View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by SHAREOK repository

A New Multi-core CPU Resource Availability Prediction Model for Concurrent Processes

Khondker S. Hasan, John K. Antonio, and Sridhar Radhakrishnan

***Abstract*— The efficiency of a multi-core architecture is directly related to the mechanisms that map the threads (processes in execution) to the cores. Determining the CPU resource availability of a multi-core architecture based on the characteristics of the threads that are in execution is the art of system performance prediction. Prediction of CPU resource availability is important in the context of making process assignment, load balancing, and scheduling decisions. In distributed infrastructure, CPU resources are allocated on demand for a chosen set of compute nodes. In this paper, a prediction model is derived for multi-core architectures and empirical evaluations are performed with real-world benchmark programs in a heterogeneous environment to demonstrate the accuracy of the proposed model. This model can be utilized in various time-sensitive applications like resource allocation in a cloud environment, task distribution (determining the order for faster processing time) in distributed systems, and others.**

***Index Terms* — CPU Availability, Execution Efficiency, Hyper-threading, Multi-core Prediction Model, Prediction Algorithm.**

I. Introduction

SYSTEM efficiency prediction involves estimating the system's behavior for a set of tasks to be executed on it. Making such predictions is complicated due to the dynamic nature of the system and its workload, which can vary drastically in a short span of time [2]. For any prediction approach, it can be useful to know a priori certain characteristics of tasks that are planned to be assigned to the system. As an example, it is useful to know the CPU requirement of the task, which is the fraction of time a task requires the CPU. It is also useful to know the maximum amount of main memory a task will consume during its execution (memory requirement). This information is useful to forecast the amount of time that may be consumed for memory paging activities.

Manuscript received December 23, 2016; revised January 16, 2017.

Khondker S. Hasan, Ph.D., is an Assistant Professor at the University of Houston - Clear Lake, Houston, TX 77058, USA. He received his Ph.D. in Computer Science from the University of Oklahoma, OK, USA in 2014 and M.S. in Computer Science from the Wichita State University, Kansas, USA in 2001. Dr. Khondker is a current IAENG Member # 156478. phone: 1-281-283-3842; (e-mail: HasanK@ uhcl.edu).

John K. Antonio, Ph.D., received the Ph.D. degree in electrical engineering from Texas A&M University, College Station in 1989. He is currently Senior Associate Dean of Engineering and Professor of Computer Science at the University of Oklahoma (OU), Norman, OK 73019, USA (e-mail: antonio@ou.edu).

Sridhar Radhakrishnan, Ph.D., is Professor and Director of the School of Computer Science at the University of Oklahoma, Norman, OK 73019, USA, where he joined in 1990. He received his Ph.D. in Computer Science from Louisiana State University in 1990 (e-mail: sridhar@ou.edu).

In time-shared systems, the utilization of the CPU and the speed of the computer's response to its users have improved as a result of advances in CPU scheduling [2]. To realize this increase in performance, it is customary to keep many processes running in the system [3]. The success of approaches for assigning concurrent processes (or threads) to multi-core or many-core systems relies on the existence of reasonably accurate CPU resource availability model. This is because there is a strong relationship between a thread's total execution time and the availability of CPU resources used for its execution [1]. Therefore, predicting the resource availability that results when threads are assigned to a processor is a basic problem that arises in many important contexts. Accurate resource availability prediction model is important because there exists a wide range of applications and scientific models (e.g., geological, meteorological, economical and others) that requires extensive use of CPU resource, repeatedly. These models fall into this category, where the program remains the same and the data on which it operates changes over time. The distributed task assignment problem in general is NP-Hard and it is further complicated by the changing dynamics (changes to CPU availability) of the compute nodes [1].

Given a set of tasks each with varied CPU requirements, and a multi-core system, we provide a CPU availability prediction model for the efficiency with which the tasks are executed. To facilitate scientific and controlled empirical evaluation, real-world benchmark programs with dynamic behavior are employed on LINUX systems that are parameterized by their CPU usage factor. Extensive experimental studies and statistical analysis are presented in this paper to measure the accuracy of the introduced prediction models.

Given the CPU requirements of tasks in a run queue, Beltrán et al. [2] provide an analytical model to estimate the CPU availability (which is the percentage of CPU time that will be allocated) for the new task prior to its placement in the run queue. It is shown that in certain cases this information can be used to schedule the execution of tasks in such a way that the completion time of all the tasks is minimized [2]. They considered a batch of tasks (each with its own CPU requirements) and their analytical model determined the CPU availability using the sum total of the CPU requirement of each of the tasks in the batch. Using this sum total posed a challenge in that the CPU availability prediction is precise only when the order of task execution is known a priori. To address this challenge, our (Khondker et al. [1]) analytical model provided tight upper- and lower-bounds on CPU availability. The bounds are necessary because the actual CPU availability depends on the order of execution of tasks in the batch. Thus the analytical model in

ISBN: 978-988-14047-3-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Khondker et al. [1] is oblivious of the CPU scheduler.

