LEARNING SCHEDULER PARAMETERS FOR ADAPTIVE PREEMPTION

Prakhar Ojha[1], Siddhartha R Thota[2], Vani M[1] and Mohit P Tahilianni[1]

[1]Department of Computer Engineering,
National Institute of Technology Karnataka, Surathkal, India
[2]Department of Information Technology,
National Institute of Technology Karnataka, Surathkal, India
prakharojha992@gmail.com, sddhrthrt@gmail.com, vani@nitk.edu.in,
tahilianni@nitk.edu.in

ABSTRACT

An operating system scheduler is expected to not allow processor stay idle if there is any process ready or waiting for its execution. This problem gains more importance as the numbers of processes always outnumber the processors by large margins. It is in this regard that schedulers are provided with the ability to preempt a running process, by following any scheduling algorithm, and give us an illusion of simultaneous running of several processes. A process which is allowed to utilize CPU resources for a fixed quantum of time (termed as timeslice for preemption) and is then preempted for another waiting process. Each of these 'process preemption' leads to considerable overhead of CPU cycles which are valuable resource for runtime execution. In this work we try to utilize the historical performances of a scheduler and predict the nature of current running process, thereby trying to reduce the number of preemptions. We propose a machine-learning module to predict a better performing timeslice which is calculated based on static knowledge base and adaptive reinforcement learning based suggestive module. Results for an "adaptive timeslice parameter" for preemption show good saving on CPU cycles and efficient throughput time.

KEYWORDS

Reinforcement Learning, Online Machine Learning, Operating System, Scheduler, Preemption

1. INTRODUCTION

Scheduling in operating systems is based on time-sharing techniques where several processes are allowed to run "concurrently" so that the CPU time is roughly divided into "slices", one for each runnable process. A single core processor, which can run only one process at any given instant, needs to be time multiplexed for running more processes simultaneously. Whenever a running process is not terminated upon exhausting its quantum time slice, a switch takes place where another process in brought into CPU context. *Linux processes* have the capability of preemption [8]. If a process enters the RUNNING state, the kernel checks whether its priority is greater than the priority of the currently running process. If this condition is satisfies then the execution is interrupted and scheduler is invoked to select the process that just became runnable or any

Jan Zizka et al. (Eds) : ICAITA, SAI, CDKP, Signal, NCO - 2015
pp. 149–162, 2015. © CS & IT-CSCP 2015 DOI : 10.5121/csit.2015.51513

another process to run. Otherwise, a process is also to be preempted when its time quantum expires. This type of time sharing relies upon the interrupts and is transparent to processes.

A natural question to ask would be - *How long should a time quantum last?* The duration, being critical for system performances, should be neither too long nor too short [8]. Excessively short periods will cause system overhead because of large number of task switches. Consider a scenario where every task switch requires 10 milliseconds and the time slice is also set to 10 milliseconds, then at least 50% of the CPU cycles are being dedicated to task switch. On the other hand if quantum duration is too long, processes no longer appear to be executed concurrently[15]. For instance, if the quantum is set to five seconds, each runnable process makes progress for about five seconds, but then it stops for a very long time (typically, five seconds times the number of runnable processes). When a process has exhausted its time quantum, it is preempted and replaced by another runnable process. Every time a process is pushed out to bring in another process for execution (referred as context switch) several other elementary operations like swap-buffers, pipelines clearances, invalidate cache etc. take place making process switch a costly operation. [16] So preemption of a process leads to considerable overhead.

As there does not exist any direct relation between timeslice and other performance metrics, our work proposes a machine-learning module to predict a better performing timeslice. The proposed adaptive time slice for preemption displays improvements in terms of the the total time taken (Turnaround Time) after the submission of process to its completion, in-return creating more processor ticks for future. Most of present work has hard-wired classifiers which are applicable only to certain types of jobs. Having a reinforcement learning agent with reward-function, which learns over time, gives the flexibility of adapting to dynamic systems. The subsequent sections will briefly discuss the fundamentals of reinforcement learning framework, which strives to continuously improve self by learning in any new environment.

Following sections in this paper are organized as follows: Section 2 gives an overview of the related previous works and Section 3 explains the theory of our reinforcement learning framework. Section 4 shows how we approach the problem in hand by proposing a novel design, integrate RL modules and run simmulations and then followed by implementation details of knowledge base created and self-learning systems in Section 5. The results and analysis of our system's performace is evaluated in Section 6 followed by conclusions and discussions in Section 7.

2. RELATED WORKS

Attempts have been made to use historical-data and learn the timeslice parameter, which judges the preemption time for a given process, and make it more adaptive. Below section briefly discusses earlier works in relevant fields by applying machine learning techniques to CPU resources and Operating system parameters.

