OPERATING SYSTEMS PROJECT 2

MEASUREMENTS FOR COST OF ACCESSING TLB

BY

SAI KIRAN REDDY MALIKIREDDY(U63053381)

KIRANMAYI KARRI(U81100483)

Table of Contents

1. INTRODUCTION

This report gives a brief description of how we measure the cost and size of a TLB. These measurements are implemented using c programming and tested on various system architectures such as Intel core I7(C4 lab machine), AMD Opteron etc,. The detailed description of each execution and results are reported in further sections.

2. SYSTEM CONFIGURATION

Below are the system configurations of the machines on which we tested.

2.1 HARWARE ENVIRONMENT

Configurations	AMD Opteron	C4 Lab	Intel Xeon(R)	Intel Core I7
Processor	AMD Opteron	Intel(R) Core(TM) i7	Intel Xeon(R)	Intel(R) Core(TM) i7
Model	AMD Opteron QEMU	Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz	Intel Xeon(R)-E7310 @ 1.60GHz	Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz
Cycle Time	2399.998MHz	3392.20MHz	1601.61MHz	3520.33MHz
RAM Size	8GB	3GB	115 GB	16GB
Number of Processors	4	8	8	8

2.2 SOFTWARE ENVIRONMENT

Operating System: Linux

Compiler : gcc

Language : C Programming

3. PROJECT OVERVIEW

In this Project, we are measuring the size and cost of accessing a TLB. To determine the TLB size and to know whether we have a two level hierarchy , we are going to access it by varying page numbers. The Project consists of two programs tlb.c and run.c.

tlb.c : In tlb.c is used to measure the cost of each page access with the inputs given as arguments through command line. It's been coded based on the requirements given in the project document. The following are the requirements which we met for a successful execution.

- Set the program to run on single CPU for which we used set affinity
- Calculate the Pagesize using getconf PAGESIZE or getpagesize() and found the page size to be 4KB
- Create a large array with [NUMOFPAGES * PAGESIZE/SIZEOF(INT)] entries
- So if we measure for 1 page the size of array would 1024 and for 2 pages it would be 2048 and the loop will help in updating the values.
- Various timers like clock_gettime, gettimeofday and rdtsc are used in the program to calculate the access time. Loop overhead is also calculated to be more precise for calculating the access time.
- The average access time is calculated by total/ MAX_NUM_TRIALS*NUMOFPAGES and the overhead for the for loop was removed to get the final output.

run.c : In run.c is used to execute the tlb.c code using system() command, where iterates in powers of 2 using math.pow.

Based on the outputs obtained, and from our analysis we concluded certain aspects which are detailed in further sections.

3.1 QUESTIONS

3.11 For timing, you'll need to use a timer such as that made available by gettimeofday(). How precise is such a timer? How long does an operation have to take in order for you to time it precisely? (This will help determine how many times, in a loop, you'll have to repeat a page access in order to time it successfully)

Description

For the calculation of the cost measurements we need to use predefined timer function which returns the time taken to call or execute a module of code or function. Some of the timers which can be used are gettimeofday(), clock(), rdtsc(), clock_gettime() etc., Further details are specified below.

How it works:

gettimeofday() : This function returns return's the number of seconds since the midnight (0 hour), January 1, 1970 UTC which is expressed in terms of microseconds and seconds using timeval structure. The resolution of gettimeofday() is 10microsecs. The main disadvantage with this function is that it provides less accurate time.

clock_gettime(): This function is used to retrieve the time of the specified clock id. Here there are many types of clock id, two popular clock id's are
CLOCK_REALTIME return's the current wall clock time of the day. This means that the time may jump forwards and backwards as the system time-of-day clock is changed. Any user with some right privileges can modify this.
CLOCK_MONOTONIC return's the elapsed clock time , from certain fixed point. It doesn't affect with respective to changes made to the system day clock.

rdtsc(): Read time stamp counter, this measures the cost in CPU cycles, to conversion from cycles to nanoseconds has been done in the program.

Program Implementation

To measure the cost of the TLB access, we have experimented the program with many types of timers and reported our observations.

Observations

During the experimentation for these measurements we have learned few interesting aspects.

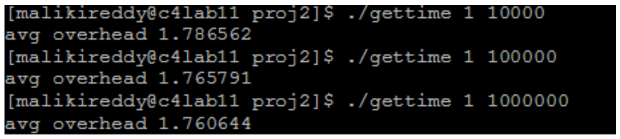
clock_gettime() : Due to the limitations in the precision when gettimeofday() used, we have implemented our programs using clock_gettime() which we observed to be more accurate when compared to the gettimeofday() timer.

rdtsc() could also be another option.

Results

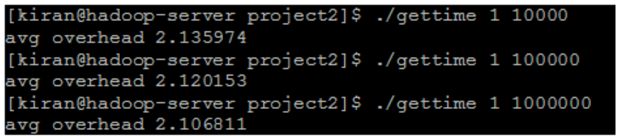
To determine how many number of iterations to be used we have considered measurements with empty loops using these three timers. As part of the experiments, we tried with number of trials ranging from 10000 to 1000000, and found that the reliable **consistency occurred at 100000**.
Note: The results with the usage of rdtsc() are represented in clock cycles. We have then converted the clock cycles to nanoseconds in code , they can found in the below table.

Measurement on C4 lab machine



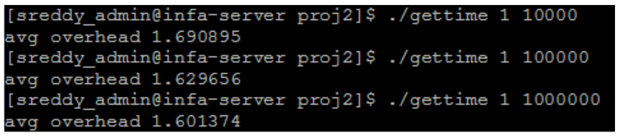
```
[malikireddy@c4lab11 proj2]$ ./gettime 1 10000
avg overhead 1.786562
[malikireddy@c4lab11 proj2]$ ./gettime 1 100000
avg overhead 1.765791
[malikireddy@c4lab11 proj2]$ ./gettime 1 1000000
avg overhead 1.760644
```

Measurement on Intel Xeon(R)



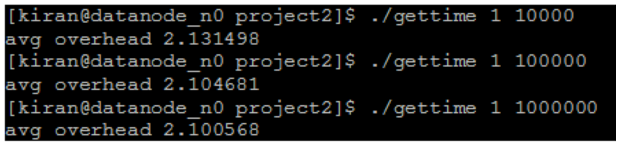
```
[kiran@hadoop-server project2]$ ./gettime 1 10000
avg overhead 2.135974
[kiran@hadoop-server project2]$ ./gettime 1 100000
avg overhead 2.120153
[kiran@hadoop-server project2]$ ./gettime 1 1000000
avg overhead 2.106811
```

Measurement on Intel Core-i7



```
[sreddy_admin@infa-server proj2]$ ./gettime 1 10000
avg overhead 1.690895
[sreddy_admin@infa-server proj2]$ ./gettime 1 100000
avg overhead 1.629656
[sreddy_admin@infa-server proj2]$ ./gettime 1 1000000
avg overhead 1.601374
```

Measurement on AMD Opteron

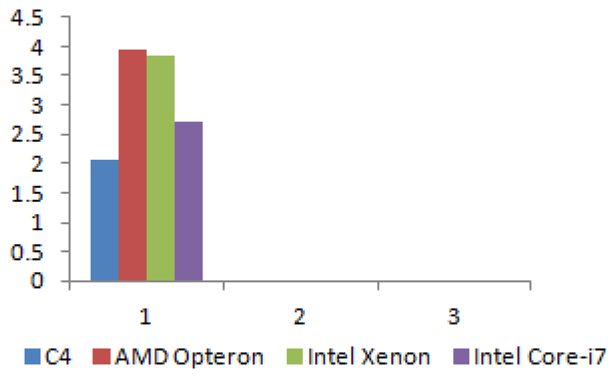


```
[kiran@datanode_n0 project2]$ ./gettime 1 10000
avg overhead 2.131498
[kiran@datanode_n0 project2]$ ./gettime 1 100000
avg overhead 2.104681
[kiran@datanode_n0 project2]$ ./gettime 1 1000000
avg overhead 2.100568
```

Conclusion

From the above experiments, we could say that the accuracy can be occurred with '100000' number of trials.

