Improved Optimum Dynamic Time Slicing CPU Scheduling Algorithm based on Round Robin approach

Conducted By:

MD. Shihab Ullah - 11201047

Supervised By:

Dr. Jia Uddin

Assistant Professor

Department of Computer Science and

Engineering

BRAC University

Declaration

This is to certify that this final thesis report entitled ***'Improved Optimum Dynamic Time Slicing Round Robin Algorithm based on Round Robin approach'*** is submitted by the authors for the purpose of obtaining the degree of Bachelor of Science in Computer Science. We hereby declare that all the instances of work presented in this thesis are original and inspirations for the work that we have made use of have been duly accredited with proper referencing.

Signature of Supervisor	Signature of Author
Dr. Jia Uddin	MD. Shihab Ullah

Abstract

In time-shared systems, selection of the time quantum plays a pivotal role in performance of CPU. In this paper, the static use of dynamic time quantum as CPU Time Slice is reviewed and a new algorithm for CPU scheduling named Improved Optimum Dynamic Time Slicing Round Robin Algorithm (IODTSRR) is proposed for process and thread scheduling. The proposed algorithm is based upon dynamic nature of allocation, calculation of the value of time quantum which varies according to the state of queue along with the capability of executing ready processes arriving at the same or different time. The concept of multi-threading by using Dummy Thread is introduced to hold the added processes in the queue during all arrival time intervals respectively. The performance is compared with Optimum Dynamic Time Slicing Using Round Robin (ODTSRR) and the results revealed that the proposed algorithm is much better specifically in response time and turnaround time. As process gets fully or partially executed while others arrive simultaneously, the context switch rates, waiting time and throughput improves hence resulting in optimized CPU performance.

Keywords — scheduling algorithm; randomized control trial; time quantum; context and thread switching; response time; turnaround time; waiting time; fairness; multi-threading; synchronization; arrival time interval; dynamic queue; first come shortest job first (FCSJF); improved optimum dynamic time slicing round robin algorithm (IODTSRR); optimum dynamic time slicing using round robin (ODTSRR)

Acknowledgements

First and foremost, unparalleled recognition and appreciation to Almighty Allah (SWT) who gave me the opportunity, determination, strength and intelligence to complete my thesis with gradual hardship under Brac University.

Due to sole guidance, continuous supervision and consultation, special thanks to Dr. Jia Uddin for laying the foundation for our thesis concept along with its development process.

For careful notifications and responses, sincere gratitude are also due to honorable faculties Mr. Rubel Biswas, Amitabha Chakrabarty, and Hossain Arif respectively.

Furthermore, due to consistent but unflinching support and encouragement, I would really like to express my over-whelming indebtedness to my parents Mrs. Shahida Akhter and Md. Baki Ullah. Also few peers and friends who were worth mentioning for desired result and completion throughout the whole time.

Table of Contents

Chapter 1
Introduction

1.1 Background

Process Management is one of the mandatory and fundamental tasks for operating system of any given platform to run different applications. It sententiously depends upon CPU scheduling algorithms which derive the overall extent of performance of operating system. In single processor system, non-preemptive scheduling were used, i.e, only one process can be executed at a time, any other process or processes must wait until the CPU becomes free. Operating systems today, are moving towards multitasking environments due to emergence of running multiple process of different level as per usage. Thus, multi-level programming for process management are implemented to maximize CPU utilization by having some processes running all the time. But imprudent use and allocation of the CPU can dwindle the efficiency of system in multiprogramming environments. More than one processes are being kept in memory to achieve maximum CPU utilization. As a result, process scheduling still remains an elemental activity of fulfilling the purpose of the operating system. CPU scheduling is imperative because it can have immense impact on CPU utilization, overhead and inclusive performance of the system. Scheduler requires conscientious consideration to ensure fairness and avert process starvation in the CPU. This allocation involves a scheduler and dispatcher of long-term, middle-term, short-term basis respectively.

Since the era of multi-programming and multi-threaded scheduling, the default Round Robin algorithm and its various modification has been eventually used for CPU as non-monotonic scheduling in newer versions of operating system platforms such as Windows, Linux, Unix, AIX, Mac-OS respectively. There exists no “pure” form of scheduling algorithm or RR variations which contemplated lack of efficient queue handling and implementation of core multi-threading programming (where processes are executed like threads) etc. As a result of trivial queue implementation and conventional modification, the existing RR variations tempted to increase CPU overhead, idleness while scheduling in different types of processes and system-cases. So unlike previous RR variations, the attained time-slicing CPU scheduling algorithm will be able to remove the kernel-based complexity of different schedulers by simultaneously scheduling and executing ready processes.

1.2 Motivation

This section discusses three key aspects and factors which essentially motivated me for research and experimentation for unique but effecting CPU scheduling approaches.

1.2.1 Better CPU time slicing

As per my research and knowledge, till now, the median which has been calculated trivially has been outperformed the other way of calculation for allocating CPU time slice unit. But in different input cases, overall performance are negligible due to imbalance of amount of quantity of time quantum. So I searched for better efficient way to calculate the "perfect time quantum" which will be ultimately faster but effective way to calculated and allocate CPU time slices respectively. Thus, ensuring either lower-cost or less amount of run-time while searching for perfect time quantum ensuring more efficient execution.

1.2.2 Potentiality of secure multi-threading programming

There is prospective to use secure multi-threading where resources are created, used and disposed within the thread i.e. concurrently executing multiple processes along with that of scheduling respectively. Thread switches cost lesser than context switches and can be faster without falling in dead-lock conditions. Hence, it can be useful in maximizing overall performance of scheduling and execution of processes arriving in different time by using only one synchronized and dynamic queue instead of separate ready queue and blocking queue.

1.2.3 Minimizing context switches costs and performance consistency

Due to multi-processing architecture and development of resourceful computability in the operating system, I find large possibility of more consistent performance with careful and intelligent memory allocation and data structures. Despite of different variation of input cases, hyper-threading can bring notable improvement in decreasing the total time for scheduling and running processes simultaneously using modern 64-bit processor architecture. On the other hand, thread switches are faster and cheaper making the cost and amount of context switches between two processes non-effective in case of longer lists of processes with high burst time.

1.3 Challenges and constraints

Initially, I faced some minor but general challenges while conducting my research analysis regarding study materials and implementation. Following are the indications narrating and discussing the constraints.

1.3.1 Scarcity of test cases

Generally, CPU scheduling algorithm are analyzed and experimented with as usual input or test cases derived from various authenticated and followed textbooks related to the study of Operating Systems. All the input cases are lists of one of more processes with its given ID#, amount of burst time and amount of arrival time in specific unit respectively. Due to few datasets, I also had to consider those trivial test cases along with those of different articles which were used for their particular experimentation as per similarity of the input case structures. Otherwise, the availability of most of the possible types of input cases would not be established for more thorough analysis.

Despite of that, I have also created some of my own sample input cases derived from above stated resources for more solid and further comparative experimentation of outcome.

1.3.2 Handling concurrent scheduling and programming complexities

Most of the previous implementation and simulations of state-of-the-art RR variations were handling either processes arriving at the same time or via multi-programming instructions to the default queues for those arriving at different times respectively. Hence, urge of necessity of multi-threading programming for cost and resource effective execution of processes seemed viable and implementable for next generation CPU scheduling algorithm. Despite of its programming advantages, handling the con-current situation, such as, scheduling a process in the dynamic queue accordingly while executing or continuing allocating CPU time slices for running processes simultaneously entering the critical section code.

Programming a RR variation for time-shared multi-core system not only requires a deep understanding of the hardware architecture of the system, but also careful tailoring of the program to that specific hardware [16]. The coding implementation for exact simulation were tough due to continuous but dynamically required trade-off between run-time and space complexities while considering enough thread-safe resource management and its correctness as per algorithm.

1.3.3 Balancing process prioritization of processes

Prioritization of processes has been crucial while effectively scheduling and completing execution of process in lesser time. It also significant in decreasing CPU overhead and inner starvation of the processes respectively.
Previously, proper as well as ultimately required priority of processes while scheduling were balanced via variations of priority queue scheduling. The other extension and due versions of RR algorithms at best give priority to the processes arriving in ascending order.
But it does not necessarily balance the priority of the process list as longer ones arriving at the same time get equal priority and effect overall waiting and turn-around time. Thus, it consecutively provides enough room for efficiency by coming up with a fixed and consistent standard of process-ordering while maintaining fruitful delivery of faster performance.

