ISSN (Online): 2230-7893
www.IJCEM.org

Improving Performance of Real Time Scheduling Policies for Multicore Architecture

N. Ramasubramanian[1], Ashutosh[2], Akhilesh Kumar Verma[3], Manvendra Singh[4] and Praveen Kumar Yadav[5]

[1,2,3,4,5] Department of Computer Science and Engineering, National Institute of Technology, Tiruchirappalli, Tamilnadu, India

Abstract

Increase in demand of multicore for embedded systems has lead to the need of efficient real time scheduling policies. In this paper a new implementation for global EDF is proposed for scheduling in multicore systems. It uses a heap global data structure to speed-up scheduling decisions. Ready tasks are enqueued in a logical global queue, and the M highest priority tasks are selected to run on the M cores. The logical global queue is implemented using a set of distributed run-queues, one for each core. Each run-queue is implemented using a linked list. The max heap is used for keeping track of deadlines of the earliest deadline tasks currently executing on each run queue. Deadlines are used as keys and if B is a child node of A, then deadline (A) <=deadline (B). Therefore, the node in the root directly represents the CPU where the tasks need to be pushed. Node of the heap is implemented using the structure data structure. The average total turnaround time and percentage of tasks completed is analyzed as the number of cores increases for some task set. The performance of different processor cores is compared based on percentage of tasks completed.

Keywords: *EDF, Global Scheduling, Multicore Real Time Scheduling, Load Balancing*

1. Introduction

The presence of multicore and multiprocessor has lead to the development of embedded systems which require multiprocessor real-time task scheduling. The current researches are more focused on determining proficient schedulability test for scheduling algorithm and developing algorithm for providing tighter schedulable upper bound. The primary requirement of schedulability test is the worst-case behavior of tasks including execution times, arrival times, and resource access behaviors. These facts are better explained by Carpenter et al. in [1] where categorization of scheduling problem is done for multiprocessor system.

Real-Time schedulers for multiprocessor systems are of two types that is partition and global schedulers. As per work done by Lelli et al. in [2] portioned scheduler are more efficient for applications which statically loaded at start-up and their run time changes rarely. Whereas global schedulers are more useful for open systems, where application can leave the system any time. If jobs are scheduled on multiple processors, and a job can be dispatched from the priority run queue to any of the processors, the system is called dynamic scheduler. If jobs are partitioned into subsystems, and each subsystem is bound statically to a processor, have a static system. In partitioned schedulers the priority of the jobs is predefined offline while in dynamic scheduler priority is determined when the task comes in the run queue and its characteristics are analyzed. Each processor is associated with a separate instance of a uniprocessor scheduler for scheduling the tasks assigned to it and a separate local ready queue for storing its ready jobs. In other words, the priority space associated with each processor is local to it. The different per-processor schedulers may all be based on the same scheduling algorithm or use different ones.

Global scheduling algorithm is one of the most widely researched class of multiprocessor real-time scheduling algorithm. In this algorithm, all the scheduling decisions are made for entire system on the basis of a queue where all tasks are placed and are globally accessible. The tasks can freely migrate between all processors in the system. Dellinger et al. have shown in [3] that this approach is very much efficient in providing optimal scheduling on a multiprocessor and tasks can be added to the system at run-time without difficulty. But this approach increase overhead due to cache misses and migration cost.

Cluster scheduling is another class of scheduling algorithm which is intermediate between global and partition scheduling algorithm. Clustered schedulers reside in the middle, where the available processors are partitioned into clusters to which tasks are statically assigned, but in each cluster tasks are globally scheduled. It takes the good of both the worlds. Each node in a cluster is an independent computing system with its own operating system and possibly with dedicated peripheral devices. The mounting interest has been fuelled in part by the availability of powerful microprocessors and high-speed networks as off-the-shelf commodity components as well as in part by the

