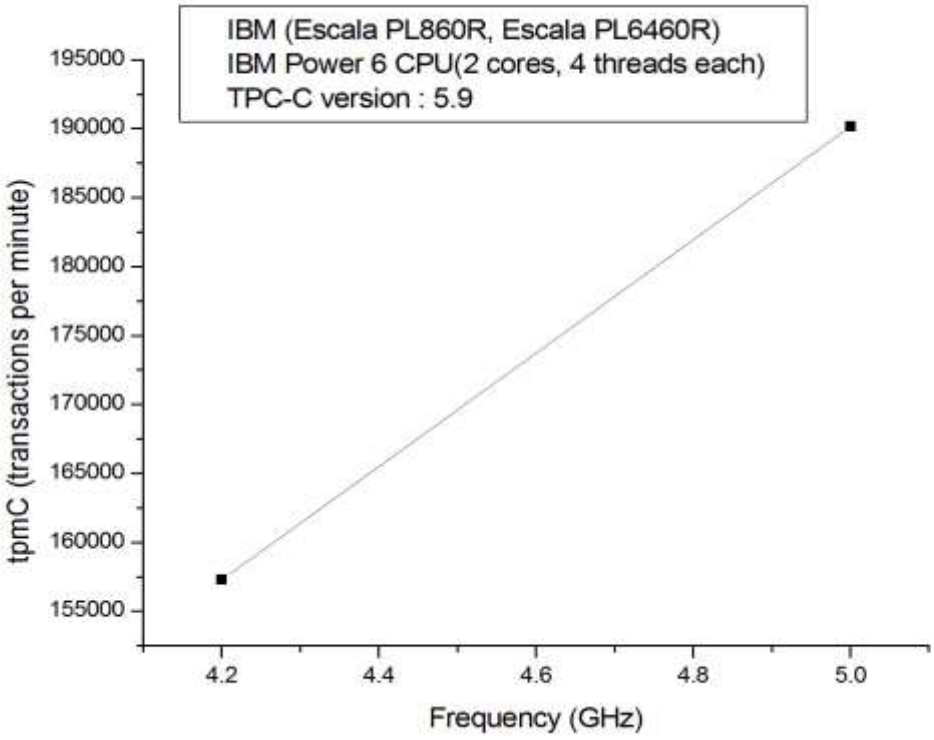
CSL 309 – Architecture of High Performance Computing

