Applying Machine Learning Techniques to Improve Linux Process Scheduling

Atul Negi
Department of Computer and Information Sciences
University of Hyderabad
Hyderabad, INDIA 500046
Email:atulcs@uohyd.ernet.in

Kishore Kumar P.
Member of IT
DE Shaw India Software Pvt Ltd.,
Hyderabad, INDIA 500016
Email: kishoregupta_os@yahoo.com

***Abstract*— In this work we use Machine Learning (ML) techniques to *learn* the CPU time-slice utilization behavior of known programs in a Linux system. Learning is done by an analysis of certain *static* and *dynamic attributes* of the processes while they are being run. Our objective was to discover the most important *static* and *dynamic attributes* of the processes that can help best in prediction of CPU burst times which minimize the process *TaT* (Turn-around-Time). In our experimentation we modify the Linux Kernel scheduler (version 2.4.20-8) to allow scheduling with customized time slices. The "Waikato Environment for Knowledge Analysis" (Weka), an open source machine-learning tool is used to find the most suitable ML method to characterize our programs. We experimentally find that the *C4.5 Decision Tree algorithm* most effectively solved the problem. We find that predictive scheduling could reduce TaT in the range of 1.4% to 5.8%. This was due to a reduction in the number of *context switches* needed to complete the process execution. We find our result interesting in the context that generally operating systems presently never make use of a program's previous execution history in their scheduling behavior.**

I. Introduction

Generally process schedulers allocate CPU time slices to a process according to a scheduling algorithm that does not use any previous execution history of the process. Here we suggest that better resource utilization could be done if we "recognize" a program and predict its resource requirements. For example, consider a process which exhausted its allocated CPU time and is pre-empted although it needed a small slice of additional CPU time for completion. Such preemption, increases number of *context switches* (also called as a process switch or task switch). It causes invalidation of caches and pipelines, swapping of buffers and so on [6]. Thus ultimately this increases *TaT* of the program. (TaT is the delay between process submission and process completion). Thus we observe that by characterizing or recognizing programs it may be possible to understand their previous execution history and predict their resource requirements. In this paper we address the problem of, how to minimize the TaT of programs by using ML techniques. We discover certain static and dynamic characteristics of a program are taken as features which the machine learning techniques use to predict CPU burst times. We call Special Time Slice or STS as the CPU burst time that minimizes turnaround time.

The rest of this paper first discusses related work in Section II, an overview of machine learning techniques in Section III, we review Linux process scheduling in Section IV, and then describes our implementation in Section V. Section VI describes our experimentation. Section VII presents the conclusions and describes possible future work.

II. Related Work

The work [10] by Suranauwarat and Taniguchi, presents an approach to remember the previous execution behavior of certain well-known programs. They study the process times of these programs in various similarity states. The knowledge of the *program flow sequence* (PFS) is used to extend the CPU time slice of a process. PFS of a process is computed from its past execution history. PFS characterizes the process execution behavior and is used to decide whether the program executing currently needs additional time. They conclude from experimental results that overall processing time is reduced for known programs. To schedule a process they search for its name in the PFS knowledge base and thus improve its behavior.

In the paper by Smith et al. [7], the authors predicted the application run times using historical information. They present a technique for deriving predictions for the run times of parallel applications from the run times of *similar* applications that have executed in the past. They use 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 for identifying "good templates" for a particular workload. A template defines the similarity between two applications. They use Genetic Algorithm (GA) techniques to determine those application characteristics that yield the best definition of similarity for making predictions. GAs are probabilistic techniques and are well known for exploring large search spaces. However from a machine learning perspective, GAs are expensive in terms of computation [15] and also their results are considered fragile.