Chapter 2
Background Analysis

2.1 Overview

The following section elaborates various nomenclature related to CPU scheduling algorithm for further conceptual clearance.

2.1.1 Scheduling Nomenclature

It is necessary to be familiar with different scheduling terminologies defined below: [1]

Ready Queue: The processes which reside in Main Memory and waiting for the CPU time are put in a queue called ready queue.

Concurrent Queue: A ready queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.

CPU Utilization: It is defined as the amount of time CPU is in use. Maximizing CPU utilization is usually the aim of any scheduling algorithm.

Context Switch: Context switch is a process of keeping and restoring context of a pre-empted process, so the execution can be carried on from the same position at later time. Context switching is wastage of time and memory which results in increase in overhead of the scheduler.

Turnaround Time: It is defined as the total time which is used to complete the process, from entering in to the ready queue till its complete execution.

Waiting Time: It is defined as the total amount of time a process waits in ready queue.

Response Time: It is defined as the time consumed by the system to give first response to a particular process.

Starvation: It means the long process blocks the way of short process vice versa and the higher priority process out run the lower priority processes.

Priority: Give preferential treatment to processes with higher priorities.

2.1.2 Scheduling Metrics

Characteristics of good scheduling algorithms are mentioned as follows [1]:

- Minimum CPU overhead, number of context switches and waiting, turnaround, response time.
- Maximum CPU utilization and throughput.
- Avoid indefinite blocking or starvation.

  Enforcement of priorities

2.2 Literature Review

Round Robin is the simplest, fairest and most widely used scheduling technique in timeshared systems. Use a fixed time slice for scheduling also known as time quantum. It choose process from head of ready queue and run that process for at most 1 time slice, and if it is not completed, add it to the end of the ready queue. If that process terminates or blocks before its time slice is completed, choose another process from the head of the ready queue, and run that process for at most 1 time slice. It achieves the fairness of resource allocation and result in minimized response time as compared to the Shortest Job First and First Come First Serve algorithms. But, due to the static time quantum concept it increases the turnaround time and waiting time resulting in dilapidation of system performance. Response time is good for short

processes, while long processes may have to wait. Fairness factor which penalizes I/O-bound processes (may not use full time slice). Starvation is not possible as every process is getting the equal share of time, and the CPU Overhead is low. [1]

Aashna Bisht et al. [1] proposed Enhanced Round Robin algorithm (ERR). ERR allocates CPU to a process for designated time quantum after the completion of which, it checks the remaining CPU burst time of the process currently in execution, if the remaining CPU burst time of the currently running process is less than (average burst time/time quantum) value, then CPU is again allocated to the currently running process for remaining CPU burst time.

Rami J. Matarneh et al. [1] proposed an algorithm named "Self- Adjustment Time Quantum in Round Robin Algorithms Depending on Burst Time of the Now Running Processes algorithm the time quantum is repetitively adjusted according to the burst time of the currently running processes using median.

Lalit Kishor and Dinesh Goyal [1] proposed median based round robin algorithm. This algorithm is a blend of two techniques, the processes are arranged in ascending order first, and then the time quantum is set according to the value of median.

H.S. Behera and Brajendra Kumar Swain [1] proposed an algorithm named "A New proposed precedence based Round Robin with dynamic time quantum scheduling algorithm for soft real time systems" in which precedence value is allocated to all the processes according to their priority and burst time. RR algorithm is then applied on the processes on the basis of their precedence. This Proposed algorithm is developed by taking dynamic mean time quantum in to account. Time quantum is computed dynamically by taking the mean of priority values and burst times.

Ali Jbaeer Dawood et al. [1] proposed an algorithm "Improved Efficiency of Round Robin Scheduling Using Ascending Quantum and Minimum-Maximum Burst Time" in which processes were arranged in ascending order with shortest remaining burst time and calculated the time quantum by multiplying the average summation of minimum and maximum burst time by (80) percentage. The (80) percentage is chosen depending to two reasons: First, if the TQ calculated depending only on the summation the algorithm is become as the Short Job First (SJF). Second, the rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.

Radhe Shyam and Parmod Kumar [3] on the article "Improved Round Robin with Shortest Job First Scheduling" proposed an algorithm combining Round Robin with shortest Job first scheduling. The TQ studied to improve the efficiency of RR and performs degrades with respect to context switching, Average Wait Time and Average turned around time. The processes were ascending with shortest remaining burst time and then TQ are given to that ascending process to CPU and also continue to allocate TQ again if the remaining burst time is less than 1 time quantum. While ready queue is not empty or any new process enter in the queue, the execution is regulated. The proposed algorithm (IRRSJF) performs better than Round Robin (RR), Improved Round Robin (IRR), FCFS and some other scheduling algorithm in terms of reducing the number of context switches, average waiting time and average turnaround time.

Anju Muraleedharan et al. [4] lodged an article "Dynamic Time Slice Round Robin Scheduling Algorithm with Unknown Burst Time" where their approach mainly focuses on how round robin will perform if the processes burst times are unknown at the beginning. They

propose a refinement to simple RR by altering the time quantum while execution. First and foremost, put a small value to initial time quantum, and carry through the first cycle with this time quantum. In succeeding cycles, they multiply the time quantum by two if no processes finished its work. It will scrutinize the number of processes completed in each cycle. If at least one of them is completed, then continue the next iteration with an unchanged time quantum. By this method, some of the problems with static time quantum are partially solved. If the time slice used is larger, then the average waiting time will reduce, in normal cases. But a factor that affects the average waiting time of the RR is the arrival time of the jobs – if several new processes arrived in during the execution, then it may cause an increase in the average waiting time.

Wasim Firuj Ahmed and Sahana Parvin Muquit [5] brought up a new method in their article named "Improved Round Robin Scheduling Algorithm with Best Possible Time Quantum and Comparison And Analysis with the RR Algorithm" to find the best possible time quantum to make the traditional Round-Robin algorithm an efficient one. Instead of usual fixed time quantum, calculated value of the time quantum unit becomes a rounded-up magnitude of the square root of the multiple of the median and highest burst time respectively applied in RR algorithm.

Mohammad Salman Hafeez and Farhan Rasheed [1] proposed an algorithm in "An Optimum Dynamic Time Slicing Scheduling Algorithm Using Round Robin Approach" article where the time quantum is determined dynamically with median value and the continuity of the execution of a process with the remaining burst time lesser than the time quantum set by the median value.

Following that, an article paper called "An Enhanced Round Robin CPU Scheduling Algorithm", Jayanti Khatri [6] proposed an algorithm is similar to traditional Round Robin algorithm with a small improvement. The proposed algorithm (ERR) allocates the processor to the first process of the ready queue for a time interval of up to 1 time quantum. Then it checks the remaining burst time of the currently running process and if the remaining burst time is less than or equal to 1 time quantum, the processor again allocated to the same process. After completing the execution, this process is removed from the ready queue. If the remaining burst time of the currently running process is longer than 1 time quantum, the process will be added at the tail of the ready queue.

Omar Hani Mohammad Dorgham and Dr. Mohammad Othman Nassar [7] came up with attempts to introduce an alternative method in RR algorithm in the paper "Improved Round Robin Algorithm: Proposed Method to Apply SJF using Geometric Mean" which combines two algorithms together and calculate a dynamic time quantum using the geometric mean method to enhance the CPU utilization and minimize the waiting and turnaround time in CPU scheduling. Geometric Mean can be calculated by applying the nth root of the product of n numbers, where these numbers is considered as the burst times of the processes, and then take the ceil of the result as a time quantum. On the other hand, the proposed algorithm will take into consideration two cases, first one, when the processes have an arrival times, and the second case, when there is no arrival time. Algorithm will depends on the arrival time, and here the proposed algorithm applies the second algorithm which it is First Come First Serve to take the proper process, where the first process enter the ready queue will be execute first.

Accordingly, if there exist a process which need more than 1 time quantum, the remaining burst time will be compared with the other processes burst times by applying the SJF algorithm to choose the proper process to start execution in the CPU. Without Arrival Time, the algorithm applies the SJF only to choose the lowest burst time between all processes, where the shortest burst will enter the CPU for the execution and if the lowest burst time needs more than 1 time quantum, the remaining burst time will be compared between other burst times by applying the SJF algorithm, then it will choose the lowest burst time again.