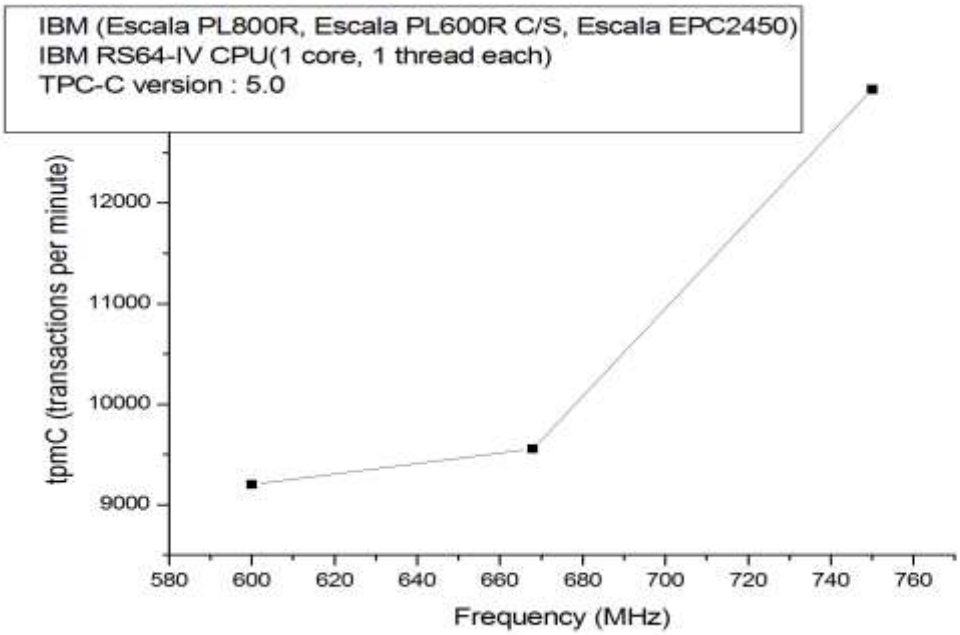
Project Report

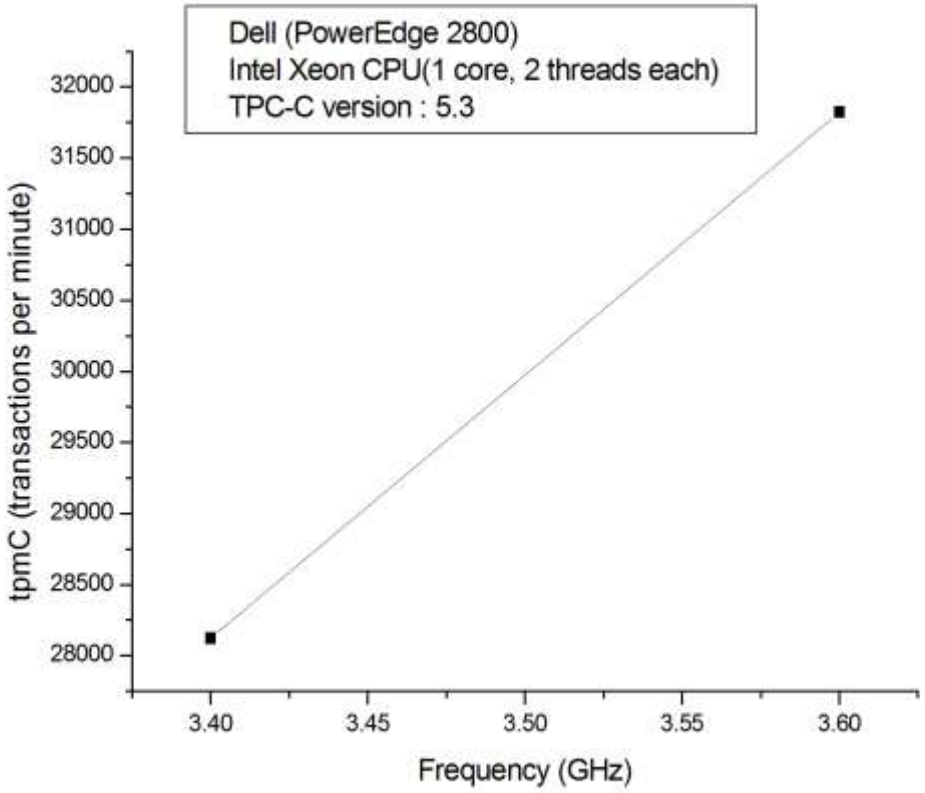
Part 1

Here we are using data from tpc.org to see the variation in Throughput (measured in tpmC) with changes in different system variables for similar types of systems. The results used for plotting these graphs are for the TPC-C benchmark.