In the paper by Gibbons [5], the authors used Statistical Regression methods for prediction. While regression methods

work well on numeric data but cannot readily be used for nominal data. In the paper by Fredrik et al. [4], the authors described an *application signature model* used for predicting application performance on a given set of grid resources. In this model, they introduced the notion of application intrinsic behaviour to separate the performance effects of the runtime system from the behaviour inherent in the application itself. The signature model is used as the basis for performance predictions. So their approach combines the knowledge of application intrinsic behaviour with run-time predictions of resources. They also define *application intrinsic metrics* as metrics that are solely dependent on the application code and problem parameters.

From this brief review of related literature, we draw the following conclusions:

- It is possible to profitably predict scheduling behavior.
- However the success of the approach depends upon the ML technique used to train on previous programs execution behavior.
- More importantly for the ML techniques to succeed, it needs a suitable characterization of the program attributes (features) that are most significant for prediction.

III. An introduction to machine learning techniques used

In this section we bring in some background information about the machine learning techniques which were used in the experimentation. The field of machine learning (ML) [15] is concerned with the question of how to construct computer programs that automatically improve with experience. ML methods are also similar to Pattern Recognition methods as [16]. The most important aspect is that while in the training or learning stage several attributes are used to describe the objects being studied. In our case the objects being studied are programs and we have come up with the best attributes that can describe programs for prediction. [8]. However the success of a ML method also depends upon the classifiers used. Certain classifiers are more successful for different data or problem domains. Here we just give a cursory introduction to the two most effective methods which were used [8].

The Learning Algorithms (Classifiers): For our experiments, we selected a representative set of standard machine learning algorithms with different model classes. All of these are available in the "Waikato Environment for Knowledge Analysis (Weka) [9] , an open source machine learning tool. The selected learner types (classifier types) were: "Trees" and "Lazy" which are listed below:

- C4.5 (or J48 in Weka [9]) is a Reduced-Pruned Decision Tree Learner. Decision Tree learning is one of the most widely used and practical methods for inductive inference. It is a method for approximating discrete-valued functions, and is robust to noisy data and capable of learning disjunctive expressions.
- IB k Learner (k-NN): This is an instance based learner and assumes all instances are points in an n-dimensional space. The nearest neighbors (NN) of an instance are defined in terms of the standard Euclidean Distance. The class label of a new instance is found from the k instances nearest to it by assigning to it the majority label of the k-NN [15] [9].

IV. A review of Linux process scheduling concepts

In this section, we review the Linux process scheduling concepts which we use to develop the modified scheduler as described in Section V.

Scheduler: The idea behind a scheduler is simple. Its objective is to best utilize processor time. Assuming there are runnable processes, a process should always be running. If there are more processes than processors in a system, some processes will not always be running. These processes are waiting to run. Deciding which process runs next, given a set of runnable processes, is the fundamental decision the scheduler must take.

Time slice: Linux uses a tick to allocate processor time to individual processes. There are several different hardware timers that the kernel can access. The kernel uses the CPU clock speed given as HZ in the kernel to set the length of a single tick to approximately 2.5 nanoseconds [2]. Consequently, every 2.5 nanoseconds, the system takes over, decrements the amount of ticks that the current process has, and returns the CPU to the process for another tick. If the process running is out of CPU time, the scheduler is invoked. If there are more processes in the run-queue that have been allocated CPU time, the scheduler will find the best one and switch to it. The best process is selected using the $task\_timeslice()$ macro defined in $sched.c$. This $task\_timeslice()$ macro calculates a weight for the process using bonus which is calculated from *nice* values etc.. The highest weight is the best process to schedule [14] [2].

Process Descriptor: The process descriptor [13] is defined in $sched.h$ and holds all information about a process. The amount of ticks of CPU time that a process receives before another process is scheduled is stored in the process descriptor. When all processes in the run-queue have exhausted their CPU time, the scheduler recalculates the amount of CPU time for each process using the macro $task\_timeslice()$. The $task\_timeslice()$ uses the process' *nice* value in the process descriptor to determine the number of ticks the process gets. Here a process is referred to as *p*. Variables in the process descriptor are referred to as *p→variable*. For example *p→nice* refers to the nice value of process *p*.

