Volume 2, Issue 2, February 2012 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Studies on Performance Aspects of Scheduling Algorithms on Multicore Platforms

N. Ramasubramanian, Srinivas V.V., Chaitanya V.
Department of Computer Science and Engineering
National Institute of Technology - Tiruchirappalli

Abstract— **The theory of scheduling has expanded rapidly during the past years. As multi-core architectures begin to emerge, operating system issues are to be considered for best use of multi-core processes. Due to the architectural differences in the state of art multi-core processors such as shared caches, memory controllers etc., it becomes the responsibility of the operating system to make use of intelligent scheduling mechanisms instead of simply scheduling tasks. In this paper, we try to explore the rapidly expanding area of scheduling by classifying the multi-core scheduling into traditional shortest job scheduling for multi-core, tree based threaded scheduling and block level scheduling. We have conducted simulation of the traditional shortest job first for multi-core processors the details of which are discussed below. Tree based scheduling is achieved by constructing a binary search tree (BST) data structure; similarly we have demonstrated block scheduling which form a part of software scheduling for reducing the processes execution time. Research by, [13] shows that applications do not make use of the entire processing power of multi-core processors. There has been considerable progress in the design of thread schedulers. We demonstrate a mechanism of handling various cores as compared to the traditional mechanism of handling a single processor core. Results show improved values of execution time as compared to traditional scheduler.**

Keywords— **Thread scheduling, multi-core, kernel, BST-tree, block scheduling**

I. INTRODUCTION

Single-threaded processor performance is becoming power limited, so processor architects are increasingly turning to multi-core designs to improve processor performance. A multi-core processor is an integrated circuit composing of two or more individual processors [2]. The performance gained using a multi-core processor depends on the proximity of multiple-cores on same die, which in-turn allows the cache coherence circuitry to operate at a much higher rate [3]. The coupling between the cores in a multi-core environment can be considered as either loose or tight. Combining CPU's on same die improves the performance of snooping cache. As a result of this there is little degradation of signals.

II. NEED FOR SCHEDULER

Multi-core is the latest technology which has grabbed the market of processors [4]. A number of applications do not in reality exploit the power of the processing cores. Like every multitasking operating system, linux achieves simultaneous execution of multiple processes by rapidly cycling through the process that are ready to run [5]. Determining when to switch and which process should be allowed to run is called scheduling [6]. An ideal scheduler should protect lower priority process from starvation. Some of the issues related to schedulers that needs to be addressed are:

- Persistent starvation - This is the scenario where two tasks share a CPU yet one task completely starves out execution when compared to the other task.
- Initial affinity problem - A task that is assigned affinity to a particular CPU might never have run on that particular CPU. When applications are run at a certain minimal frequency, in rare occasions they tend to run on a single CPU, even when the other CPU's are idle [10].

Schedulers must be crafted carefully so that the process appears to be running continuously [1], [11].

III. EXISTING METHODOLOGY

Process is program in execution holding a specific address space. A process consists of multiple-threads of control. The other way of looking at a process is, it is a way to group related resources together [10]. Here resources refer to open file descriptors, child processes, pending alarms, signal handlers and accounting information. Scheduling in traditional uni-processor system consisted of processes having a single thread of control running on top of the kernel [7]; see Fig. 1, whereas scheduling in multi-processor system consists of a single process having multiple threads of control

running on top of multiple cores. Threads have a program counter that keeps track of the next instruction to execute. When multiple threads are running a few fields are unique for each and every thread [8]. Fig. 2.