Chapter 3
Research Design and Feasibility

3.1 Research Methodologies

While constructing and developing the quantitative research, I have analyzed the journals publishing various latest and also significant RR variations especially along with ODTSRR algorithm. Primarily, I begin testing hypotheses derived from theory and/or being able to estimate the size of a phenomenon of interest. Hence, my long-term research initiatives were followed by considering ODTSRR as randomized control trial and denoted the main factors such as experimental, control group, along with data sampling process etc.

Based on time-to-time research and analysis, I took attention to the key focus, intervention, control of variables derived and followed Quasi-experimental approach as the quantitative research method. After repetitive pre- and post-test study designs, studies determined those and other fundamental blocks for proving overall improvement in performance of ODTSRR initially but along with being thorough with other RR variations accordingly.

Independent variables falls under the experimental group refer number of processes, total arrival time, arrival and burst time of each processes etc. Also potential constituents like con-current execution of processes in a single scheduling queue, way of calculating and allocating time quantum, following standard protocol for prioritization of processes are notable for comparative research and experimentation.

On the other hand, control group consisted the constant flows of fixed and required variable such as the flow of IF and ELSE statement along with the static variable stored in register allocating time slices, null checking of ready queue, variables holding arrival and burst time of processes respectively.

After in-depth observation and calculation of various types of sample datasets and input cases gained from previous relevant researches, I were able to accumulate some amount of reliable experimental data with coherence and patterns for building numerous test cases for analyzing research outcomes. Despite some randomness in data sets used for several trials, most data were collection on basis of situational characteristics. Convenient sampling and experimental modification through some of the datasets are performed longitudinally to statistically control and analysis the underlying influences on the dependent, or outcome, variable.

Consequently, proposed algorithm has been programmed via Java SE language for fast and accurate simulation of each and every test case, in the spirit of rigorous comparison of outcomes of different metrics measuring performances.

3.2 Overview of Feasibilities

Following section emphasize various factors which denotes the overall feasibility of this research and experimentation.

3.2.1 Theoretical Feasibility

Feasibility of the research theory mainly pertained to consider major factors of study aiming consistent and better performance. Specifically the following figure shows that most of the processes to be scheduled have shorter burst time i.e. less than 10 milliseconds respectively. Proper resource and time slice allocation may impose probability of more efficiency in scheduling processes in fixed priorities.

While applying Little's Law in series of test case simulation, IODTSRR sounds highly feasible for scheduling, better than other RR variations respectively. Also by avoiding process starvation, no aging required due to fixed standard of prioritization. Thus, balancing resources with minimum possibility of waiting in queue unnecessary which is promising for lowering response and waiting time.

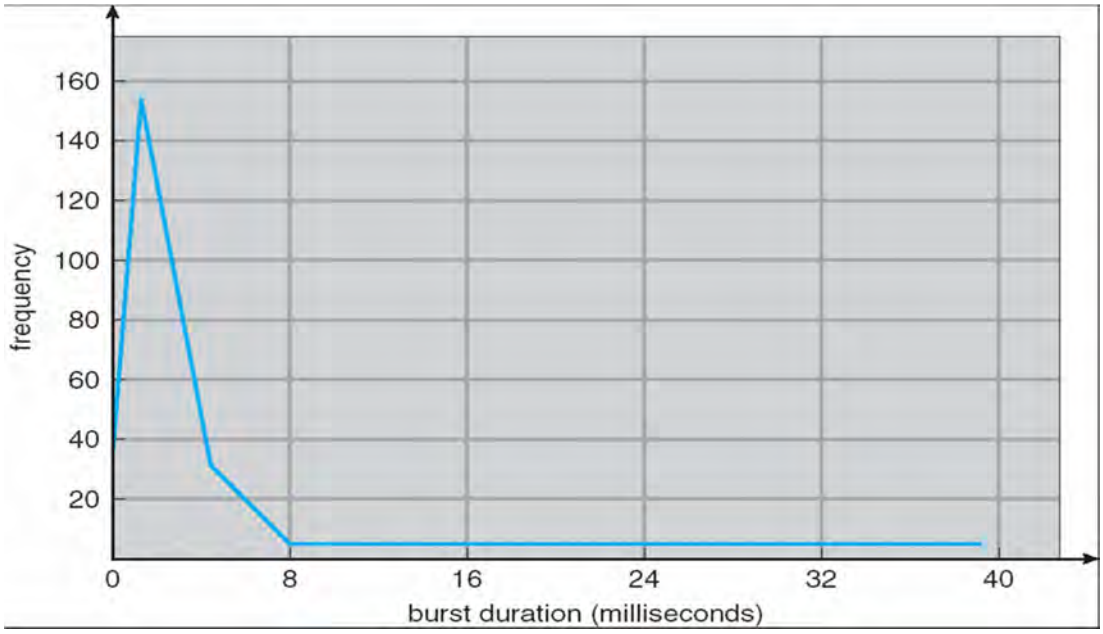

Fig 1. Histogram of CPU Bursts Duration

3.2.2 Programming Compatibility

For running parallel applications, the scheduler may use explicit thread and data placement to achieve the best performance.

Most thread library implementations provide support for pinning threads to assign threads to specific CPUs (i.e., hardware threads) and to restrict their migration. Thread libraries on operating systems such as both Windows and POSIX, etc. give clients some control over thread scheduling. Since the architecture of many-core systems is still evolving, portability is needed to allow the same program to run well on different kinds of many-core systems [16].

3.2.3 Technical and Operational Plausibility

The proposed method is derived to be implemented either totally in software, totally in hardware, or as a hardware/software combination. It has potential for targeting single CPU architectural platform for many-core processors.

While running in multi-threaded environment, the proposed method have feasibility of being flexible and independent of individual kernel configurations. It provides capability for user-level memory management, I/O, interrupt handling tasks etc on behalf of kernel.

For Windows kernel, the algorithm can be plausible for implementing in a Windows kernel synchronization mechanism in IPC (Inter-Process Communication) of WinFSP as it provide similar characteristics to KQUEUE i.e. kernel portion of I/O completion ports. By profiling with xperf, system threads were handled for transitioning from the signaled to non-signaled state by EventSet and EventWait synchronization events. The processed scheduler may impose its dynamic queue mutually with KQUEUE along with core I/O Queues which must provide additional services, such as IRP cancelation, IRP expiration, etc. Proper configuration can be done with independence of re-setting compiler-internal and library operations, dynamic and static linking with linking library.

Kernels in UNIX or LINUX distribution have easy approach to implement any new or custom scheduling algorithm. In this cases, the feasibility may be ensured by using any of the implementation pattern. There are only three files in the default directory kernel/sched which are : core.c , debug.c , fair.c etc. They are responsible for dispatching tasks i.e. both processes and threads as they are treated as same in the Linux kernel scheduler.

System Administrator may have create new .c file and attempt to duplicate the key functions such as CONFIG_SMP, CONFIG_FAIR_GROUP_SCHED etc and implement structure for e.g. sched_class fair_sched_class inside the default fair.c file holding the scheduler which runs CFS (Completely Fair Scheduling) algorithm. Initial functions will have to be wiped away from memory after the kernel have executed them during startup. Then he/she have to set the init() routine as general scheduling class initialization routines to initial modified structures along with the functions. At runtime, scheduling class which is used is configurable - just switching via the use of function pointer, and so inside core scheduling related file.

Chapter 4
Algorithm

4.1 Implementation Framework

Framework assumption is extremely crucial while scoping the design and implemental criterias of any good algorithm wheter its for Scheduling or finding the shortest path from complex graph. I considered maximum traditional implementation in OS kernel and process scheduler as a “whole and complete” scheduling dispatcher rather than most of the CPU scheduling algo according to configuration parameters, process behavior, and user requests.

IODTSRR follows the time-sharing policy for adjusting process priorities to balance the throughput of processes that use a lot of CPU time, while allows task parallelism at high level of abstraction. This results in formulation of more tractable scheduling problems which in classical form are computationally hard.

4.2 Synopsis of Core Design

In broader perspective, the principal requirement from any short-term CPU scheduler like IODSRR is process handling. However, some kernel i.e those of Linux distributions treat processes and thread kind of equally where thread is executed and get allocation of virtual memory spaces as light-weighted processes.

It is core conditionality of the scheduling dispatcher like IODTSRR to be able to provide high-level guidance while balancing the level of abstraction and the amount of control while imposing in Kernel as scheduler respectively.

As mentioned earlier, designing have also considered hardware and software compatibility of single CPU architectural platform for many-core processors. In a single processor system, no kernel process and no time-sharing process runs while a runnable real-time process exists while also having compatibility for multi-processor scheduling along with adjusting itself for concurrent thread scheduling too.