rapidly maturing software components available to support high performance and high availability applications. Cluster is a collection of inter-connected and loosely coupled stand-alone computers working together as a single, integrated computing resource. Clusters are commonly, but not always, connected through fast local area networks. Clusters are usually deployed to improve speed and/or reliability over that provided by a single computer, while typically being much more cost-effective than single computers of comparable speed or reliability. Due to large number of computing nodes, it becomes difficult for a single application to use all the computing power at its disposal. Hence, is allowed multiple applications to use our cluster simultaneously. In this case, a situation may arise in which two or more applications need the same resources at the same time. This ultimately requires for a scheduling mechanism to resolve contentions. The basic problem for such schedulers is to find the best solution in a trade-off between under-utilization of resources versus slowing down applications due to lack of resources at particular points of time. Petridou et al. have shown in [4] that cluster based scheduling can also be used for single-hop lightwave networks.

One of the example of global scheduling algorithm, Earliest Deadline First (EDF) has been used to scheduling of various critical application, but it does not provide mechanism for managing time redundancy, so real-time tasks can meet deadlines even in the presence of faults. Beitollahi et a.l in [5] have shown extension of this by adding time redundancy for EDF policy for scheduling periodic and preemptive tasks.

The study aims at finding the efficiency of a Global Earliest Deadline First algorithm in CPU scheduling. Here Heap data structure is used to find the processor on which the incoming task should be assigned. Here a min-heap of all the processors is maintained according to the deadline of the jobs in the runqueue of each processor, so that the incoming task is assigned to the processor having the latest deadline in its runqueue. The main objectives are :

A. Efficient implementation of Earliest Deadline First (EDF) scheduling in a multiprocessor system.
B. Using heap data structure to reduce the time needed to find the most suitable processor for the incoming task.
C. Finding total turnaround time (TTR) for different number of tasks for each 8 core, 16 core, 32 core and 64 core processor.
D. Finding the percentage of tasks that have been completed within the deadline by the system for different number of tasks for each 8 core, 16 core, 32 core and 64 core processor .

2. Related Works

Chishiro et al. have evaluated global and portioned semi-fixed priority scheduling algorithm in [6] for multicore systems. It infers that overhead of a fixed priority scheduling without cache is comparable to semi-fixed priority scheduling. Because of dual scheduler P-RMWP has higher overhead than G-RMWP.
Lee et al. have implemented EDF scheduling in uniprocessor system in [7] for checking feasibility of preemption. It shows the need of new preemption policies for controlling preemption of job in order to get better EDF schedulability. The result infers that cp-EDF finds schedulable tasks which are previously seems to be unschedulable by EDF under old pre-emption policy.
UmaMaheswari C. Devi et al. in [8] have tied to implement generic resource sharing framework using simple shared object like queue and stakes. The proposed queue based spin lock impose is appropriate for both hard and soft real time system as it have very little overhead whereas lock free algorithm overhead is higher and can be used in absence of kernel support for less number of processor and object calls. Lopez et al. have tried to implement utilization bound on uniprocessor system for EDF scheduling. The result shows that bound depend on number of processor but does not depend on number of tasks. Liu et al have tied to guarantee bound response time through their work in [10] by applying soft real time scheduling analysis techniques. Result shows that with bounded response time, system can be efficiently supported on multiprocessors.

3. Methodology

3.1 Data Structure

3.1.1 Structure

Structured data structure are used to implement a process, CPU and run-queue.

(i) A structure proc1 is made which work like a process contains different information like process identity, release time, execution time, deadline , remaining time to execute .
(ii) CPU
Process is executed on CPU. The structure “CPU” has following information: status of CPU, pointer to run-queue, CPU identity, and latest deadline of task which is executing on CPU.

3.1.2 Linked List

The Linked list "run_queue" is used to implement run-queue of the CPU. In run-queue all the tasks that are allocated to CPU are sorted in increasing order of their deadlines.