Graphs:



Comparision of overheads on various system architectures

The above graph represents the results tested on various systems, the costs of the overhead has variation on each different architecture due to the processor speed etc.,

Challenges

Determining the accuracy of the timers?

From our analysis , using various timers and running them on different architecture repeatedly varying number of trials, we determined that rdtsc() is more accurate than the other two. Here, we also considered our theoretical analysis weighing pros and cons of each timer.

3.1.2 Write the program, called tlb.c that can roughly measure the cost of accessing each page. Inputs to the program should be the number of pages to touch and the number of trials. How many trials are needed to get reliable measurements?

Description

"A translation lookaside buffer (TLB) is a cache that memory management hardware uses to improve virtual address translation speed"[3].

How it works: Whenever CPU accesses the virtual memory, there is a need of converting the virtual address to physical address. To do so there is a need of access to page table. This is performed with the use of a small cache which stores the recently accessed pages. This cache is called TLB. On a access to virtual memory , the CPU searches the Translation lookaside buffer(TLB) and retrieves the virtual page number of the page that is required to be accessed. This technique is called a TLB lookup. If there is an entry found in the TLB, this is called TLB hit. If it's not found, this is called as TLB miss. The reason that TLB is very useful because it reduces the cost of accessing to the page and increases the speed.

Program Implementation

To measure the cost of the TLB access, we have experimented the program with various types of timers and reported our observations. In this code, we initially calculated the loop overhead for all the three timers used. As per the requirement, we took two arguments as inputs ('number of pages' and 'number of trials') to the code.

Below is the code which was specified in the project requirement document.

```
int jump = PAGESIZE / sizeof(int);
for (i = 0; i < NUMPAGES * jump; i += jump) {
a[i] += 1;
}
```

The above code is used to update one array integer per page. The maximum number of pages to be updated or accessed should be given as the argument to the code. The access to the pages are repeated for several number of trials to determine the accurate cost. For the cost calculation for this access is measured by starting a timer before the loop runs and end timer is issued after the execution of the loop . Now, to find the cost of accessing each page, average is been calculated as a (difference between the end time and start time) over (number of pages * number of trials) and finally we have excluded the overhead of the overhead from the average.

They are certain key terms used in this program.

PAGESIZE : To determine the pagesize , we have used a function called getpagesize(), in our code. This can also be determined with the shell command "getconf PAGESIZE".

NUMPAGES : This value is provided as input to the code using argument.

Jump: This is used to access each page and is calculated as pagesize over size of int.

Observations

During the experimentation for these measurements we have learned few interesting aspects.

With the usage of three clocks to determine the cost, we could observe that rtdsc() generates results more accurately when compared to the other two timers.

Calculation of the overhead time : We performed this process in the same code where we run the empty loop between the timer for 100000 iterations and with no other operation in the loop. Then reported the overhead time using the elapsed time over number of iterations.

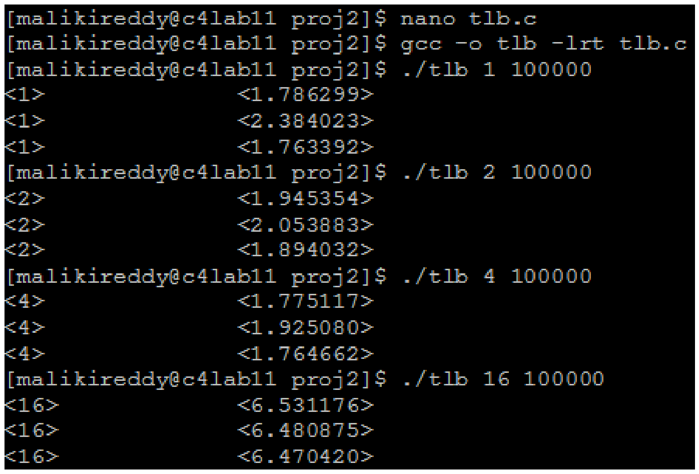
Exclusion of Overhead time : In the initial coding stage this hasn't been excluded from the total cost of the access, later from our analysis and discussions we found that there is a need to exclude the execution time, so we have measured this overhead time and then excluded the resultant time from the actual result.

Results

To obtain accurate results we have consider the statistics of at least three runs and calculated the average of those.
Note: The results with the usage of rdtsc() are represented in clock cycles. We have then converted the clock cycles to nanoseconds in the code , they can found in the below table.

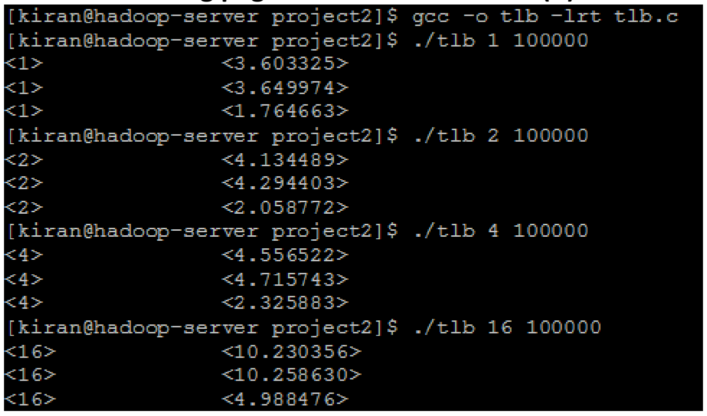
Inputs
Argument1: Number of Pages
Argument2: Number of trials

Cost of accessing page taken on C4 Lab Machine



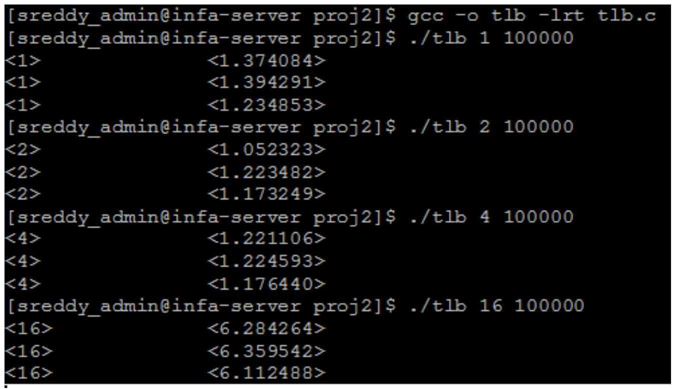
```
[malikireddy@c4lab11 proj2]$ nano tlb.c
[malikireddy@c4lab11 proj2]$ gcc -o tlb -lrt tlb.c
[malikireddy@c4lab11 proj2]$ ./tlb 1 100000
<1>                 <1.786299>
<1>                 <2.384023>
<1>                 <1.763392>
[malikireddy@c4lab11 proj2]$ ./tlb 2 100000
<2>                 <1.945354>
<2>                 <2.053883>
<2>                 <1.894032>
[malikireddy@c4lab11 proj2]$ ./tlb 4 100000
<4>                 <1.775117>
<4>                 <1.925080>
<4>                 <1.764662>
[malikireddy@c4lab11 proj2]$ ./tlb 16 100000
<16>                <6.531176>
<16>                <6.480875>
<16>                <6.470420>
```

Cost of accessing page taken on Intel Xenon(R)