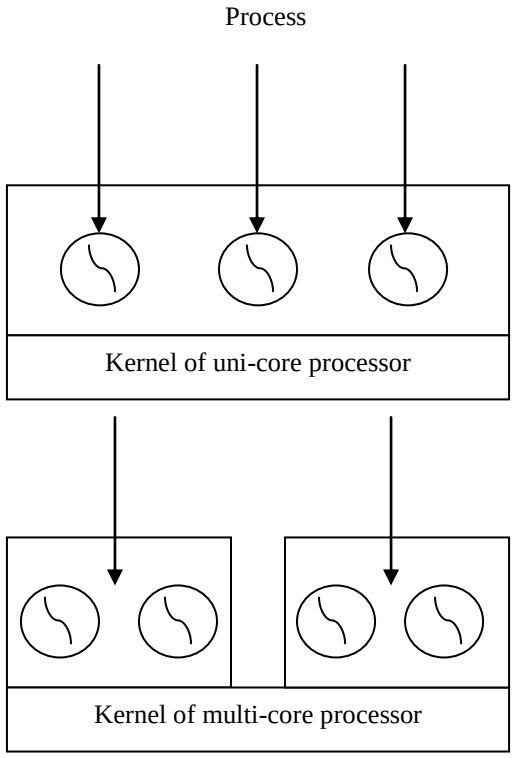

Fig. 1. Single thread of execution in uni-core environment vs Multi-threaded execution in multi-core environment

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child process	State
Pending alarms	
Signals and handlers	
Accounting information	

Fig. 2. Description of fields in process and thread table.

IV. KERNEL LEVEL SCHEDULING MECHANISM

Generally a kernel process has a higher priority compared to a user process [9]. Under special circumstances where the user process requires higher priority than the kernel process, the *nice* command in linux can be used to modify the scheduling priority of the process. The scheduler maintains 16 queues of runnable process. Not all of them are used at a given instant of time. Fig. 3 shows the queue and processes that are in place at the instant kernel begin to run. The array rdy_head has one entry for each queue with the entry pointing to the process at the head of the queue. Similarly rdy_tail is an array whose entries point to the last process on the queue. Based on the priority level of the task, the process fall into one of the 16 queues. The scheduling is done in round-robin fashion. If the running process exhausts the quantum, it is moved to the tail of the queue and given a new quantum. If a process is blocked and after some quantum of time it is awakened, it is moved to the head of the queue, if the process has remaining time quantum.

Given the queue structure, the scheduling mechanism is simple. The first step is to *enqueue* the process along with a pointer to the process table entry. Once *enqueue* is complete, it calls the function *sched* which determines one among the 16 queues where the process has to be inserted. If the queue was previously empty, then the rdy_head and rdy_tail are made to point to the same location when the process is added. If a process is added to the head of the queue then p_nextready gets the current value of rdy_head and the rdy_head is pointed to the new process.

A process that is running is blocked by *dequeue* operation. The process to be de-queued is likely to be at the head of the queue. In case, when a signal is sent to a process that is currently not in execution, then the de-queue procedure has to traverse the array to find the process that is not running. The likely hood of finding the victim is at the head of the queue. Once the victim process is removed, the pointers are adjusted accordingly.

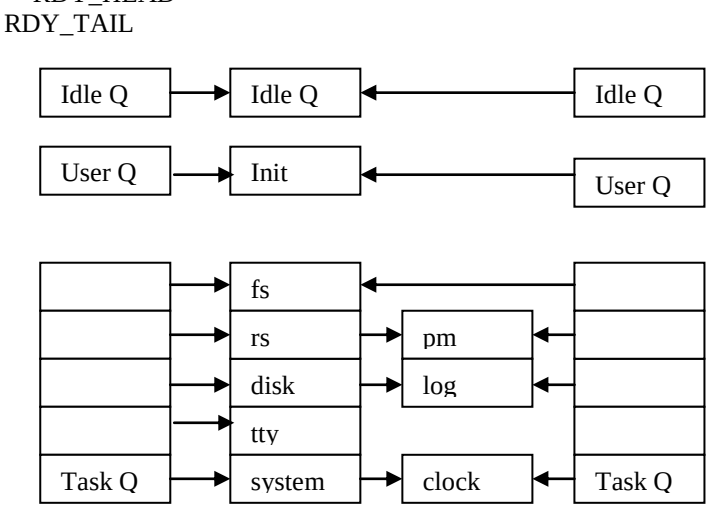

Fig. 3. Queuing system

One important consideration to be noted is that, the kernel processes maintains a common stack. Hence the integrity of the stack has to be checked periodically. At the beginning of the de-queue, a test is performed to verify whether the process operates in kernel space. If the process operates at the kernel stack, a check is made to see that the distinctive pattern at the end of the stack is not overwritten.