Context switching: The *context* of a process [3] consists of the contents of its (user) address space and the contents of hardware registers and kernel data structures that relate to the process. In multi-processing environments, a *context switch* (or process switch or task switch) is when one process is suspended from execution , its context is recorded and another process starts its execution in the CPU.

V. Implementation

In this section, we briefly describe the kernel modifications and process characterization in our implementation. The kernel

modifications subsection shows the kernel data structures modified for finding STS. The process characterization section shows, how we find the static and dynamic characteristics of a process.

A. Kernel modifications

Our idea of kernel modifications is similar to that of [12], but the difference is that we describe the modifications to O(1) scheduler of Linux Kernel 2.4.20-8.

1) Process descriptor: The Linux process descriptor is defined in $sched.h$ as a structure called *task_struct*. More than one process can request a larger amount of processor time and the amount of extra time to be given is specific to the process. Therefore the extra time a process gets is stored in the process descriptor. A single integer field, *special_time_slice* is added to the process descriptor and is sufficient to store the number of ticks that a CPU-bound process should receive. This single variable can also be used to distinguish between CPU-bound process from the other processes in the system. If *p→special_time_slice* is less than 0 then this process should be allocated the standard amount of ticks defined by the macro $task\_timeslice()$ in $sched.c$. If *p→special_time_slice* is greater than 0 then the process is given *special_time_slice* ticks of processing time instead of using the standard macro, *task_timeslice()*. This requires that when processes are created, the *special_time_slice* variable must be initialized to -1 in *fork.c*. The next section describes how the modified scheduler uses the *special_time_slice* variable in the process descriptor.

2) System calls: Two system calls have been added to the kernel for controlling CPU-bound processes. One system call is necessary to allocate more processor time for a single process. This system call performs two simple tasks. First, the *special_time_slice* variable is set to the number of ticks requested by the process via an argument to the system call. Second, the *time_slice* variable of the process descriptor is set to the value passed to the system call. The process then returns to run on the processor with an increased amount of processor time and special handling from the scheduler. Therefore the process gets *special_time_slice* ticks of processor time, each time the scheduler recalculates the process time_slices.

The second system call is used to return a process to the normal state in which *p→time_slice* is calculated using *task_timeslice()* and *p→special_time_slice* is less than 0. The next time the scheduler runs the *time_slice* is set to *task_timeslice(p)*, which is the standard method of assigning *p→time_slice*.

3) Modifications to O(1) scheduler data structures: The *scheduler_tick()* function of $sched.c$ has been modified to handle any process with *special_time_slice* > 0 differently from other processes. If this value is exhausted (decremented to 0) then the scheduler gives the processor to the next process. The run-queue is a circular doubly linked list of process descriptors in the *TASK_RUNNING* state. The first thing that the new scheduler does is check whether process currently running has been granted more processor time (*p→ special_time_slice > 0*). If this is true, and the process still has some CPU time left (*p→time_slice > 0*), then the scheduler immediately gives the processor back to the current process. This guarantees that each processor-bound process uses all of its processor time immediately, even if the scheduler is called in the middle of its running. If the CPU-bound process has used all of its CPU time (*p→time_slice > 0*), then the scheduler continues its normal operation and switch to the next process.

Calculating the goodness [2] for each process remains the same. Since CPU-bound processes have large time_slice values, they are not favored over other processes. This is acceptable since the CPU-bound processes are typically the only things using the processor on a system. The CPU-bound process may be last to run, but there should not be much waiting to run in front of them, and from the previous modification these processes is guaranteed to stay on the processor until they have used all of there CPU time.

