

Project Report

CSCI 5593 Advanced Computer Architecture

Final Project Report

May.3.2015

Professor Gita Alaghband

**Implementation of Retiming Techniques  
on the Loop Constructs**

by

**Atrayee Bhadra(atrayee.bhadra@ucdenver.edu)**  
**Sharmila Kannan(sharmila.kannan@ucdenver.edu)**  
**Sindhuja Nandikonda(sindhuja.nandikonda@ucdenver.edu)**

## Contents

Abstract.....	3
1. Introduction and Motivation.....	3
2. Background - Performance Metrics.....	4
2.1 CPU clock.....	4
2.2 CPU cycles.....	4
2.3 Branch misses.....	4
3. Implementation.....	5
3.1 Re-Timing Technique.....	5
3.1.1 Importance of Loop Transformation.....	5
3.1.2 Factors of Re-Timing.....	5
3.1.2(a) Iterative Re-Timing.....	5
3.1.2(b) Instruction Level Re-Timing.....	5
3.1.3 Loop constructs along with Re-timing.....	6
3.1.4 Goal of implementaiton.....	7
3.1.5 Algorithm and Explanation.....	7
3.2 Skew the Pivot.....	10
3.2.1 Quicksort and Pivot.....	10
3.2.2 Pivot position Vs Input distribution.....	11
3.2.3 Quicksort - context switching problems.....	12
3.2.4 Parameters affecting sorting performance.....	12
4. Experimental Results.....	13
4.1 Machines and tools handled.....	13
4.2 Analysis of Performance Metrics .....	13
4.2.1 Re-Timing Technique.....	15
4.2.2 Skew the Pivot.....	15
5. Conclusion.....	17
6.What we have learnt.....	18
References.....	19

## Abstract

One of the most stringent issues in obtaining optimal performance from today's computers is the decay in CPU performance. This decay in CPU performance is caused by the delay in execution of the loop constructs, and mispredicted branch instructions.

The performance of software and hardware are inter-related. Whatever changes that are made in software, will reflect in the hardware performance. This report suggests that changes made in loop constructs using a loop re-timing technique, provide optimal CPU performance. Also, the count of mispredicted branch instructions can be minimized by selecting the right position as pivot.

**Key words:** Loop construct, Retiming, Branch prediction, Pivot skew, and conditional loop.

## 1. Introduction and Motivation

Improving the performance of loop structures has been the focus of a large number research projects. Significant results have been attained with both software and hardware improvements. But there will always be a decay in the CPU performance. This effect of decay is mainly because of the execution of control instructions.

Control instructions may alter the linear execution of the program. Whenever the execution path is altered, the modern systems that has the advantage of pipeline, instruction-level-parallelism, pre-fetching will then be a failure. The reason is that already fetched instructions in the pipeline will have to be discarded because of the altered execution path, and therefore new instructions have to be brought to the CPU. Overall, it causes a deterioration in unit performance.

There are a lot of techniques to improve the performance of loop execution, by achieving parallelism among loop instructions. All the techniques comes with the overhead of either code expansion, CPU overhead, branch mispredictions, or lengthy loop iterations. One important optimization of loop execution consists of improving the CPU performance by reducing loop iterations, CPU clock time and also to predict with some degree of certainty.

As part of our project, we will experiment with the loop instructions and analyze how the changes and efforts made in software, affects the Hardware performance. The analysis is based on the performance metrics CPU clock, branch misses, cpu cycles.

The next section presents the background of the Performance metrics used for our project. Section 3, explains two software techniques, its algorithm and examples, as part of our project implementation. Then in section 4, Analysis of experimental results that we got from our implementation is dealt. Later, demonstration of what we have learnt from this project is provided in section 5. Finally, a conclusion

about the techniques used and its effects on hardware is presented.

## 2. Background - Performance Metrics

The performance metric used for our project implementation are CPU clock, CPU cycles, branch misses.

### 2.1 CPU clock

CPU clock is the latency to complete a task, including disk accesses, memory accesses, input/output activities, operating system overhead- everything. CPU time recognizes to distinguish the time when processor is computing and does not include the time waiting for I/O or running other programs.

$$\text{CPU clock} = \text{Instruction count} * \text{Clock cycles per instruction} * \text{Clock cycle time}$$

#### Instruction count:

Instruction (IC) count is a dynamic measure: the total number of instruction executions involved in a program. It is dominated by repetitive operations such as loops and recursions. We can reduce the instruction count by adding more powerful instructions to the instruction set. However, this can increase either CPI or clock time, or both.

#### Clock cycles per instruction:

We can reduce CPI by exploiting more instruction-level parallelism. If they add more complex instructions it often increases CPI.