During the scheduling process described above, the process table is updated with the following entries, time_quantum_left, priority, maximum_allowed_priority. Every time a process is scheduled, a check on the time_quantum_left is done.

A. Working of nice

Nice is a UNIX system call which assigns scheduling priority to processes. A positive nice value refers to process having lower priority and a negative value denotes higher priority process. Generally task drivers and servers are given large quanta so that they run before they get blocked. But if they run longer than the given quanta they may get pre-empted.

.

V. SCOPE FOR IMPROVEMENT

When tasks are created they are placed on a given CPU's run queue. Processes are either short lived or long lived. Traditional linux scheduler provides the functionality of scheduling by balancing the workload among the CPU's [12] in round robin fashion. The scheduler does not take the process execution time into consideration. Using traditional SJF for multicore, BST and Block scheduling mechanism, we propose methods for scheduling by taking the execution time of each process and the load of each processor core.

VI. SIMULATED SJF FOR MULTI-CORE

In the shortest job for multi-core we have come up with a model where the processes arrive at a Poisson rate and the execution time of the processes are considered to be random.

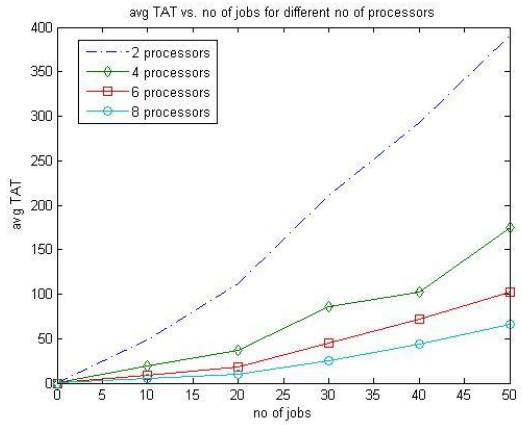

Fig. 4. Average turn around time

Based on the number of processors assigned to the simulation environment, the processes are transmitted to the processors using the shortest job first algorithm from a M/M/1 queue. The execution of the processes by the processors is assumed to be threaded and the results obtained as a result of execution of the simulated traditional SJF for multi-core scheduling are given below.

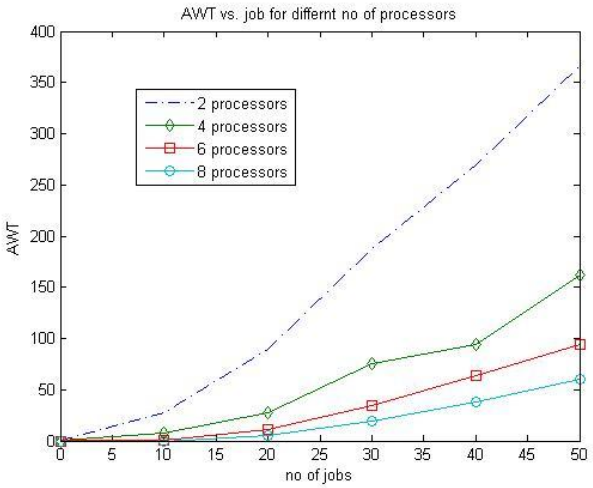

Fig. 5. Average waiting time

The results show that the as the number of jobs increase, the average turnaround time decreases for increased number of cores. Similarly the average waiting time for the processes also decreases as the number of cores increase.

