How large is the TLB?

Johan Montelius

November 2, 2020

Introduction

The Translation Lookaside Buffer, TLB, is a cache of page table entries that are used in virtual to physical address translation. Since the cache is of finite size we should be able to detect a difference in the time it takes to perform a memory operation depending on if we find the page table entry in the cache or not.

We will not have access to any operations that will explicitly control the TLB but we can access pages in a way that forces the CPU to have more or less page hits. We will set up a number of virtual pages in memory and if we access them linearly we should be able to see some changes when we have more pages than the TLB can handle.

1 Measuring time

To explore the difference between hitting the TLB and having to find the page table entry in memory, we will write a small benchmark program that accesses more and more pages to see when the number is to large to fit in the TLB. In order to do this we first of all need a way to measure time.

1.1 process or wall time

First of all we need to decide if we want to measure the *process time* or the *wall time*. The process time is the time our process is scheduled for execution whereas the wall time also includes the time the process is suspended. The process is suspended if the operating system has to schedule another process or if our process is waiting for something, for example I/O. In this experiment we will do fine with the process time.

Look up the manual entry for the procedure `clock()`, it will give us a time stamp that we can use to calculate the elapsed process time. There is then a macro, `CLOCKS_PER_SEC`, that we use to translate the timestamp into seconds. This is the time we would normally use if we are not doing any I/O or other things that we also want to include in our measurements.

1.2 the test rig

We want to measure how long time it takes to access a page and see if the time increases when we increase the number of pages. We will set a maximum

number of pages (we will start with 16) and then measure the time it takes to access one page, two pages, three pages etc. To get more reliable figures we need to do a multiple of references in each test.In order to make it easier to compare the numbers we will make the same number of references in each test. We therefor set up a loop in a loop where the inner loop will access the number of pages in the test and the outer loop is adjusted to make the total number or references the same.

The increment of the sum is only there to have something to measure. We will later remove it and replace it what we really want to measure.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define PAGES (16)
#define REFS (1024*1024)

int main(int argc, char *argv[]) {

  clock_t c_start, c_stop;

  printf("#pages\t proc\t sum\n");

  for(int pages = 1; pages <= PAGES; pages+=1) {

    int loops = REFS / pages;

    c_start = clock();

    long sum = 0;

    for(int l = 0; l < loops; l++) {
      for(int p = 0; p < pages; p++) {
        /* dummy operation */
        sum++;
      }
    }

    c_stop = clock();

    {
      double proc;

      proc = ((double)(c_stop - c_start))/CLOCKS_PER_SEC;
```

```
        printf("%d\t %.6f\t %ld\n", pages, proc, sum);
      }
    }
    return 0;
}
```

In the example above I've used a million references but this might differ from machine to machine. Experiment with less or more references to see how many you would need to get measurements that do not fluctuate to much. In the `printf()` statement you can change the sequence `%.6f\t` to write more or less decimals. How many significant figures is it reasonable to show?

1.3 don't let i go

Do the numbers look right? Is the time to do the first and second test slightly longer than the other tests? Is this just a glitch in the system or is it real? Hmm, there is something going on here? What is happening?

When you see strange numbers try to figure out what the problem could be. In this case the numbers show that it takes more time to run the outer loop one million times, where we inside the loop only loops once, compared to making the outer loop 500.000 times and the inner loop twice. If you think about it is what we would expect since there is an overhead in preparing for the inner loop and jumping out of the loop. The more time we spend inside the loop the less the overhead will make a difference.

1.4 optimize but not too much

We will in the end try to estimate what it costs to do one memory reference. The problem is that test rig we now have set up actually makes tons of memory references by itself. The `sum`, `l` and `p` are probably stored on the stack so we will make multiple references to the stack in each test. We could try to reduce the memory overhead by turning on some optimizations. Try the following if you are using gcc.

```
> gcc -o tlb -O tlb.c
```

This will turn on the first level of optimizations. I think that you have a substantial improvement. You can verify what happened by looking at the generated assembly. Using the `-S` flag we can ask the compiler to generate an assembler file, try the following:

```
> gcc -o tlb-opt.s -S -O tlb.c
 :
```

```
> gcc -o tlb-reg.s -S tlb.c
```

Now compare the two files and see how they differ. Look for the call to `clock` where you will find the loop. My guess is that the regular version will be filled with references like `-16(%rbp)` i.e. a reference using the base stack pointer. In the optimized version we have references to different registers such as `%ebx` and `%edx`.

How fast can we do this loop? Try using `-O2` the second level of optimizations. What is happening now, is this really true? One should be careful when doing measurements, sometimes the quickest way to do something is not to do it at all.

2 The benchmark

So now for the actual benchmark. We're going to allocate a huge array that we will access in a linear fashion. The question is if it takes more time to access 4 pages compared to 16, 32 or even more pages.

2.1 a first try

We first define the page size and we assume that we have a page size of 64 Bytes. This is of course not the case since we know that the page size of our machine is probably 4 KiByte but we will use it as a first step to see what it is that we are measuring.

