Correlation based Empirical Model for Estimating CPU Availability for Multi-Core Processor in a Computer Grid

Khondker S. Hasan
Department of Computing Sciences
University of Houston – Clear Lake
Houston, TX, USA
HasanK@UHCL.edu

Mohammad A. Rob
Management Information System
University of Houston – Clear Lake
Houston, TX, USA
Rob@UHCL.edu

Amlan Chatterjee
Department of Computer Science
California State University,
Dominguez Hills, CA, USA
Achatterjee@csudh.edu

***Abstract*— Techniques for estimating and predicting the availability of CPU resources for a task while executing in a single- and multi-core systems are introduced. These predictions have a significant effect in areas such as dynamic load balancing, scalability analysis, distributed task queue scheduling, and others. Analytical models proposed in this paper is useful for predicting the availability of CPU resources while executing a new task. The model can predict the allocation of CPU for a new task without prior knowledge of the run queue and system state. To validate the introduced prediction models, a dynamic monitoring tool (simulator) is developed. This real-time monitoring utility is responsible for measuring states and availability of resources based upon receiving resource request from client program. Extensive experimental studies with real-world benchmark programs and statistical analysis are performed to measure the accuracy of models. The performance of introduced monitor tool is evaluated as well while extracting resource availability data from the computer grid.**

***Keywords*— CPU availability; Computer grid; CPU Load; Distributed system; Multi-core processors; Performance Prediction;**

I. INTRODUCTION

Correlation refers to any of a broad class of statistical relationships involving dependence. Correlations are useful because they can indicate a predictive relationship that can be exploited in practice. The approach of proposing the new model in this paper is correlation based empirical model using time series analysis. Data points are measured typically at successive time instants because time series analysis assumes that sequential values in the data file that embodies repeated quantities are taken at equally spaced time intervals [3]. A sequence of methods for analyzing time series data in order to extract meaningful statistics and other characteristics of the data has been employed in this paper. The introduced mathematical models are based on an empirical average-case study in two different extreme situations, when the CPU is lightly and heavily loaded. Additionally, the scheduling pattern of tasks was observed for a long period of time to incorporate the accuracy.

When the number of threads assigned to a multi-core processor is less than or equal to the number of CPU cores associated with the processor, then the performance of the CPU is predictable, and is often nearly ideal. When the number of assigned threads is more than the number of CPU cores, the resulting CPU performance can be more difficult to predict. For example, assigning two CPU-bound threads to a single core results in CPU availability of about 50%, meaning that roughly 50% of the CPU resource is available for executing either thread. Alternatively, if two I/O-bound threads are assigned to a single core, it is possible that the resulting CPU availability is nearly 100%, provided that the usage of the CPU resource by each thread is fortuitously interleaved. However, if the points in time where both I/O-bound threads do require the CPU resource overlap (i.e., they are not interleaved), then it is possible (although perhaps not likely) that the CPU availability of the two I/O bond threads could be as low as 50% [1].

Predicting the availability of CPU resources when the number of threads assigned to the processor exceeds the number of cores is important in making thread assignment and scheduling decisions. Precise values of CPU availability are difficult to predict because of a dependence on many factors, including context switching overhead, CPU usage requirements of the threads, the degree of interleaving of the timing of the CPU requirements of the threads, and the characteristics of the thread scheduler of the underlying operating system (or Java Virtual Machine) [1]. In this paper, a thread is modeled by a series of alternating work and sleep phases. For the purposes of this study, the work portion of a phase is CPU-bound and requires a fixed amount of computational work (i.e., CPU cycles). The sleep portion of a phase length can be varied to realize different CPU load utilization factors for a thread. Due to the complex nature of the execution environment, an empirical approach is employed to evaluate proposed CPU availability prediction models and formulas.

In this paper, prediction of a new CPU availability model for time-shared systems has been introduced. The introduced mathematical forecasting model can be utilized in dynamic load distribution, scalability analysis, parallel systems modeling, and others. Therefore, the first step in this static analytical model is usually a prediction of CPU assignment or availability for a new task in computer systems that rely on a priori knowledge about the systems state. Because of the dynamic nature of current computer system and their workload, making such predictions is not simple. Workload in a system can vary drastically in a short interval of time. The model presented in this paper can predict

CPU availability for a new task using the computational requirements of the current tasks in the run queue and by the operating system scheduler. The empirical results presented in this paper exhibits low prediction errors for the empirical proposed models.