VII. BST SCHEDULER

The main idea behind the BST scheduler is to maintain balance in providing processor time to tasks and reducing the latency for getting CPU access [14]. When the time for the tasks is out of balance then the out of balance tasks should be given excess time to execute. The scheduler is named as BST scheduler due to the following reasons; refer [15] for details related to completely fair scheduler.

- Binary search tree is used as data structure for the run queues instead of the traditional arrays.
- Fast interactive response
- Fair to all tasks
- Improved load balancing for multi-core.

A. Internal Design

A Binary Search Tree has the self-balancing property. Operations pertaining to Binary Search Tree occur in O(log n) time. This means that the insertion and deletion of tasks can be done in fixed time

efficiently. BST's design uses time-ordered BST-tree to build a time-line of future task execution. It does not make use of the traditional array switch mechanism. BST makes use of the concept of sleeper fairness, where tasks that are not runnable receive a comparable share of processor time at a later point of time when they need it.

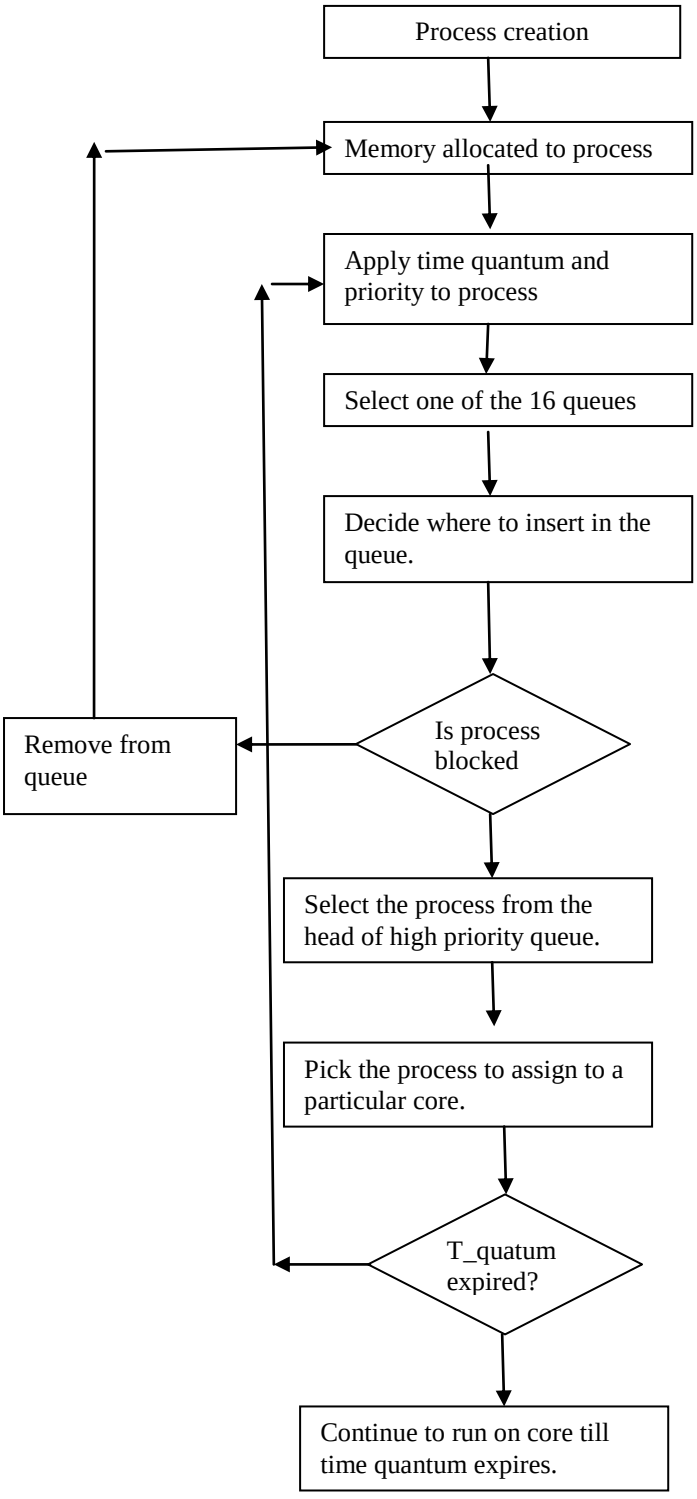

Fig. 6. Flow diagram of traditional scheduler

VIII. INTERNAL DATA-STRUCTURE OF BST SCHEDULER

The internal of BST scheduler makes use of the following data-structure.