#### Clock cycle time:

Clock time (CT) is the period of the clock that synchronizes the circuits in a processor. It determines the speed of the processor. It is the reciprocal of the clock frequency.

### 2.2 CPU cycles

Cycles per instruction (CPI) is an effective average. It is averaged over all of the instruction executions in a program.

CPI is affected by instruction-level parallelism and by instruction complexity. Without instruction-level parallelism, simple instructions usually take 4 or more cycles to execute. Instructions that execute loops take at least one clock per loop iteration. Pipelining (overlapping execution of instructions) can bring the average for simple instructions down to near 1 clock per instruction. Superscalar pipelining (issuing multiple instructions per cycle) can bring the average down to a fraction of a clock per instruction.

For computing cycles per instruction as an effective average, the cases are categories of instructions, such as branches, loads, and stores. Knowledge of how the architecture handles each category yields the cycles per instruction for that category.

### 2.3 Branch misses

Branch misses is used to measure the branch misprediction rate. Branch prediction guesses the next instruction to execute and inserts the next assumed instruction to the pipeline. Wrong guess is called branch misprediction. Branch misprediction occurs when a CPU mispredicts the next instruction to process in branch prediction, which is aimed at speeding up execution.

### **3. Implementation**

#### **3.1 Re-Timing Technique**

Re-Timing technique is a loop transformation technique. Loop transformation reconstructs the loop.

There are a number of transformations, that can be applied to the loops for performance improvement. Some loop transformation techniques are

- Loop distribution
- Loop fusion
- Loop interchanging
- Loop tiling
- Re-timing
- Loop unrolling
- Loop skewing etc.,

In general, to fruitfully parallelize loops, the loops must be transformed to expose parallelism that may not be available in the original form.

##### **3.1.1 Importance of Loop Transformation**

The time consuming part of any program is the execution of loops. The importance and goal of loop transformation depends on the target architecture. Few advantages of loop transformation are:

- Improve data reuse and data locality
- Reduce the number of latches or registers needed
- Reduce memory cost
- Pipelining of instructions
- Reduce the clock cycle time of the pipeline
- Reduce overheads associated with the executing loop
- Maximize parallelism

##### **3.1.2 Factors of Re-Timing**

Re-Timing technique is one of the loop transformation techniques. It comprises of two factors and they are Iterative retiming and Instruction level retiming. These two factors have their own importance. They can be implemented for loop transformation either together or separately, according to the goal of the transformation.

### 3.1.2(a) Iterative Re-Timing

As the name describes, the iterative retiming is the retiming technique, that is implemented for the optimization of loop iterations. It helps in reducing the length of the loop iterations. Thereby, aids in the overall performance by reducing the CPU clock of the pipeline.

```
for i= 0 to 100
{
a[i] = a[i-2] + 3
b[i] = a[i] * 5
}
```

Program 1

```
a[0] = a[-2] + 3
for i= 0 to 99
{
a[i+1] = a[i-1] + 3
b[i] = a[i] * 5
}
```

Program 2

In the above program1, it takes 100 iterations to finish the loop.

However, when the iterative retiming technique has been implemented in program2, the loop iterations now will extend from 0 to 99, since the initial array value is initialized outside the loop. This shows the reduction in the length of the loop iteration.

### 3.1.2(b) Instruction Level Re-Timing

The instruction level retiming technique, aims in transformation in the instruction level. It helps in removing the dependences between instructions of the same iteration. In order to do so, instruction level transformation is applied in such a way that, all the iterations are executed simultaneously. Thereby, all the true dependences will be removed which in turn helps in reducing the CPU clock of the pipeline.

Consider the same programs 1 and 2 as an example. In program 1, there is a RAW dependency of  $a[i]$  values inside the loop which causes a stall in the pipeline for every iteration to calculate  $b[i]$ .

However, when the instruction level retiming technique has been implemented in program2, it can be clearly seen that the true dependency has been removed by initializing the first  $a[0]$  value outside the loop. Thus, there will not be any stall in the pipeline in order to calculate  $b[i]$ .

As previously mentioned, these two factors can either be implemented alone or together. In this program2, both the factors have been implemented together, introducing parallelism.

### 3.1.3 Loop Constructs along with Re-Timing

Along with the Re-timing loop transformation technique, implementation of new branch prediction instructions improves the accuracy in most instances of the single control instructions found in loop structure. The branch prediction instructions are:

- Predicted if (*pif*)
- Predicted auxiliary if (*paf*)
- Predicted ultimate if (*puf*)

#### Predicted if (*pif*)

In this if construct, the decision of this "if" is stored a number of loop iterations, and used throughout the restructured loop. At every occurrence of a *pif* instruction, the previous decision is retrieved from the branch register and used in fetching the target instructions, while the current decision is stored back in the branch register for a future iteration.