II. Relevant Work

Research that is closely related to this paper falls under two different prediction models which are static CPU availability prediction and dynamic CPU assignment prediction models. In general, there are four major approaches to obtain performance predictions in a computer system: code analysis, benchmarking, statistical prediction, and analytical modeling. There are two new response-time prediction models [7] also available. The first one is a mixed model based on two widely used models, CPU availability and Round Robin models. The second one, called Response Time Prediction (RTP) model, is a completely new model based on a detailed study of different kinds of tasks and their CPU time-consuming. Queuing models have been widely used for modeling processor performance due to their simplicity. Almost all queuing model conclude that the best workload descriptor for performance prediction is the number of tasks in the run queue.

Most of the existing prediction models assume that in a time-shared system, CPU is equally distributed among all the processes in the run queue by following the Round Robin scheduling technique [6]. These models use the number of tasks in the run queue as the system load index. As a result, the CPU assignment prediction for a new arriving process at any given instance when there are N processes in the run queue is simply $\frac{1}{(N+1)}$. This model is correct only when all the processes in the run queue are CPU bound. This kind of processes always shares the processor time in a balanced way, just like the RR model assumes. But, when the processes are I/O bound or a mix, this model fails to provide accurate predictions and incurs large prediction errors though in most of the cases there are heterogeneous tasks and resource requirements occurs.

Next in a multi-processor system, the most widely used approach known as symmetric multi-processing (SMP), where each processor is self-scheduling. Each processor has its own private ready queue of ready processes. The scheduling proceeds by having the scheduler for each processor examine the ready queue and select a process to execute. Virtually all modern operating systems support SMP, including Windows 7, Solaris, Linux, and Mac OS X. On SMP systems, it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor.

Multi-core processors are the common trend in computer hardware to place multiple processor cores on the same physical chip. Each core has a register set to maintain its architectural state and thus appears to the operating system to be a separate physical processor [3]. That is, from the operating system perspective, a system with two separate physical chips and a dual-core processor are same. Therefore, the scheduling of multi-processor and multi-core processor are similar as both systems have separate private run-queue and uses symmetric multi-processing. Multi-core processors have further advantages as researchers have discovered that when a processor accesses memory, it spends a significant amount of time waiting for the data to become available because of cache miss hit. This situation is called memory stall and processor can spend up to 50% of its time waiting for data to become available from memory. Figure 1 shows a memory stall situation.

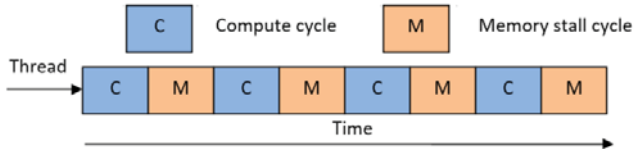

Figure 1: Memory stall situation [3].

To overcome this situation, most of the recent hardware designs have implemented multithreaded processor cores in which two or more hardware threads are assigned to each core. That way, if one thread stalls while waiting for memory, the core can switch to another thread. Figure 2 shows a dual-threaded processor core on which execution of thread-1 and the execution of thread-2 are interleaved.

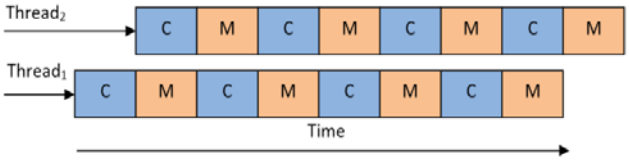

Figure 2: Multithreaded multi-core system [3].

From an operating system perspective, each hardware thread appears as a logical processor that is available to run a software thread. Thus, on a dual-threaded, dual-core system, four logical processors are presented to the operating system.