- min_vruntime - It is a monolithic increasing variable taking the smallest virtual run time among all the tasks in the run queue.
- load - The total number of running tasks in the run queue is accounted through this variable.
- se_vruntime - holds the difference between the min_vruntime and the executed time.

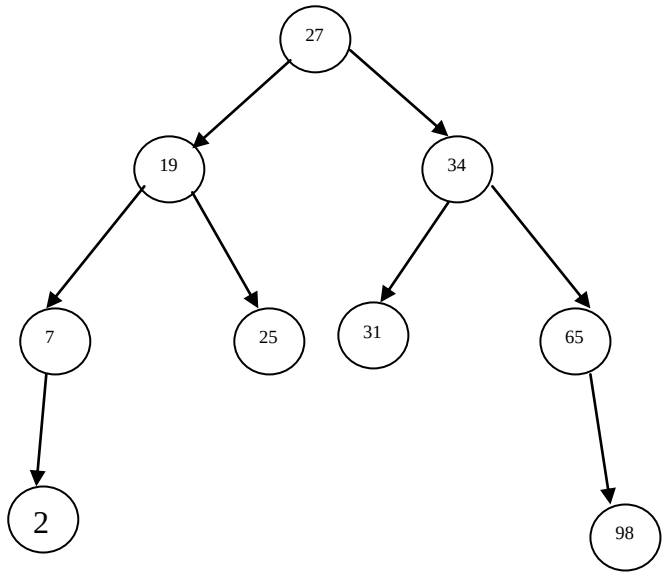

Fig. 7. BST Datastructure

IX. WORKING OF BST SCHEDULER

A task that is to be executed is scheduled based on the min_vruntime value i.e. the left most element of the BST-tree. Now the se_vruntime is computed. Once the value becomes greater than the threshold, some other task is made as the leftmost task. With tasks stored in time-ordered tree, tasks which are in grave need of processor are allocated processor first. The flow diagram shown in Fig. 6 describes the entire process involved in scheduling of tasks using BST tree. Initially the process/thread is created and the time quantum is computed and placed in the BST tree. The initial step involves, checking whether the process is blocked. If the process is blocked due to some operation, then the process is moved into the ready queue and the process id associated with the left most leaf of BST tree is relocated to some other leaf. The process-id of the next process which is going to make use of the processor is associated with the left most leaf of the BST tree. If the process is not blocked then the process continues until the time quantum expires.