To remember the previous execution behaviour of certain well-known programs, [10] studies the process times of programs in various similarity states. The knowledge of the program flow sequence (PFS), which characterizes the process execution behaviour, is used to extend the CPU time slice of a process. They also use thresholding techniques by scaling some feature to determine the time limit for context switching. Their experimental results show that overall processing time is reduced for known programs. Works related to Thread schedulers on multi

core systems, using Reinforcement learning, assigns threads to different CPU cores [6], made a case that a scheduler must balance between three objectives: optimal performance, fair CPU sharing and balanced core assignment. They also showed that unbalanced core assignment results in performance jitter and inconsistent priority enforcement. A simple fix that eliminates jitter and presents a scheduling framework that balances these three objectives by algorithm based on reinforcement learning was explored.

The work in [7] has addressed similar problem based on making fixed classifiers over hand picked features. Here timeslice values were tried against several combination of attributes and patterns emerged for chosing better heuristic. However, their approach was compatible to only few common processes like random number generation, sorting etc. and unlike our work, not universally adaptive for any application. Reward based algorithms and their use in resolving the lock contention has been considered as scheduling problem in some the earlier works[2]. These hierarchal spin-locks are developed and priority assigned to processes to schedule the critical-section access.

Application run times are predicted using historical information in [1]. They derive predictions for run times of parallel applications from the run times of similar applications that have executed in the past. They use some of the following characteristics to define similarity: user, queue, load leveler script, arguments, network adapter, number of nodes, maximum run time, submission time, start time, run time. These characteristics are used to make a template which can find the similarity by matching. They use genetic algorithms (GA), which are are well known for exploring large search spaces, for identifying good templates for a particular workload.

Statistical Regression methods, which work well on numeric data but not over nominal data, are used for prediction [5]. An application signature model for predicting performance is proposed in [4] over a given grid of resources. It presents a general methodology for online scheduling of parallel jobs onto multi-processor servers, in a soft real-time environment. This model introduces the notion of application intrinsic behaviour to separate the performance effects of the runtime system from the behaviour inherent in the application itself. Reinforcement Learning is used for tuning its own value function which predicts the average future utility per time step obtained from completed jobs based on the dynamically observed state information. From this brief review of related literature, we draw the following conclusions:

- It is possible to profitably predict the scheduling behaviour of programs. Due to the varied results in all above discussed works, we believe that the success of the approach depends upon the ML technique used to train on previous programs execution behaviour.

- A suitable characterization of the program attributes (features) is necessary for these automated machine learning techniques to succeed in prediction.

In specific to using reinforcement learning in realms of scheduling algorithms, most of the work is concentrated around ordering the processes like to learn better permutations of given list of processes, unlike our work of parameter estimation.

3. REINFORCEMENT LEARNING FRAMEWORK

Reinforcement learning (RL) is a collection of methods for approximating optimal solutions to stochastic sequential decision problems [6]. An RL system does not require a teacher to specify correct actions, instead, it tries different actions and observes their consequences to determine which actions are best. More specifically, in any RL framework, a learning agent interacts with its environment over a series of discrete time steps $t = 0, 1, 2, 3. . .$ Refer *Figure.1*. At each time t, the agent observes the environment state s_t , and chooses an action a_t , which causes the environment to transition to a new state s_{t+1}, and to reward the agent with r_{t+1} . In a Markovian system, the next state and reward depend only on the current state and present action taken, in a stochastic manner. To clarify notation used below, in a system with discrete number of states, S is the set of states. Likewise, A is the set of all possible actions and $A(s)$ is the set of actions available in states. The objective of the agent is to learn to maximize the expected value of reward received over time. It does this by learning a (possibly stochastic) mapping from states to actions called a policy. More precisely, the objective is to choose each action at so as to maximize the expected return R, given by,

$$R := E\left(\sum_{i=0}^{\infty} \gamma^i \, r_{t+i+1}\right) \qquad (1)$$

where γ is the discount-rate parameter in range [0,1] , which allows the agent to trade-off between the immediate reward and future possible rewards.