#### Predicted auxiliary if (*paf*)

This if construct's decision is stored until the first *paf* is encountered. This instruction is the initialization process of the branch register and does not cause the branch to be executed.

#### Predicted ultimate if (*puf*)

This construct has no real action i.e., will not make any decisions. It is used to retrieve the last stored decision and correctly fetch the target instructions.

Combining the retiming techniques along with the three loop constructs mentioned above, improves the accuracy in branch prediction occurring in loops that have no loop carried dependencies.

Before implementation of re-timing technique, the results from one iteration is used in the same iteration that had been computed in. However, after retiming the instructions, the result of the current iteration will be used in the next iteration of the loop. This new view of instruction sequence allows the program to anticipate branch decisions.

In ordinary pipeline model, a new iteration starts every  $n$  cycles, where  $n$  is called the initiation interval. The execution periods of several iterations in the ordinary pipeline are overlapped. In order to resolve the dependency between the iterations, instead of looking at the original iteration, a static schedule is constructed. Static schedule is a new loop body, that consists of nodes from different original iterations.

In the model which we present, has a loop pipeline composed of: Prologue and Epilogue.

#### Prologue

The prologue is a repeating schedule consists of control and the data instructions that are to be executed repeatedly and all the necessary delays are introduced into the structure of the loop. This forms the new loop body.

Prologue has two loop constructs, predicted if(*pif*) and predicted auxiliary if(*paf*).

#### Epilogue

It will wrap up the execution of the loop. Epilogue has one construct- predicted ultimate if(*puf*).

### **3.1.4 Goal of Implementation**

The loop iterations are long in the ordinary pipeline. Thus we aim at reducing the length of the repeating schedule of the loop constructs. Also, we aim at implementing new construct of prologue and epilogue.

### **3.1.5 Algorithm and Explanation**

To start with the retiming technique algorithm, consider the following program 3. The loop should be modeled into a direct acyclic graph(DAG) using the algorithm. Then the obtained values of prologue and epilogue from the DAG, will help in retiming the loop constructs.

```

X[0]=1
for(i=0;i<=10;i++)
{
    X[i]=X[i-1]*(-1);
}
for(i=1;i<10;i++)
{
    if(X[i]<0)
    Y[i]=Y[i-1]+7
    else
    Y[i]=[i-1]-5;
}

```

Program 3

From the above program, the loop instructions that are to be modeled into DAG are :

```

if(X[i]<0) → A
Y[i]=Y[i-1]+7 → B
else
Y[i]=[i-1]-5; → C

```

Program 3(a)

The graph is called a conditional data flow graph, because the loop includes conditional instructions. A conditional data flow graph (CDFG)  $G = (V, E, d, t)$  is a data flow graph in which nodes are differentiated by a function  $t$  according to their type into control nodes and regular ones. Control instructions are represented by a diamond shaped symbols, while all others are represented by circles. In the graph,

- Nodes represent instructions. In the above example, the instructions are modeled into nodes A, B and C.
- Edges represent the dependencies between the nodes.
- Weights on the edges represents the delays occurring while execution.
- Delays are computed as difference between the iteration where some data value is produced and the iteration when that information is utilized.

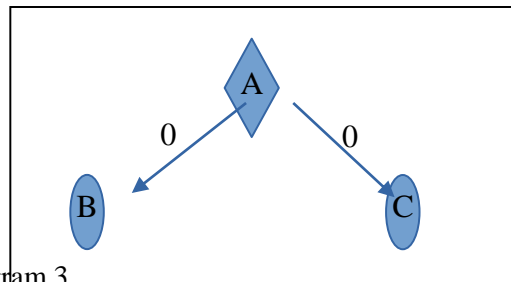


Figure 1. CDFG for program 3

Here is the algorithm, that the CDFG uses to calculate the values of *pif*, *paf* and *puf* loop



constructs, which then will be used for the retiming transformation of loop structure.

Algorithm:

Begin

```

    V={A,B,C}, Q={∅}, increment =0
    for all v ∈ V
        do level(v)=0 //initialize levels
    While V≠ ∅
        for all v ∈ V
            if indegree(v)=0 then
                Q=Q U {v} //separating control
instructions from others
                V=V-{v}
            end if
            while(Q ≠ ∅) //now, Q={ A, the control
instruction}
                get(Q,u) //u represents A here
                if(t(u)==condition) do increment=increment
+1
                    else do increment=increment+0
                    endif
                for all v ∈ V , such that for every u -> v do
                    level(v)=level(v)+increment ; indegree

```

Figure 2. Algorithm- Loop transformation

The nodes whose value is greater than 0, will be in *pif*'s. The number of each conditional statement is the maximum level minus its own level.