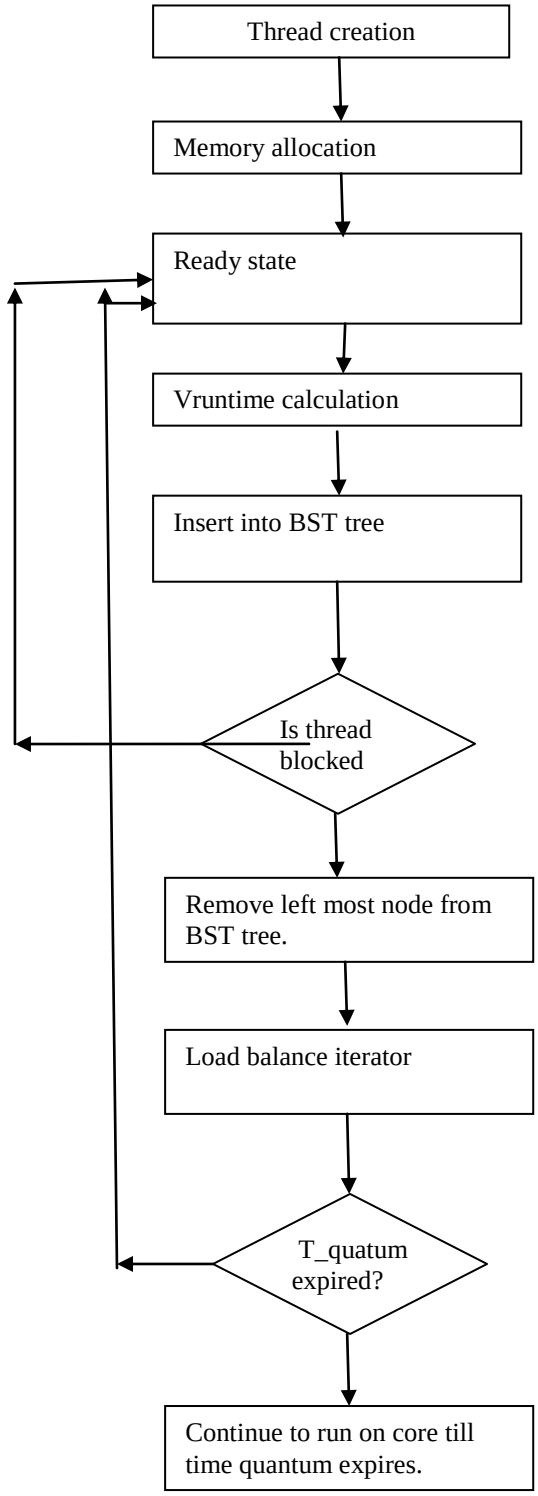

Fig. 8. Flow diagram of BST scheduler

```
File  Edit  View  Terminal  Help
root@athul:/home/atul/Desktop/interbench-0.30# make
make: interbench is up to date.
root@athul:/home/atul/Desktop/interbench-0.30# ./interbench
997733 loops_per_ms read from file interbench.loops_per_ms

Using 997733 loops per ms, running every load for 30 seconds
Benchmarking kernel 2.6.28-18-generic at datestamp 201005240952

---Benchmarking simulated CPU of Audio in the presence of simulated ---
Load  Latency +/- SD (ms)  Max Latency  % Desired CPU  %Deadlines Met
Video  0.146  +/- 0.958       8.52           100            100
X      0.023  +/- 0.0343      0.497          100            100
Burn   0.111  +/- 0.1113      0.119          100            100
Write  0.046  +/- 0.34        6.34           100            100
Read   0.230  +/- 0.426       4.303          100            100
Compile0.013  +/- 0.0133      0.02           100            100
root@atul:/home/atul/Desktop/interbench-0.30#
```

Fig. 9.Output obtained from traditional scheduler

```
File  Edit  View  Terminal  Help
root@athul:/home/atul/Desktop/interbench-0.30# make
make: interbench is up to date.
root@athul:/home/atul/Desktop/interbench-0.30# ./interbench
997733 loops_per_ms read from file interbench.loops_per_ms

Using 997733 loops per ms, running every load for 30 seconds
Benchmarking kernel 2.6.28-18-generic at datestamp 201005240952

---Benchmarking simulated CPU of Audio in the presence of simulated ---
Load  Latency +/- SD (ms)  Max Latency  % Desired CPU  %Deadlines Met
Video  0.036  +/- 0.0849      0.809          100            100
X      0.025  +/- 0.0648      0.804          100            100
Burn   0.011  +/- 0.0115      0.075          100            100
Write  0.033  +/- 0.13        2.23           100            100
Read   0.026  +/- 0.026       0.303          100            100
Compile0.013  +/- 0.0133      0.02           100            100
root@atul:/home/atul/Desktop/interbench-0.30#
```