```
#define PAGESIZE 64
```

Then we allocate a huge array and why not call it `memory`. Note that we cast the `PAGESIZE` to a long integer i order for the multiplication to work even if we start to use a total memory that is larger than 4GiByte (and we will in the end). We also run through all pages and write to them just to force the operating system to actually allocate the pages and set up the page table entries.

```
char *memory = malloc((long)PAGESIZE * PAGES);

for(int p = 0; p < PAGES; p++) {
  long ref = (long)p * PAGESIZE;
  /* force the page to be allocated */
  memory[ref] += 1;
}
```

We also add some nice printout so that we can look at our figures later and see what parameters we used. The initialization and printout is added before the code that starts the sequence of test.

```
printf("#TLB experiment\n");
printf("#  page size = %d bytes\n", (PAGESIZE));
printf("#  max pages = %d\n", (PAGES));
printf("#  total number or references = %d Mi\n", (REFS/(1024*1024)));
printf("#pages\t proc\t sum\n");
```

Then it is time to do the benchmark and we simply replace the dummy (`sum++`) operation with a page reference that we use to increment the sum.

```
/* this will replace our dummy operation */
long ref = (long)p * PAGESIZE;
sum += memory[ref];
// sum++;
```

If you have everything in place you should see something like the printout below. You can compare these number to the dummy that we performed before (I've used 10 million references).

```
#TLB experiment
#  page size = 64 bytes
#  max number of pages = 16
#  total number or references = 10 Mi
#  time for all references in sec  (1000000)
#pages  proc  sum
1  0.040  10485760
2  0.060  10485760
3  0.043  10485759
4  0.036  10485760
  :
  :
```

There is a small difference in accessing a memory references compared to just doing a dummy addition but the difference is not very large. If you try using up to 64 pages you might notice something strange and predictable behaviour. We have sorted out why the initial tests show slightly higher values but there might also be other test that show higher values. The difference likely has to do with the the size and structure of the first level data caches.

The initial preparations of the benchmark are important since it tells us what order of magnitude the differences have to be in order to detect them. If TLB misses only increase the execution time by 10 percent then we will have a hard time separating what is actually an effect of TLB misses and what is an effect data caches etc.

2.2 running some tests

Let's first increase the page size to something more reasonable such as 4 KByte. This is probably the size that the machine that you're running on uses but if you're not running on a x86 architecture you should look it up.

```
#define PAGESIZE (4*1024)
```

First see if there is any difference using 16, 32 or why not 64 pages. Hmm, switch back to a page size of 64 bytes and then try 64 pages again. How large is the memory that we use when we use 64 pages of size 64 byte each? How many real pages are used then?

What happens if we step up to 512 pages or why not 4096? Since we're probably not that interested in what happens between 2378 pages and 2379 we can change the benchmark to try double as many pages in each iteration.

```
for(int pages = 4; pages <= PAGES; pages *=2) {
```

Ouch, that hurts. There is definitely a penalty that we take when using more pages.

2.3 turn it into a graph

So you now have some nice printouts of the time it takes to reference a page depending on the number of pages that we use. As you see there is definitely a penalty as we use more pages. We can create a nice plot of the values using a tool called `gnuplot`.

First we save the output in a file called `tlb.dat`. This is easily done using the redirection operator of the shell.

```
$ gcc -o tlb  -O tlb.c
$ ./tlb > tlb.dat
```

You can run `gnuplot` in interactive mode (and you should learn how to do so to quickly generate a graph) but it's better to write a small script that generates the plot. Write the following in a file `tlb.p`.

```
set terminal png
set output "tlb.png"

set title "TLB benchmark, 4 KiByte pages, 10 Gi operations"

set key right center

set xlabel "number of pages"

set ylabel "time in s"
```

```
# use log scale if we use doubling of number of pages
set logscale x 2

plot "tlb.dat" u 1:2 w linespoints title "page refs"
```

Now run gnuplot from the command line and let it execute the script.

```
> gnuplot tlb.p
```

You should now find a file called `tlb.png` with a nice graph. Experiment with gnuplot and add more lines in the same graph. If you generate data for the dummy test, using pages of 64 bytes and 4K bytes you can print them in one graph using the following directive.

```
plot "tlb4K.dat" u 1:2 w linespoints  title "page size 4K bytes", \
     "tlb64.dat" u 1:2 w linespoints  title "page size 64 bytes", \
     "dummy.dat" u 1:2 w linespoints  title "dummy"
```