Being immune to I/O interruption, cache-memory loss and management complexities are considered in design development. I also emphasized in developing dynamic but easily configurable structure of algorithm in respect of assigning it to any OS kernel as main scheduler while following Multi-threading and synchronization nomenclature etc.

4.2.1 Abstraction

When the list of processes to be scheduled has same arrival time, time quantum is also calculated dynamically using the proposed calculation of median. The processes are arranged in ascending order with the shortest remaining CPU burst time and placed accordingly in the priority queue. The median value is set as the time quantum which will be the value of CPU Time Slice. If the process is in its execution state and consumed its time slice and its remaining CPU burst time is less than or equal to the time quantum, the CPU will continue its execution till it finishes, Otherwise the process will be placed at the end of the dynamic queue (Here, it is performing like a Synchronized Ready queue). After all the processes in the ready queue are at least once attended by the CPU, it will again sort the process in ascending order with shortest remaining burst time. If a process is suspended by CPU for I/O wait or other reasons, the very same queue will dynamically perform as a Synchronized Blocking queue where the process will be placed in and will stay there until the waiting state is over.

If the list of processes to be scheduled has different arrival times, time quantum for executing processes is calculated differently before and after all the processes arrived in the queue. Let, N be the total time required for the arrival of all the processes and k be the arrival interval of the processes arrived already. At first, add the processes in the queue (one or multiple) arriving in k^{th} interval (where k<=N and initially k = 0). The queue execute those processes based on my modified *FCSJF* algorithm till the next $(k+1)^{th}$ time interval arrives. When the summation of the arrival interval of the processes become equivalent to the total arrival time of all processes, I perform ODTSRR with my modified median to the processes either added or have remaining CPU Burst Time existing in the same queue.

First Come Shortest Job First (FCSJF)

The time quantum for executing the process is also calculated dynamically using the proposed calculation of median. The process ready for the execution are placed in the dynamic queue. At first processes are arranged in ascending order with the shortest remaining CPU burst time on basis of early arrival. After sorting, if the processes is in its execution state and consumed its time slice, the processes only with remaining CPU burst times will be placed at the end of the queue. After all the processes in the queue are at least once attended by the CPU, it will again sort the process in ascending order with shortest remaining burst time respectively.

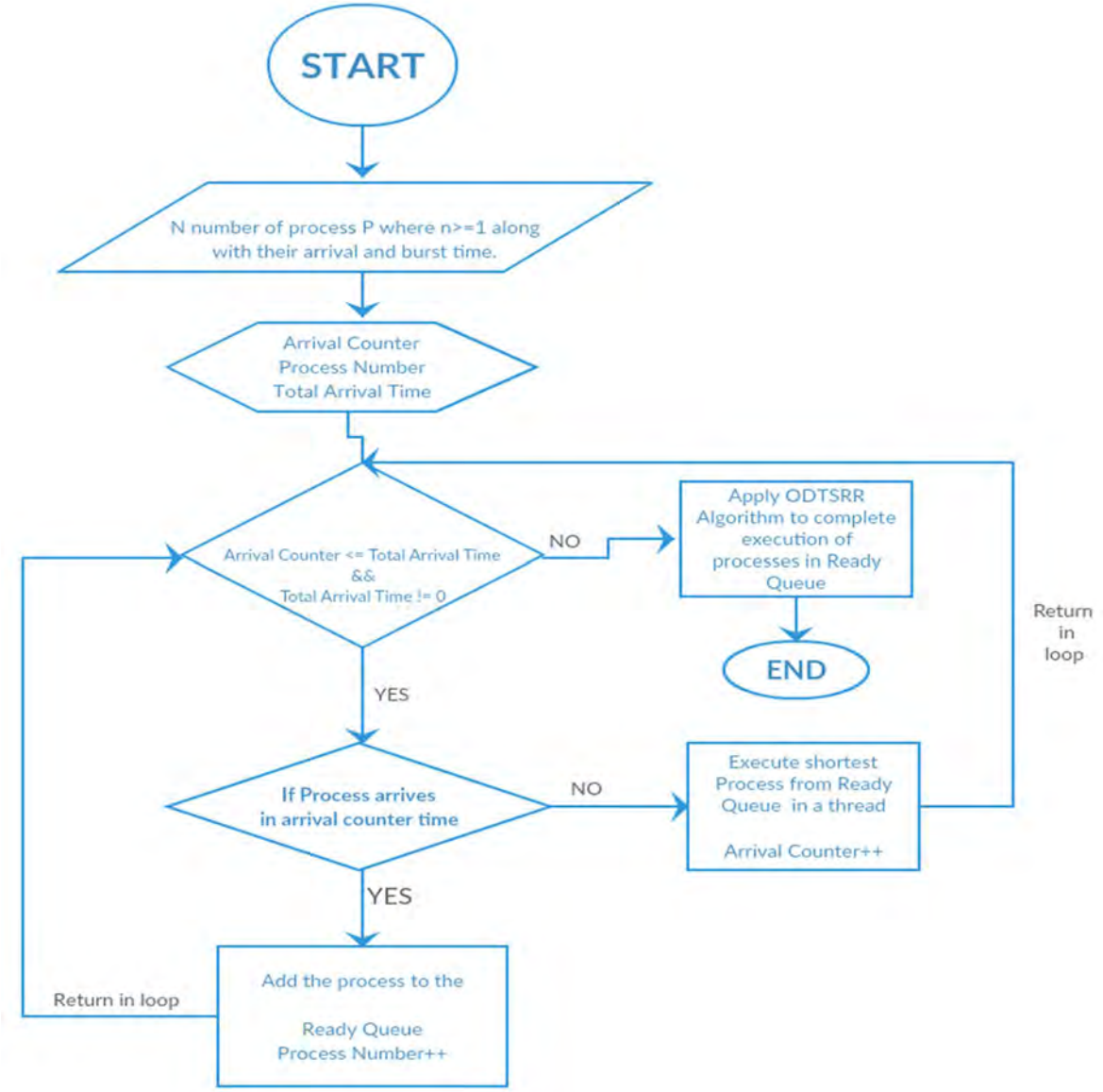

Fig 2. Flowchart of basic design of IODTSRR algorithm

4.2.2 Policies

In this algorithm, the processes are arranged in ascending order according to their burst time's existent upon early arrival in the ready queue. Instead of using static time slices. I also used optimum time slicing approach where heuristic and multi-level programming reformation dynamically change the basis of calculating Time Quantum (i.e. Time Quantum is calculated differently before and after the completion of arrival of all the processes to be scheduled).

For scheduling and executing the ready queue where all the process have same arrival time, I modified ODTSRR algorithm where my proposed modified median is calculated and used only if there exist even number of processes in the queue respectively.

4.2.3 Pseudo Code

Begin

1. **Initialize**, Ready_queue, Number_of_process, Arrival_counter, Total_arrival_time.
2. **IF** (Total_arrival_time**> 0** && Arrival_counter != Total_arrival_time)
3. **While** (Arrival_counter<= Total_arrival_time)
4. **If** (Arrival Time of Process[Number_of_process] == Arrival_counter)

// Number_of_process indicates which process to arrive next

Add the Process to Ready_queue;

Number_of_process = Number_of_process + 1; // *Referring* the next immediate Process

End*of***If**

5. Else

Sort Elements in ascending order of remaining CPU burst time based on the order of arrival

6. If (Ready_queue != **NULL)**

Arrival_counter = Arrival_counter +1;

7. *Start* **Execution** // Processes placed in Ready_queue

Time Slice = ***1;***

8. If (*Remaining Burst Time****>0)***

Place the *process* with remaining burst time at the **End** of the ***Ready Queue***.

End*of***If**

End*of***If**

End*of****Else***

End*of****While***

End*of***IF**

9. *ELSE*

10. While (Ready_queue**!=NULL**)

11. For, *Sort* Elements in Ascending Order

Calculate **Median_Value**

If (Odd)

Select Middle; //Trivial median value

If (Even)

Select ***Ceiling (***Middle + (Middle-1))/2; // *Optimizing* the median value

12. *Start* **Execution** //Ready Queue

Set **TQ = Median_Value;**

13. *For, all Processes Entering CPU*

IF (*Remaining Burst Time* **<= *Time Quantum)***

Time Slice = TQ + Remaining Burst Time // Continued Execution **of the process**

Else

END of Iteration.

Place the *process* at the **End** of the ***Ready Queue***.