III. The Correlation-based Availability Model

In this section, an analytical framework is developed for estimating CPU availability associated with executing concurrent threads on a multi-core processor. Most of the modern OS uses priority-based algorithm in which higher-priority tasks get longer time quanta and lower-priority tasks get shorter time quanta. Each processor maintains its own run-queue and schedules itself independently. Each run-queue contains two priority arrays: active and expired. A runnable task is considered eligible for execution on the CPU as long as it has time remaining in its time slice. When a task has exhausted its time slice, it is considered expired and is not eligible for execution again until all other tasks have also used their time quanta. The OS maintains a list of all runnable tasks in a run-queue data structure. In Linux, each CPU has a run-queue made up of 140 priority lists that are serviced in FIFO order. The scheduler chooses the task with the highest priority from the active array for execution on the CPU. For multi-core machines, it means that each processor is scheduling the highest priority task from its own run-queue structure. When all tasks have used their time slices (that is, the active array is empty), the two priority arrays

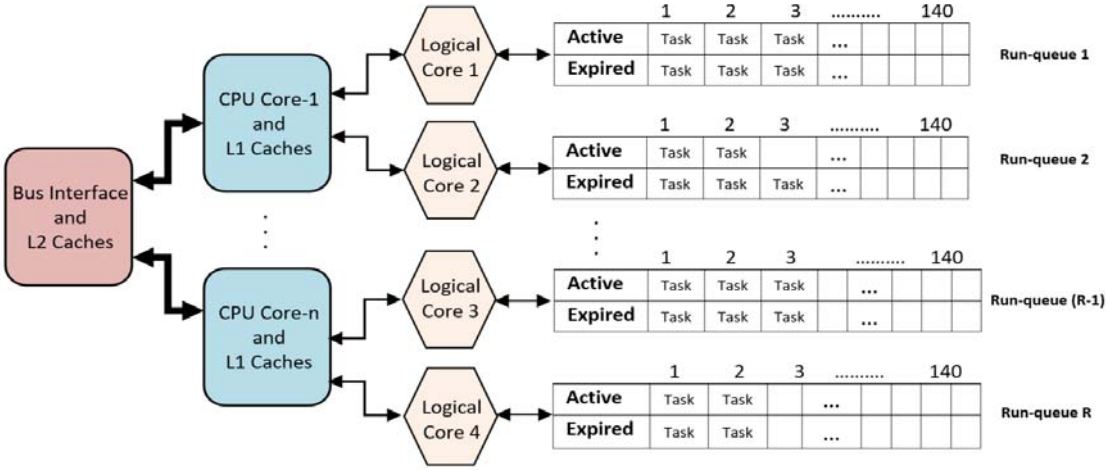

Figure 3: Multi-core processor with two hardware threads and associated private run-queue.

are exchanged. Figure 3 illustrates a multi-core processor containing two hardware threads and awaiting tasks in the run-queue.

All CPU bound processes in the run queue consume processor time for their execution. On the other hand, I/O bound processes fail to utilize allocated processor time to them in RR scheduling as they are busy with doing some I/O operation. The prediction error occurs in RR scheduling when a node has two running CPU bound processes in its run queue and another node has two I/O bound processes in its run queue. The RR model predicts the same CPU assignment for a new process. It is not correct as the new task will have a shorter response time in the second node than in the first node. To handle these situations and to describe the CPU prediction accurately, this paper has proposed a new CPU assignment model that distinguishes different CPU requirements for processes and overcomes the limitations of the RR model.

The proposed static CPU availability prediction model required prior knowledge about the tasks in the run queue and the unloaded CPU usage for the processes [2]. So, this model requires information about the number of CPU bound tasks *n*, the number of I/O bound task (*m*), and the unloaded CPU usage $\hat{U}_i$ for process P_i. For CPU bound process, $\hat{U}_i$ value is always 1.0. Empirical average-case studies in two different extreme situations have been deployed to propose the desired new model because Linux scheduler behavior is different in these two situations. In first *t* cases, two (2) CPU- and two (2) I/O-bound tasks in the run queue with very low CPU usage ($\hat{U} = 0.02$) has been used. It has been observed that the scheduler assigns a single time slice or quantum to every CPU bound process and these processes have consumed all the CPU time that is allocated to them, unlike I/O bound processes.