From the Figure1, Nodes B and C belongs to *pif*. And, the *Puf* for nodes A, B and C are 2, 0, 0 respectively.

Along with the values of *pif*, and *puf* obtained from the algorithm, the re-timing technique have been implemented in the program 3. The implementation is shown in the

**program 3(b).**

```

X[0]=1
for(i=0;i<=10;i++)
{
    X[i]=X[i-1]*(-1);
}
Paf(X[1]<0)
for( i=0;i<= 9 ;i++)
{
    Pif(X[i+1]<0)
        Y[i]=Y[i-1]+7;
    else
        Y[i]=Y[i-1]+5;
}
Puf(x[10]<0)
    Y[10]=y[9]+7;
Else
    Y[10]=Y[9]+5;

```

The *pif*, *paf*, and *puf* are the three new loop constructs added to the original program i.e., the loop transformation have been implemented. the *paf* and *pif* together known as Prologue and the *puf* forms the epilogue.

A Boolean list is then created to perform the function of the *pif*, *paf* and *puf*. Since they are of Boolean type, they have to store either "true" or "false" in the register. However, the values changes according to the program.

For example, consider Program 3(b),

When ever the *paf* is true, push its Boolean result on to the list.

Enter into the *pif*, then push the Boolean value of its result on to the list.

Now pop the list and based on the Boolean value returned, it decides which instruction to execute next. Repeat this for all the iterations.

### 3.2 Skew the Pivot

The Quicksort is a "divide and conquer" sorting algorithm. So called because, it recursively breaks the input sequence into smaller and smaller subsequences to sort.

It uses a binary recursion tree. This breakdown into sub-sequences depends on the Pivot element.

#### 3.2.1 Quicksort and Pivot

```
QuickSort (array A, length n)
  If n=1
  Return;
  P=choosePivot(A,n)
  Partition A around P
  Recursively sort left subsequence
  Recursively sort right subsequence
```

Figure 3. Algorithm-Quicksort

The pseudocode of the Quicksort algorithm is given in Figure 3. When a pivot element is chosen, the partition of the entire array is done around the pivot. Where, the elements smaller than the pivot are on the left-hand side of the pivot, called the left subsequence and, the elements greater than the pivot are on the right-hand side of the pivot called the right subsequence. The left subsequence is taken for sorting, where again the pivot is chosen according to the given condition and then again there will be two partitions for the left subsequence. Similarly, there will be two partitions to sort for the right sequence also. This partition of elements will be carried on, until no partition is possible within the subsequences. since, they are recursive, all the sorted sub-sequence will be merged when they are done sorting.

The pivot plays a very important role in the quick sort algorithm. Choosing the pivot element, can affect the hardware performance. Because, the sorting is mainly dependent on the partition done by the pivot. In the next section, we present that factors that affect the prediction and runtime of quicksort- pivot position and input distribution. Later, the effects of quicksort without recursion is provided.

### **3.2.2 Pivot Position Vs Input Distribution**

For any sorting algorithm, there will be runtime of best case, average case and worst case. likewise, the quicksort have all the three cases of runtime. The factors that affect the runtime and prediction, are the pivot position that we choose and also the input distribution.

#### **Pivot Position:**

The choice of pivot can make significant difference in the speed of the algorithm,

branch prediction. for every choice of pivot position, there is some pros and cons.

##### 1.First element as Pivot:

Its clearly sub-optimal. Assume the input is in decreasing order, then the pivot will be the biggest, so the whole array will move to the left of it. Then for every recursive call, the first element (Pivot) will be the biggest again, therefore once again the whole array will move to the left of it. And, it goes on and on...

##### 2. Last element as pivot:

Be it any end, it has the same result as the first element as pivot. It also has the sub-optimal solution.

##### 3.Median as pivot:

When the median is the pivot and when the prediction happens, even during the best case, only 50% will be right decision and not more. This also wastes the cpu-clock for the instructions that are not executed.

##### 4. Random element as pivot

Choose the pivot in such a way, quarter of all elements will be on the left side of the partition. Then the CPU could guess "right" all the time. Best case- 75% correct prediction. This results in a significantly reduced number of branch misses.

#### **Input distribution:**

The sorting performance may vary according to the input distribution. The input distribution may be

- Random order input
- Decreasing order
- Already Sorted input

### 3.2.3 Quicksort - Context switching problems

The performance of sorting varies with the quicksort that has no recursion call. In the quicksort with recursion method, there is overhead of recursive calls and consequent context switching. The recursive calls will have to wait in the pipeline till the execution on one call is finished. This decreases the runtime efficiency of the quicksort.