Variation in Throughput with frequency

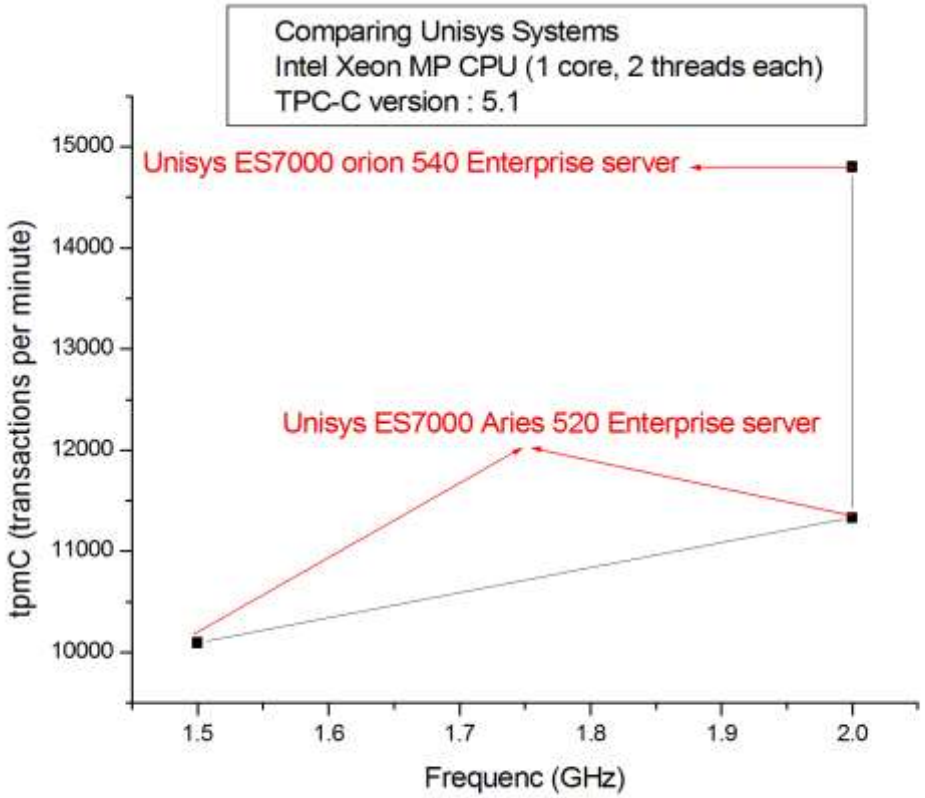




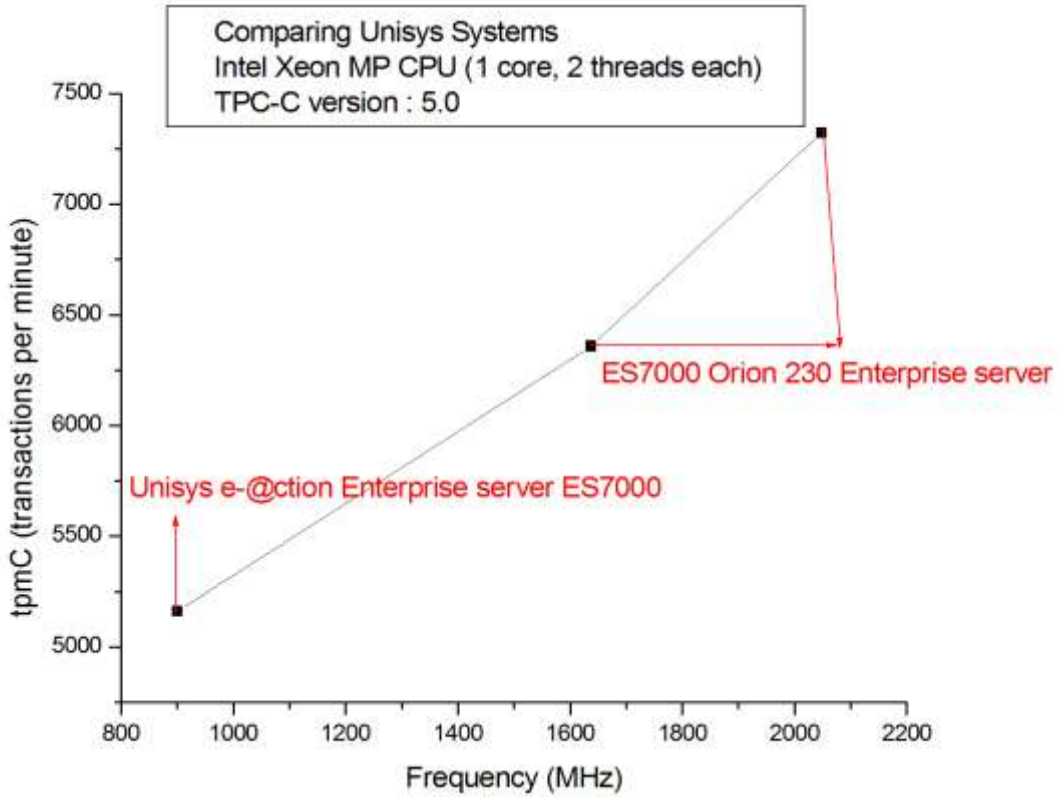


These results are for the IBM and Dell systems. Looking at the graphs for these systems, the throughput is not exactly linear with respect to the frequency. This shows that other system variables also have an effect.

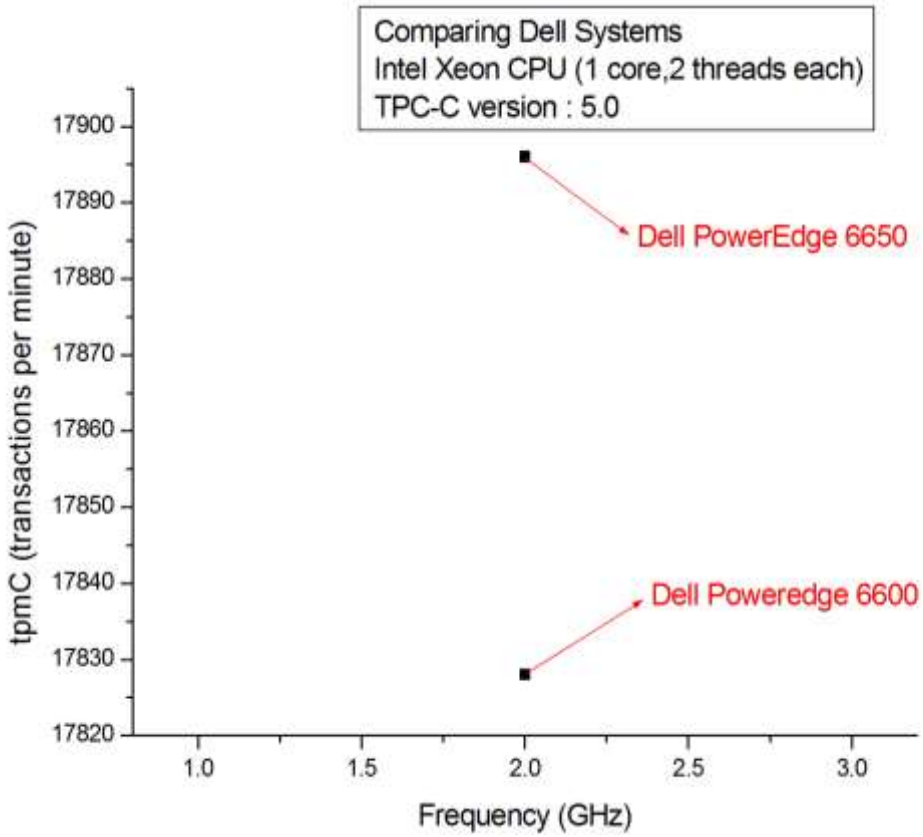
Following graphs depict the comparison done among different systems from the same organization:



The above graph indicates that increase in frequency will increase the throughput for the same system as expected. Also for same frequency **ES7000 Orion 540 Enterprise server has higher performance than ES7000 Aries 520 Enterprise server.** Similar inferences can be made from the graph shown below:



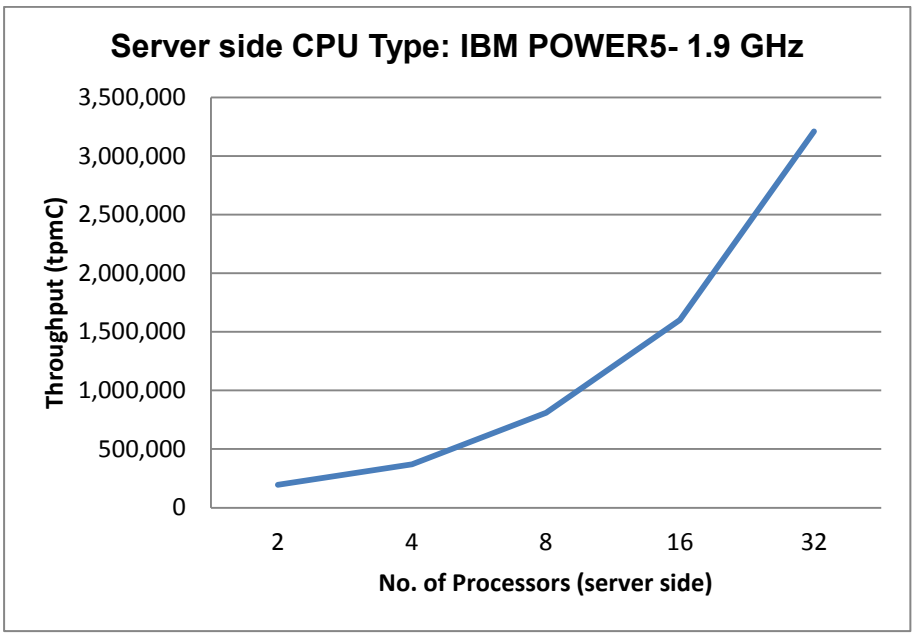
The graph below depict that Dell PowerEdge 6650 has higher performance in comparison with Dell PowerEdge 6600 for the same frequency.