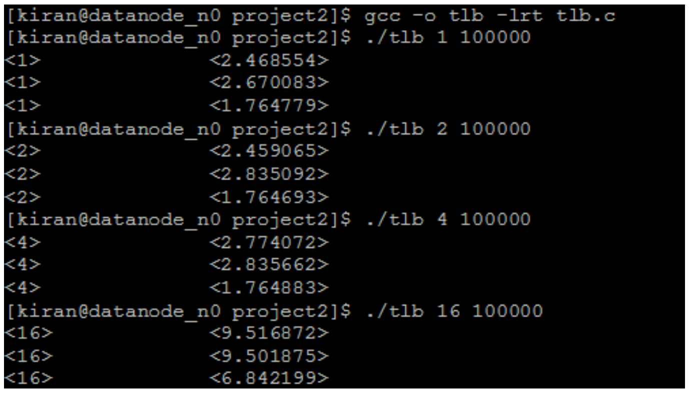
```
[kiran@hadoop-server project2]$ gcc -o tlb -lrt tlb.c
[kiran@hadoop-server project2]$ ./tlb 1 100000
<1>                <3.603325>
<1>                <3.649974>
<1>                <1.764663>
[kiran@hadoop-server project2]$ ./tlb 2 100000
<2>                <4.134489>
<2>                <4.294403>
<2>                <2.058772>
[kiran@hadoop-server project2]$ ./tlb 4 100000
<4>                <4.556522>
<4>                <4.715743>
<4>                <2.325883>
[kiran@hadoop-server project2]$ ./tlb 16 100000
<16>               <10.230356>
<16>               <10.258630>
<16>               <4.988476>
```

Cost of accessing page taken on Intel Core-i7



```
[sreddy_admin@infa-server proj2]$ gcc -o tlb -lrt tlb.c
[sreddy_admin@infa-server proj2]$ ./tlb 1 100000
<1>                 <1.374084>
<1>                 <1.394291>
<1>                 <1.234853>
[sreddy_admin@infa-server proj2]$ ./tlb 2 100000
<2>                 <1.052323>
<2>                 <1.223482>
<2>                 <1.173249>
[sreddy_admin@infa-server proj2]$ ./tlb 4 100000
<4>                 <1.221106>
<4>                 <1.224593>
<4>                 <1.176440>
[sreddy_admin@infa-server proj2]$ ./tlb 16 100000
<16>                <6.284264>
<16>                <6.359542>
<16>                <6.112488>
```

Cost of accessing page taken on AMD Opteron



```
[kiran@datanode_n0 project2]$ gcc -o tlb -lrt tlb.c
[kiran@datanode_n0 project2]$ ./tlb 1 100000
<1>                <2.468554>
<1>                <2.670083>
<1>                <1.764779>
[kiran@datanode_n0 project2]$ ./tlb 2 100000
<2>                <2.459065>
<2>                <2.835092>
<2>                <1.764693>
[kiran@datanode_n0 project2]$ ./tlb 4 100000
<4>                <2.774072>
<4>                <2.835662>
<4>                <1.764883>
[kiran@datanode_n0 project2]$ ./tlb 16 100000
<16>               <9.516872>
<16>               <9.501875>
<16>               <6.842199>
```

Conclusion

From our experiments, we can conclude that for minimum '100000' trials we could generate accurate results.

Challenges

Whenever we are using clock_gettime() , function to retrieve the cost of accessing, to compile this code we have to include a compiler option '-lrt' which links the "librt.so" Real Time shared library.

3.1.3 Now run this program (name it run.c) using system() (check man system) while varying the number of pages accessed from 1 up to a few thousand(perhaps incrementing by a factor of two per iteration) . Your program should output in this format (or similar): <no. of pages>,< corresponded access time>. What you want to see is at least one, ideally two jumps, as in the plot above. Run this program on different machines and gather some data. How many trials are needed to get reliable measurements?

Description

This program is used to run the tlb.c program using **system()** function in such a way to produce

How it works: With the help of this function one can execute any command line command by passing as an argument to system() function. Note that the system() function doesn't capture the output of the program. If we want to capture the output to a file then probably we need to use another argument to do so.

Importance: From the programmer point view, this provides an ease to execute or repeat the execution of a shell command.

Program Implementation

This program executes the previous code (refer 3.1.2) tlb.c using a function called system(). We assumed certain conditions as part of this code such that it would meet the given project requirements. We know from 3.1.2 section of this document that we need to pass the arguments(number of pages , number of trials) to the program. The functionality of this code is being implemented as follows. The number of trials to the code are taken through command line argument as we have experimented with various number of trials at each time to determine the reliable measurements. Second argument is dynamically calculated in the run.c code in a way where it's been incremented in factors of 2 for maximum of 14. Now we consider three arguments for system(). To measure the cost of the TLB access, we have used various types of timers and reported our observations.

Observations

During the experimentation for these measurements we have learned few interesting aspects. We could see that there is fluctuation in the costs as the number of pages are being increased. This might be because there isn't any refresh or drop of cache data before every run.

Results

To obtain accurate results we have consider the statistics of at least three runs and calculated the average of those.

Note: The results with the usage of rdtsc() are represented in clock cycles. We have then converted the clock cycles to nanoseconds manually , they can found in the below table.

Table representing results in clock cycles and nanoseconds.

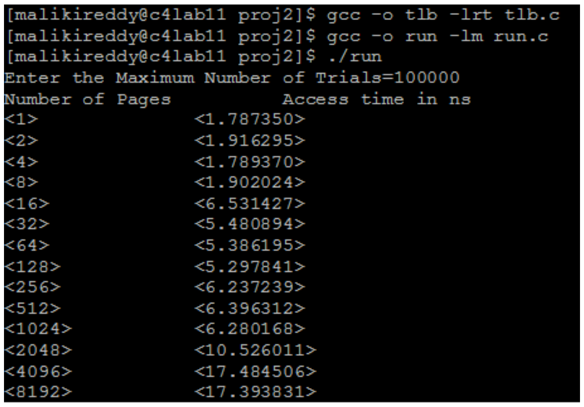
Conversion formula:

Time in nanoseconds = (1/processor speed(hertz))*number cycles

<u>Executed using clock_gettime</u>

Assumed as a jump if there is minimum difference of 5ns

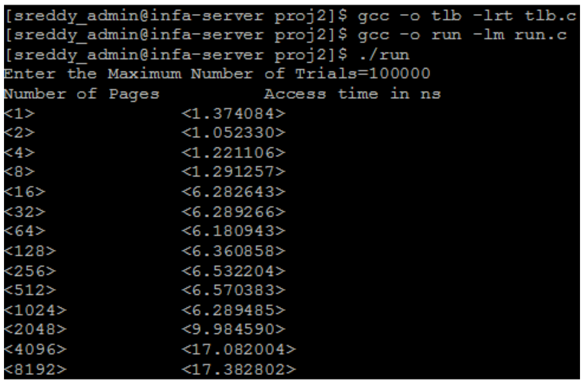
Cost of accessing taken on C4 Lab Machine



```
[malikireddy@c4lab11 proj2]$ gcc -o tlb -lrt tlb.c
[malikireddy@c4lab11 proj2]$ gcc -o run -lm run.c
[malikireddy@c4lab11 proj2]$ ./run
Enter the Maximum Number of Trials=100000
Number of Pages          Access time in ns
<1>               <1.787350>
<2>               <1.916295>
<4>               <1.789370>
<8>               <1.902024>
<16>              <6.531427>
<32>              <5.480894>
<64>              <5.386195>
<128>             <5.297841>
<256>             <6.237239>
<512>             <6.396312>
<1024>            <6.280168>
<2048>            <10.526011>
<4096>            <17.484506>
<8192>            <17.393831>
```