As I/O bound processes cannot consume allocated time quantum, the CPU-bound processes steal a fraction from their time quantum to better utilize the CPU. Because of this allocation, the I/O bound processes are given higher priority, by assigning them in higher priority queue, for processing tasks than CPU bound processes. On average, this behavior is coherent with the Linux scheduler mechanisms. In second case, when I/O bound processes are very close to being CPU bound processes, i.e., $\hat{U}$ value is very close to 1.0, it leads to scheduling of I/O bound process only once in the scheduling round similarly as CPU bound processes [2]. The combination of these two extreme situations gives the general equation to obtain the fraction of CPU time that corresponds to any process in the run queue.

Let the number of CPU core is *c*, the number of hardware thread is ξ, and the number of run-queue is R. Then the number of run-queue R = (c × ξ). As the static model derived for measuring CPU availability is based on the prior information about the run-queue, for a multi-core processor with two hyper-threading requires (2 × c) separate predictions (one for each logical processor). Therefore, the definitive expressions for the static prediction of each logical processor, denoted using α_{lp} where $lp \in \{0, 2, ..., R-1\}$, is:

$$\alpha_{lp} = max\left(\frac{1-\sum_{i=1}^{m} U_{i,lp}}{n+1}, \frac{1}{n+m+1}\right) \quad (1)$$

with the expression of shared CPU usage is the following:

$$U_{i,lp} = \frac{\hat{U}_{i,lp}}{1+\hat{U}_{i,lp}.(n-1)+\sum_{j=1}^{m}\hat{U}_{j,lp}} + \frac{\hat{U}_{i,lp}^2}{m+1+\hat{U}_{i,lp}.(n-1)} \cdot \sum_{j=1,j\neq i}^{m}\left(1-\hat{U}_{j,lp}\right) \quad (2)$$

Here, the first part of the Eq. 2 computes the shared CPU usage of the I/O bound process in the run queue. The second part of the equation computes the usage of the CPU bound process in the run queue. All processes in the run queue can take advantage of their processor quantum. Finally, to get the average CPU load,

denoted by δ, of the multi-core processor, the following equation can be employed:

$$\delta = \sum_{lp=0}^{R-1} \frac{\alpha_{lp}}{c} \quad (3)$$

This average CPU load availability, δ, can be used as a parameter for calculating the priority of nodes in a grid for assigning tasks to achieve faster response time.

IV. Empirical Studies

A. Overview

The purpose of the experimental studies is to empirically measure CPU availability of computers in a grid while assigning a new task. A simple Java based system monitor application has been designed and implemented to reside on each system node. The implemented monitor tool is based on client-server architecture in which client sends resource measure request to all computers of the grid using UDP datagram. The UDP protocol is utilized to reduce the overhead of connection-oriented protocol. Upon receiving the request, system nodes will talk with respective operating system to gather lower-level date (like *pid*, *utime*, *stime*, *idle time* etc.) for each process to compute the CPU usage by each process of the run-queue. These parameters are then passed to the CPU availability model for computing CPU load average. Along with CPU availability, OS name, number of cores, total free physical memory, and other resource information are send to the requested node for making job scheduling decision. The algorithm for the developed client-server model is given below:

Algorithm 1: Client-side of the Monitoring tool

Input: Get the Monitor Connection Data
Output: The Measured Availability Data

Set the N, IP, Port, and Resource Linked Hash Map
For (i ← 1 to N) (N is the number of nodes in the grid) do
    Retrieve the IP and PORT number of N_i
    Send resource request to N_i via datagram packet
    Receive resource response from N_i via datagram packet
    Add the available resource of N_i into Grid resource Linked Hash Map
Store the data received from Server into separate files
Clean useless data and handle errors, if event occurs.
Close connection and files

The Algorithm 1 shows the high-level pseudocode for the client-side of the monitor tool. The Algorithm 2 shows high-level statements of the server side while extracting resource availability information from the system and the run-queue. The monitor is implemented in real client-server environment which becomes a real-time monitoring utility responsible for extracting computer node's state information, interpreting it and computing the model to predict CPU assignment. This application can serve as a flexible state measurement module for heterogeneous cluster or grid application that require a CPU assignment prediction, such as load balancing and scheduling algorithms or resource discovery service of a grid.