If we combine the observations from each of these graphs the net variation of throughput with frequency is not directly proportional.

Variation in Throughput with no. of processors

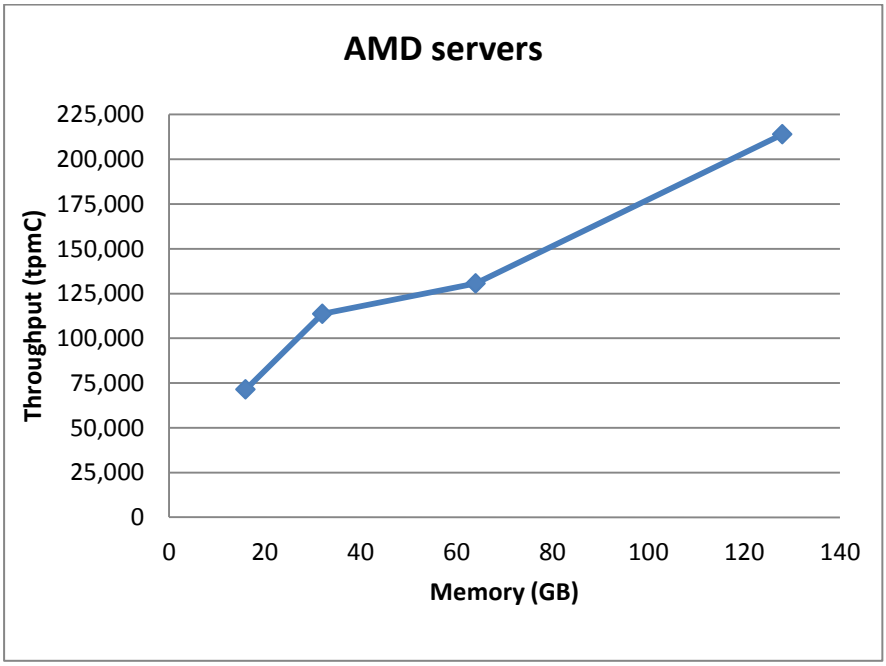
This graph has been plotted for IBM Power5 systems with a fixed frequency of 1.9 GHz. The variation in throughput is studied with respect to the no. of server side processors.



This shows that variation in throughput with the no. of processors is directly proportional.

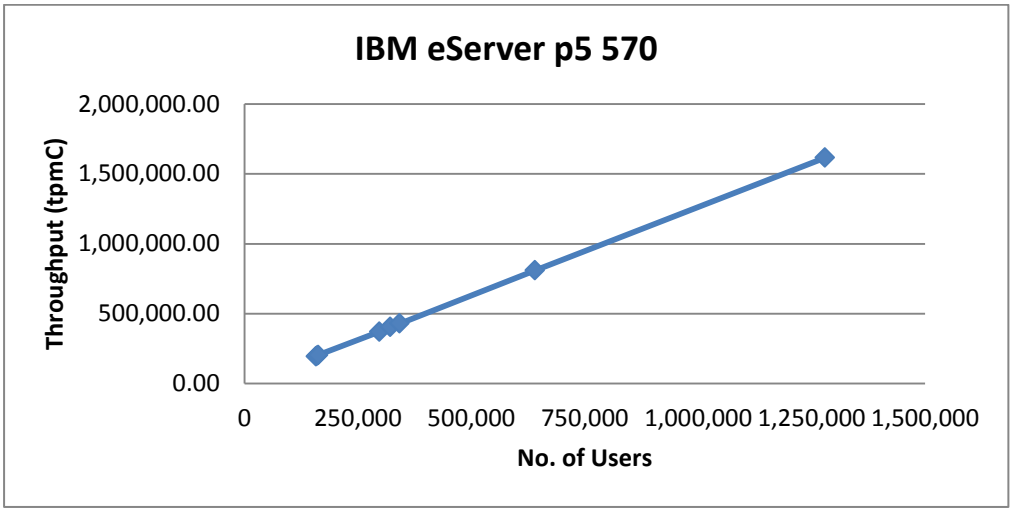
Variation in Throughput with memory

This is studied for AMD servers. And the graph shows that the throughput increases with larger memory but the increase is not directly proportional.



Variation in Throughput with no. of users

The graph plotted for IBM eServer p5 570 compares the throughput with the no. of users.



The throughput is observed to increase linearly with the no. of users.

Part 2

To study the effect of effect of address pattern on cache by Workload Generator

Cache is used to exploit the spatial and temporal locality. A program is said to have good locality of reference if it can reuse data while it is still in the upper levels of memory, which requires that the use and reuse of data be close together in time. The property of programs to reuse data and instructions they have used recently is called locality, and locality of reference refers to locality in the context of data. The job of cache memory is to hold frequently referenced data close to the processor. A cache miss occurs when a program requests data that is not found in cache memory.

Workload Generator

Arguments:

- -**seq** < length = % of ins * repeat ,:... >
- -**ins** < total # of instruction >
- -**load** < approx % of load instruction >

