we found enough pages, we can link these pages together and walk to check timing. I haven't got time to derive the actual expectation, but it should be much larger than the current turning point 512. I have implemented a prototype for this algorithm, but not yet tested due to time limit.

In my experiments, I didn't measure the size of huge page TLB. One reason is it is gradually become unified TLB, and measures the small page TLB should be enough in future. 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 are all embodied inside the continuous huge pages.

4. MEMORY ALLOCATOR

Memory allocator is a major interface that operating systems exposed. On Linux, users request memory throught the system call *brk* or the memory mapped file through *mmap*. Operating systems will ensure that if the requests are granted, then visiting these address would not cause protection errors. However, this does not mean operating system will allocate the physical page, say, setting up the virtual to physical mapping immediately. Rather than to fit user's request once and for all, operating systems may employ a lazy strategy. In this configuration, only when a page fault occurs, will the operating system check the need to allocate physical memory. In this section, I try to verify three 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. At last I will check the benefit of on-demand allocation.

4.1 Methodology

To determine whether operating systems allocate pages on demand, I just allocate a large memory area, and measure the time we walk on it for first time. It should be quite different from 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 operating systems then can only: 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 with/without memory pressure. To prevent operating systems to have many free pages, I will allocate a big memory pool and grab most of the memory. Then I come to check if the timing in allocation can have some change. 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

4.2.1 Allocation Cost

To address questions we mentioned, I measured several parts of the overhead in figure 4:

- 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.

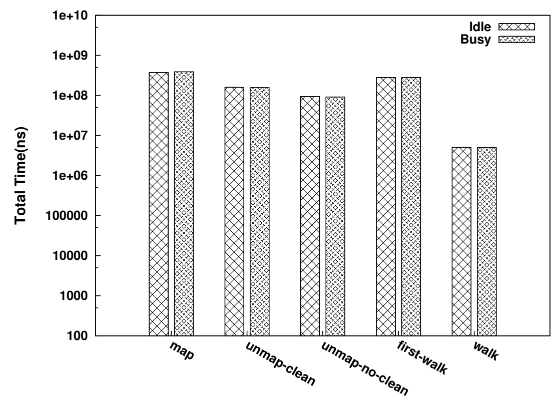

Figure 4: The different part of allocation time. *map* means the entire time including the mapping, walking, and unmapping. *unmap* has two options, to zero out the page (suffixed with "-clean"), and don't do extra things (suffixed with "-no-clean"). The first walk means the duration of the first time we visit the allocated space, and walk means the time to do memory walk after we have touched all the pages several times.

- regular memory walk after all the pages are assured to exist in memory(I monitored the swap space used, and it's not grown at any experiment time).

To compare the zero-out strategy, we also measured the allocation time when memory is sparse 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 4 we can find out the allocation is on demand, since regular walk is much cheaper comparing to first time walk right after mmap returns. And also the performance retains regardless of the memory pressure. According to that fact, it is only possible 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 tell if it actually does page zero out or not.

4.2.2 Benefit from On-Demand Allocation

The on demand allocation is an optmization for virtual memory. To see its effect, I compared the cost of random walking and sequential walking on the 128MB newly allocated memory. For space limitation, I only present the result on 4B stride (or 32M random read) as in figure 5. Further results are uploaded to [7], it is consistent with my conclusions here.

Figure 5 exhibits interesting result: random walk runs faster with small pages, but slower with large page. In the former case, not bring in unnecessary pages saves time, while in the later one there are only a few pages to bring in, and poor locality slows it down.

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

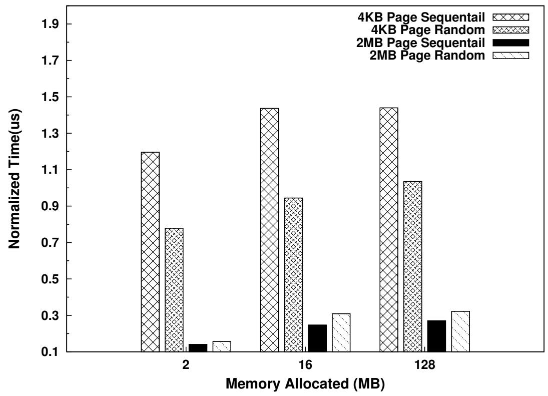

Figure 5: The time of doing random walk and sequential walk through the newly allocated 128MB memory. For small page space, random walk would not bring in the page not needed, thus performs better than sequential walk. But for large page space, random walk is slower.