End*of*IF

End*of*For

End*of* WHILE

End*of* ELSE

End

The above pseudo code of the algorithm is explained step by step as follows:

Step 1: Initialize the Number_of_process as the serial number of the process, Arrival_counter as a counter of the arrival time intervals (k), Total_arrival_time as the total value of the arrival times of all the processes along with the Ready Queue, where the processes are placed by the Long-term scheduler.

Step 2: In the 2^{nd} step, the algorithm defines an IF statement for a check whether the processes arrives at the same time or not. If multiple processes arrives in different times and also the counter of the arrival time intervals are not equal to Total_arrival_time, then perform the following tasks. Else go to **Step 11**.

Step 3: While the counter of the arrival time intervals is less than or equal to Total_arrival_time, perform the following tasks respectively.

Step 4: In the 4^{th} step, the algorithm defines an IF statement for a check whether the arrival time of the process arrives at the same arrival time interval or not. If yes, we add all the processes arriving in that arrival time interval by referring the Number_of_process to the next immediate process. **Else** perform the following task.

Step 5: *Sort* all the processes in the queue in ascending order of remaining CPU burst time based on the order of their arrival.

Step 6: Next arrival time interval is referred to Arrival_counter. If there are any processes available in the queue, perform the following task.

Step 7: *Set* CPU Time Slice = 1. Here, Time Quantum is not calculated and necessary to refer it as CPU Time Slice. Start the execution of the first process from the sorted queue.

Step 8: *If* the remaining CPU burst time of the process is greater than zero, place the process at the end of the queue.

Step 9: All the processes has arrived. They have been fully or partially executed.

Step 10 to 14: Check whether the queue is empty or not and perform ***IODTSRR*** algorithm accordingly. Here, the previous algorithm (*ODTSRR*) set the *Time Quantum* and *CPU Time Slice* equal to the median value obtained from my proposed calculation respectively.

4.3 Analysis

The following section has huge fundamental research impact in order to proof the correctness and performance magnitude by continuous analysis via thorough observation, comparison, experimentation, simulation. The in-depth analytical insights are discussed in the following sub section for clear understanding.

4.3.1 Algorithmic Analytics

The FCFS is better for a small burst time. The SJF is better if the process comes to processor simultaneously. While Round Robin is better to adjust the average waiting time desired [2], I have considered the above relative insights correspondingly. The processes are arranged in ascending order according to their burst times upon early arrival in the ready queue. Instead of using static time slices. I also used optimum time slicing approach where heuristic and multi-level programming reformation dynamically changes the basis of calculating Time Quantum. For scheduling and executing the ready queue where all the process have same arrival time, I modified and used ODTSRR algorithm where my proposed modified median is calculated and used only if there exist even number of processes in the queue respectively.

It might be considered as modified form of existing Optimum Dynamic Time Slicing Using Round Robin Scheduling Algorithm (ODTSRR). Rather than the use of different queues during scheduling, I used a concurrent dynamic queue as both ready and blocking queue respectively. The CPU Time Slice for process execution is calculated dynamically on basis of complete arrival of processes.

4.3.2 Comparative Findings

One or multiple processes can be scheduled in the queue upon instant arrival while the current process gets fully or partially executed. Unlike ODTSRR, where no other processes can be scheduled in the queue before the execution of the current processes is fully completed, scheduling and execution can be done simultaneously. Due to dynamic nature of the calculation of time quantum and process handling in the queue, lesser CPU overhead and higher throughputs are unavoidable for larger input respectively. Whereas, the performance of ODTSRR decreases significantly and gradually when scheduling processes arriving in different times as both scheduling and execution cannot be done at the same time.

Importantly, in IODTSRR, if multiple processes with same remaining CPU burst time are scheduled, the processes with earlier arrivals will get the chances to be executed foremost. Hence, CPU will be implementing real time enforcement of priorities to always avoid Dead-lock condition and starvation as being functioned like a blocking queue respectively.

4.3.3 Asymptotic Complexity

IODTSRR calculates the value of median in O (n (log n)) in both average and worst cases respectively for total n number of processes in the queue respectively. [8].

When one or more processes in the scheduler all arrives at the same time, IODTSRR and ODTSRR will be executed in O (n (n+1) (log n)) for all n > 0 in every cases.

When processes arrives at different times, ODTSRR runs in total of O (n^2(n+1) log n) and O (n (n+1) log n) during worst cases (i.e. neither one nor multiple processes arrives at the same time). Thus, exponential growth of total runtime occurs when there are large number of processes to be scheduled.

Whereas, complete execution of processes arriving at different times with IODTSRR will be in O (n^2 (log n)) in best and average cases. In worst case scenarios, the runtime complexity will be equal to either (n^2 (log n)) or (n (n+1) (log n)) + (n^2 (log n)) which fairly depends on the pattern of arrival of the process along with the variance of its burst time. Consequently, quick but constant rate of improvement in scheduling performance can be further noticed. In the Table 1, the total amount of runtime cost occur when both algorithm is scheduled for 5<= Tn <=200 are given in the next page as follows.

TABLE I

Calculation of Asymptotic Complexity of ODTSRR and IODTSRR

Amount of Processes	ODTSRR	IODTSRR
5	125.81	38.44
10	1210	210
20	11475.08	1066.84
50	220951.05	8579.8
75	812276.54	21235.07
100	2040200	40200
150	7442558.52	184542.61
200	18592782.57	184542.61

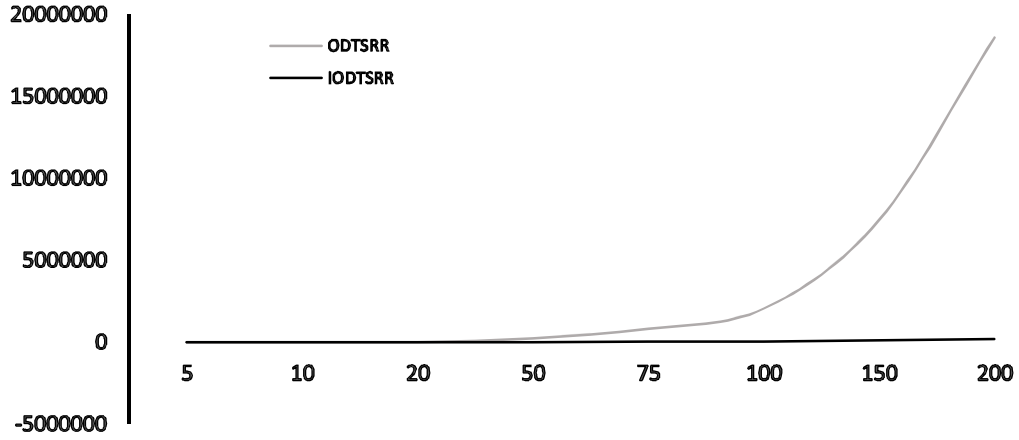

Fig. 1. Difference of the asymptotic complexities between ODTSRR and IODTSRR for 5 <= ∑n <= 200.

The above presentation of data in tabular format (Table 1) and graph chart (Fig 1) both theoretically conclude promising performance results of IODTSRR having not only constant but also quadratic rate of improvement in each and every cases comparative to ODTSRR.

Specifically, when there are 50 or more than that number of processes assumed to be of same priority have to be executed fairly while taking lesser time.

Chapter 5
Experimental Details

5.1 Estimations

The processes are homogenous and independent of both full and reduced level of computational time requirements. The system runs the algorithm only when it is in a stable state, that is, p < l/R. Irrespective of I/O bound, all processes are CPU bounded .The time unit is considered to be millisecond (ms). All attributes like burst time, number of processes and the time slice of all the processes are known before submitting the processes to the processor as performance metrics.

In-depth analysis and calculation of the evaluation metrics states overall and average performances of most of the state of the art variations in RR algorithms published periodically especially before and after ODTSRR are either less or equal to that of ODTSRR respectively.

5.2 Experimental Framework

The experiment consists of a number of input and output parameters. The input parameters consist of *Burst Time*<**BT**>, *arrival time*<**AT**>and *total number of processes* <**Tn**>. The output parameters consist of average response time, average waiting time, average turnaround time and number of context switches, fairness factor, throughput and CPU overhead.

5.3 Procedure

Suggested algorithm can work effectively with large number of data input and cases. In order to prove the supremacy of performances and standard of our proposed algorithm, same data sets used in the experimentation of ODTSRR along with mentioned RR variations are considered directly as majority of test cases of the proposed algorithm.