3.1.2 Heap

A new data structure is introduced to speed-up the search for a destination CPU inside a push operation. The requirements for the data structure are:
(i) O (1) complexity for searching the best CPU
(ii) Less-than-linear complexity for updating the structure.

Also, it can be implemented using a simple array. A min heap is developed to keep track of deadlines of the earliest deadline tasks currently executing on each run-queue. Deadlines are used as keys and the heap-property is: if B is a child node of A, then deadline (A) >=deadline (B). Therefore, the node in the root directly represents the CPU where the task needs to be pushed.
Node of the heap is implemented using structure having a pointer head of type "run queue", CPU identity, and latest deadline of task which is executing on CPU.

3.2 Implementation of different operations

3.2.1 Push Operation

When a task is activated on CPU k, first the scheduler checks the local run-queue to see if the task has higher priority than the executing one. In this case, a pre-emption happens, and the pre-empted task is inserted at the head of the queue; otherwise waken up task is inserted in the proper run-queue, depending on the state of the system. In case the head of the queue is modified, a push operation is executed to see if some task can be moved to another queue. A function "check" is made to check whether the upcoming task is missing its deadline, if it is executed on current CPU. It will return "1" id deadline is not missed else return "0".

Now function is defined to find the CPU**,** where to push the task. The result of expression (deadline of second task in run queue **–** deadline of first task in run queue – execution time of second task) is evaluated. If it is greater than execution time, then the CPU is selected to execute the task.

3.2.2 Pull Operation

A pull operation looks at the other run-queues to see if some other higher priority tasks need to be migrated to the current CPU. Pushing or pulling a task entails modifying the state of the source and destination run-queues: the scheduler has to dequeue the task from the source and then enqueue it on the destination run-queues. The function "check_pull" is defined to check which CPU is idle or where the pull operation to be performed.

A function "pull_op" is defined to find the CPU, from where the task is pulled. Here the difference of deadlines of second and first task is greater than the execution time of second task is checked.

3.2.2 Enqueue Operation

Enqueue operation is adding a task to CPU's run-queue to execute. It is implemented using pointer. When a task comes to the CPU, it is added to run-queue based on its deadline (maintaining tasks in sorted deadline).
Here a pointer "link" is used points to the head of the run-queue when task is to be added. This link pointer is updated to point to second node and so on until proper location is found for the task.

3.2.2 Dequeue Operation

Dequeue operation is to remove the task from run-queue, when it is completed or it will miss the deadline. Here the node of corresponding task is deleted from run-queue. Update the link pointer accordingly.

3.3 Implementation of HEAP, CPU idle check

3.3.1 Implementation of HEAP

Min heap is implemented using the property:
Deadline (A) >= deadline (B),then A will be child node of B. The heap is built using heap

property for multicore processor. When a task is allocated to some core then this core is added to the heap. Hence number of processors that are busy is total number of nodes in the heap.

3.3.2 CPU Idle check

Here the function "check_idle()" checked whether the CPU is idle. If the ID of the processor is not in the heap list then CPU is idle.

3.4 Calculations of Average Turnaround Time

Average turnaround time can be calculated for tasks as:
Avg TTR = (Sum of TTR of each completed task / total number of completed tasks) *100

ISSN (Online): 2230-7893
www.IJCEM.org

4. Result

In Fig.1 shows the graph between number of tasks and average total turnaround time (ATTR) for multicore processors. Data for number of tasks and corresponding ATTR is given in Table 1, based on these data different graphs are plotted between number of tasks and corresponding ATTR for 8, 16, 32 and 64 core processors. Fig.1 infers that:

(i) As the number of tasks increases for a multicore processor, if the tasks have almost same value of execution time then ATTR is increased and if the tasks have different execution time then ATTR is decreased. ATTR also depends on the dataset.

(ii) For the same dataset, if the number of cores are increases then ATTR is decreases.