Algorithm 2: Server-side of the Monitoring tool

Input: Incoming request from client with control data
Output: Actual CPU Usage, Model Prediction Usage (P_c), and Prediction Error

Waiting for an incoming message via UDP from Client node
Receive a resource request message from client node
Fetch PID for all the processes, *P*, in the run-queue
Retrieve the number of processor cores, *c*, for nodes
For (i ← 1 to *P*) do
    Fetch utime, stime, startTime, and state for the process P_i
    If (state ≠ running)
        m=m+1 (here m is I/O bound process count)
    Else
        n=n+1 (here n is CPU bound process count)
    CPU utilization $CU_i = \frac{utime_{Processn} + stime_{Process}}{utime_{CPU} + stime_{CPU}} \times \varepsilon$
For (i ← 1 to m) (each I/O bound process in the run-queue) do
    Compute the Unloaded CPU usage of the process
    Compute the cumulative Unloaded CPU usage for all I/O bound processes in the run-queue
For (i ← 1 to m) (each I/O bound process in the run-queue) do
    Compute shared CPU usage by evaluating the CPU availability
    Compute cumulative shared CPU usage for all I/O bound processes in the run-queue
    Compute CPU Availability using Round Robin (RR) Model
IF (RR > Total Shared CPU Usage)
    CPU Availability = RR
Else
    CPU Availability = Total Shared CPU Usage
Return CPU Availability data, Prediction (P_c) to client-tool
Close Client connection and clean memory

B. Empirical Environment

The computer grid that has been used to run this model consists of eight (8) heterogeneous computer nodes. This heterogeneous grid consists of 6 Linux machines each consists of 8 cores. The other two machines are MAC consists of 24 and 2 cores. Detail configurations of the computers in the grid are shown in Table 1.

Table 1: Configuration of the machines utilized to construct a distributed grid for the empirical work.

OS Name	OS Version	Number of Processors	CPU Speed	Physical Memory
Linux	2.6.32-22	8	2.67 GHz	6.0 GB
Linux	2.6.32-22	8	2.67 GHz	6.0 GB
Linux	2.6.32-22	8	2.67 GHz	6.0 GB
MAC OS X	10.6.8	24	2.66 GHz	12.0 GB
Linux	2.6.32-22	8	2.67 GHz	6.0 GB
MAC OS X	10.6.8	2	3.06 GHz	4.0 GB
Linux	2.6.32-22	8	2.67 GHz	6.0 GB
Linux	2.6.32-22	8	2.67 GHz	6.0 GB

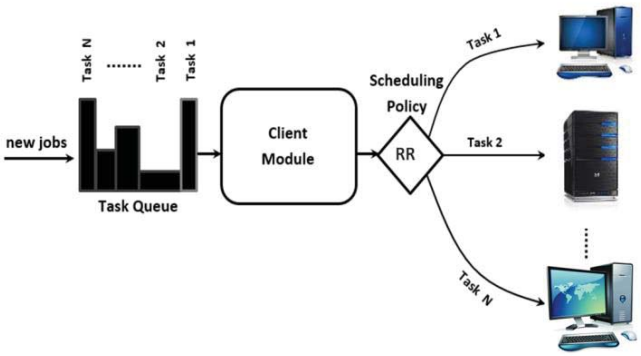

Figure 4: Distributed architecture of the monitor-tool for dispatching resource availability query to multi-core nodes.

The JVM used for these experiments is JDK 1.7. The above specified machines used for empirical studies are lightly loaded.

Real-world benchmark programs like fast furrier transformation, modular arithmetic, image rendering, matrix chain multiplication is utilized to conduct empirical case studies. These benchmark programs are chosen to facilitate variable CPU and I/O requirements. Before running test cases, CPU load of each machine are measured accurately. Figure 4 shows the diagram of monitoring tool implemented for carrying out the empirical case studies accurately. The Average CPU load for machines were found 0.01782 per core in a scale of 1.0. The Table 2 shows the available CPU and memory resources of all computer nodes in the grid.

Table 2: The machines utilized to construct a distributed grid for the empirical work.