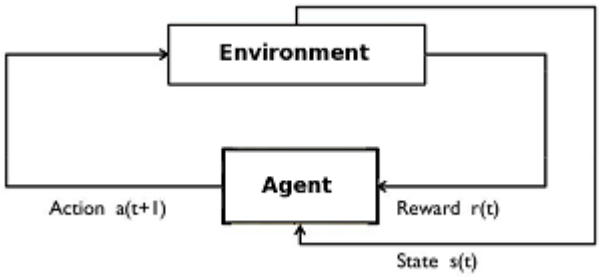


Fig.1 Concept of Reinforcement learning depicting iteraction between agent and environment

Two common solution strategies for learning an optimal policy are to approximate the optimal value function, V*, or the optimal action-value function, Q*. The optimal value function maps each state to the maximum expected return that can be obtained starting in that state and thereafter always taking the best actions. With the optimal value function and knowledge of the probable consequences of each action from each state, the agent can choose an optimal policy. For control problems where the consequences of each action are not necessarily known, a related strategy is to approximate Q*, which maps each state and action to the maximum expected return starting from the given state, assuming that the specified action is taken, and that optimal actions are chosen thereafter. Both V* and Q* can be defined using *Bellman -equations* as

$$Q^*(s,a) = \sum_{s' \epsilon S} P^a_{ss'} \left[R^a_{ss'} + \gamma \max_{a' \epsilon A(s')} Q^*(s', a') \right] \qquad (2)$$

where s' is the state at next time step, $P^a_{ss'}$ is its probability of transission and $R^a_{ss'}$ is the associated reward.

4. OUR APPROACH

4.1. Problem formulation

In this paper, we want to study the application of machine-learning in operating systems and build learning modules so as to make the timeslice parameter flexible and adaptive. Our aim is to maintain the generality of our program so that it can be employed and learned in any environment. We also want to analyze how long it takes for a module to learn from its own experiences so that it can be usefully harnessed to save time. Our main approach is to employ reinforcement learning techniques for addressing this issue of continuous improvement. We want to formulate our learning through the reward-function which can self-evaluate its performance and improve overtime.

Our prime motivation is to reduce the redundant preemptions which current schedulers do not take into account. To explain using a simple example, suppose a process has a very little burst time left and it is swapped due to the completion of its timeslice ticks, then the overhead of cache-invalidation, pipeline clearing, context switching etc. reduces the efficiency. Hence having a flexible timeslice window will prevent the above scenario. This would also improve the total time taken after the submission of process to its completion, in-return creating more processor ticks for future.

4.2. Module Design

Figure.2 gives an over all view of our entire system. It describes how our reinforcement learning agent makes use of the patterns learned initially and later on after having enough experiences it develops a policy of itself to use the prior history and reward-function.

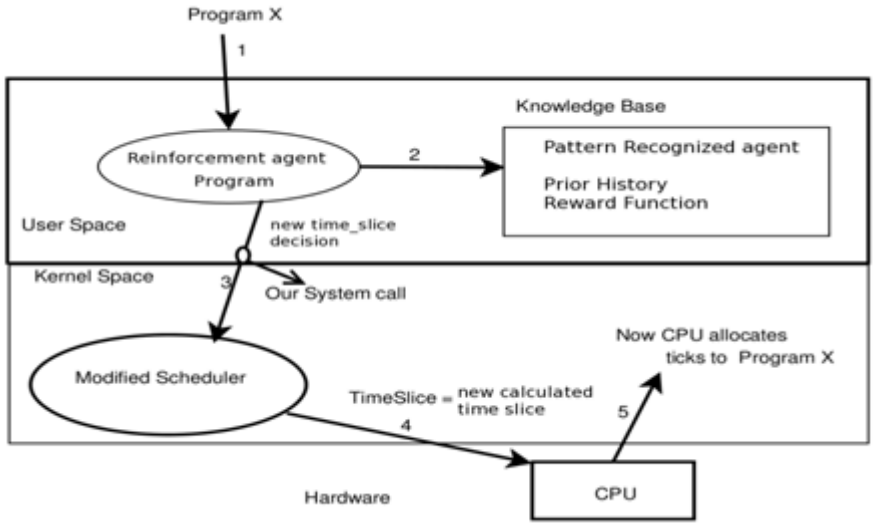


Fig.2 Bird's eye view of our design and implementation pipeline

Formally, these below steps capture the important end-to-end flow mechanism.

1. Program X passes its requirements in user-space for acquiring resources from computer hardware. These requirements are received by our agent.

2. Reinforcement learning agent uses its knowledge base to make decision. It uses patterns recognized in the initial stages to have a kick start with reasonable values and not random values. Later on knowledge base develops its history and reward function after sufficient number of experiences.

3. The information is passed from the user-space to kernel-space via a system call which will have to be coded by us. This kernel call will redirect the resource request to our modified scheduler.

4. The number of ticks to be allocated is found in the fields of new_timeslice and forwarded to CPU. And finally, CPU allocates these received orders in form of new ticks.

As the intermediate system call and modified scheduler are the only changes required in the existing systems, we provide complete abstraction to the CPU and user-space.