Jumps at 16,2048,4096

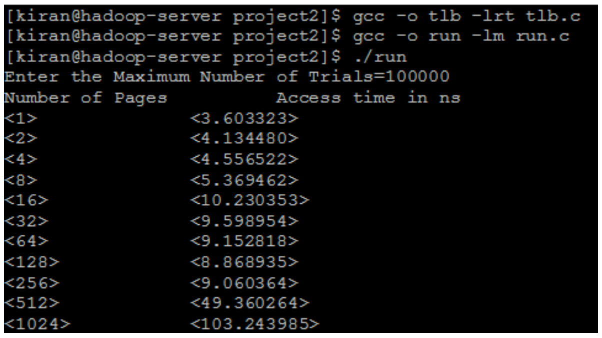
Cost of accessing taken on Intel Core-i7



```
[sreddy_admin@infa-server proj2]$ gcc -o tlb -lrt tlb.c
[sreddy_admin@infa-server proj2]$ gcc -o run -lm run.c
[sreddy_admin@infa-server proj2]$ ./run
Enter the Maximum Number of Trials=100000
Number of Pages          Access time in ns
<1>               <1.374084>
<2>               <1.052330>
<4>               <1.221106>
<8>               <1.291257>
<16>              <6.282643>
<32>              <6.289266>
<64>              <6.180943>
<128>             <6.360858>
<256>             <6.532204>
<512>             <6.570383>
<1024>            <6.289485>
<2048>            <9.984590>
<4096>            <17.082004>
<8192>            <17.382802>
```

Jump at 16,2048,4096

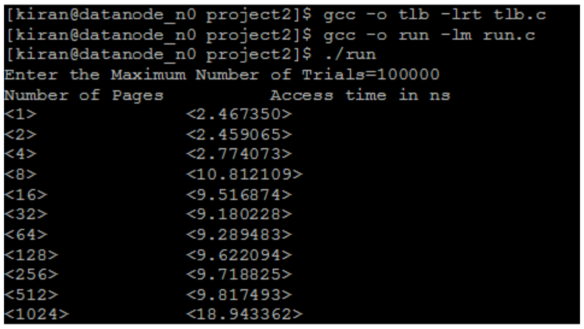
Cost of accessing taken on Intel Xenon(R)



```
[kiran@hadoop-server project2]$ gcc -o tlb -lrt tlb.c
[kiran@hadoop-server project2]$ gcc -o run -lm run.c
[kiran@hadoop-server project2]$ ./run
Enter the Maximum Number of Trials=100000
Number of Pages          Access time in ns
<1>               <3.603323>
<2>               <4.134480>
<4>               <4.556522>
<8>               <5.369462>
<16>              <10.230353>
<32>              <9.598954>
<64>              <9.152818>
<128>             <8.868935>
<256>             <9.060364>
<512>             <49.360264>
<1024>            <103.243985>
```

Jumps at 16,512,1024

Cost of accessing taken on AMD Opteron

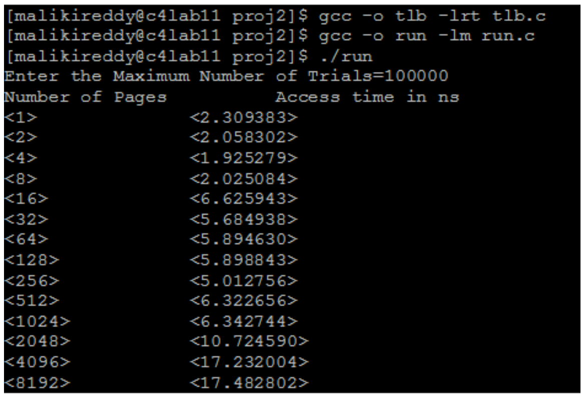


```
[kiran@datanode_n0 project2]$ gcc -o tlb -lrt tlb.c
[kiran@datanode_n0 project2]$ gcc -o run -lm run.c
[kiran@datanode_n0 project2]$ ./run
Enter the Maximum Number of Trials=100000
Number of Pages          Access time in ns
<1>               <2.467350>
<2>               <2.459065>
<4>               <2.774073>
<8>               <10.812109>
<16>              <9.516874>
<32>              <9.180228>
<64>              <9.289483>
<128>             <9.622094>
<256>             <9.718825>
<512>             <9.817493>
<1024>            <18.943362>
```

Jumps at 16,1024

Executed using gettimeofday

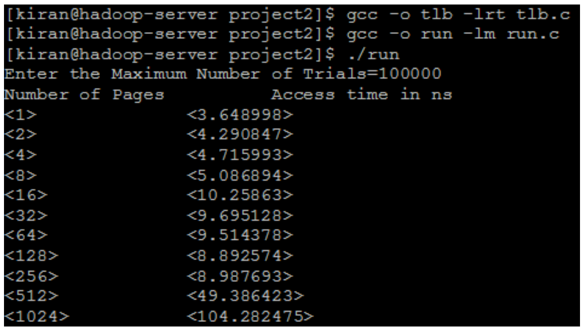
Cost of accessing taken on C4 Lab Machine



```
[malikireddy@c4lab11 proj2]$ gcc -o tlb -lrt tlb.c
[malikireddy@c4lab11 proj2]$ gcc -o run -lm run.c
[malikireddy@c4lab11 proj2]$ ./run
Enter the Maximum Number of Trials=100000
Number of Pages          Access time in ns
<1>                <2.309383>
<2>                <2.058302>
<4>                <1.925279>
<8>                <2.025084>
<16>               <6.625943>
<32>               <5.684938>
<64>               <5.894630>
<128>              <5.898843>
<256>              <5.012756>
<512>              <6.322656>
<1024>             <6.342744>
<2048>             <10.724590>
<4096>             <17.232004>
<8192>             <17.482802>
```

Jumps at 16,2048, 4096

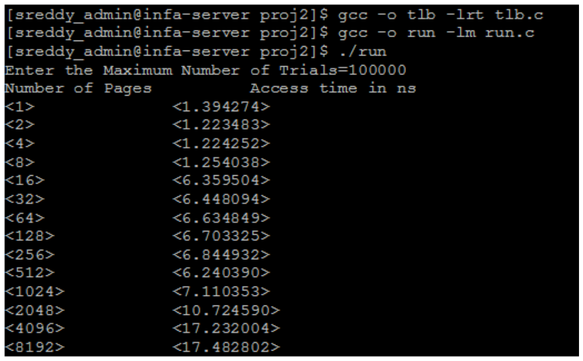
Cost of accessing taken on Intel Xenon(R)



```
[kiran@hadoop-server project2]$ gcc -o tlb -lrt tlb.c
[kiran@hadoop-server project2]$ gcc -o run -lm run.c
[kiran@hadoop-server project2]$ ./run
Enter the Maximum Number of Trials=100000
Number of Pages          Access time in ns
<1>                <3.648998>
<2>                <4.290847>
<4>                <4.715993>
<8>                <5.086894>
<16>               <10.25863>
<32>               <9.695128>
<64>               <9.514378>
<128>              <8.892574>
<256>              <8.987693>
<512>              <49.386423>
<1024>             <104.282475>
```

Jumps at 16,512,1024

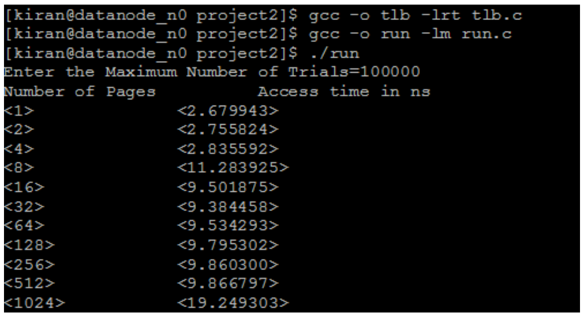
Cost of accessing taken on Intel Core-i7



```
[sreddy_admin@infa-server proj2]$ gcc -o tlb -lrt tlb.c
[sreddy_admin@infa-server proj2]$ gcc -o run -lm run.c
[sreddy_admin@infa-server proj2]$ ./run
Enter the Maximum Number of Trials=100000
Number of Pages          Access time in ns
<1>                <1.394274>
<2>                <1.223483>
<4>                <1.224252>
<8>                <1.254038>
<16>               <6.359504>
<32>               <6.448094>
<64>               <6.634849>
<128>              <6.703325>
<256>              <6.844932>
<512>              <6.240390>
<1024>             <7.110353>
<2048>             <10.724590>
<4096>             <17.232004>
<8192>             <17.482802>
```