Chapter 6
Comparative Analysis and Outcome

6.1 Introduction

I have compared our proposed algorithm on basis of all mandatory performance metrics and results with only that of ODTSRR as it's the considered standard algorithm in my comparative analysis. Instead of the traditional approach where the processes in blocking and ready state are stored in two separate queues, a concept of only one synchronized blocking queue is considered.

Thus, it can perform much faster than both ready and blocking queue together for being able to handle concurrency dynamically.

6.2 Illustration

Example 1

The above algorithm is exhibited along with considerable empirical evidence as follows:

Let's assume 8 processes named P1, P2, P3, P4, P5, P6, P7 and P8 are given with their respective CPU burst time and arrival time.

Input Table

Process ID	Arrival Time	CPU burst time
P1	**1**	**20**
P2	**2**	**69**
P3	**3**	**53**
P4	**4**	**94**
P5	**5**	**82**
P6	**6**	**36**
P7	**7**	**100**
P8	**8**	**7**

A queue with eight processes P1, P2, P3, P4, P5, P6, P7 and P8 has been considered for illustration purpose. The processes are arriving at time 1, 2, 3, 4, 5, 6, 7, 8 with burst time 20, 12, 15, 60, 42, 9, and 19 respectively.
First, we initialize our queue along with the important variables. Set the Number_of_process = 8, Arrival_counter = 1(the arrival time of **P1**) as arrival time interval and Total_arrival_time = 8 (i.e. the arrival time of the last process i.e. **P8**). Therefore, the processes arrive in different time interval so the dynamic queue will not perform IODTSRR directly.

When Arrival_counter = 1 or 1st second, only P1 is added to the queue as no other processes arrived at that time.The Number_of_process is referred to P2.
Then we execute it for only one millisecond (ms), keep it in the queue and increment the Arrival_counter by 1.

When Arrival_counter = 2, P2 is added along with P1 in the queue and the Number_of_process is referred to P3. As no more processes will arrive at 2nd second, the Arrival_counter will be incremented by 1 indicating the arrival time of one or more processes coming in the next arrival time interval (k+1th) . Sort the processes' serial ID for execution in the queue according to their burst time in ascending order then to their arrival time in descending order respectively. The first available process in the sorted queue will be executed with a constant CPU Time Slice unit only till (k+1)th interval. Here P1 is executed for another 1 millisecond (ms) with remaining burst time of 18 millisecond (ms).

At 3rd second, P3 arrives and added to the queue. The Number_of_process is referred to P4. As no more processes will arrive at 3rd second, the Arrival_counter will be incremented by 1. After sorting the order of the processes in the queue, P1 will be executed for 1 ms again having 17 ms remaining.

At 4th second, P4 arrives and added to the queue as usual. The Number_of_process is referred to P5. The Arrival_counter will be incremented by 1. As no more processes will arrive at 3rd second, the order of the processes are sorted in the queue and P1 will be executed for another 1 ms accordingly.

At 5th second, P5 added to the queue as usual. The Number_of_process is referred to P6 and also Arrival_counter will be incremented by 1. As no more processes will arrive at 5th second, P1 will be executed for another 1 ms with remaining burst time of 15 ms after sorting the order of the processes in the queue.

At 6th second, P6 arrives and added to the queue. The Number_of_process is referred to P7 and Arrival_counter will be incremented by 1. As no more processes will arrive at 6th second, P1 will be executed for another 1 ms with remaining burst time of 14 ms after sorting the order of the processes in the queue accordingly.

At 7th second, P7 added to the queue. The Number_of_process is referred to P8. As no more processes will arrive at 7th second, P1 will be executed for another 1 ms with remaining burst time of 13 ms after sorting the order of the processes in the queue. Arrival_counter will be incremented by 1 indicating next process will arrive at k+1th arrival time interval respectively.

At 8th second, P8 arrives hence added to the queue. Both Arrival_counter and the Number_of_process will be incremented by 1 indicating whether next process (one or multiple) will arrive at k+1th arrival time interval respectively. P8 will be executed for only 1 ms with remaining burst time of 6 ms after sorting the order of the processes in the queue. Since, Arrival_counter is now greater than Total_arrival_time and all the processes have arrived in the queue whether partially executed with a single context switch, this very queue will start executing its processes using IODTSRR algorithm.

Now, the processes P1, P2, P3, P4, P5, P6 , P7 and P8 in the queue are arranged in the ascending order of their burst time in then in descending order of their arrival time which gives the sequence P8, P1, P6, P3, P2, P5, P4 and P7. The time quantum value is set equal to the median value obtained from proposed calculation i.e. 45 (it's the ceiling of round value of 44.5). CPU time slices for a time quantum of 45 milliseconds (ms) is allocated to the processes P8, P1, P6, P3, P2, P4 and P7 respectively.

During first cycle, the remaining burst time for the processes P8, P1, and P6 will be exactly equal to zero. Now it is turn to execute P3, P3 CPU burst time is 53 and the

allotted time slice is 45 which means it need 8 ms to complete its execution and it is less than or equal to the time quantum i.e. 45. CPU will continue its execution till it finishes, it will take 8 ms more to complete its execution. Next process in the ready queue is P2 and its CPU burst time is 69 ms. 45 ms will be allotted to P2, but the remaining burst time for P2 will be 69 - 45 = 24, which is also less than or equal to the time quantum i.e. 45. CPU will also continue its execution for another 24 ms to finish it.
The following process in the queue is P5 and its CPU burst time is 82 ms. 45 ms will be allotted to P5 as usual but the remaining burst time for P2 will be 82 - 45 = 37, which is also less than or equal to the time quantum i.e. 45. CPU will let continuing the execution of the process till it completes.

Except the only context switches for P1 during the completion of arrival of all the processes, it also means that P8, P1, P6, P3, P2 and P5 processes will be completed without any context switch after that, thus completing in 15, 28, 64, 117, 186 and 268 ms respectively.

Next process following in the queue is P4 and its CPU burst time is 94 ms. 45 ms will be allotted to P4; the remaining burst time for P4 will be 94 - 45 = 49, which is greater than the time quantum i.e. 45.CPU will stop its execution after 45 ms and place it at the end of the queue.

The immediate process in the queue is P7 and its CPU burst time is 100 ms. 45 ms will be allotted to P7, but the remaining burst time for P7 will be 100 - 45 = 55, which is also greater than the time quantum i.e. 45. Hence, CPU will stop its execution after allotting 45 ms to P7 and place it at the end of the queue.

One cycle of execution is completed. Again, the processes in the ready queue will be sorted in ascending order with respect to their remaining CPU burst time. There are only two processes left in the queue i.e. P4 and P7 respectively. After sorting the order will be same in this case as P4's remaining burst time is less than that of P7. Now, time quantum will be set to 52 after taking the proposed improved median value. As the process at the front of queue is P4 with its remaining CPU burst time 49 ms, time quantum 52 ms will be allotted to the process P4 and it will complete its execution and leave CPU just after 49 ms. Now there is only one process remaining in the ready queue P7 with its remaining burst time 55 ms, time quantum 52 ms will be allotted to P7 but the remaining burst time for P7 will be 55 - 52 = 3, so the CPU will complete its execution.

Thus, P2 and P5 processes will be completed with two context switches making total no. of context switches to 3 which might degrade the overall performance either gracefully or in negligible margin

Gantt Chart														
IODTSRR														
P1(19)	**P1(18)**	**P1(17)**	**P1(16)**	**P1(15)**	**P1(14)**	**P1(13)**	**P8**	**P1**	**P6**	**P3**	**P2**	**P5**	**P4**	**P7**
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**15**	**28**	**64**	**117**	**186**	**268**	**362**	**462**

Average Response time: **124.75 ms**

Average waiting time: **125.63 ms**

Average turnaround time: **183.25 ms**

No of Context switches: **3**

Fairness: **Yes**

Starvation: **No**

Gantt Chart							
ODTSRR							
P1	**P8**	**P6**	**P3**	**P2**	**P5**	**P4**	**P7**
1	**21**	**28**	**64**	**117**	**186**	**268**	**362**	**462**

Average Response time: **126.38 ms**

Average waiting time: **126.25 ms**

Average turnaround time: **184 ms**

No of Context switches: **0**

Fairness: **Yes**

Starvation: **No**

Example 2

Now, let us look at another test case and analysis of ODTSRR and IODTSRR are shown below:

Let 5 Processes P1, P2, P3, P4 and P5 all are arriving at zero millisecond (ms) with their burst time of 140, 75, 320, 280 and 125 ms respectively.