4.3. Modelling an RL agent

We present here a model to simulate and understand the Reinforcement learning concepts and understand the updates of Bellman equation in greater depth [6]. We have created this software with an aim to visualize the results of changing certain parameters of RL functions and as a precursor for modelling scheduler.

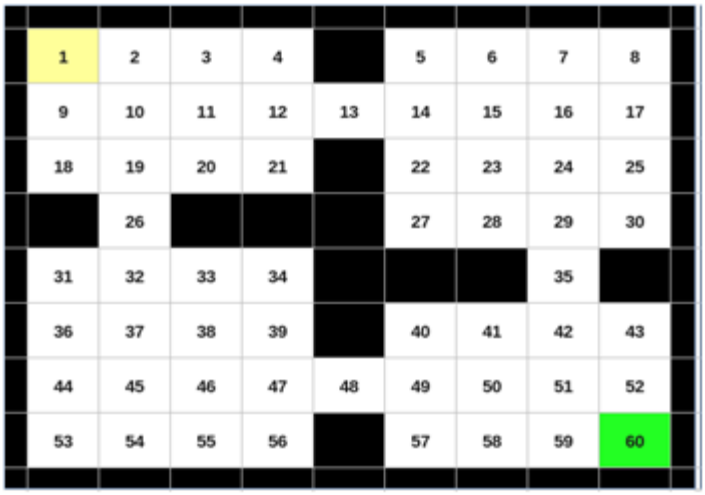


Fig.3 Maze showing the environment in which RL-agent interacts.

- *Work-Space*: Checkerboard setup with a grid like maze.

- *Aim*: To design an agent which finds its own way from the start state to goal state. The agent is designed to explore the possible paths from start state and arrive at goal state. Each state has four possible actions N, S, E & W. Collision with wall has no effect.

- *Description*: *Figure.3* depicts the maze which consists of rooms and walls. The whole arena is broken into states. Walls are depicted by dark-black solid blocks denoting that the agent cannot occupy these positions. The other blocks are number 1,2,3.....60 as the

possible states in which agent can be. Agent is situated at S_1 at time t=0 and at every future action it tries to find its way to the goal state S_{60}.

- *Reward-function*: Transition to goal state gives a reward of +100. Living penalty is 0. Hence the agent can take as long time as it wants to learn the optimal policy. This parameter will be changed in case of real time schedulers. *Reward Updating policy* has Temporal difference updates with learning rate (alpha) =0.25

Initially the agent is not aware of its environment and explores it to find out. Later it learns a policy to make that wise decision about its path finding. Code (made publicly available) is written in C language for faster excution time and the output is an HTML file to help better visualize the reward updates and policy learned. Results and policies learned will be described in later sections.

4.4. Simulation

As the scheduler resides deep in the kernel, measuring the efficacy of scheduling policies in Linux is difficult. Tracing can actually change the behavior of scheduler and hide defects or inefficiencies. For example, an invalid memory reference in the scheduler will almost certainly crash the system [8]. Debugging information is limited and not easily obtained or understood by new developer. This combination of long crash-reboot cycles and limited debugging information can result in a time-consuming development process. Hence we resort to a good simulator of the Linux scheduler which we can manipulate for verifying our experiments instead of changing kernel directly.

LinSched: Linux Scheduler simulation

LinSched is a Linux scheduler simulator that resides in user space [11]. It isolates the scheduler subsystem and builds enough of the kernel environment around it that it can be executed within user space. Its behaviour can be understood by collecting relevant data through a set of available APIs. Its wrappers and simulation engine source is actually a Linux distribution. As LinSched uses the Linux scheduler within its simulation, it is much simpler to make changes, and then integrate them back into the kernel.

We would like to mention few of the essential simulator side APIs below, which we experimented over. One can utilize them to emulate the system calls and program the tasks. They are used to test any policy which are under development and see the results beforehand implementing at kernel directly. *linsched_create_RTrr(...)* -creates a normal task and sets the scheduling policy to normal. *void linsched_run_sim(...)* -begins a simulation. It accepts as its only argument the number of ticks to run. At each tick, the potential for a scheduling decision is made and returns when it is complete. Few statistics commands like *void linsched_print_task_ stats()* and *void linsched_print_group_stats()* give more detailed analysis about a task we use. We can find the total execution time for the task (with *task_exec_time(task)*), time spent not running (*task->sched_info.run_delay*), and the number of times the scheduler invoked the task (*task->sched_info.pcount*).