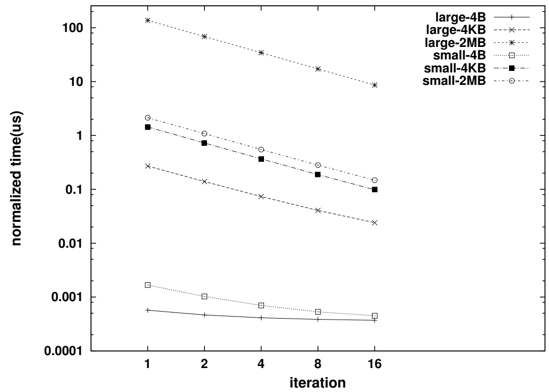

Figure 6: Memory walk time on 128MB mapped memory. I use different stride, and iterations for each walk, and measure the average time cost. The larger the stride size is, the worse performance I will get by using huge pages. Due to space limitation, I didn't draw more stride result. Actually the small page start to beat huge ones from the point of 32KB stride.

the page immediately to avoid a page fault can be somewhat a good choice. This period is too small to measure 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.

5. HUGE PAGES

Huge pages are an optimization for performance. On latest Intel x86-64 platform, the page size can be even 1GB. This provides good chance for applications 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, it can be cost, like to prepare these huge pages will rearrange memory so it is physical continuous. Moreover, when page fault happens, it's much heavier than small page faults. In this section, I try to show the cost of huge pages.

5.1 Methodology

The experiments focuses on the cost of allocating huge pages. Preparation time can be quite long, but it is a one time cost, and will be much less possible to become a bottleneck. So I will care more about what happened during using the huge pages. What I did was allocating same amount of memory using both large and small pages, and comparing the time elapsed with both sequential walking, and random accessing. Also, I will increase the walking iteration, so that I can find out what frequency of use can make huge pages competitive in performance , even it has much larger overhead to be set up and zeroed out.

5.2 Experiments

Figure 6 shows the cost of walking on newly allocated small and huge pages, total memory size is 128MB. The stride we used to walk on the pages are from 4B to 2MB, with multiple iterations in each walk. It is quite clear that huge pages performs bad when iteration number is small, and stride is large. With the iteration increasing, its average performance becomes better.

Random walking shows similar trend as in figure 7. Small pages still perform better at the beginning, but normalized difference is smaller. The result again illustrates the benefit of on demand allocation: setting up the mapping and zeroing out pages at the very beginning is not a must, since the applications may not reference them all.

5.3 Discussion

Although there are some inconveniences, huge pages are useful, especially when used by VMM. If no huge pages are used, then on a 4 level paging system, a page fault can require up to 20 times memory reference. With the huge pages, this can be reduced to minimum. The problem for the huge page is that it must be configured and requested explicitly, thus may lead to memory pressure. Another thing is the recognition of the cost of huge pages: for large data not frequently referenced, using large pages may not get benefits.

During experimentation, I found a strange thing:in Linux normal users can get huge page through *mmap* interface using anonymous mapping, without causing protection errors. However, the similar access through *shmget* interface will not work, unless super user privilege is granted. I didn't see any differences between these 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 code sharing. Shared code usually appears as a shared library or say dynamic linking library. With shared library, code size are largely reduced. Moreover, shared libraries are 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 will focus on comparing the typical applications' code size by linked them statically and dynamically.