The Gantt chart of my proposed algorithm and ODTSRR illustrating the scheduling and execution of the process in each step is shown in Table 2 and 3 perceptively.

In accordance with, the result of summation of average of turn-around, waiting and response times of every process are mentioned as follows.

Gantt Chart					
IODTSRR					
P2	**P5**	**P1**	**P4**	**P3 (180)**	**P3**
0	**75**	**200**	**340**	**620**	**760**	**940**

Average Response time: **247 ms**

Average waiting time: **247 ms**

Average turnaround time: **435 ms**

No of Context switches: **1**

Fairness: **Yes**

Starvation: **No**

Gantt Chart						
ODTSRR						
P2	**P5**	**P1**	**P4 (140)**	**P3 (180)**	**P4**	**P3**
0	**75**	**200**	**340**	**480**	**620**	**760**	**940**

Average Response time: **219 ms**

Average waiting time: **275 ms**

Average turnaround time: **463 ms**

No of Context switches: **2**

Fairness: **Yes**

Starvation: **No**

6.3 Comprehensive differentiations

The experiential comparison in scheduling approaches, performances, data structure along with limitations between ODTSRR and my proposed IODTSRR are explained as follows:

Improved ODTSRR:

1. One or multiple processes can be scheduled in the queue upon instant arrival while the current process gets executed (fully or partially).
2. Scheduling and execution can be done simultaneously.
3. Average Response Time greatly improved due to FCSJF where every processes arriving during in every arrival time intervals gets fair chance of execution with constant CPU Time Slices. The summation of it equals to the desired Time Quantum respectively.
4. Average Waiting Time also decreases significantly due to FCSJF along with the proposed calculation of the median value for setting it as Time Quantum. As a result, the longer processes avoid indefinite blocking and starvation.
5. The average amount of time to complete the execution of one process is less than or equal to that of previous algorithm.
6. Overall performances either remain same or improve significantly if the queue has multiple numbers of processes arriving at the same time.
7. Less CPU overhead due to concurrent scheduling and execution of processes in the queue.
8. More throughputs in most of the cases.
9. Due to the dynamic nature of the Time Quantum, it is calculated differently before and after the completion of arrival of all the processes to be scheduled.
10. Instead of the traditional approach where the processes in blocking and ready state are stored in two separate queues, a concept of only one synchronized blocking queue is considered. It can perform much faster than both ready and blocking queue together for being able to handle concurrency dynamically. Importantly, it will allow

CPU to always avoid Dead-lock condition while functioning like a blocking queue simultaneously.

11. Its multi-level programming requires hardware virtualization causes overhead of the main memory.
12. Less number of context switches as a process gets fully executed if its remaining burst time is not only less than but also equal to the current value of the Time Quantum. The optimizations of the value of Time Quantum with the proposed median value decrease the chance of unnecessary context or even thread switches respectively.
13. If multiple processes with same remaining CPU burst time are scheduled, the processes with earlier arrivals will get the chances to be executed foremost. This enforcement of priorities increases fairness of the algorithm without any starvation.
14. Fairness is always guaranteed due to FCSJF where time slice for all Processes are 1 leading to desired TQ. Also for improvement in median value in ODTSRR.

ODTSRR:

1. No other processes can be scheduled in the queue before the execution of the current processes is fully completed.
2. Scheduling and execution cannot be done at the same time.
3. Average response time is much higher as the time quantum is set to the trivial median formula, the waiting time for each process cannot get better even though they arrive.
4. Average waiting time is also higher in case processes arrive at different times.
5. Average turnaround time is greater than or equal to that of proposed algorithm.
6. Overall performance is constant only when all the process arrives at the same time. It decreases significantly when it arrives at different time.
7. More CPU overhead.
8. Less or equal throughput in many cases.
9. Static nature of calculation of Time quantum done dynamically.
10. The concept of tradition ready queue was used for process scheduling.
11. Does not require hardware virtualization hence no memory overhead.
12. The number of Context switches is constant.
13. Real time enforcement of priorities was not implemented. It might affect the fairness and create chances of starvation.

Chapter 7
Experimentation

7.1 Explanatory Results

Overall performances of above algorithms on all types of input dataset used for analysis are stated in Fig 2 and Table 5 respectively. The test cases are used as input individually in respect of its types and complexities, where each test case represents any queues or list of processes to be scheduled and executed.

TABLE V

Complete projection of sum of the average
Turn-around, waiting and response time of all input cases

Metrics	**ODTSRR**	**IODTSRR**
Mean Turnaround Time	79.15	75.08
Mean Waiting Time	52.66	48.06
Mean Response Time	41.95	41.73

Complete projection of sum of the average turn-around, waiting and response time for n amount of test cases (where $\sum n = 32$).

Consecutively, illustration of the performance gap between my proposed algorithm and ODTSRR when both are executed on the list of processes in queue (where at least one or more process arrives in different time) are stated as follows in Table 6 along with corresponding graph chart in Figure 3.

TABLE VI

Cumulative difference of sum of the average turn-around,
Waiting and response time of list of processes
Scheduled in dissimilar arrival times

Metrics	**ODTSRR**	**IODTSRR**
Mean Turnaround Time	216.97	212.3
Mean Waiting Time	154.64	149.83
Mean Response Time	153.95	148.58

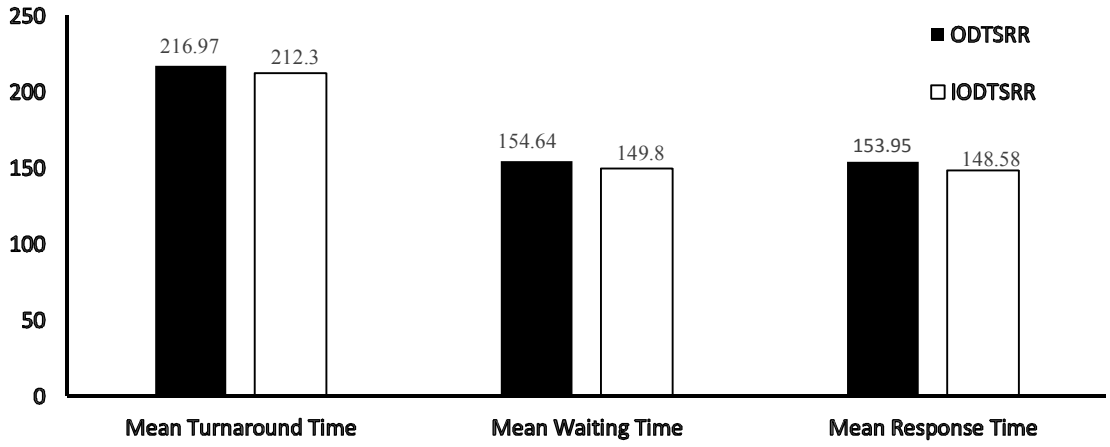

Fig Cumulative difference of sum of the average turn-around, waiting and response time of n number of test cases scheduled in dissimilar arrival times(where $\sum n = 21$ respectively).

Thus, all the tabular data and graphs resulted above from competitive test cases ensures that my proposed method ultimately takes lesser turn-around, waiting and response time than ODTSRR for scheduling processes at the same or different time period of their arrivals. Especially when there are large amount of processes in queue to be executed, no matter it have same or different priorities or extreme variation of their burst time consequently.

Chapter 8

Outcomes and Limitations

8.1 Outcomes and Contribution

Here, I discussed the various insightful outcomes and contribution of the research underlying to improve application throughput and overall system utilization.

8.1.1 Empirical Contribution

My research accumulated and developed improved datasets with reliability, the programming runs explicit thread and data placement to achieve the best performance for their parallel applications. Faster cache allocation and management at the end of every scheduling dispatched via minimum required flag pointer (Total arrival time) are found along with being compatible with virtualization of modern hardware architectures.

Derived Blocking queue can be able to handle thread communication delays while dispatching multiple number of scheduling with IODTSRR concurrently. allow clients to specify a scheduling policy and priority for their threads. Furthermore, thread libraries typically provide some support for binding a thread where to run threads and allocate data to one or more processors. The dynamic queue utilizes a coherent shared memory model with a single shared address space and runs the application thread in hardware thread (HWT) although this may have NUMA characteristics due to hierarchical caches.[16]

As kernel scheduler allows the user to provide a small amount of additional information such as whether to emphasize load balancing or proximity to data, and whether gang scheduling is required. Because scheduling dispatcher does not require the developer to supply architecture-dependent information, such as the specific cache level threads should share, it helps to preserve the application portability that general-purpose developer's need [16].