A slight modification to the recalculation of the *time_slice* variables checks each process to see if it can be allotted more than *task_timeslice(p)* ticks of processing time. If the *p→special_time_slice* > 0, then *p→time_slice* is set to p→*special_time_slice*. There is a small amount of overhead added to the scheduling algorithm, because each process has to be checked to see if it is a CPU-bound process. All other processes receive the same time_slice value, as they would under normal operation of the scheduler. The rest of the scheduling algorithm remains the same.

B. Process characterization

This section describes, how we characterized the programs. We studied the execution behaviour of several programs to find out the characteristics that can be used to predict the STS. We have taken representative programs: matrix multiplication, sorting programs, recursive Fibonacci number generating programs and random number generator programs. Any Machine learning technique [15] requires fully labeled (categorized) data for training. We categorized our data into classes and trained the classifiers (machine-learning techniques) [15].

The experimental procedure is divided into two phases. In the first phase, we create the data set from the program's run traces and make the data base with the static and dynamic characteristics of the programs and train it with machine learning techniques. In the second phase, we classify this data according to the fitness function, "STS" by using machine-learning techniques. The trained classifiers are then used on a different data set called as a *test data set*. Often due to limitations in size of the data, we train classifiers using a leave-one-out technique [16] which is a standard technique.

1) Creating the dataset: To characterize the program execution behaviour , we need to find the static and dynamic characteristics. We used *readelf* [1]and *size* commands to get the attributes as shown in Table I. To find STS, a script ran the matrix multiplication program of size 700 x 700 multiple times with different values of STS on P4 Linux

TABLE I

ATTRIBUTES AND THEIR MEANING

Attribute	Definition
Hash	This is symbol hash table size in bytes.
Dynamic	This dynamic linking information size in bytes.
Dynstr	This is size (bytes)of strings needed for dynamic linking.
msh	This is symbol hash table size in bytes.
Dynamic	This dynamic linking information size in bytes.
Dynstr	This is size (bytes)of strings needed for dynamic linking, most commonly the strings that represent the names associated with symbol table entries .
Dynsym	This is size (bytes) of the dynamic linking symbol table .
Got	This is the global offset table size in bytes.
Plt	This is the procedure linkage table size in bytes.
RoData	This is read-only data size (bytes) that typically contribute to a non-writable segment in the process image.
Ryl.Dyn	This is the size (bytes) of the relocation information size.
Text	This is the ”text” or ”executable instructions”, size (bytes)of a program.
Data	This is the size(bytes) of the initialized data that contribute to the size of the program's memory image size.
Bss	This section holds uninitialized data size
Total_size	This is total size (bytes) of the program.
Program_Name	This is name of the program and a nominal attribute.
Input_Size	Value of InputSize which depends on the programs type.
Input_Type	Type of the Input which is a nominal value.
SpecialTimeSlice(STS)	Value of the best time slice.
STSclass	Class of the STSs

TABLE II

THE INSTANCE

Attribute	Value
Hash	44
Dynamic	12
Dynstr	32
Dynsym	56
Got	8
Plt	32
RoData	208
Ryl.Dyn	48
Text	2400
Data	5200
Bss	12000
Total_size	18000
Program_Name	mm
Input_Size	800
Input_Type	i1 input_type1
SpecialTimeSlice (STS)	110
STSclass	t2 (101 ticks-149 ticks)

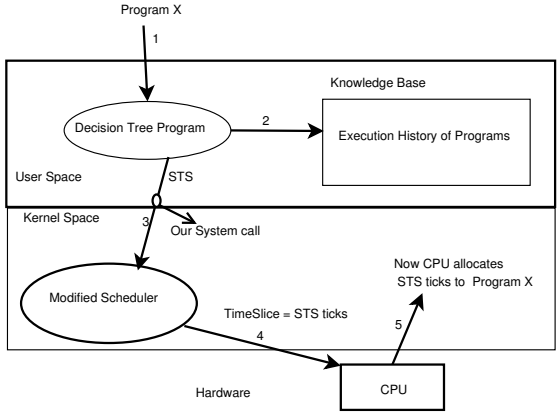

Fig. 1. The design of modified scheduling process.