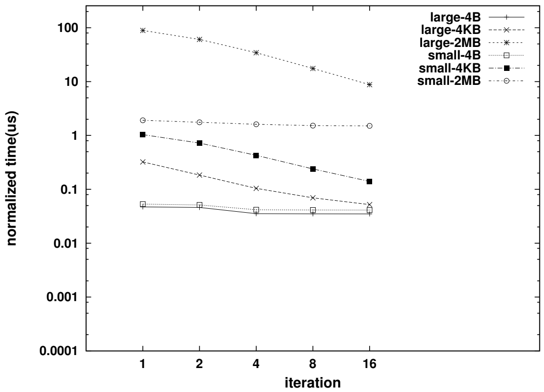

Figure 7: Random walking is similar with sequential walking, when in small stride size the huge pages perform better. Here the stride is just used to determine the amount of memory reference, rather than walk pattern. I use stride to label it for comparison with sequential walk. One thing changes in random walk is that for 4B stride, both of the two configurations perform worse than sequential walk. The reason may be related with locality, but it is not what we address here.

6.1 Methodology

It's quite straightforward to measure the code size. I compiled LLVM [8] 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 with prefork mechanism, to compare the performance of shared code and static code. But finally I gave up, it is because the *fork* system call on Linux will make parent-children sharing the same code region, then the difference in code size will not be small. It is possible to synthesis some benchmark, but it is not quite persuading, since it must reflect the pattern how real applications will use shared library Due to these considerations, I didn't finish the experiments on this.

6.2 Experiments

name	shared	static	name	shared	static
httpd	1.6M	7.3M	bugpoint	3.5M	75M
clang	283M	459M	clang-check	135M	152M
clang-tblgen	6.0M	6.0M	llc	908K	146M
lli	844K	103M	llvm-ar	484K	14M
llvm-as	248K	17M	bcanalyzer	564K	2.6M
llvm-config	2.1M	2.1M	llvm-cov	132K	2.2M
llvm-diff	1.3M	15M	llvm-dis	384K	14M
opt	2.0M	73M	llvm-extract	420K	23M
llvm-link	344K	30M	llvm-mc	916K	20M
llvm-nm	480K	15M	objdump	1.2M	22M
llvm-prof	824K	16M	llvm-ranlib	280K	14M
llvm-readobj	424K	3.8M	llvm-rtdyld	268K	4.5M
llvm-size	396K	3.8M	llvm-stress	604K	14M
llvm-tblgen	22M	22M	macho-dump	188K	2.2M

Table 1: Code size comparison for LLVM code suite and Apache Server using static libraries and shared libraries. The unit is byte.

The code sizes of LLVM and Apache are shown in Table 1. Obviously the statically linked programs are much larger in code size than the corresponding dynamically linked program. Increasing in code size is usually not acceptable to industrial use, making deployment more difficult. When it comes to concern of code size, then shared library serves well for this purpose.

6.3 Discussion

Originally I tried to compile firefox [4] to see the code size change. But after some attempts I found they had abandoned support of static linking long ago in their build scripts, so I moved to LLVM. The reason size is such huge is because the build is a default debug build with debug information. However, even the optimized version still occupies 1GB on disk.

Another fact is measuring performance difference between static programs and dynamic problems are not easy. As mentioned before, the system call *fork* provides a chance to share the read only code. Thus although programs are larger in size, it may not occupy too much memory for code regions. To compare the performance, a set of independent programs are needed.

On the other hand, Although shared objects are extensively used in current systems, it is not correct to say that static library is no longer required. Some problems raise 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.

7. CONCLUSION

In this work I mainly finished four parts of experiments, to explore the details of virtual memory subsystem. I wrote about 2000 lines C and inline assembly with comment. The code, the experiment raw data, and also the tex source of this paper, could be downloaded from [7]. The results and conclusions from each part basically confirms my knowledge about how things are working. There are some parts that can not be explained well now, and I will seek to find the answers in the future.

8. 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/.

[4] Firefox. http://www.mozilla.org/en-US/firefox/.

[5] Intel Software Developer Manual, Volume-3. http://www.intel.com/.

[6] Perf on Linux. https://perf.wiki.kernel.org/.

[7] Source code of this project and paper. git://github.com/jerryz920/vm_measure.git.

[8] 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.

[9] 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.