Jumps at 16,2048,4096

Cost of accessing taken on AMD Opteron

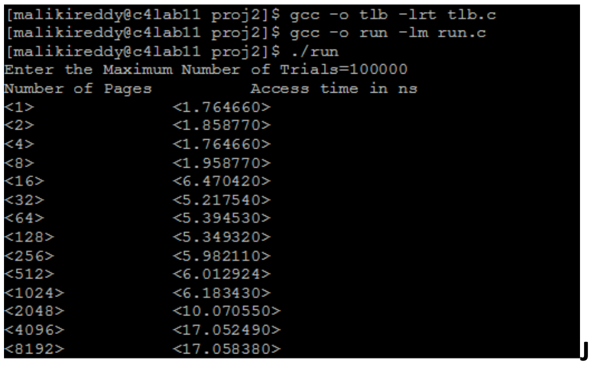


```
[kiran@datanode_n0 project2]$ gcc -o tlb -lrt tlb.c
[kiran@datanode_n0 project2]$ gcc -o run -lm run.c
[kiran@datanode_n0 project2]$ ./run
Enter the Maximum Number of Trials=100000
Number of Pages          Access time in ns
<1>                <2.679943>
<2>                <2.755824>
<4>                <2.835592>
<8>                <11.283925>
<16>               <9.501875>
<32>               <9.384458>
<64>               <9.534293>
<128>              <9.795302>
<256>              <9.860300>
<512>              <9.866797>
<1024>             <19.249303>
```

Jumps at 8,1024

Executed using rdtsc

Cost of accessing taken on C4 Lab Machine

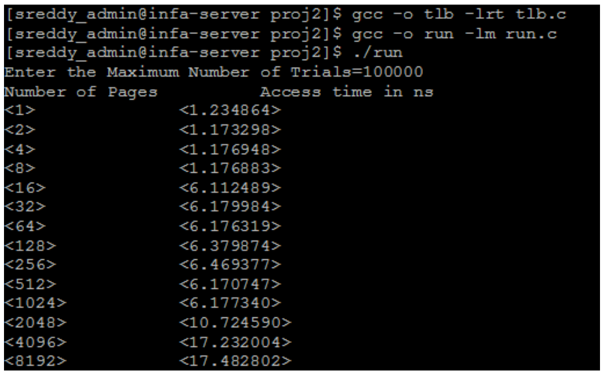


```
[malikireddy@c4lab11 proj2]$ gcc -o tlb -lrt tlb.c
[malikireddy@c4lab11 proj2]$ gcc -o run -lm run.c
[malikireddy@c4lab11 proj2]$ ./run
Enter the Maximum Number of Trials=100000
Number of Pages          Access time in ns
<1>                <1.764660>
<2>                <1.858770>
<4>                <1.764660>
<8>                <1.958770>
<16>               <6.470420>
<32>               <5.217540>
<64>               <5.394530>
<128>              <5.349320>
<256>              <5.982110>
<512>              <6.012924>
<1024>             <6.183430>
<2048>             <10.070550>
<4096>             <17.052490>
<8192>             <17.058380>
```

J

Jumps at 16,2048,4096

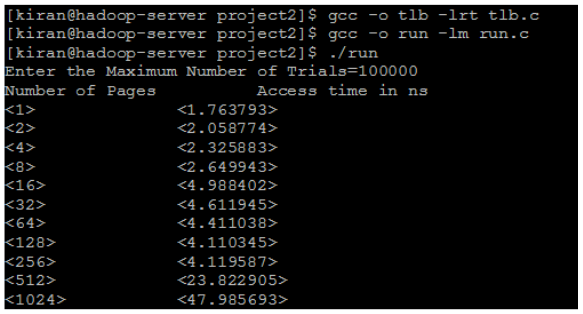
Cost of accessing taken on Intel Core-i7



```
[sreddy_admin@infa-server proj2]$ gcc -o tlb -lrt tlb.c
[sreddy_admin@infa-server proj2]$ gcc -o run -lm run.c
[sreddy_admin@infa-server proj2]$ ./run
Enter the Maximum Number of Trials=100000
Number of Pages          Access time in ns
<1>                <1.234864>
<2>                <1.173298>
<4>                <1.176948>
<8>                <1.176883>
<16>               <6.112489>
<32>               <6.179984>
<64>               <6.176319>
<128>              <6.379874>
<256>              <6.469377>
<512>              <6.170747>
<1024>             <6.177340>
<2048>             <10.724590>
<4096>             <17.232004>
<8192>             <17.482802>
```

Jumps at 16,2048,4096

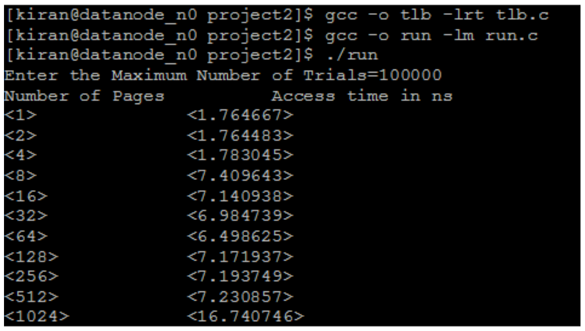
Cost of accessing taken on Intel Xenon(R)



```
[kiran@hadoop-server project2]$ gcc -o tlb -lrt tlb.c
[kiran@hadoop-server project2]$ gcc -o run -lm run.c
[kiran@hadoop-server project2]$ ./run
Enter the Maximum Number of Trials=100000
Number of Pages          Access time in ns
<1>                <1.763793>
<2>                <2.058774>
<4>                <2.325883>
<8>                <2.649943>
<16>               <4.988402>
<32>               <4.611945>
<64>               <4.411038>
<128>              <4.110345>
<256>              <4.119587>
<512>              <23.822905>
<1024>             <47.985693>
```

Jumps at 16,512,1024

Cost of accessing taken on AMD Opteron



```
[kiran@datanode_n0 project2]$ gcc -o tlb -lrt tlb.c
[kiran@datanode_n0 project2]$ gcc -o run -lm run.c
[kiran@datanode_n0 project2]$ ./run
Enter the Maximum Number of Trials=100000
Number of Pages          Access time in ns
<1>                  <1.764667>
<2>                  <1.764483>
<4>                  <1.783045>
<8>                  <7.409643>
<16>                 <7.140938>
<32>                 <6.984739>
<64>                 <6.498625>
<128>                <7.171937>
<256>                <7.193749>
<512>                <7.230857>
<1024>               <16.740746>
```

Jumps at 8,1024

Comparison of outputs for various timers on all the system architectures.

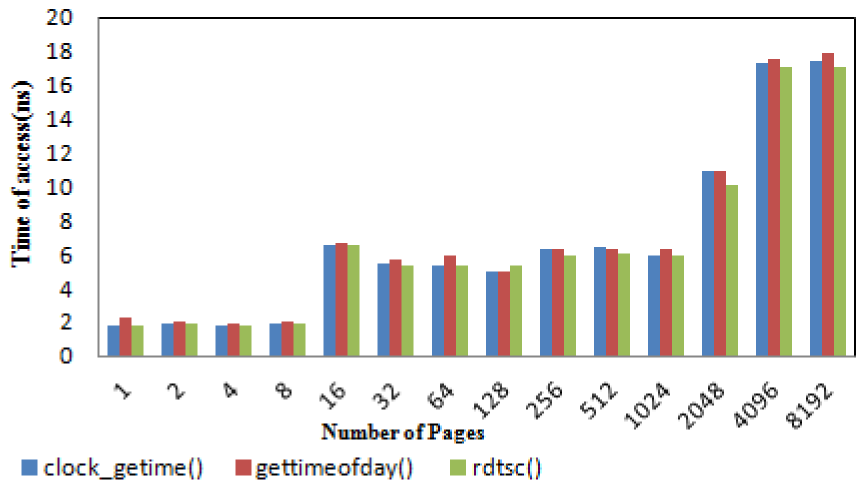
C4Lab 3392.20 MHz (Speed)	clock_gettime	gettimeofday	rdtsc
Pages	Cost of accessing(in ns)		
1	1.782993	2.436732	1.764094
2	1.948588	2.054475	1.892344
4	1.771172	1.925384	1.763456
8	1.973276	2.025466	1.905464
16	6.579753	6.625883	6.434567
32	5.486846	5.684003	5.293563
64	5.382644	5.898845	5.398564
128	4.984669	5.012235	5.346783
256	6.23287	6.282497	5.985677
512	6.392243	6.324613	6.013562
1024	5.943365	6.336008	5.997534
2048	10.881426	10.904837	10.07655
4096	17.340397	17.484964	17.05352
8192	17.380996	17.84644	17.05953