Table 1: Number of tasks and corresponding average total turnaround time for different cores

No_of_tasks	Ttr_core8	Ttr_core16	Ttr_core32	Ttr_core64
10	31	27.3	27.3	27.3
20	28.18	24.77	24.77	24.77
30	19.06	22.7	23.12	23.12
40	14.65	20.9	22.12	22.12
50	15.12	19.9	20.9	20.9
60	13.66	20.16	22.35	22
70	14.54	14.82	21.68	21.84

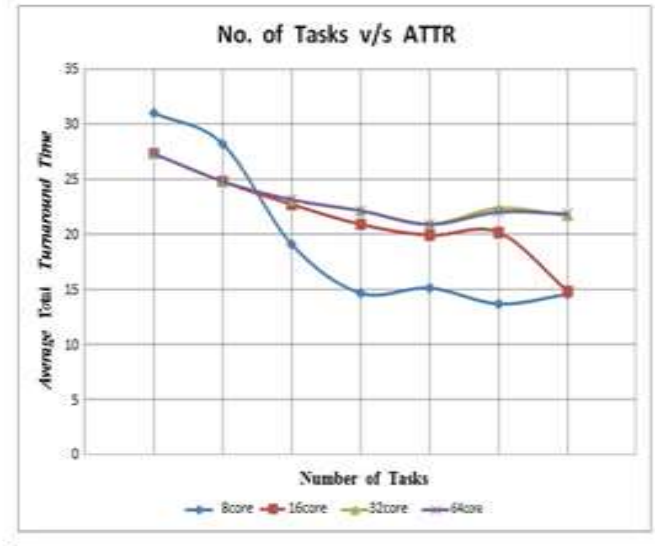

Fig. 1 Number of tasks v/s Average total turnaround time.

In Fig2 the graph between total number of tasks and percentage of tasks completed is shown for different multi-core processors. Data for number of tasks and corresponding ATTR is given in Table 2, based on these data different graphs are plotted between number of tasks and corresponding percentage of tasks completed for 8, 16, 32 and 64 core processors.

Fig. 2 shows that, for same multi-core processor, increase in the number of tasks results decrease in percentage of completed tasks. For same task set if we increase number of cores, then percentage of completed tasks also increases.

Table 2: Number of tasks and percentage of number of completed task

No_Task	Perc_core 8	Perc_core 16	Perc_core 32	Perc_core64
10	100	100	100	100
20	95.45	100	100	100
30	67.74	93.54	100	100
40	53.65	82.93	100	100
50	49.02	80.39	100	100
60	41.66	71	96.67	100
70	40	60	94.28	100

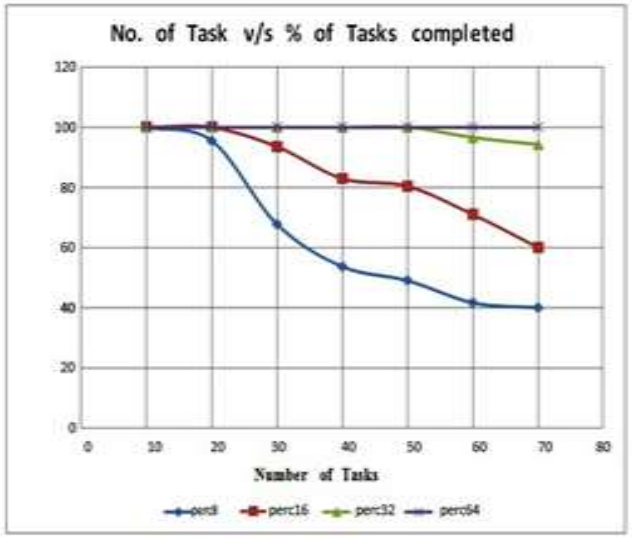

Fig. 2 Percentage of task completed v/s number of tasks

5. Conclusions and Future Work

From the result shown in Fig. 1 and Fig. 2, it is clear that when the number of tasks increases for a multicore processor, if the tasks have almost same value of execution

time then ATTR is increased and if the tasks have different execution time then ATTR is decreased.
For the same dataset, when the number of cores increases then ATTR is decreased. For same multi-core processor, if the number of tasks increases then percentage of completed tasks decreases and if number of cores increases, then percentage of completed tasks also increases

Future scope of this work could be the reduction of task migration overhead and reduction of CPU cycle for enqueue and dequeue.