System and selected the STS, which gives minimum TaT. The first 18 (characteristics) attributes are used to predict the target attribute STSclass.

We build the data set of 84 execution instances of five programs: matrix multiplication, Quick-sort, Merge-sort, Heap-sort and a recursive Fibonacci number generator and the Table II shows an example of a training example or an instance. We collected the data like the above for 5 programs with different input sizes and different best STSs. Data of 84 instances of the five programs was collected and made into 11 categories based on the attribute STSclass with each class having an interval of 50 ticks.

2) Training and Testing methodology: We performed two types of tests on the training examples with all the learners described in the section III (Overview of machine learning algorithms) on the data sets collected in the first phase. The tests are:

- Use Training Set [9]: The classifier is evaluated on how well it predicts the class of the instance it was trained on.
- Cross-Validation [9]: The classifier is evaluated by cross-validation, using the number of folds that are entered in the Folds text field (Weka). Recognition accuracy was tested via cross-validation. In this test, the training examples are divided into 10 parts and the classifier classifies by taking one part as a test set and other 9 parts as training set. Likewise, we continue for all parts.

The results of these tests explained in the next section ”Evaluation and Results”.

3) Extracting the best attributes: Attributes selection[1] [11] involves searching through all possible combinations of attributes in the data to find which subset of attributes works best for predicting the program execution behavior. Here our goal is to predict the target attribute. To do this, two objects must be set up ; “an attribute evaluator ”and “a search method”. The *evaluator* determines what method is used to assign a value (weight or worth) to each subset of attributes. Here, we used *Exhaustive* and *Genetic* search methods, and *Co-rrelation based feature selection* (CfsSubsetval) [9] evaluation method. A detailed description of feature selection is presented in [11].

C. The design of modified scheduling process

The Figure 1 shows the design of modified scheduling process and the steps to minimize TaT of a program. The steps

[1]Extracting the best attributes is nothing but feature selection

to minimize TaT of a program are as shown in the Figure 1 with numbers from 1 to 5.

1) The program 'X' is given to C4.5 decision tree as an input.
2) The decision tree will classify 'X' and output the STS.
3) We send this STS information to modified scheduler through a system call.
4) The scheduler instructs the CPU such that CPU allocates STS ticks to 'X'.
5) The CPU allocates STS ticks to 'X' and it will run with minimum TaT.

D. Overall method

Our method is explained as follows :

1) Run the programs with different special_time_slices with modified O(1) scheduler and find STS (best special_time_slice) which gives minimum turn-around-time (TaT).
2) Build the knowledge base of static and dynamic characteristics of the programs from the run traces obtained in step 1 and train them with the C4.5 decision tree algorithm.
3) If a new program comes, classify it and run the program with this predicted STS.
4) If the new program instance is not in the knowledge-base, go to step 1.

E. Experimental environment

For this work, we used GNU/Linux 2.4.20-8 operating system, 1.6GHz Intel P4 processor, 128MB RAM main memory, 512KB Cache memory, *Vi* editor and GNU gcc compiler and so on. WEKA(Waikato Environment for Knowledge Analysis), an open source machine-learning tool was used to find the most suitable ML method to characterize our programs.

VI. Experiments and Results

We explain the experimentation process by taking the matrix multiplication program as an example.

A. Effectiveness of STS

A script ran the matrix multiplication program of size 700 x 700, multiple times with different values of STS on P4 Linux System. The Table III shows how the turn-around-time changed as the CPU burst time (STS) of the process changed. From Table III, STS (or Best special_time_slice) is

TABLE III
EFFECT OF SPECIAL_TIME_SLICE ON TURN-AROUND-TIME(TAT).

Average TaT(secs)	special_time_slice (no.of ticks)
16.372123	100
16.900567	150
15.863590	200
.....	....
15.880436	1200
...	...