I7-4376 3520.33 MHz (Speed)	clock_gettime	gettimeofday	Rdtsc
Pages	Cost of accessing(in ns)		
1	1.374084	1.394637	1.234888
2	1.052337	1.223489	1.173202
4	1.221106	1.224253	1.176448
8	1.291251	1.254038	1.176445
16	6.284264	6.359535	6.112489
32	6.386764	6.448094	6.176315
64	6.186764	6.348094	6.176314
128	6.360646	6.716538	6.379876
256	6.539586	6.846867	6.472363
512	6.570086	6.241936	6.176319
1024	6.289748	6.336229	6.171372
2048	9.984261	10.904111	10.075009
4096	17.0803974	17.384732	16.052872
8192	17.3804996	17.842728	16.053232

XENON 1601.61 MHz (Speed)	clock_gettime	gettimeofday	rdtsc
Pages	Cost of accessing(in ns)		
1	3.603322	3.606544	1.764623
2	4.134148	4.235979	2.058771
4	4.556522	4.715018	2.325889
8	5.369462	5.086875	2.646902
16	10.230354	10.258625	4.988434
32	9.598954	9.695012	4.611654
64	9.152818	9.514375	4.412655
128	8.868903	8.892578	4.107547
256	9.060364	8.987464	4.117843
512	49.360062	49.384621	23.822914
1024	103.248339	104.2820017	47.939931

AMD 2399.998 MHz (Speed)	clock_gettime	gettimeofday	rdtsc
Pages	Cost of accessing(in ns)		
1	2.468545	2.679574	1.764665
2	2.460065	2.755409	1.764681
4	2.779407	2.835119	1.764662
8	10.812099	11.301252	7.940971
16	9.516874	9.501875	6.984443
32	9.180228	9.384451	6.842171
64	9.283845	9.534211	6.498459
128	9.623522	9.798426	7.178034
256	9.710123	9.862632	7.173604
512	9.817493	9.866797	7.213337
1024	18.984611	19.284812	16.458403

Graphs



Comparision of time of access with three different timers

The above graph represents the results tested using various timers which impacts the cost accordingly.

Conclusion

From the above analysis and observation, we could conclude that though there are some inconsistency in the cost of accessing as the number of pages increasing(for this, our analysis can be found in), we could one or two jumps. From our previous analysis , we could also see that its sufficient to run for 100000 trails.

Challenges

Why a system with one architecture have one jump whereas the other having two or more?

From our analysis, we could observe this was due to the number of caches used and their size.

Why there was inconsistency for few pages?

This impact was due to the cache. Please refer to section 3.1.5

3.1.4 Next, **graph the results**, making a graph that looks similar to the one above. **Include your conclusion about the TLB hierarchy.**

Description

As part of this project we have executed the code on 4 different systems with different architectures. Below are the graphs which were reported using these results.

Implementation

The results which were plotted are the output generated from run.c (which in turn executes the tlb.c code for several times), which ensures that we get reliable measurements.

Observations

During the experimentation for these measurements we have learned few interesting aspects.

Though we have a jumps at certain number of pages we could see some inconsistency during the execution of few number of pages. From our analysis we could observe that the reason behind this inconsistency could be the impact of *caches* i.e., whenever a page is accessed it may or may not exists in the cache, if it does so then the access time will be less when compared to access this from memory.

Results

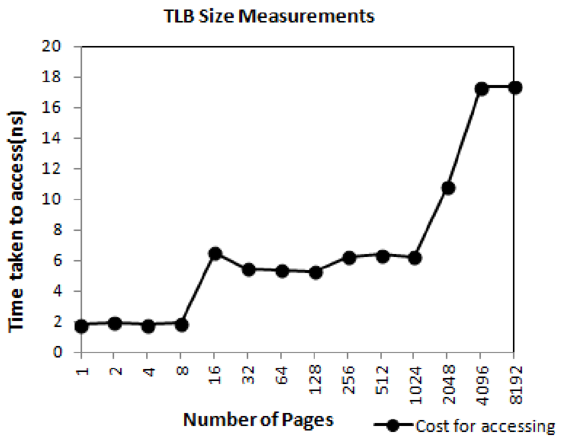
To obtain accurate results we have consider three timer's in tlb.c, and then compared the results.
Note: Though these three functions are being implemented as part of the same program, we have executed each module at a time by commenting the other two. This reason for this is that whenever we execute this code thru run.c , we expected the results to be produced in order of the timer used.

Note: The results with the usage of rdtsc() are represented in clock cycles. We have then converted the clock cycles to nanoseconds , they can found in the below table.

Graphs:

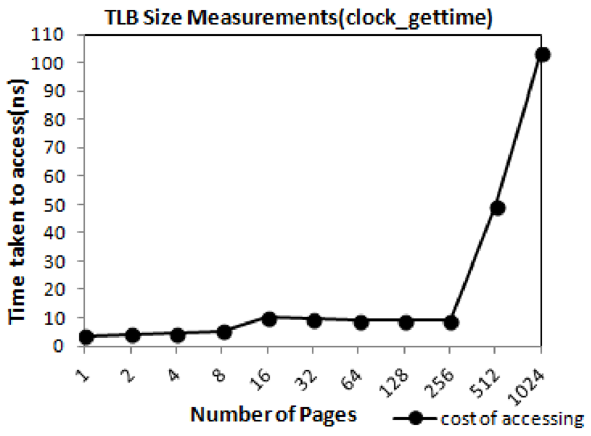
Executed using clock_gettime

Cost of accessing taken on C4 Lab Machine



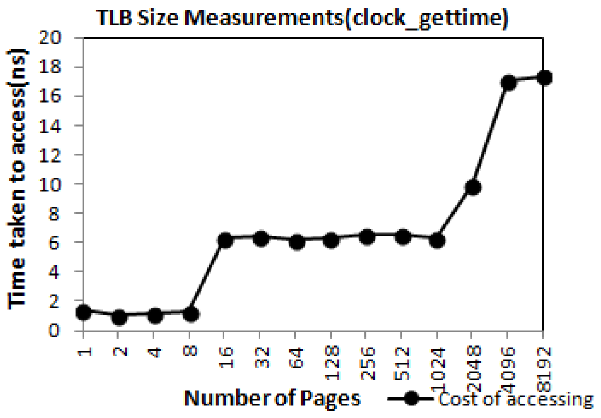
Jumps at 16,2048,4096

Cost of accessing taken on Intel Xenon(R)