8.1.2 Theoretical Contribution

The proposed scheduling method have shown immense contribution in fixing consistent and effective standard of process-ordering in a scheduler which differs it from other RR variation as per its excellency.

The dynamic allocation of the CPU Time slice with time quantum which is calculated and used as time slice each time for decreasing any possible CPU Idleness and resource wastes. Also by careful instructional approaches, not bringing software-based architectural complexity of scheduling processes while multi-threading via arrival time interval for single many-core processors bring a lot of theoretical senses in effective development of CPU scheduler.

8.1.3 Methodological Contribution

Introduction of concept of dummy-threading while coding the IODTSRR algorithm has been a major turn-up as a portrait of clever trade off against memory allocation for efficient and faster scheduling performances.

Despite the traditional fixed thread-to-processor mappings, the proposed programming of the algorithm have compatibility to it would be reasonable for the operating system to control only the number of processors granted to a parallel application and leave the control of the threads to the application (i.e. to the compiler and the programmer)[16].

Threads distribution widely among the different CPUs is beneficial. On the other hand, the threads of array-based programs typically share data heavily, so scheduling the threads on nearby CPUs to share data in common caches provides the best performance.

8.2 Limitations

Primarily, the algorithm targets single CPU architectural platform for many-core processors. It will be essential to reduce shared cache misses since the on-die caches of many-core processors will be relatively small for at least the next several years, and memory latencies have grown to several hundred cycles [7].

Similarly, choosing which threads run concurrently on a processor is important since cache contention and bus traffic can significantly impact application performance. It can also be important to decide which threads to run on each core since simultaneous-multithreaded (SMT) cores share low-level hardware resources such as TLBs among all threads. Interrupts, in OS terms, creates both hardware and software-based boundaries in two flavors:

Hardware interrupts - those initiated by an actual hardware signal from a peripheral device. These can happen at, (nearly) any time and switch execution from whatever thread might be running to code in a driver. Hardware peripherals can rapidly make threads ready/running that ware waiting for data from that hardware, without any latency resulting from threads that do not yield or waiting for a periodic timer reschedule.

Software interrupts - those initiated by OS calls from currently running threads. Either interrupt may request the scheduler to make threads that were waiting ready/running or cause threads that were waiting/running to be preempted.

On multicore systems, the OS has an interprocessor driver that can cause a hardware interrupt on other cores, so allowing the OS to interrupt/schedule/dispatch threads onto multiple cores.

High memory requirements, been a considerable resource problem in processing, it has not yet been assessed how to optimally distribute partitions and/or alignment sites to processors, in particular when the number of cores is significantly smaller than the number of partitions. I find that, by distributing partitions (of varying lengths) monolithically to processors, the induced load distribution problem essentially corresponds to the well-known multiprocessor scheduling problem.

Chapter 9
Inference

9.1 Recommendations

As a result, one challenge for mainstream many-core programming is to develop mechanisms that provide the ability to do HPC-style customization for performance but do not compromise portability and programmability [16].

It will be essential to reduce shared cache misses since the on-die caches of many-core processors will be relatively small for at least the next several years, and memory latencies have grown to several hundred cycles [7]. Similarly, choosing which threads run concurrently on a processor is important since cache contention and bus traffic can significantly impact application performance. It can also be important to decide which threads to run on each core since simultaneous-multithreaded (SMT) cores share low-level hardware resources such as TLBs among all threads.

The proposed algorithm is not suitable for parallel scheduling and divisible computational task. Thus, Co-scheduling is a scheduling policy proposed to avoid these difficulties. Co-scheduling consists in granting simultaneously (in the same time quantum) the processors to the threads of the same application. It has been demonstrated in that co-scheduling performs quite well in a wide range of conditions and for various models of parallel applications. In the performance of a parallel application using barrier synchronization was studied. The co-scheduling policy (called here gang scheduling) has been compared, both theoretically and in practice, with blocking. In blocking the thread releases a processor as soon as it completes its share of the work. For coarse-grain parallelism blocking performs well. For parallel application co-scheduling is better. Thus, co-scheduling is postulated in parallel systems. Observe that co-scheduled applications

occupy several processors at the same moment of time and as such are multiprocessor tasks [15].

9.2 Further Insights and Works

For future work and potential research studies, I have great hope to modify the IODTSRR further and recommend making it cross-kernel compatible scheduler as per due points mentioned below:

- Divisible Task Concept
- Formulation of algorithms for scheduling preemptive multiprocessor
- Experimentation with tasks having linear speedup on parallel processors.
- Formulation of low-order complexity algorithms for scheduling preemptive
- Scheduling multiprocessor tasks with linear speedup for mean completion time criterion
- Formulation of polynomial-time algorithms for scheduling preemptive
- Scheduling divisible tasks on other architectures.

Soon in the near future, practical verifying divisible task concept for other applications and other architectures. Can portfolio huge possibilities for many-core hardware threads running much software threads respectively.

9.3 Conclusion

I believe, from the above illustration of results and comparative analysis, IODTSRR as "complete" CPU scheduling algorithm has overall ascending performances not specifically scheduling and executing processes arriving in different times than other but also that of one or multiple processes arriving at the same time respectively. The suggested architectural implementation of a singular dynamic queue rather than handling that of both ready and block queue shows significant possibilities of decreasing but consistent run-time complexity of executing and scheduling processes. Unlike other variations of RR algorithm, the approached method effectively calculates the Time Quantum being feisty and performs its dynamic allocation as CPU time slice unit too. It ensures lower-cost along with decrease of context switches between longer processes due to CPU resources utilization by multi-threading programming. Despite of hardly sufficient input cases available i.e., both custom and derived for testing the benchmarks, the stated performance can be observed by operating in the offered dynamic queue architecture. For very long list of process scheduling, the trade-off with space and negligible cost in thread and context switches against CPU idleness can be worth considering. Further extension of research in the future, faster modification of the

algorithm with intelligent CPU resource utilization and con-currency handling of scheduling can be possible. IODTSRR may have practical employment and fruitful experimentation when accordingly imposed in the real time operating system and along with other version of known OS platforms such as Windows, Linux, MacOS etc.

References

[1] Mohammad Salman Hafeez, and Farhan Rasheed, "An Optimum Dynamic Time Slicing Scheduling Algorithm Using Round Robin Approach," *IJCSIS*, vol. 14, pp. 778-798, Jun. 2016.

[2] Andysah Putera, and Utama Siahaan, "Comparison Analysis of CPU Scheduling : FCFS, SJF and Round Robin," *IJEDR*, vol. 4, no. 3, pp. 124-131, Jul. 2016.

[3] Radhe Shyam, and Parmod Kumar, "Improved Round Robin with Shortest Job First Scheduling," *IJARCSSE*, vol. 5, no. 3, pp. 156–162, Mar. 2015.

[4] Anju Muraleedharan et al. (2016, February). Dynamic Time Slice Round Robin Scheduling Algorithm with Unknown Burst Time. *INDJST* [Online]. 9(8). pp. 1-6. Available: http://www.indjst.org/index.php/indjst/article/view/76368

[5] Wasim Firuj Ahmed, and Sahana Parvin Muquit, "Improved Round Robin Scheduling Algorithm with Best Possible Time Quantum and Comparison And Analysis with the RR Algorithm," *IRJET*, vol. 3, no. 3, pp. 1357-1361, Mar. 2016.

[6] Jayanti Khatri, "An Enhanced Round Robin CPU Scheduling Algorithm," *IOSRJCE*, vol. 18, no. 4, pp. 20-24, Jul.-Aug. 2016.

[7] Omar Hani Mohammad Dorgham, and Dr. Mohammad Othman Nassar, "Improved Round Robin Algorithm: Proposed Method to Apply SJF using Geometric Mean," *IJASCSE*, vol. 5, no. 11, pp. 112-119, Nov. 2016.

[8] M. Naftalin and P. Wadler, "Queues," in Java Generics and Collections, 1st ed. Sebastopol: O'Reilly Media, 2006, ch. 14, sec. 5, pp. 210-211.

[9] Centre For Innovation In Research and Teaching. (2017). *Quantatitive Approaches*. Retrived from https://cirt.gcu.edu/research/developmentresources/research_ready/quantresearch/approaches

[10] University of Winconsin. *Data Collection Methods*. Retrived from https://people.uwec.edu/piercech/ResearchMethods/Data%20collection%20methods/DATA%20COLLECTION%20METHODS.htm