200 and corresponding TaT = 15.863590sec. Standard TaT (with unmodified scheduler) is 16.466sec. The microseconds saved per second = (16.466880 - 15.863590) / 16 = 60320 micro-seconds. About 60320 micro-seconds saved per second. Therefore on 1.6GHz Intel P4 machine, we can save 1600MHz = 16 x 10^8 clocks/second = [16 x 10^8 x no.of pipelines x (60320)] low-level operations, with special_time_slice (STS) = 200.

B. Selecting the best machine Learning technique

We used WEKA for training and testing the data set of 84 instances of those five programs to find a good ML technique [8]. The table IV shows the best machine learning technique particularly for our data set. From the table IV, C4.5 is the best technique[2] to predict the STS [8].

TABLE IV
ML TECHNIQUES AND PREDICTION ACCURACIES ON 84 INSTANCES.

ML Technique	% Classification with CrossValidation Test	% Classification with Use Training Set Test
C4.5(J48)	91.1667%	94.0476%
IBK	89.5432%	93.6480%

C. Finding the best attributes which can characterize STS class

We applied, *exhaustive search* with *CfsSubset* [11] evaluation method on the training set and find the best attributes, which can characterize the STS Class. The Table V shows the best attributes.

TABLE V
THE BEST ATTRIBUTES.

Attribute	Its Rank
InputSize	1
ProgramSize	2
BSS	3
RoData	4
Text	5
InputType	6

D. Effectiveness of our method in reducing the Turn-around-time.

Table VI shows that our Modified Scheduler [3] reduces turn-around-time of matrix multiplication program with different input sizes. In Table VI, MS is Modified Scheduler and UMS is Unmodified Scheduler.

[2]IBK is also a good technique but it is costly in terms of computation as compared to C4.5

[3]UnModified Scheduler is O(1) Scheduler of Linux Kernel 2.4.20-8 and Modified scheduler is modified O(1) Scheduler of Linux Kernel 2.4.20-8.

TABLE VI

THE COMPARISON OF UNMODIFIED SCHEDULER AND MODIFIED SCHEDULER OF MATRIX MULTIPLICATION PROGRAMS.

Matrix Sizes	TaT with Unmodified Scheduler	TaT with Modified Scheduler	Reduction in TaT
700 x 700	16466 millisecs	15864 millisecs	602 millisecs
750 x 750	19568 millisecs	19228 millisecs	340 millisecs
800 x 800	23820 millisecs	23190 millisecs	630 millisecs
850 x 850	28789 millisecs	28112 millisecs	677 millisecs
900 x 900	33540 millisecs	32029 millisecs	1511 millisecs
950 x 950	39997 millisecs	39388 millisecs	609 millisecs
1000 x 1000	47370 millisec	46110 millisecs	1260 millisecs
1050 x 1050	54912 millisecs	54204 millisecs	708 millisecs
1100 x 1100	63782 millisecs	61580 millisecs	2200 millisecs
1150 x 1150	72704 millisecs	70264 millisecs	2440 millisecs
1200 x 1200	85776 millisecs	80654 millisecs	5122 millisecs

Influence of STS on a highly loaded system : We ran the matrix multiplication programs as a batch program on a highly loaded system to test the effectiveness of our modified scheduler. There were 10 programs in a batch. The Table VII shows the results. We ran four programs which have operations like opening several files, writing large number of random numbers in them and closing them to make the system highly loaded. We monitored the *load-average* using "*top*" command.

TABLE VII

COMPARISON OF MODIFIED SCHEDULER AND UNMODIFIED SCHEDULER ON A HIGHLY LOADED SYSTEM

Matrix Sizes	Batch Size	load-average	TaT with UMS	TaT with MS	Reduction in TaT
800 x 800	10	7.38	165.57 secs	163.23 secs	2.34 secs
900 x 900	10	10.34	469.23 secs	463.86 secs	5.37 secs

Decision tree overhead: We implemented *C4.5 decision tree* for training and testing and its running time is around 220 milliseconds. The overhead of this is around 4% when compared to the reduction in turn-around time in case of matrix multiplication program of size 1200 x 1200. So we can neglect the overhead of decision tree.