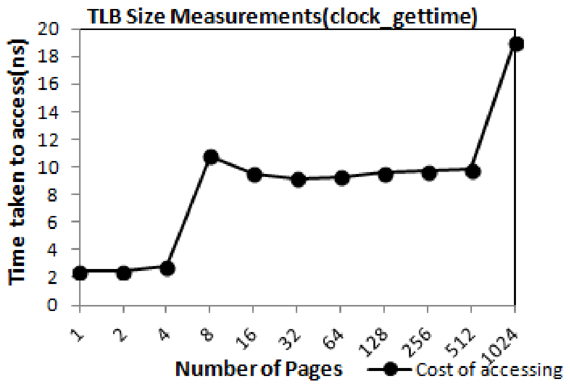
Jumps at 16,512,1024

Cost of accessing taken on Intel Core-i7



Jumps at 16,2048,4096

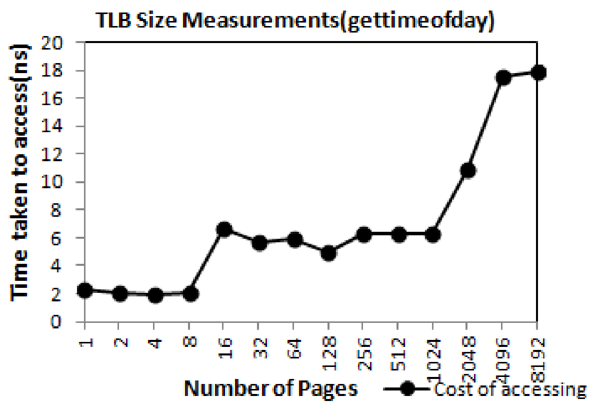
Cost of accessing taken on AMD Opteron



Jumps at 8,1024

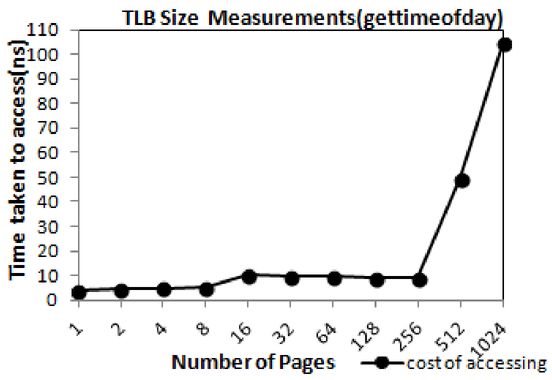
Executed using gettimeofday

Cost of accessing taken on C4 Lab Machine



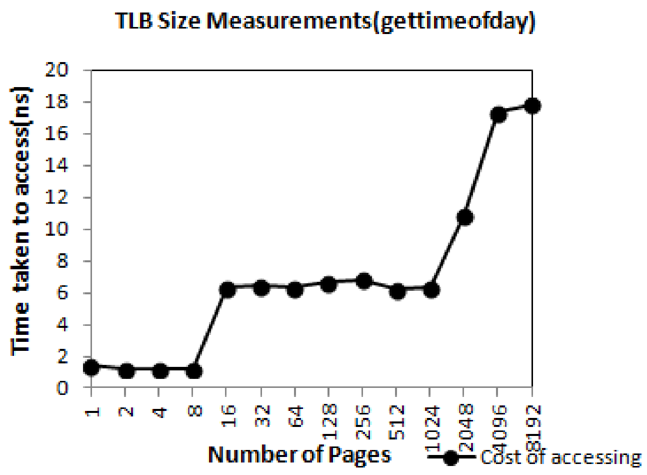
Jumps at 16,2048,4096

Cost of accessing taken on Intel Xenon(R)



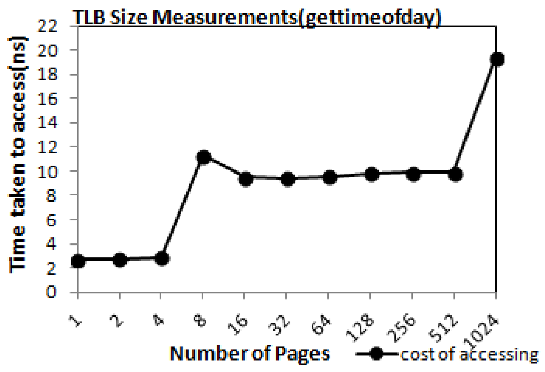
Jumps at 16,512,1024

Cost of accessing taken on Intel Core- i7



Jumps at 16,2048,4096

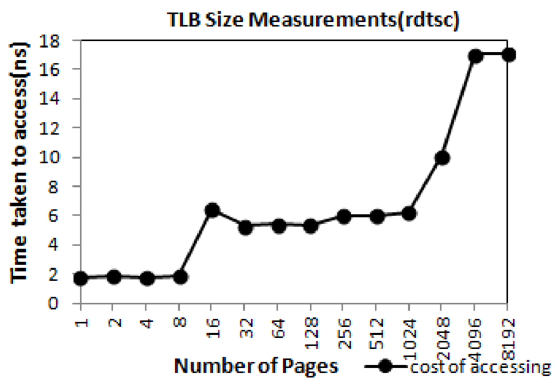
Cost of accessing taken on AMD Opteron



Jumps at 8,1024

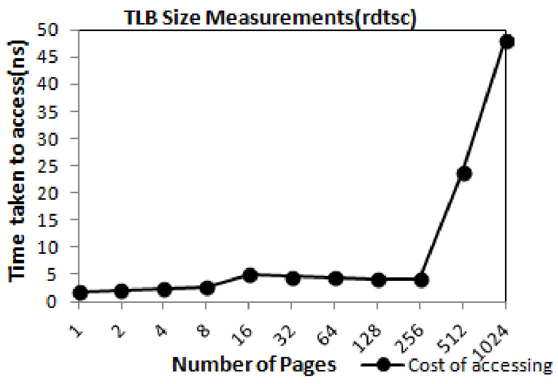
<u>Executed using rdtsc()</u>

Cost of accessing taken on C4 Lab Machine



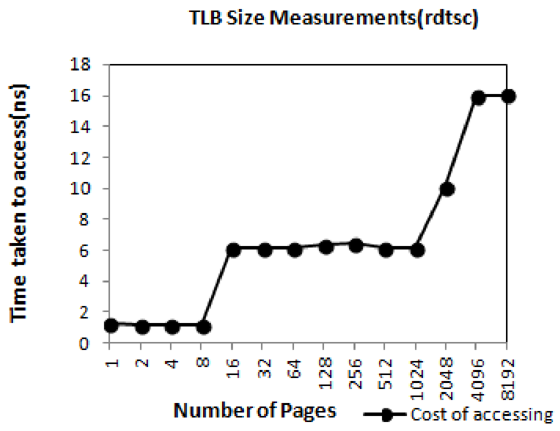
Jumps at 16,2048,4096

Cost of accessing taken on Intel Xenon(R)



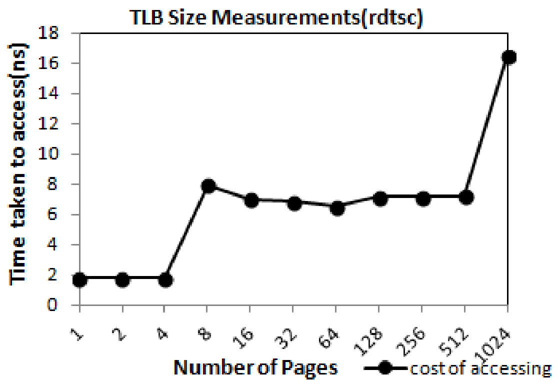
Jumps at 16,512,1024

Cost of accessing taken on Intel Core-i7



Jumps at 16,2048,4096

Cost of accessing taken on AMD Opteron



Jumps at 8,1024

The above graph represents the results tested on various systems, the costs of the page accesses has variation on each different architecture due to the processor speed etc.,

Conclusion

From the above graphs, we could conclude that there is a Two Level TLB hierarchy on C4 machine as it was found that there are atleast two jumps with the variation in number of pages. Few other machines like AMD Opteron has a one level TLB hierarchy as we could see only one jump.

Challenges

Why inconsistency observed in the results at certain number of pages?