We conducted several experiments over the simulator on normal batch of jobs by supplying it work load in terms of process creation. First 2 normal tasks are created with no difference and

ambiguity (using *linsched_create_normal_task(...)*). We next created a job which runs on normal scheduler and has a higher priority by assigning *nice value* as -5. Similarly we experimented with jobs which had lower priority of +5, followed by populating another normal and neutral priority. On the other hand, we also verified our experiments over batch tasks which are created with low and high priorities. They are all computation intensive tasks which run in blocks or batches. (using *linsched_create_batch_task(...)*). And then finally one real-time FIFO task with priority varying in range of 50-90, and one round-robin real-time task with similar priority range. Each task as created is assigned with task_id which is realistic as in real linux machines. Initially all tasks are created one after other and then after *scheduler_tick()* function times out, it is called for taking decision on other processes in waiting/ready queue. The relevant results will be discussed in subsequent sections.

5. IMPLEMENTATION AND EXPERIMENTS

5.1. Knowledge Base Creation

5.1.1. Creating Dataset

To characterize the program execution behaviour, we needed to find the static and dynamic characteristics. We used readelf and size commands to get the attributes. We built the data set of approximately 80 execution instances of five programs: matrix multiplication, quick sort, merge sort, heap sort and a recursive Fibonacci number generator. For instance, a script ran matrix multiplication program of size 700 x 700 multiple times with different nice values and selected the special time slice (STS), which gave minimum Turn Around Time (TaT). After collecting the data for the above programs with different input sizes, all of them were mapped to the best priority value. Data of the above 84 instances of the five programs were then classified into 11 categories based on the attribute time slice classes with each class having an interval of 50 ticks.

We mapped the variance of timeslice against total Turnaround Time (TaT) taken by various processes like Insertion sort, Merge sort, Quick sort, Heap sorts and Matrix multiplication with input ranging from 1e4, 1e5, 1e6 after experimenting against all possible timeslices.

5.1.2. Processing Dataset

After extracting the features from executable filles, by readelf and size commands, we refine the number of attributes to only those few essential features which actually help in taking decision. A few significant deciding features which were later used for building decision tree are: *RoData* (read only data), *Hash* (size of hash table), *Bss* (size of uninitialized memory), *DynSym* (size of dynamic linking table), *StrTab* (size of string table). The less varying / non-deciding features are discarded. The best ranked special time slices to each instance to gauge were classified to the corresponding output of decision tree. The processed result was further fed as input to to the classifier algorithm (decision trees in our case) to build iterative if-else condition.

5.1.3. Classification of Data

To handle new incoming we have built a classifier with attributes obtained from previous steps. Decision tree rules are generated as the output from classification algorithm. We used WEKA (Knowledge analysis tool) to model these classifiers. Most important identified features are

RoData , *Bss* and *Hash* . Finally groups are classified into 20 classes in ranges of timeslice. Few instances for Decision Tree Rules are mentioned below.

- *if* {(RoData<=72) AND (bss <= 36000032) AND (bss <= 4800032) AND (bss <=3200032) } *then* class=**13**
- *if* {(RoData<=72) AND (bss <= 36000032) AND (bss <= 4800032) AND (bss > 3200032) } *then* class=**2**
- *if* {(RoData<=72) AND (bss <= 36000032) AND (bss > 4800032) AND (bss <= 7300032) } *then* class=**5**
- *if* {(RoData<=72) AND (bss <= 36000032) AND (bss > 4800032) AND (bss > 7300032) AND (bss <= 2000032) } *then* class=**3**
- *if* {(RoData<=72) AND (bss > 36000032) AND (bss <= 4800032) } *then* class=**7**
- *if* {(RoData<=72) AND (bss <= 36000032) AND (bss > 4800032) AND (bss > 7300032) AND (bss > 2000032) } *then* class=**0**
- *if* {(RoData<=72) AND (bss > 36000032) AND (bss <= 4800032) } *then* class=**4**

To give a better visualization of out features, we present in *Table.1* various statistics obtained for Heap sort with input size 3e5 and priority (nice value) set to 4. These statistics help us decide the lowest Turnaround Time and lowest number of swaps taken for best priority class.

Table.1 Statistics obtained for Heap sort with input size 3e5 showing the classifier features.