A context switch (also sometimes referred to as a process switch or a task switch) is the switching of the CPU (central processing unit) from one process or thread to another.

A context is the contents of a CPU's registers and program counter at any point in time. A register is a small amount of very fast memory inside of a CPU (as opposed to the slower RAM main memory outside of the CPU) that is used to speed the execution

of computer programs by providing quick access to commonly used values, generally those in the midst of a calculation. A program counter is a specialized register that indicates the position of the CPU in its instruction sequence and which holds either the address of the instruction being executed or the address of the next instruction to be executed, depending on the specific system.

#### The Cost of Context Switching:

Context switching is generally computationally intensive. That is, it requires considerable processor time, which can be on the order of nanoseconds for each of the tens or hundreds of switches per second. Thus, context switching represents a substantial cost to the system in terms of CPU time and can, in fact, be the most costly operation on an operating system.

### 3.2.4 Parameters affecting sorting performance

#### Number Of Instructions:

In quick sort, compare and swap are the major operations we perform. Number of instructions executed are proportionate to number of comparisons done. When we use median as pivot the number of comparisons will be less obviously because each recursive call processes a list of half the size. Consequently, we can make only  $\log_2 n$  nested calls before we reach a list of size 1.

#### Branch mispredictions:

Performance of quick sort is more dependent on the number of branch misses taken place because more the branch mispredictions, more the time taken by the program to complete its execution.

Non-median pivots are beneficial if we are unconcern about the run time of the program.

## 4. Experimental Results

### 4.1 Machines and tools handled

The experiments have been performed in HYDRA, a multicore cluster architecture.

Hardware of HYDRA has cluster of 16 nodes (1- Master node, 15 compute nodes) works on CentOS release 6.7 operating system, and uses Gigabit ethernet and InfiniBand for cluster interconnection. Each node(node 0 to 15) has 2- 6 core processors.

#### Features of HYDRA:

core- 268 CPU cores

Memory- 544 GB RAM

Storage- 7 TB disk space

The tools used to analyze the performance of program execution was "perf".

Profiling is a method used to analyze the execution of a program by measuring cpu performance, trace points, branch execution etc. "perf list" command is used to list all the currently known hardware and software events.

#### Sample Profiling:

**perf stat -e branch-misses bpsh 1 ./filename**

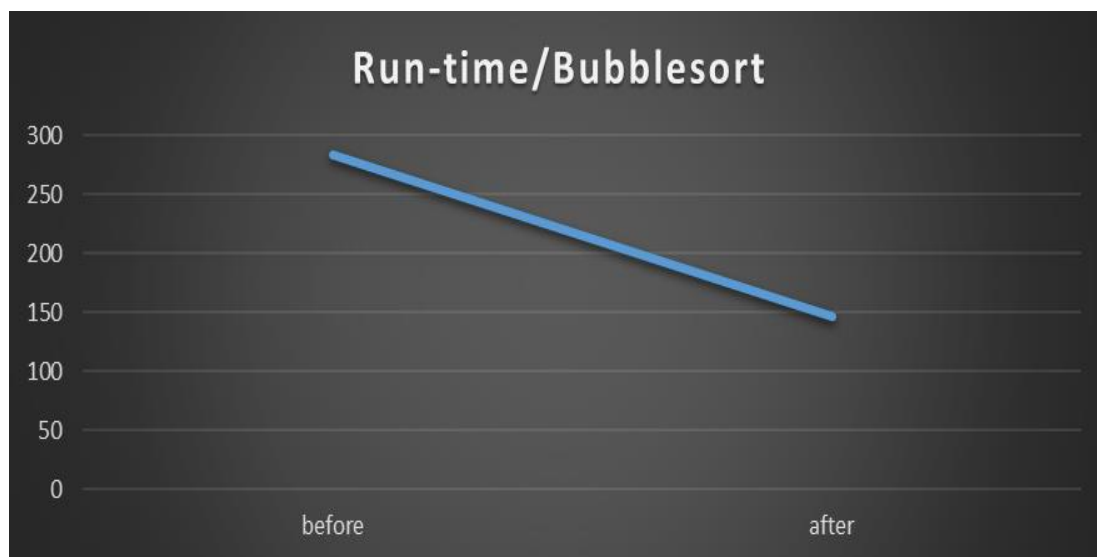
stat => to obtain event counts.