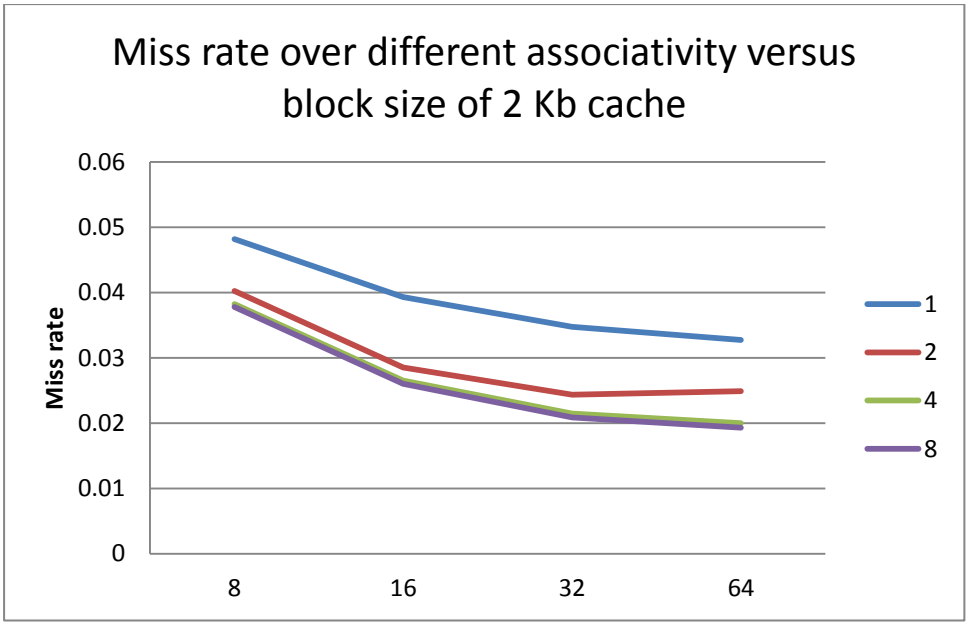
Example : *./workgen -seq 10:20*2:10:20*2 -ins 100000 -load 20*
Address trace will have 10000 instructions then 20000 instructions in a loop 2 times. Then 10000 instructions and 20000 instructions in a loop 2 times. And load instructions approx 20 %.

Cache Simulator

Arguments:

- -**cap** < capacity in kbytes >
- -**block** < size of block in (8,16,32,64) >
- -**assoc** < associativity in (1,2,4,8) >
- -**out** < standard output to file >

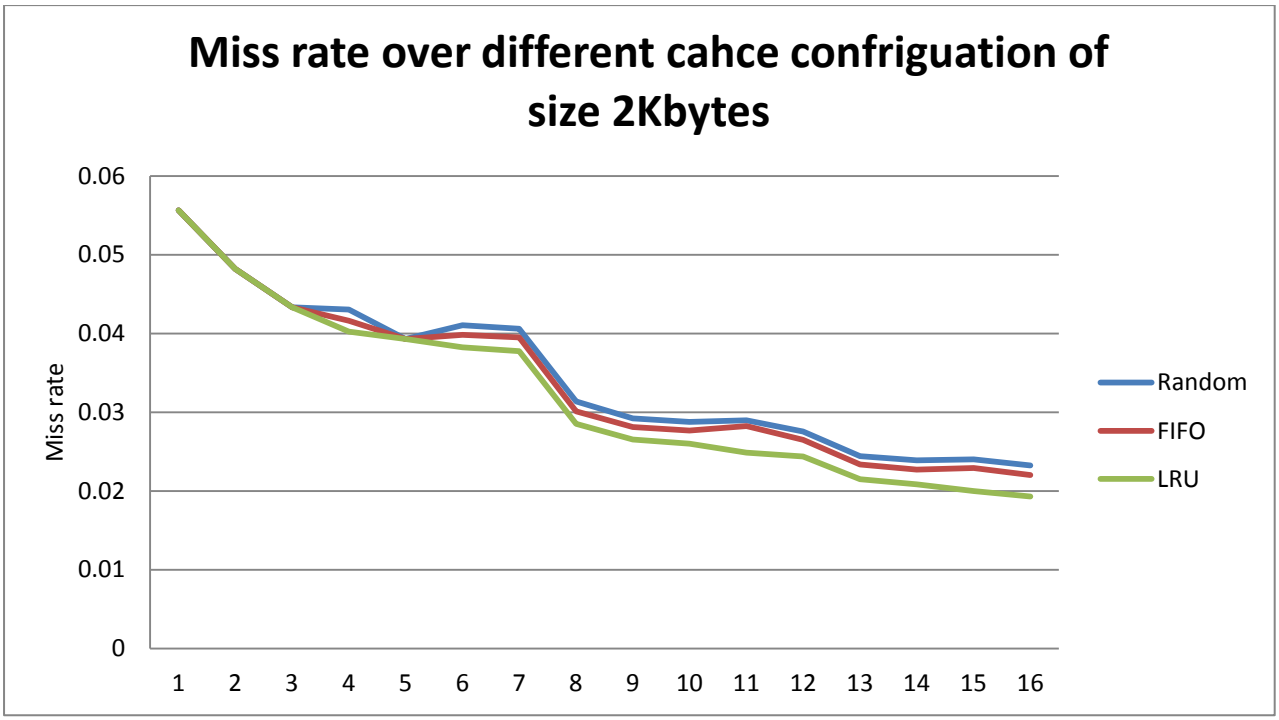
Example : *./cache-sim -cap 2 -block 32 -assoc 4 -out output tracefile*



Above graphs show that for a fixed associativity the miss rate decreases as we increase the block size from 8 to 64 Kb. These shows that for the given address trace larger blocks improve hit ratio as more of the nearby instructions are brought into the cache.

The Cache configurations considered in the following graphs are:

Number of sets	32	256	64	128	128	64	32	64	32	16	16	32	16	8	8	4
Block Size	64	8	32	8	16	8	8	16	16	16	64	32	32	32	64	64
Associativity	1	1	1	2	1	4	8	2	4	8	2	2	4	8	4	8
S no.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16



From graph above, LRU replacement is always below the FIFO and random policy. Hence it is better than others. For simulation on address pattern, LRU has been used.