Feature Name	Value	Feature Name	Value
User time (seconds)	0.37	Voluntary context switches	1
Minor (reclaiming frame) page faults	743	Involuntary context switches	41
Percent of CPU this job got	98%	File system outputs	8
Elapsed (wall clock) time (h:mm:ss)	0:00.38	Socket messages sent	0
Maximum resident set size (kbytes)	1632	Socket messages received	0
Signals delivered	0	Page size (bytes)	4096

5.2. Self-Improving Module

The self-learning module which is based on Reinforcement learning technique is proved to improve over time with its experience until converged to saturation. The input to this module is the group decision from the knowledge base in the previous step as the output of the if-else clause. Further, reinforcement learning module may give a new class if it decides from its policy learned over time of several running experiences. In the background this self improving module would explore for new classes which it could assign to a new incoming process. We modelled the scheduler actions as a markov decision process where decisions for assigning a new time slice solely based over current state and it need not have to take into account of the previous decisions. The policy mapping for states and their aggregate reward associated is done using the Bellman equations

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} P^a_{ss'} [R^a_{ss'} + \gamma V^*(s')] \quad (3)$$

Figure.4 shows how using an array implementation of Doubly Linked List, we generated the above module. Temporal difference (TD method) was used for updating the reward-function with experiences and time. The sense of reward for the scheduler agent was set to be a function of inverse of waiting time of the process. The choice of such a reward function was to avoid the bias introduced by the inverse of total turnaround time (TaT), which is the least where compared to waiting time. This is because TaT is also inclusive of total number of swaps which in turn is dependent over the size of input and size of text, whereas waiting time does not depend over the size of input. We set the exploration vs. exploitation constant to be 0.2 which is still flexible under temperature coefficient mentioned above.

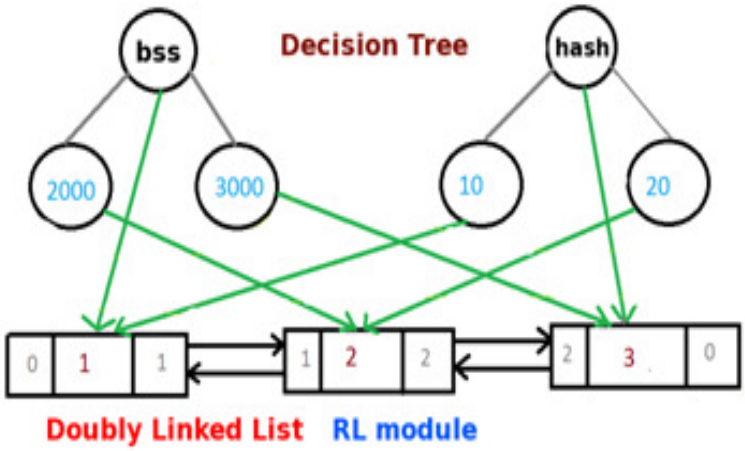


Fig.4 Integration of self learning module with decision tree knowledge base.

Input to this module is the class decision from knowledge base obtained in the previous step which is the output of the decision tree. It outputs a new class which RL module decides from its policy generated over time of running. Reward sense is given by the inverse of waiting time of the process. We have used exploration vs. exploitation ε-greedy constant as 0.2.

In our experimental Setup, we used WEKA (Knowledge analysis tool) for Decision trees and attribute selection. For compilation of all programs we used gcc (GNU_GCC) 4.5.1 (RedHat 4.5.1-4). To extract the attributes from executable/binary we used readelf & size command tools. For graph plots and mathematical calculations we used Octave.

6. RESULTS AND ANALYSIS

Below we present a few test cases which characterise the general behaviour of scheduler interaction with knowledge base and self improving module. We also analyse the effectiveness of integrating Static knowledge base and self-learning module by calculating time saved and number of CPU cycles conserved. Programs were verfified after executing multiple times with different nice values on Linux System. Their corresponding figures show how the turn-around-time changed as the CPU allotted timeslice of the process changed.

Experiments show that there does not exist any direct evident relation between time slice and CPU utilization performance metrics. Refer *Figure.5* and *Figure.6* plot of TaT vs. timeslice class allotted. Hence it is not a simple linear function which is monotonic in nature. One will have to learn a proper classifier which can learn the pattern and predict optimal timeslice. Below we show the analysis for 900x900 matrix multiplcation and merge sort (input size 3e6). *Table.2* shows their new suggested class from knowledge base. For Heapsort (input size 6e5) and Quicksort (input size 1e6) we have only plotted their TaT vs. Timeslice graphs in *Figure.6*, which is similar in wavy nature as *Figure.5*. We have omitted explicit calculations to prevent redundancy in paper, as their nature is very similar to previous matrix multiplication.

Effectiveness analysis for Matrix Multiplication with input size of 900x900 random matrix elements.

- Turnaround Time (normal) - 27872216 ms
- Turnaround Time (with KB)- 24905490 ms
- Time saved = 2966726 ms
- Time saved per second - 109879 ms
- No. of clock cycles saved - 2.4MHz x 109879
- No. of Lower operations saved - 109879 / (pipeline clear + context switch etc.)