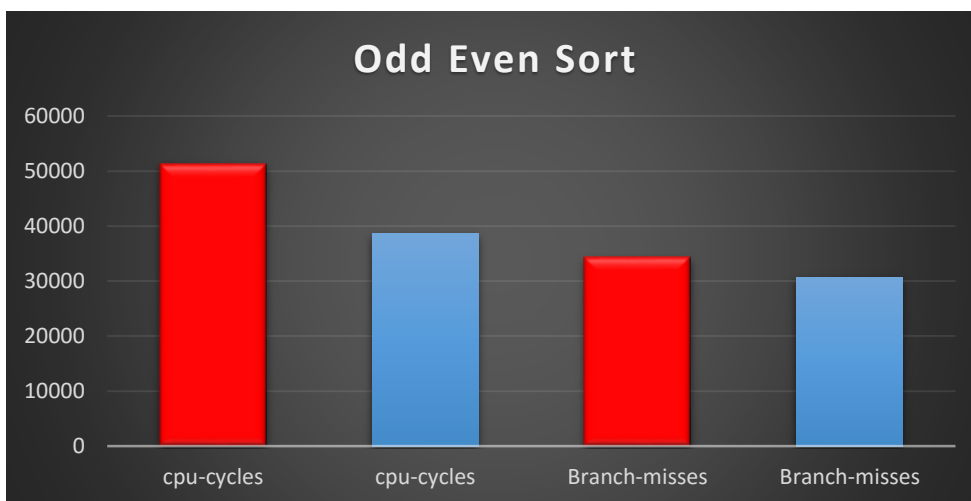
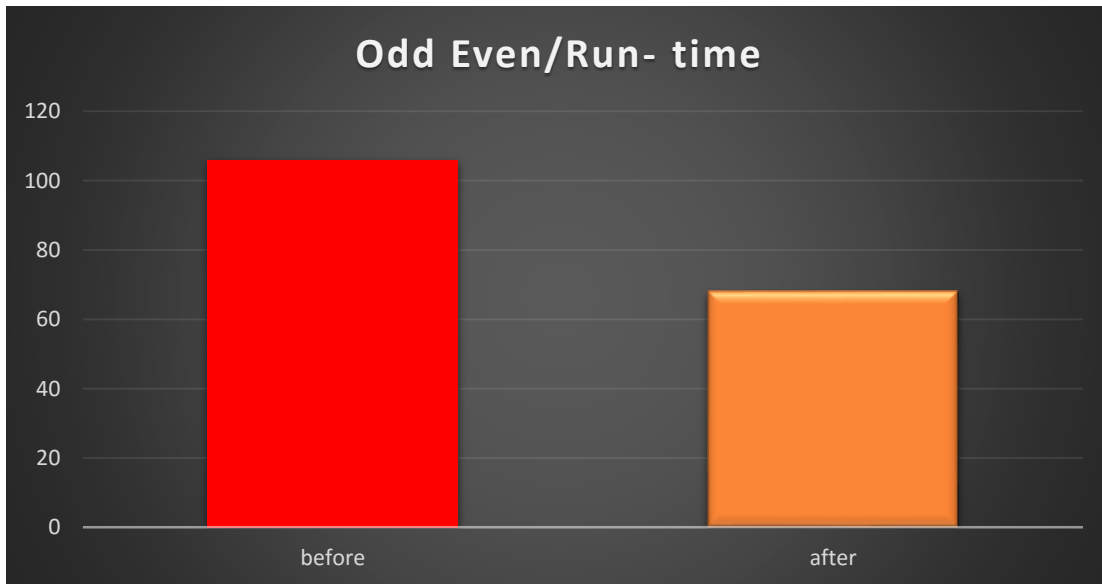
-e => to denote the event



then the syntax for the particular node, the program is run with the arguments to get the required parameter to be profiled.

### 4.2 Analysis of Performance Metrics

#### 4.2.1 Re-Timing Technique





- **Before Retiming** 
- **After Retiming** 

#### 4.2.2 Skew the Pivot

Following values were collected after running programs on hydra:

##### Branch Misses:

<i>Pivot Element/Input</i>	Ascending Input	Descending Input	Random Input
Random Element	29691	30808	32772
Median Element	31061	33178	33619
First Element	32999	33108	32986
Last Element	32759	33838	33262