References

[1] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah, "A categorization of real-time multiprocessor scheduling problems and algorithms". In Joseph Y. Leung, editor, Handbook on Scheduling Algorithms, Methods, and Models, pp. 30.1–30.19, Chapman Hall/CRC, Boca Raton, Florida, 2004.

[2]Juri Lelli, Giuseppe Lipari, Dario Faggioli, Tommaso Cucinotta, "An efficient and scalable implementation of global EDF in Linux", In proceedings of the 7th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), Porto (Portugal), July 2011, pp. 6-16.

[3] Matthew Dellinger, Aaron Lindsay, Binoy Ravindran, "An Experimental Evaluation of the Scalability of Real-Time Scheduling Algorithms on Large-Scale Multicore Platforms", ACM Journal of Experimental Algorithmics, Vol. 17, No. 1, August 2011.

[4] Sophia G. Petridou, Panagiotis G. Sarigiannidis, Georgios I. Papadimitriou and Andreas S. Pomportsis, "Clustering-based scheduling: A new class of scheduling algorithms for single-hop lightwave networks", International Journal Of Communication Systems, 2008, pp. 863-887.

[5] Hakem Beitollahi, Seyed Ghassem Miremadi, Geert Deconinck,"Fault-Tolerant Earliest-Deadline -First Scheduling Algorithm", IEEE International Symposium on Parallel and Distributed Processing, 2007, pp. 1- 6.

[6] Hiroyuki Chishiro and Nobuyuki Yamasaki, "Experimental Evaluation of Global and Partitioned Semi-Fixed-Priority Scheduling Algorithms on Multicore Systems", IEEE 15th International Symposium on, Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2012 pp. 127 – 134.

[7] Jinkyu Lee, Member, Kang G. Shin," Preempt a Job or Not in EDF Scheduling of Uniprocessor Systems" , IEEE Transactions on Computers, 2012, pp. 1-10.

[8] UmaMaheswari C. Devi, Hennadiy Leontyev, and James H. Anderson, "Efficient synchronization under global EDF scheduling on multiprocessors", 18th Euromicro Conference on Real-Time Systems, 2006, pp.74 – 84.

[9] J. M. López, J. L. Díaz, D. F. García, "Utilization Bounds for EDF Scheduling on Real-Time Multiprocessor Systems", Journal on Real Time System, Springer,2004 ,pp39-68 .

[10]Cong Liu and James H. Anderson, Supporting Soft Real-Time Parallel Applications on Multicore Processors, IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2012, pp 114 – 123.

N. Ramasubramanian is an Associate Professor in the Department of Computer Science and Engineering at National Institute of Technology – Tiruchirappalli, India. His research interests are multi-core architecture, advanced digital design and distributed system, reconfigurable system, and real time embedded system.

Ashutosh has completed his graduation in the Computer Science and Engineering at National Institute of Technology-Tiruchirappalli, India. His areas of research include multi-core computing, parallel programming networking and real time embedded system.

Akhilesh Kumar Verma has completed his graduation in the Computer Science and Engineering at National Institute of Technology-Tiruchirappalli, India. His areas of research include multi-core computing, and real time embedded system.

Manvendra Singh has completed his graduation in the Computer Science and Engineering at National Institute of Technology-Tiruchirappalli, India. His areas of research include multi-core computing, parallel programming networking and embedded system.

Praveen Kumar Yadav is a MS research scholar in the Department of Computer Science and Engineering at National Institute of Technology-Tiruchirappalli, India. His areas of research are computer architecture, artificial intelligence, knowledge representation and real time embedded system.