The miss rate varies for the same cache configurations if we change the structure of the address sequence as per the loop optimization techniques used by the compiler. Some of these techniques and their effect are:

*The Cache configurations considered in the following graphs are:

Number of sets	8	16	32	4	32	64	16	8	64	128	32	16	64	128	256	32
Block Size	64	64	64	64	32	32	32	32	16	16	16	16	8	8	8	8
Associativity	4	2	1	8	2	1	4	8	2	1	4	8	4	2	1	8
S no.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Optimization Strategies

Loop interchange

One major purpose of loop interchange is to improve the cache performance for accessing array elements. Cache misses occur if the contiguously accessed array elements within the loop come from a different cache line. Loop interchange can help prevent this. The effectiveness of loop interchange depends on and must be considered in light of the cache model used by the underlying hardware and the array model used by the compiler.

For example

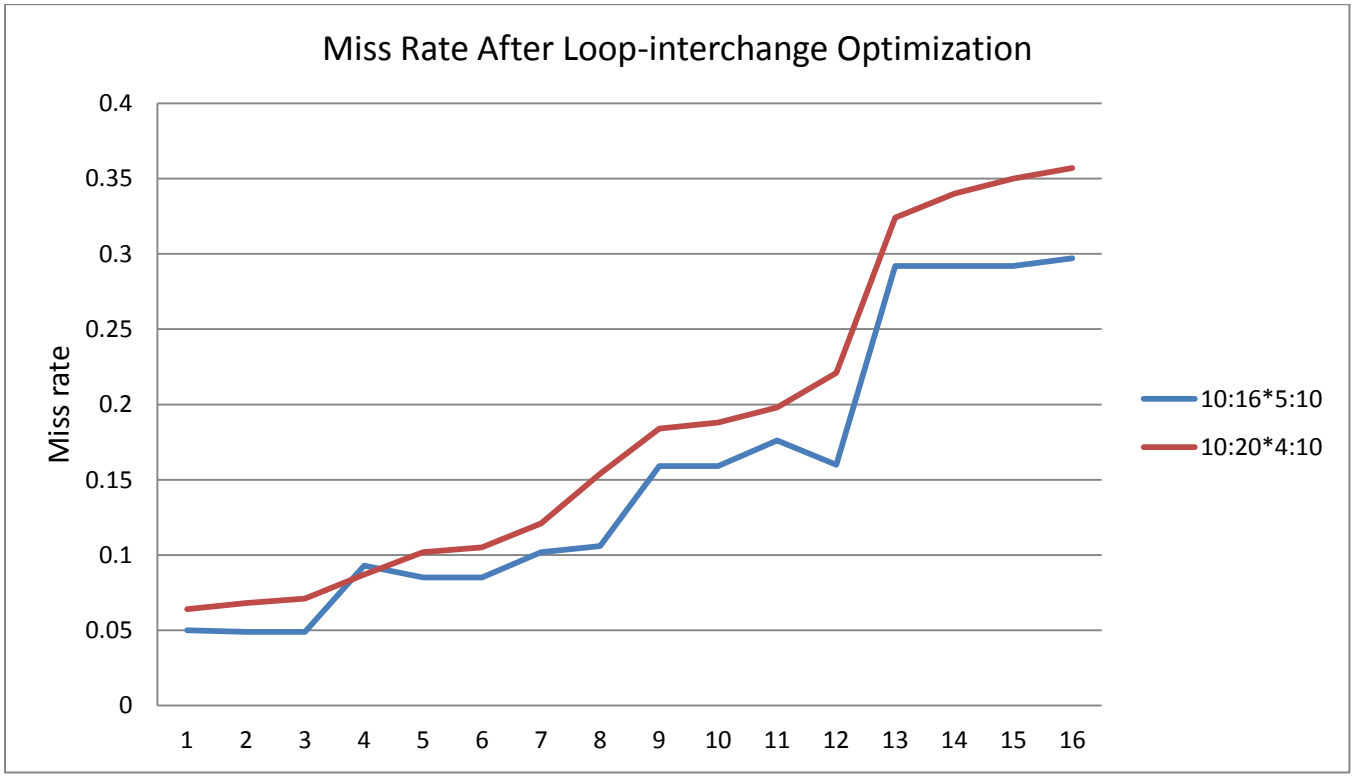
Instruction pattern **A. 10:20*4:10**

```
{
//Some sequential code
}

for j from 0 to 4                  //4 times 20 instruction = 80
 for i from 0 to 5                 //5 times 4 instruction = 20
   a[i,j] = i + j                  //4 instructions

{
//Some sequential code
}
```

After interchanging the loops in above code, it will still have 80 instructions to execute but the pattern will be **B. 10:16*5:10**

```
{
//Some sequential code
}

for j from 0 to 5                    //5 times 16 instruction = 80
 for i from 0 to 4                   //4 times 4 instruction = 16
   a[i,j] = i + j                    //4 instructions

{
//Some sequential code
}
```

After plotting the miss rate of various cache simulate over both the pattern, it is evident that second one is more optimize for cache. It is so because as the contiguously accessed elements within a loop are reduced in number the possibility of need to move to another cache line is also reduced thus decreasing the miss rate.