Node Name	Response Time	Number of CPU Cores	CPU Load Average	Available Physical Memory
Linux	2.04	8	0.0155170	6.0 GB
Linux	2.14	8	0.0191543	6.0 GB
Linux	2.17	8	0.0182963	6.0 GB
MAC OS X	1.65	24	0.0167428	12.0 GB
Linux	2.06	8	0.0125637	6.0 GB
MAC OS X	1.86	2	0.0215643	4.0 GB
Linux	2.11	8	0.0171522	6.0 GB
Linux	1.86	8	0.0187421	6.0 GB

Benchmark threads are given 2.0×10^8 units of work load to accomplish in 50 quantums. A quantum consists of a work phase and a sleep phase. Each thread needs to accomplish 4.0×10^6 units of work in each work phase so that it can complete total work in 50 phases. Sleep phase lengths of the threads are calculated depending on CPU load and remains constant during its life time, but the work time can vary depending on CPU availability. When threads have completed their work, the report of thread execution containing start time, work time, sleep time, number of phases, and end time are stored into multiple files for statistical analysis. Each of the sample is timestamp for accuracy and future reference.

C. Assumptions

The following are the assumptions for the CPU availability model:

- A batch of threads are spawned concurrently in a single- or multi-core system.
- The single- and multi-core systems in which threads will be spawned are dedicated, meaning they are not loaded with other threads.
- There are no inter-thread communication or message passing among threads.
- Overhead related to the operating system's real-time process execution is negligible.
- The CPU requirement of each thread is known, which is used to estimate the aggregate CPU load for that batch.
- The CPU utilization is statistically \stationary" over time.

The model depends on the above specified assumptions. If these assumptions are not specified, then the model may produce prediction errors or may not be completely viable.

D. Empirical Case Study Results

An extensive set of empirical studies are conducted to determine the resource utilization by the developed monitoring tool and the accuracy of the introduced CPU availability prediction model in real-world environment with benchmark programs. Two distinct sets of test cases are conducted to measure monitoring tool's performance and introduced model's accuracy. All test cases are conducted independently, and run-time are measured for relative comparison.

The server monitor tool load time and memory requirement are measured through Linux system monitoring tool. It shows that for the very first time, the server program load time is 235 ms. The amount of memory requires to serve concurrent clients is 3.27 KB. While querying for resource availability, average response time from each node 1.76 ms. Cumulative response time from all nodes is 14.08 ms. If a node doesn't respond immediately, the monitor uses a threshold wait of 1000ms. After the threshold limit exceeds, it gives up as the node may be down or the network connection is broken. Execution time for the client module is 28.68 ms.

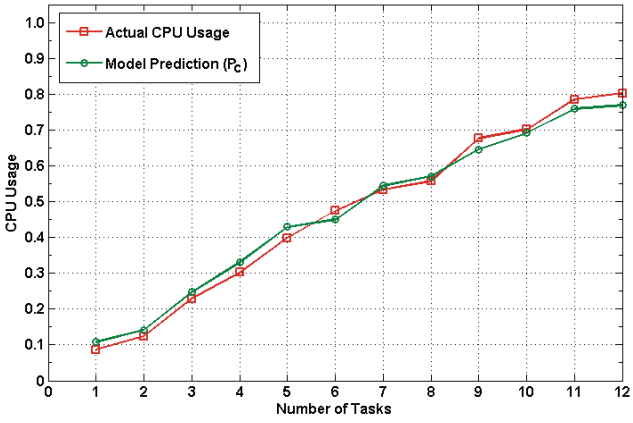

Figure 5: Comparison between actual measured CPU usage and the prediction model after dispatching 12 tasks concurrently.

Figure 5 correspond to experiments for a set of 12 benchmark programs (mix of all) to for a quad-core machine in the grid. In the Figure 5, the horizontal axis represents the number of tasks (N), and the vertical axis represents the CPU usage of the machine. The actual CPU usage line is an average of multiple run. It can be observed from Figure 5 that the CPU usage prediction model line and the actual CPU usage line follows same basic shape and pattern. The variation between actual and predicted lines are very low.