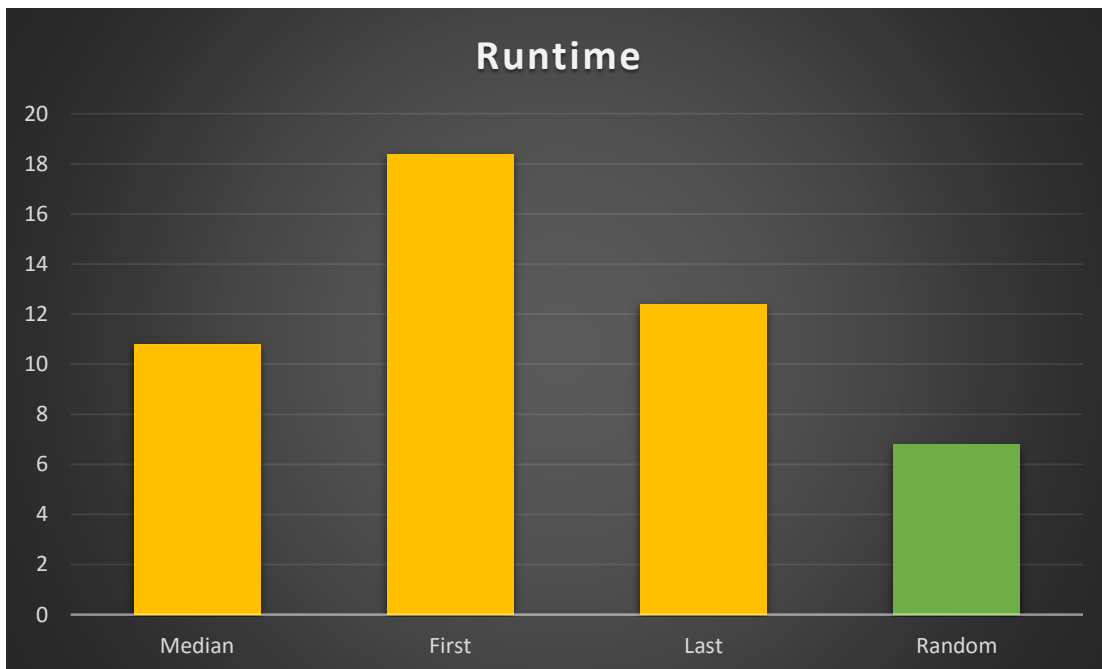
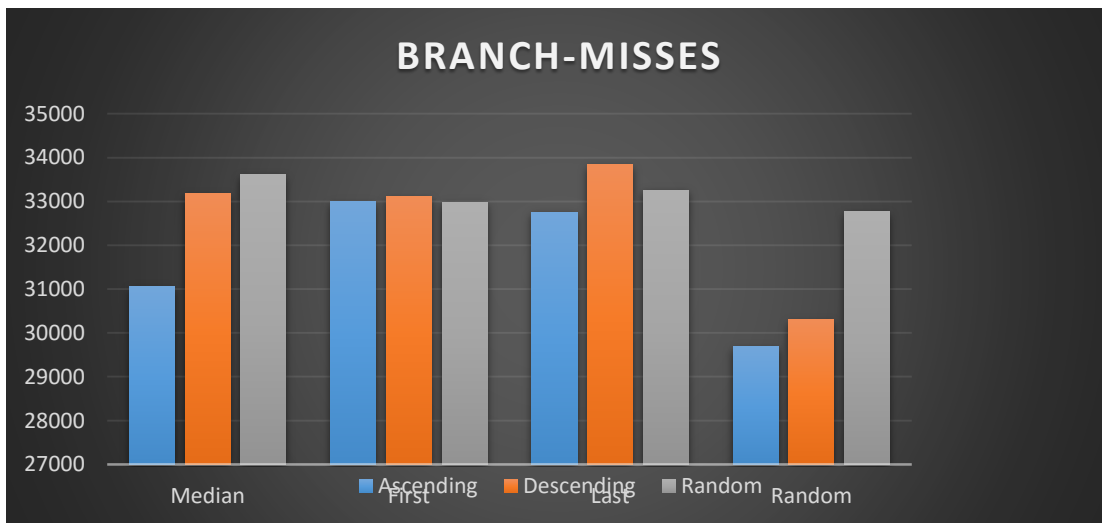
##### Instructions:

Random Element	2285330
Median Element	1948099
First Element	2265168
Last Element	2218465

##### Run-Time:

Random Element	6.8 milliseconds
Median Element	10.8 milliseconds
First Element	18.4 milliseconds
Last Element	12.4 milliseconds







## 5. Conclusion

### 5.1 Retiming Techniques

Loop transformation techniques like retiming are important in improving the quality of the loop constructs by introducing parallelism into the loop execution. We applied retiming techniques using new constructs that are predicted if, predicted auxiliary if, predicted ultimate if. These new instruction constructs improved the branch prediction accuracy in loops with no loop carried dependencies.

From our results we have observed significant reduction in number of branch misses after applying the retiming techniques. Run time is also reduced because runtime of a program is directly proportional to the number of branch misses occurred while running the program.