Loop fission

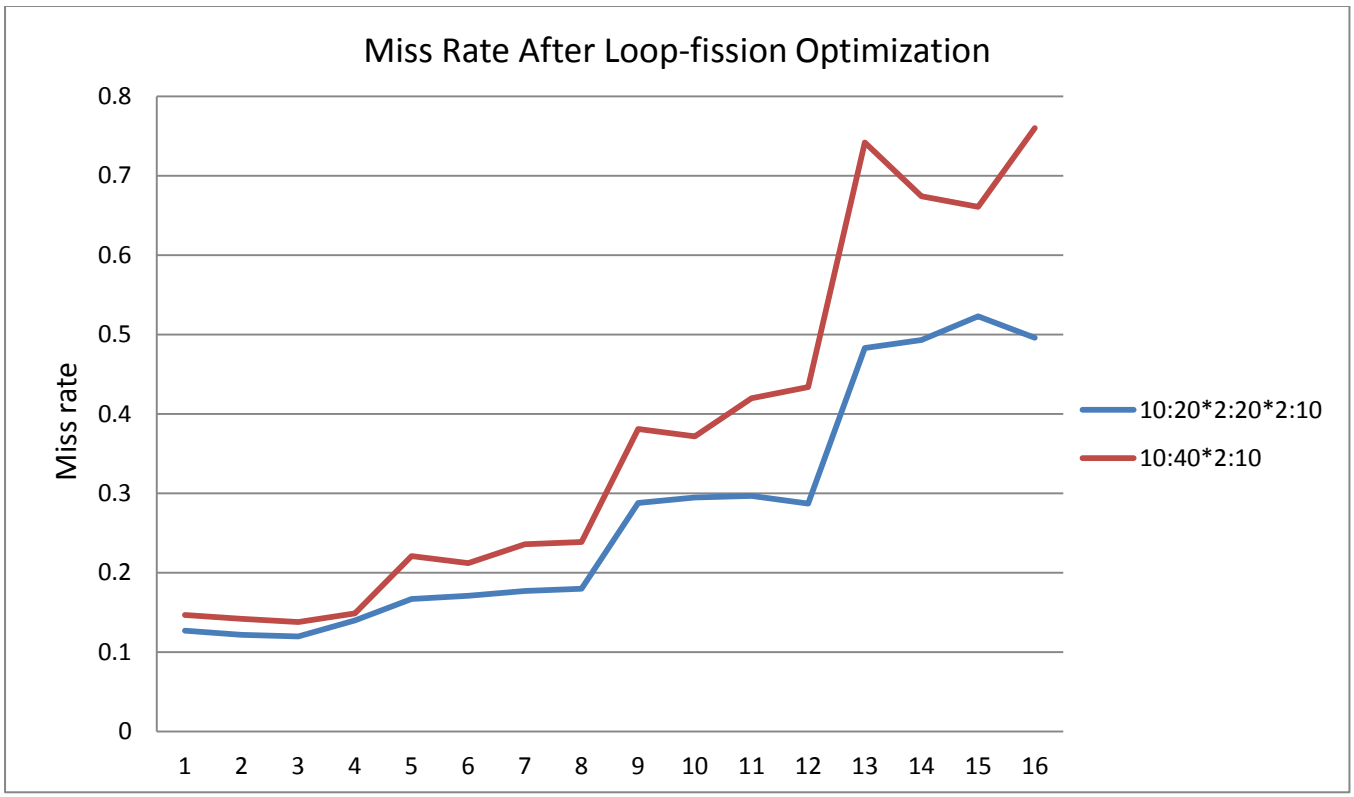
Loop fission (or loop distribution) is a compiler optimization technique attempting to break a loop into multiple loops over the same index range but each taking only a part of the loop's body. The goal is to break down large loop body into smaller ones to achieve better utilization of locality of reference. It is the reverse action to loop fusion.

Following code have pattern of **C. 10:40*2:10**

```
{
//Some sequential code
}

for i from 0 to 2 {
        //40 sequential instruction
}

{
//Some sequential code
}
```

After applying loop fission to code, the pattern will be **D. 10:20*2:20*2:10**

```
{
//Some sequential code
}

for i from 0 to 2 {
        //20 sequential instruction
}
for i from 0 to 2 {
        //20 sequential instruction
}

{
//Some sequential code
}
```



We can observe here that as the later configurations have smaller block size so fission shows better improvements as now the no. of elements contiguously accessed in a loop are reduced and are hence found more often in the smaller sized blocks. So larger loops can be broken down into smaller more manageable ones.