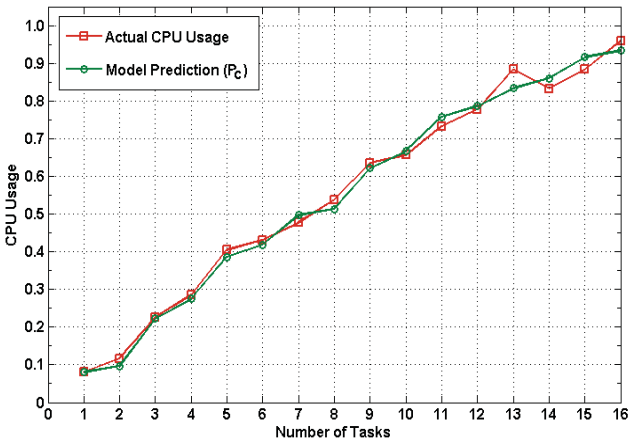

Figure 6: Comparison between actual measured CPU usage and the prediction model after dispatching 16 tasks concurrently.

Figure 6, and 7 also depicts similar shape and patters between actual and predicted measures with low variation. The CPU usage prediction model lines are smooth compared to the actual CPU usage lines.

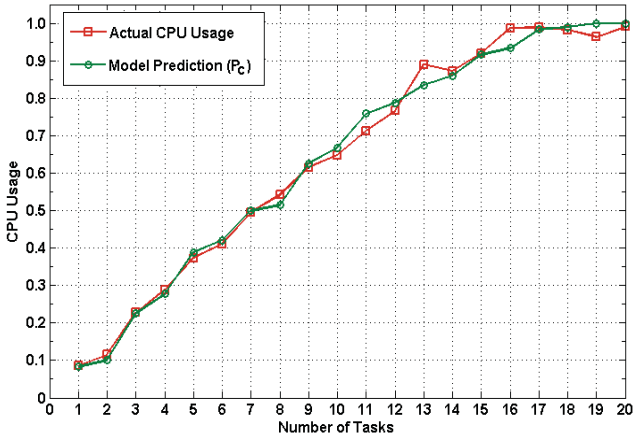

Figure 7: Comparison between actual measured CPU usage and the prediction model after dispatching 20 tasks concurrently.

The results of spawning 20 tasks from task queue to a quad-core machine in the grid shows a slim prediction error (δ_P). For each of the cases, task name (Scenario), measure of actual CPU usage (U), measure of CPU usage prediction model (P_c), and the prediction error ($\_\delta_P = | P_c \times U |\_\times 100$) are measured. After dispatching each task, real CPU usage and predicted usage are measured. These two magnitudes are then used to compute prediction error percentage. The monitor data shows introduced CPU usage prediction model achieves very low prediction error. The minimum prediction error is 1.39% for the cases in which 1 and 15 tasks spawned in the system. The maximum prediction error is 5.43% in which 13 tasks were dispatched. Regardless of CPU- or I/O-bound.

The results retrieved from the grid showing the resource availability of each node can be applied to make critical decision of node selection for job assignment based on highest resource availability for faster response. Though only few experimental results are shown in this paper, but all performed experiments are utilized for drawing the conclusions.

The data extracted using CPU availability prediction model for multi-core processor can be applied to select the best service node for better performance. The computer node in the grid which is extracting the data can use the data for arranging the worker nodes into the computer pool. The worker node which has the highest CPU availability can be assigned the highest priority of achieving task as it can process early. Figure 4 shows the distributed model which can make use of the CPU availability prediction to select the worker node with the highest available resource to process the job early.

V. Conclusion

This paper has developed analytical models (and conducted empirical studies) for estimating (and predicting) CPU availability of single- and multi-core machines in a computer grid. The model discussed in this paper has been implemented to measure CPU availability of computer nodes in a heterogeneous grid from state measurements and without prior knowledge about the processes in the run-queue. This feature helps this model to be applied in the real environments, with processes starting and ending dynamically with unknown requirements. The application developed to validate the model is light weight and separates the state measurement from the model computation. This design will help the application to be adapted in the future to other operating systems.

As the extraction of state information is time consuming, this dynamic measure application will gather resource data only if it receives a request from client instead of period measuring. The models have been verified and validated with a set of in-depth experiments which indicates lower predictions errors because its predictions are based on real system state measurements instead of predictions or prior information. This light and reasonably simple resource monitor application has the ability to accurately predict the CPU assignment for a new process on a computer in a grid. The resource availability data extracted using the prediction model for single- and multi-core processor can be applied to select the node with the highest available resource for processing the job to achieve the upmost performance.