There was certain inconsistency observed in the results at certain pages, from our analysis we could observe that this might be caused due to the pages stored in cache. Whenever a page is accessed, this might be from the TLB or memory and hence results in the inconsistency of accessing time. One solution for this can be is to refresh the caches every time so that we could find the accurate cost. The below are the commands which can be used to clear the caches, we haven't implemented this as we need root access to it which have violations of access permissions.

```
system("echo 1 >/proc/sys/vm/drop_caches");
system("echo 2 >/proc/sys/vm/drop_caches");
system("echo 3 >/proc/sys/vm/drop_caches");
```

3.1.5 **One thing to watch out for is compiler optimization. Compilers do all sorts of clever things, including removing loops which increment values that no other part of the program subsequently uses. How can you ensure the compiler does not remove the main loop above from your TLB size estimator?**

Description

Compiler optimization can be defined as a compiler that minimizes the attributes of executable code. The main advantage of this technique is used to reduce the time taken for execution and amount of memory occupation. Compiler optimization is implemented using optimizing transformations, where here it deals with a code and produces an equivalent code with fewer attributes.

Program Implementation

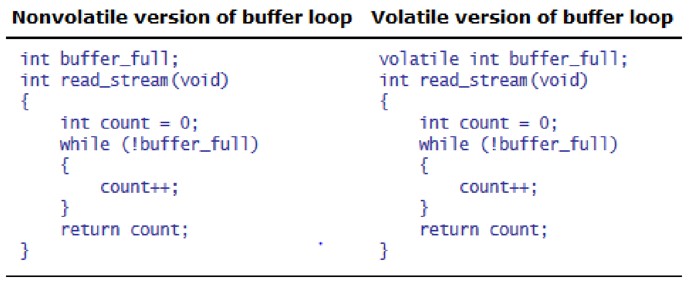
In order to avoid the optimization of loop while execution, we found there can be various options to met this requirement. Below are the few options which we have used.

Option 1: Using Volatile Keyword : When used compiler does not optimize the usage of that variable in loop

We have used 'volatile' for loop variables. When Volatile keyword is declared it means that the it says the compiler that the specific variable declared as volatile may be changed externally any time during the implementation. This means the compiler cannot perform optimization on the variable.

```
volatile int i,j,k,l,m,n;
```

Consider the below example, where the code is shown without and with the usage of volatile keyword



Nonvolatile version of buffer loop

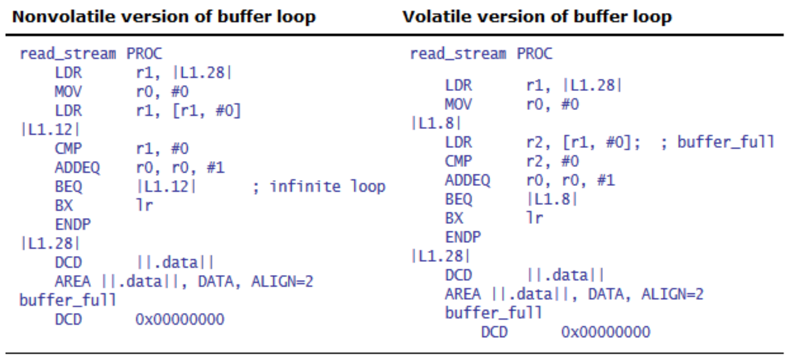
```
int buffer_full;
int read_stream(void)
{
    int count = 0;
    while (!buffer_full)
    {
        count++;
    }
    return count;
}
```

Volatile version of buffer loop

```
volatile int buffer_full;
int read_stream(void)
{
    int count = 0;
    while (!buffer_full)
    {
        count++;
    }
    return count;
}
```

Table1.1(Referenced[8])

When the above code is being compiled, below is the assembly code.



Nonvolatile version of buffer loop

```
read_stream PROC
    LDR     r1, |L1.28|
    MOV     r0, #0
    LDR     r1, [r1, #0]
|L1.12|
    CMP     r1, #0
    ADDEQ   r0, r0, #1
    BEQ     |L1.12|      ; infinite loop
    BX      lr
    ENDP
|L1.28|
    DCD     ||.data||
    AREA ||.data||, DATA, ALIGN=2
buffer_full
    DCD     0x00000000
```

Volatile version of buffer loop

```
read_stream PROC

    LDR     r1, |L1.28|
    MOV     r0, #0
|L1.8|
    LDR     r2, [r1, #0];  ; buffer_full
    CMP     r2, #0
    ADDEQ   r0, r0, #1
    BEQ     |L1.8|
    BX      lr
    ENDP
|L1.28|
    DCD     ||.data||
    AREA ||.data||, DATA, ALIGN=2
    buffer_full
        DCD     0x00000000
```

Table1.2(Referenced[8])

From the above figure, we could observe from the nonvolatile version as we have buffer_full is not declared as volatile, it is assumed by the compiler that it's value cannot be modified and omits reloading when optimization occurs. In contrast, we could see that the version of code with 'volatile usage' , the compiler assumes that buffer_full can change and hence performs no optimizations.

<u>Option 2:</u>

Another good option of what we thought would be **incrementing the main loop variables** at the end of the for loop , which doesn't have any impact on the cost calculation, but it misleads that compiler that the variable is being further used in the code, hence avoiding optimization on that variable/loop.

<u>Option 3:</u>

Usage of pragma is another option which could implement this

#pragma GCC push_options
#pragma GCC optimize ("O0")
code
#pragma GCC pop_options

3.1.6 Another thing to watch out for is the fact that most systems today ship with multiple CPUs, and each CPU, of course, has its own TLB hierarchy. To really get good measurements, you have to run your code on just one CPU, instead of letting the scheduler bounce it from one CPU to the next. How can you do that? What will happen if you don't do this?

Description

Multi core processor is a component composed of two or more core(processing units) . These cores can be utilized to run multiple instructions at the same time on various cores which in turn increases the overall speed of programs. The aim of such kind of processors is to process parallel computing which overcomes the drawback of time consumption from single processor systems. This type of core technology helps to achieve efficiency.

Some examples of multi core processor's are

Two Cores: AMD Phenom II X2 and Intel Core Duo

Quad(four) Cores: AMD Phenom II X4, Intel's i5 and i7 processors

Six Cores: AMD Phenom II X6 and Intel Core i7 Extreme Edition 980X

Eight Cores: Intel Xeon E7-2820 and AMD FX-8350

Advantages:

The main advantage of this multi core processers are to enhance the processing speed, multithreading , high performance, parallel processing.

Apart from the above benefits, it also have some disadvantages, which includes increase in the difficulty of the design of the operating system, heat radiation which indeed require heat sinks to cool the processors.

Program Implementation

In order to calculate the cost of accessing the TLB, we have bind the program to run on only one core processor. This has been implemented using a set affinity. Also the same can be implemented using a 'taskset' shell command which need to be prefixed to the executable file name.

Using Set Affinity code below , we can bound the processor to one core.

```
cpu_set_t  mask;
CPU_ZERO(&mask);
CPU_SET(0, &mask);
sched_setaffinity(0, sizeof(mask), &mask);
```

Using Taskset shell command.

taskset -c 0 ./exe

With "-c" option, you can specify a list of numeric CPU core.

Challenge

If processor is not bound to one core?

If the processor is not bound to one core, then there is a chance of context switching between the processor. And hence there is a need for calculating the context switch time which may be very difficult to dealt with. This context switching may happen between the cores which may also impact the calculation for cost of access time.

3.2 CONCLUSION

In this project, we have implemented functionalities to measure the cost of accessing a page on four different architectures. From our experiments we observed various interesting aspects such as cost of accessing results varying with vary in the architectures, increase in the number of pages in results in increase of cost.

4. REFERENCES

[1] https://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf
[2] http://www.mcs.anl.gov/~kazutomo/rdtsc.h
[3] http://en.wikipedia.org/wiki/Multi-core_processor
[4] http://unix.stackexchange.com/questions/113585/how-to-find-the-page-size-associativity-and-tlb-size-and-number-of-entries
[5] http://faculty.cs.niu.edu/~berezin/463/assns/clocks.html
[6] http://stackoverflow.com/questions/680684/multi-cpu-multi-core-and-hyper-thread
[7] http://unix.stackexchange.com/questions/23106/limit-process-to-one-cpu-core
[8] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0472c/CJAEGDEA.html
[9] http://www.gnu.org/software/libc/manual/html_node/CPU-Affinity.html