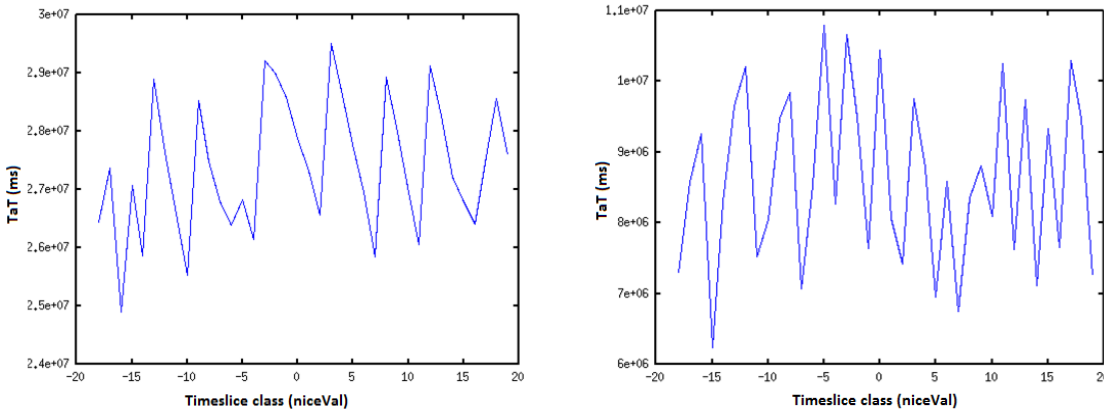


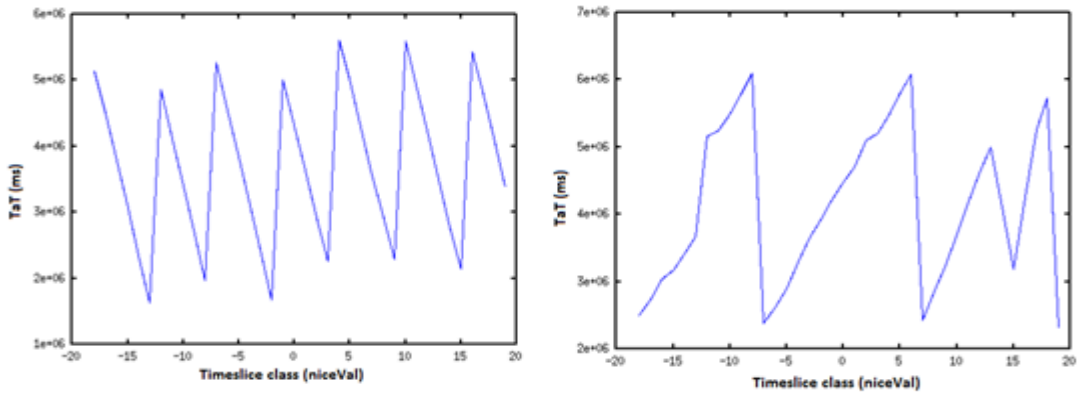
Fig.5 Timeslice class (unnormalized nice values) vs. Turn around time for (a)Matrix Multiplication and (b)Merge Sort .

Table.2 Optimal timeslice-class decisions made by knowledge base module for Matrix multiplication of input 900x900 and Merge sort over input size 3e6 elements.

Matrix Multiplcation		Merge Sort	
Turn around time (microsec)	Timeslice class suggested	Turn around time (microsec)	Timeslice class suggested
24905490	16	6236901	15
25529649	10	7067141	7
25872445	14	7305964	18
26151444	4	7524301	11
26396064	6	7639436	1
26442902	18	8055423	10
26577882	11	8273131	4
26800116	7	8302461	14
26827546	5	8537245	6
27080158	15	8569818	17
27376257	17	9255897	16
27484162	8	9483901	9
27643193	12	9499732	2
28535686	9	9660585	13
28581739	1	9844913	8
28900769	13	10217774	12

Effectiveness analysis for Merge Sort with input size of 3e6 random array elements.

- TaT (normal) - 10439931 ms and TaT(with KB)- 6236901 ms
- Time saved = 4203030 ms
- Time saved per second - 382093 ms
- No. of clock cycles saved - 2.4MHz x 382093
- No. of Lower operations saved - 382093 / (pipeline clear + context switch etc.)



(a) (b)

Fig.6 Timeslice class (unnormalized nice values) vs. Turn around time for (a)Heap Sort, (b)Quick Sort .