Acknowledgement

Authors would like to thank Ms. Hajera K. Nasreen, Research Assistant, Department of Computing Sciences, University of Houston – Clear Lake and Mr. Jonathan Mullen,

System Administrator, School of Computer Science, University of Oklahoma, for their time and coordinated support.

REFERENCES

[1] Khondker S. Hasan, John K. Antonio, Sridhar Radhakrishnan, "A model-driven approach for predicting and analysing the execution efficiency of multi-core processing", International Journal of Computational Science and Engineering (IJCSE), INDERSCIENCE Publishers, Vol: 14, No 2, Page 105 – 125, DOI: 10.1504/IJCSE.2017.082877, Olney, Bucks, UK, March 2017.

[2] Martha Beltrán, Antonio Guzmán and Jose Luis Bosque, "A new CPU Availability Prediction Model for Time-Shared Systems", IEEE Computer, Vol 57, No. 7, July 2008.

[3] Khondker S. Hasan, "Performance Enhancement and Prediction Model of Concurrent Thread Execution in JVM", 14th Int'l Conference on Modeling, Simulation and Visualization Methods (MSV-17), Organized under: The 2017 World Congress in Computer Science Computer Engineering & Applied Computing (CSCE'17), Nevada, USA, 17-20 July 2017.

[4] How To Identify Patterns in Time Series Data: Time Series Analysis StatSoft, Inc.. Electronic Statistics Textbook. Tulsa, OK 2013. http://www.statsoft.com/Textbook/Time-Series-Analysis

[5] Bill Venners, "Inside the Java 2 Virtual Machine", Thread Synchronization,URL: http://www.artima.com/insidejvm/ed2/index.html

[6] Khondker S. Hasan, Amlan Chatterjee, Sridhar Radhakrishnan, and John K Antonio, "Performance Prediction and Analysis of Compute-intensive Tasks on GPUs", The 11th IFIP International Conference on Network and Parallel Computing (NPC-14), Sept. 2014, Lecture Notes in Computer Science (LNCS), Springer, ISBN: 978-3-662-44917-2, Vol: 8707, pp 612-17, Berlin, Germany, 2014.

[7] Y. Zhang, W. Sun, and Y. Inoguchi, "Predicting running time of grid tasks on cpu load predictions", Proceedings of the 7th IEEE/ACM International Conference on Grid Computing, pp. 286–292, September 2006.

[8] "Java thread performance", The behavior of Java threads under Linux NPTL, Amity Solutions Pty Ltd – Version 1.5, July 2, 2003, URL: http://www.amitysolutions.com.au/documents/NPTL_Java_threads.pdf

[9] Khondker S. Hasan, John K. Antonio, Sridhar Radhakrishnan, "A New Multi-core CPU Resource Availability Prediction Model for Concurrent Processes", The 2017 IAENG International Conference on Computer Science (ICCS-17), Organized under: International MultiConference of Engineers and Computer Scientists (IMECS-17), Hong Kong, 15-17 March 2017.

[10] Ken Arnold and James Gosling, The Java Programming Language, Fourth Edition, Addison Wesley, 2005.

[11] Heather Kreger, Ward Harold, Leigh Williamson, Java and JMX, Building Manageable Systems, Addison-Wesley 2003.

[12] Vitaly Mikheev, "Switching JVMs May Help Reveal Issues in Multi-Threaded Apps", May 2010, http://java.dzone.com/articles/case-study-switching-jvms-may.

[13] Analysis of Multithreaded Architecture for Parallel Computing, Rafael H. Saavedra, David E. Culler, Thorsten Eicken, 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, ISBN: 0-89791-370-1, 1990.

[14] P.A. Dinda, "Online Prediction of the Running Time of Tasks," Proc. 10th IEEE Int'l Symp. High Performance Distributed Computing, pp.336-7, 2001.

[15] Inside the Linux scheduler, The latest version of this all-important kernel component improves scalability, http://www.ibm.com/developerworks/linux/library/l-scheduler/.

[16] R. Wolski, N. Spring, and J. Hayes, "Predicting the CPU Availability of Time-Shared Unix Systems on the Computational Grid," Proc. Eighth International Symposium on High Performance Distributed Computing, pp. 105-112, ISBN: 0-7803-5681-0, August 2002.