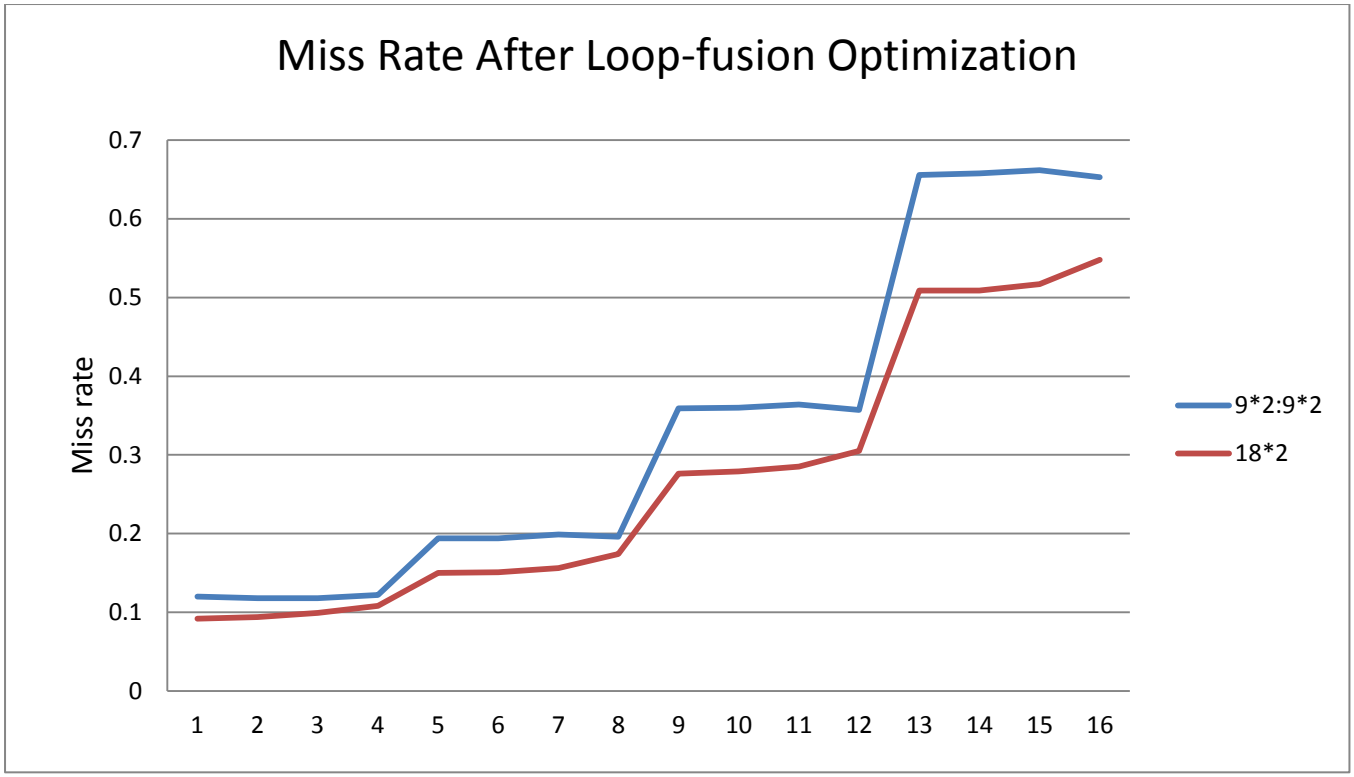
Loop fusion

Loop fusion, also called loop jamming, is a compiler optimization, a loop transformation, which replaces multiple loops with a single one. Following code have pattern of **E. 9*2:9*2:64**

```
{
//Some sequential code
}

for i from 0 to 2 {
        //9 sequential instruction
}

for i from 0 to 2 {
        //9 sequential instruction
}

{
//Some sequential code
}
```

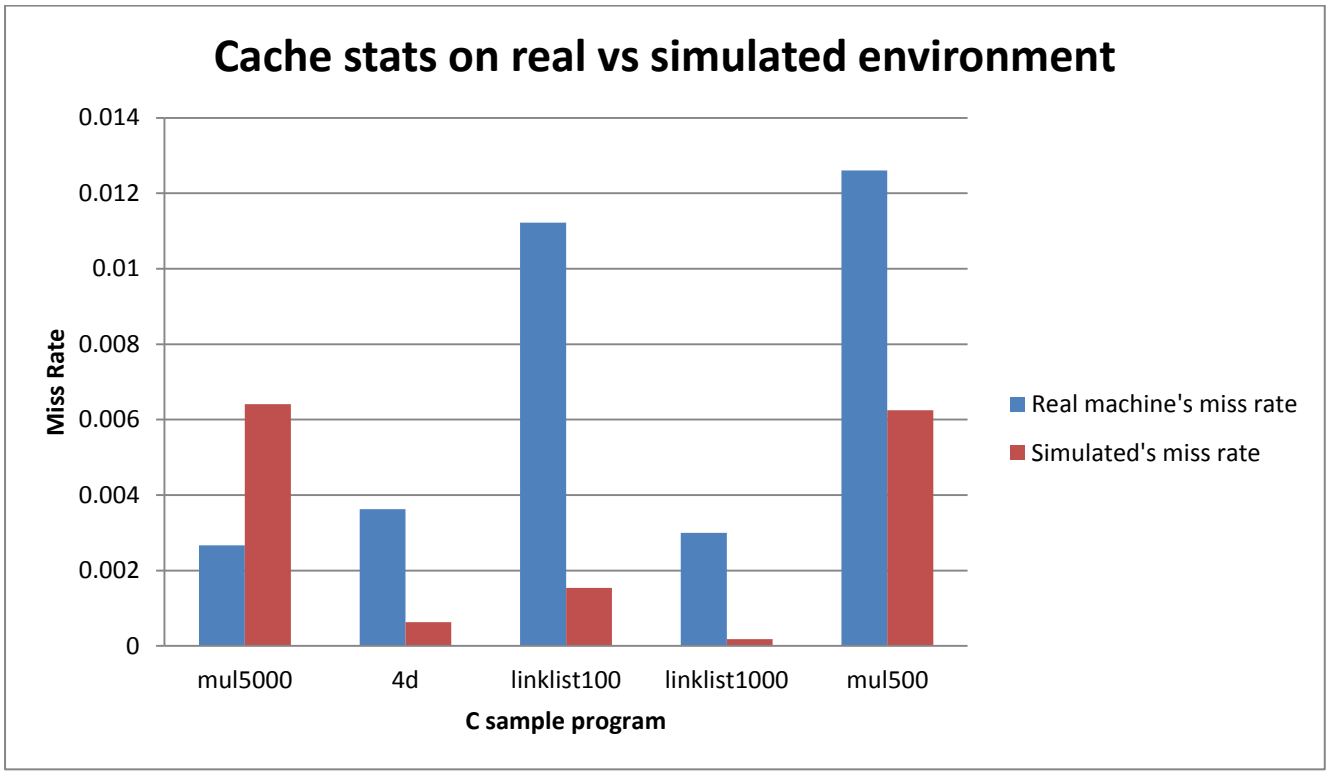
After applying loop fusion to code, the pattern will be **F. 18*2:64**

```
{
//Some sequential code
}

for i from 0 to 2 {
        //18 sequential instruction
}

{
//Some sequential code
}
```



Loop fusion is shown to be useful in cases where the elements accessed in a loop are not enough the fully utilize the spatial locality. Here if the loops are fused the no. of times we have loop decreases and the no. of array elements to be access in each loop is kept at what can be contiguously accessed in a single cache line. As this better utilizes locality of reference in such cases, the miss rate is reduced.

Comparing the cache performance of a real system with a simulator for various code snippets



In real environment, operating system has to lots of other task then just single program execution. In pipeline there are instructions and address from various different locality. Hence, miss rate increases.