Fig. 10.Output obtained from BST scheduler

X. BLOCK SCHEDULER

Block scheduling [16] is an emerging state art scheduling for process on multicore where special assumptions about process are made with respect to multiple cores. The block scheduling attempts to do an equitable distribution of workload among the multiple cores of processor. It tries to avoid forward dependency. Whenever process Pj's input depends on process Pi's output then process Pj it is said to be forward dependent on process Pi. Block scheduling tries to avoid this by rearranging processes so that the cores never remain idle. It also tries to minimize the amount of context switching. In this paper, we have implemented block scheduling using simulation.

Consider the following example for forward dependency:

P1: x=a+b P2: y=x+c
P3: z=d+e P4: s=z+v
P5: t=h+i

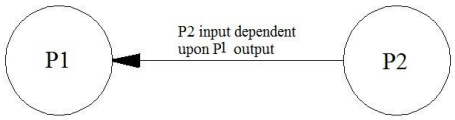

Fig. 11. Forward dependency

In this figure process P2 will not execute unless process P1 is completed. Similarly process P4 will not execute unless P3 is executed.

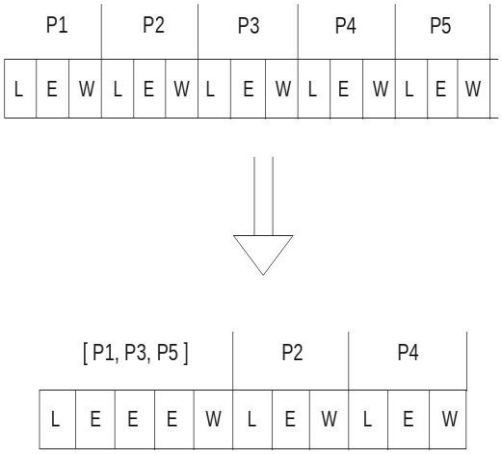

Fig. 12. Influence of scheduler on execution time

The time for execution of these processes using a traditional scheduling would be much slower than using block scheduling. Using block scheduling, the process that are independent of each other are grouped together as blocks and the remaining process that do not form the block comes separately once the block gets executed. Hence by rearranging the processes the dependency conflict that arises in case of traditional schedulers is reduced on a multi-core machine.

XI. Results of Block Scheduling

The graph is obtained by running block scheduling on single core, dual core and quad core processors. This is done by running the block scheduling program using task-set command of linux shell. The process are given affinity to one particular core in case of simulating on a uni-core processor, similarly the affinity is set to 2 for dual core model and the affinity is set to 4 in case of quad core model. The results obtained are shown in the graph.

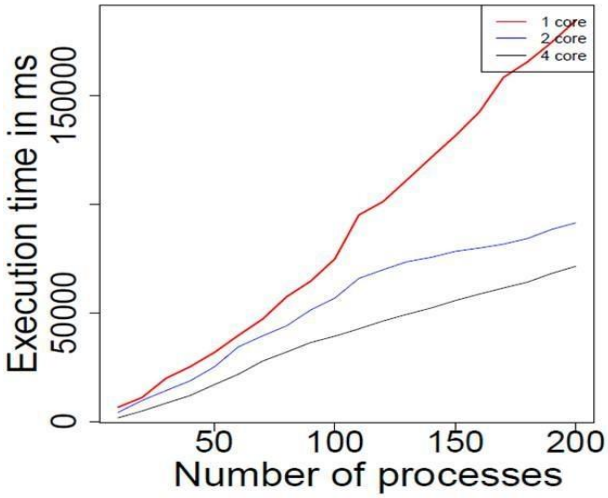

Fig. 13. Execution time for 1, 2 and 4 core processors using block scheduling.