7. CONCLUSION

From the results we can observe that the turnaround time can be optimized by reducing redundant context switches and also reducing the additional lower level register swaps, pipeline clearances etc. This in turn saves the CPU cycles which are valuable resource for runtime execution of subsequent jobs. A self-learning module proposed here has the potential of constantly improving with more experiences and is provided over a knowledge base to prevent the problem of cold-start. We have showed the non-intuitive irregularity between decreasing turnaround time and increasing time slice by wave-pattern of TaT vs. class of time. The machine learning module identifies specific time slice for given task so that its resource usage and TaT are optimized. We are also currently investigating ways to address the problem of infinite horizon in reinforcement learning, as the scheduler may run for infinite amount of time (or very large time unit) and scores rewards just for the sake of its existence. The agent may actually be performing sub-optimally, but its prolonged existence keeps collecting rewards and show positive results. This issue can be addressed by refreshing the scheduler after certain time unit, but clarity is required over how to calculate its optimal refresh period.

REFERENCES

[1] Warren Smith, Valerie Taylor, Ian Foster, Predicting Application Run-Times Using Historical Information", Job Scheduling Strategies for Parallel Processing, IPPS/SPDP'98 Workshop, March, 1998.

[2] Jonathan M Eastep, "Smart data structures: An online ML approach to multicore Data structure", IEEE Real-Time and Embedded Technology and Applications Symposium 2011.

[3] M. John Calandrino , documentation for the main source code for LinSched and author the linux_linsched files LINK: http://www.cs.unc.edu/~jmc/linsched/

[4] D. Vengerov, A reinforcement learning approach to dynamic resource scheduling allocation, Engineering Applications of Artificial Intelligence, vol. 20, no 3, p. 383-390, Elsevier, 2007.

[5] Richard Gibbons, A Historical Application Profiler for Use by Parallel Schedulers, Lecture Notes on Computer Science, Volume : 1297, pp: 58-75, 1997.

[6] Richard S. Sutton and Andrew G. Barto. , Reinforcement Learning: An Introduction. A Bradford Book. The MIT Press Cambridge, Massachusetts London, England.

[7] Atul Negi, Kishore Kumar. P, UOHYD, Applying machine learning techniques to improve Linux process scheduling 2010.

[8] Internals of Linux kernel and documentation for interface modules LINK: http://www.faqs.org/docs/kernel_2_4/lki-2.html

[9] D. Vengerov, A reinforcement learning framework for utility-based scheduling in resource-constrained systems, Future Generation Compute Systems, vol. 25, p. 728-736 Elsevier, 2009.

[10] Surkanya Suranauwarat, Hide Taniguchi, The Design, Implementation and Initial Evaluation of An Advanced Knowledge-based Process Scheduler, ACM SIGOPS Operating Systems Review, volume: 35, pp: 61-81, October, 2001.

[11] Documentation for IBM project for real scheduler simulator in User space LINK: http://www.ibm.com/developerworks/library/l-linux-schedulersimulator/

[12] Tong Li, Jessica C. Young, John M. Calandrino, Dan P. Baumberger, and Scott Hahn , LinSched: The Linux Scheduler Simulator Research Paper by Systems Technology Lab Intel Corporation, 2008

[13] McGovern, A., Moss, E., and Barto, A. G. (2002). Building a basic block instruction scheduler with reinforcement learning and rollouts. Machine Learning, 49(2/3):141– 160.

[14] Martin Stolle and Doina Precup , Learning Options in Reinforcement Learning Springer-Verlag Berlin Heidelberg 2002

[15] Danie P. Bovet, Marc, Understanding the Linux Kernel, 2nd ed, O' Reilly and Associates, Dec., 2002.

[16] Modern Operation System Scheduler Simulator, development work for simulating LINK: http://www.ontko.com/moss/
[17] Andrew Marks, A Dynamically Adaptive CPU Scheduler, Department of Computer Science, Santa Clara University, pp :5- 9, June, 2003.

AUTHORS

Prakhar Ojha, student of the Department of Computer Science Engineering at National Institute of Technology Karnataka Surathkal, India. His areas of interest are Artificial Intelligence, Reinforcement Learning and Application of knowledge bases for smart decision making.

Siddhartha R Thota, student of the Department of Information Technology at National Institute of Technology Karnataka Surathkal, India. His areas of interest are Machine Learning, Natural language processing and hidden markov model based Speech processing.

Vani M is a Associate Professor in the Computer Science Engineering Department of NITK. She has over 18 years of teaching experience. Her research interest includes Algorithmic graph theory, Operating systems and Algorithms for wireless sensor networks.

Mohit P Tahiliani is a Assistant Professor in the Computer Science Engineering Department of NITK. His research interest includes Named Data Networks, TCP Congestion Control, Bufferbloat, Active Queue Management (AQM) mechanisms and Routing Protocol Design and Engineering.