Limitations of our method: The *characteristics* of the programs and the *gain* in the *turn-around-time* depends on the architecture and operating system. Our modified scheduler is not designed with any *security features* that would prevent a user from writing their own process and requesting INT_MAX processor time via system call which is described in the Section V-A.2. Setting $p->special\_time\_slice$ to INT_MAX could significantly slow down a system.

VII. CONCLUSIONS

By considering the *static* and *dynamic characteristics* of a program, we can schedule it using a *modified scheduler* such that the *turn-around-time* of it is minimized. We show that machine learning algorithms are very effective for the *process characterization* problem. The *C4.5 decision tree* algorithm achieved good prediction $(91\% - 94\%)$, which indicates that when suitable attributes are used, a certain amount of predictability does exist for known programs. Our experiments show that 1.4% to 5.8% *reduction* in *turn-around-time* is possible and this reduction rate slowly increases with the input size of the program. From our experiments, we find the best features : *input size, program size, bss, text, rodata* and *input type* of a program that can characterize its execution behavior. We conclude that our technique can improve the scheduling performance in a single system.

Future work: Our future work will include extending our technique to parallel programs, a more comprehensive study of high performance application characteristics shall be conducted on an 8-node Linux cluster, and we may attempt predictability on the *Portable Batch Scheduler* of Linux Clusters.

ACKNOWLEDGMENTS

We thank Sasi Kanth Ala for valuable discussions throughout this work.

REFERENCES

[1] Hongjiu Lu, *ELF: From The Programmer's Perspective*, Technical Report, NYNEX Science and Technology, 500 Westchester Avenue, White Plains, NY 10604, USA, May,1995.

[2] Aivazian, Tigran, *Linux Kernel 2.4 Internals*, The Linux Documentation Project, August, 2002.

[3] Maurice Bach, *The design of the Unix operating system*, Pearson Education Asia, pp:159-170, 2002.

[4] Fredrik Vraalsen, "Performance Contracts: Predicting and monitoring grid application behavior", *In Proceedings of the 2nd International Workshop on Grid computing*, November, 2001.

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

[6] Hyok-Sung Choi and Hee-Chul Yun, *Context Switching and IPC Performance Comparison between uClinux and Linux on the ARM9 based Processor* , Linux in embedded applications, Jan., 2005. http://www.linuxdevices.com/articles/AT2598317046.html.

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

[8] Kishore Kumar. P and Atul Negi, Characterizing Process Execution Behaviour Using Machine Learning Techniques, *In DpROM WorkShop Proceedings, HiPC 2004 International Conference*, December, 2004.

[9] Garner, S .R., "WEKA: The Waikato Environment for Knowledge Analysis", *In Proc. of the NewZealand Computer Science Research Students Conference*, pp : 57-64, 1995.

[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] Mark A. Hall, *Co-rrelation based feature selection for Machine Learning*, Master Thesis, Department of Computer Science, University of Waikato, pp: 7-9, 12-14, April, 1999.

[12] Andrew Marks, *A Dynamically Adaptive CPU Scheduler*, Master Thesis, Department of Computer Science, Santa Clara University, pp :5-9, June, 2003.

[13] Robert Love, *Linux Kernel Development*, 1st ed, The Pearson Education,2004.

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

[15] Tom Mitchell, *Machine Learning*, 1st ed, pp: 52-75, 154-183, 230-244, The Mc-Graw Hill Company.Inc.International Edition,1997.

[16] O. Duda, P. E.Hart , *Pattern Classification*, Wiley, NewYork,1973.