### 5.2 Skewing The pivot

From the result of our project we have observed the following things:

#### Median Pivot:

- Its better when comparisons are costlier. When we use median as pivot the number of comparisons will be less obviously because each recursive call processes a list of half the size. Consequently, we can make only  $\log_2 n$  nested calls before we reach a list of size 1.
- It takes more run time because the branch misses will be more.

Reason: Let us consider the branch prediction unit of CPU. If the pivot is the median, then there is a 50% chance that the next element will belong to the left side of the partition and a 50% chance that it will go right. In other words, the branch predictor will have to guess, and it will be wrong half of the time and will have to flush the pipeline. This leads to increased number of branch misses, that costs a lot of time.

#### Random Pivot:

- Its better when run time is a critical factor, because from our observations when we choose random element as pivot the run time of the program was lessened considerably.
- The reason why the runtime is reduced is:  
Since the pivot is selected randomly, let's assume the pivot element as quarter element of the array, that means now we are partitioning array as  $\frac{1}{4}$  th and  $\frac{3}{4}$  th parts. Therefore there is 75% chance for the next element to be on right side of pivot. In other words the branch predictor will guess correct 75 times correctly out of 100 times. Which concludes that the number of branch misses are reduced thereby reducing the runtime of program.

### **First and Last Elements as pivots:**

They are considered as worst case scenarios in estimating the performance of quicksort. All the parameters run time, branch misses and instruction count, turns out to be large in case of choosing last and first element as pivot. One of the reasons is that quick sort belongs to divide and conquer family of algorithms, but when we chose first, last elements as pivot positions then there is no chance of dividing the input array which is making the algorithm loose its basic quality of divide and conquer and because of this now we need to compare the pivot with all the other elements of the array sequentially which increases the number of comparisons to be done and also the runtime.

### **6.0 What we have learnt**

The goal of our project is to improve the performance of the loop construct by applying retiming techniques. We have considered parameters like run time and branch misses to illustrate the improvement of the quality of loop construct after applying the retiming concept. We have learnt how different logics written by the programmer will affect the hardware parameters like branch mis-prediction, CPU cycles. We have also made a point that software and hardware performances are inter-related. As a part of our project we have implemented three new constructs called predicted if, predicted ultimate if and predicted auxiliary if. We have observed that simple for loops have limited performance, but after applying the algorithm (identified pif, puf and paf statements in the for loop construct) the performance of for loop was improved because of the removal of iterational dependencies using paf , pif, and puf constructs.

We also worked on the effect of branch miss-predictions on the performance of the quick sort algorithm. Based on the priorities we have at our hand , like if we are much concerned about run time of the program its beneficial to use random element as pivot and if we are much concerned about the cost due to swaps its better to use median as pivot.

## REFERENCES:

1. Hoare, C.A.R.: Algorithm 64: Quicksort. Commun. ACM **4**(7) (1961) 321
2. Knuth, D.E.: The Art of Computer Programming—Sorting and Searching. Volume 3. Addison Wesley (1973)
3. Martínez, C., Roura, S.: Optimal sampling strategies in Quicksort and Quickselect. SIAM Journal on Computing **31**(3) (2002) 683–705
4. Sanders, P., Winkel, S.: Super scalar sample sort. In: 12th European Symposium on Algorithms (ESA). Volume 3221 of LNCS., Springer (2004) 784–796
5. Brodal, G.S., Fagerberg, R., Moruz, G.: On the adaptiveness of quicksort. In: Workshop on Algorithm Engineering & Experiments, SIAM (2005) 130–149
6. Brodal, G.S., Moruz, G.: Tradeoffs between branch mispredictions and comparisons for sorting algorithms. In: Proc. 9th International Workshop on Algorithms and Data Structures. Volume 3608 of Lecture Notes in Computer Science., Springer Verlag, Berlin (2005) 385–395
7. Patterson, D.A., Hennessy, J.L.: Computer Architecture: A Quantitative Approach 3rd. ed. Morgan Kaufmann (2003)
8. Sedgewick, R.: Algorithms (Second Edition). Addison-Wesley Longman Publishing Co. (1988)
9. Loop transformation: [http://www.omegacomputer.com/staff/tadonki/PaperForWeb/tadonki\\_loop.pdf](http://www.omegacomputer.com/staff/tadonki/PaperForWeb/tadonki_loop.pdf)
10. Quick sort : <http://www.cs.fsu.edu/~breno/COP-4530/slides/21-anim.pdf>  
<http://www.cplusplus.com/faq/sequences/sequencing/sort-algorithms/quicksort/#pivot-choice>
11. vector header : <http://www.cplusplus.com/reference/vector/vector/>,  
<http://www.cplusplus.com/reference/vector/vector/vector/>
12. Computer architecture- a quantitative approach, John L. Hennessy and David A. Patterson