XII. Conclusion

In this paper, we have demonstrated the traditional scheduler, BST scheduler and block scheduler to minimize the execution time of the tasks. Based on the requirements of the tasks, selection of appropriate scheduler would improve upon the efficiency of execution of the multiple cores of the processor. In this paper, an attempt to optimize multicore-load balancing to achieve better response time is also made.

Acknowledgement

The authors in this paper would like to recognize National Institute of Technology, Tiruchirappalli for for providing facilities for our research.

References

[1] Saez, J.C. and Prieto, M. and Fedorova, A. and Blagodurov, S.: *A comprehensive scheduler for asymmetric multicore systems*, In: Proceedings of the 5th European conference on Computer systems, pp. 139-152, 2010.

[2] Shelepov, D. and Saez Alcaide, J.C. and Jeffery, S. and Fedorova, A. and Perez, N. and Huang, Z.F. and Blagodurov, S. and Kumar, V.*: HASS: a scheduler for heterogeneous multicore systems*, In: ACM SIGOPS Operating Systems Review, vol. 43, no. 2, pp. 66-75, 2009.

[3] Fedorova, A. and Blagodurov, S. and Zhuravlev, S.*: Managing contention for shared resources on multicore processors*, In: Communications of the ACM, vol. 53, no. 2, pp. 49-57, 2010.

[4] Kodaka, T. and Sasaki, S. and Tokuyoshi, T. and Ohyama, R. and Nonogaki, N. and Kitayama, K. and Mori, T. and Ueda, Y. and Arakida, H. and Okuda, Y. *: Design and implementation of scalable, transparent threads for multi-core media processor*, In: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 1035-1039, 2009.

[5] Jensen, M.G. and Kinter, R.C.: *Multithreading instruction scheduler employing thread group priorities*, In: Google patents US Patent 7,660,969, 2010.

[6] Kazempour, V. and Kamali, A. and Fedorova, A.: *AASH: an asymmetry-aware scheduler for hypervisors*, In: ACM SIGPLAN Notices, vol. 45, no. 7, pp. 85-96, 2010.

[7] Nesbit, K.J. and Moreto, M. and Cazorla, F.J. and Ramirez, A. and Valero, M. and Smith, J.E. : *Multicore resource management*, In: IEEE Micro, vol. 28, no. 3, pp. 6-16, 2008.

[8] Bower, F.A. and Sorin, D.J. and Cox, L.P.: *The impact of dynamically heterogeneous multicore processors on thread scheduling*, In: IEEE Micro, vol. 28, no. 3, pp. 17-25, 2008.

[9] Song, F. and Moore, S. and Dongarra, J.: *Analytical modeling for affinity based thread scheduling on multicore platforms*, In: University of Tennessee, Computer Science Tech. Rep. UT-CS-08-626, 2008.

[10] Guo, Y. and Zhao, J. and Cave, V. and Sarkar, V.: *SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems*, In: ACM SIGPLAN Notices, vol. 45, no. 5, pp. 341-342, 2010.

[11] Kato, S. and Rajkumar, R. and Ishikawa, Y.: *A Loadable Real-Time Scheduler Framework for Multicore Platforms*, In: , 2010.

[12] Wang, B.*: Task Parallel Scheduling over Multi-core System*, In: Journal of Cloud Computing, pp. 423-434, 2009, Springer.

[13] Srinivas V.V., Ramasubramaniam, N.: *Understanding the performance of multi-core architecture*, In: Accepted in International Conference in Communication, Network and Computing (CNC) (2011).

[14] Hennessy, J.L., Patterson, D.A.: *Computer Architecture: A Quantitative Approach.* In: 4th edition, Elsevier Inc., (2007).

[15] Completely Fair Scheduler, http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/

[16] Irmsher, K., *Block scheduling*, In: Eugene, OR: ERIC Clearinghouse on Educational Management, College of Education, University of Oregon, 1996.