In the present paper, we have further improved the analytical model in [1] in several ways. First, we have applied an empirical approach based on the thread assignment observation in [1, 4] to introduce a new Multi-core CPU availability prediction model. This empirical model provides a single prediction value (instead of upper- and lower-bounds) of CPU availability for a set of concurrent processes prior to their execution without explicit knowledge of the mapping between available cores and processes. Second, we have utilized real-world benchmark programs in LINUX systems to perform extensive empirical work. Finally, we provided empirical results and statistical analysis to validate the introduced new CPU availability model and its application in a commercial context.

II. Relevant Work

Existing prediction models generally assume CPU resources are equally distributed among all processes in the run queue by following a Round Robin (RR) scheduling technique [2, 7]. These models use the number of processes in the run queue as the system load index. As a result, the CPU availability prediction for a newly arriving process when there are currently N processes in the run queue is simply $1/(N + 1)$. This predictor is only accurate for CPU-bound processes, which share CPU resources in a balanced manner; consistent with the RR model assumption. But, when the processes also require I/O resources, this approach fails to provide accurate predictions and incurs large prediction errors. Thus, when there are processes in the run queue that require CPU and I/O resources, a more complex model is necessary to describe how the CPU is shared [5]. The introduced model overcomes this suitable for both CPU and I/O bound processes.

Federova et al. [9] also worked on operating system scheduling heterogeneous multi-core systems. They proposed thread-to-core assignment algorithms that optimize performance and demonstrate the need for balanced core assignment. The paper makes the case that thread schedulers for multi-core systems in a heterogeneous environment should target the following objectives: optimal performance, core assignment balance, response time, and fairness. In addition, Federova [9] introduced a practical new method for estimating performance degradation on multi-core processors, and its application to workloads of clusters nodes.

Jen-Chieh et al., [10] have worked on multi-core systems and the allied software parallelization technique trends of System On Chip (SoC) design. Their paper adopts an electronic system-level (ESL) design methodology for higher system performance and lower energy utilization for revealing a system performance prediction and analysis method for multi-core systems. Based on the scalable multi-core virtual platform, they have performed one to eight-core system performance trend prediction as well as multi-core system performance analysis. Experiments are conducted for observing the performance improvement of software parallelization. The paper also discusses issues related to hardware-software co-design and hardware cost reduction.

III. Multi-core CPU Availability Model

The primary focus of this section is to present a CPU availability prediction model for estimating CPU availability for a set of tasks on multi-core systems. 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, the efficiency of the CPU is often near to 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 (Khondker et al. [1, 4]). 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 bound threads could be as low as 50%. Therefore, considering the number of processes in the run queue as a load index is not sufficient. The aggregate CPU load (sum total) of running threads needs to be considered for deriving an accurate model.

The execution of a thread is generally modeled by a series of alternating work and sleep phases. A thread in a work phase will remain there until it has consumed enough CPU cycles to complete the allotted work of that portion of computation. After completing the work phase, the thread then enters in the sleep portion where it stays (does not consume CPU cycles) for a specific amount of time. Each thread is parameterized by a CPU usage factor. The CPU usage factor is defined as the time required to complete the work portion of a work-sleep phase on an unloaded CPU, divided by the total time of a work-sleep phase. A thread having zero sleep length has a CPU usage factor of 100%, which is also called a CPU-bound thread. Sleep phase length and the total amount of computational work to accomplish is computed based on the CPU usage factor. Its sleep length relative to that of the work portion based on CPU load factor is used to account for I/O or any other interruptions (Khondker et al. [1, 4]).

This section introduces a model for predicting the expected (on average) availability of CPU (instead of calculating the upper- and lower-bounds [1]). As discussed in the previous subsection, exact values of CPU availability are difficult to predict because of dependencies on many factors, including context switching overhead, CPU usage requirements of the threads, core hyper-threading, 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. Due to the complex nature of the execution environment, an empirical approach is incorporated into our approach to estimate expected CPU availability. The model predicts availability of CPU resources for a batch of heterogeneous threads executed concurrently on multi-core distributed systems. The model estimates the CPU requirements for the batch in and expected thread execution efficiency.

ISBN: 978-988-14047-3-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

A. Assumptions

Following are the assumptions for the introduced model:

- Threads are heterogeneous and there are no inter-thread communication.
- CPU utilization is statistically "stationary" over time.
- Batches of threads are spawned concurrently in a multi-core system.
- The number of threads in a batch is more than the number of CPU cores of the assigned machine.
- The multi-core system in which the threads are spawned was initially unloaded.
- CPU requirement of each thread is preceding known before placing into the run-queue.
- Threads are independent and there are no racing conditions.
- Overhead related to operating system's real-time process execution is negligible.

B. The Mathematical Model for Multi-core CPU

The new multi-core CPU availability model is derived based on two empirical scenarios. In the first scenario, when the value of aggregate CPU load (L) is less than or equal to the number of CPU cores (r), that is, running threads has low CPU utilization, the number of work phase overlap is generally small. This results in less contending work phases among running threads but still some context switching overhead. The CPU availability upper-bound model is based on the situation in which all work portions of threads are out of phase (interleaved). The following expression represents the context switching overhead when $L \leq r$. This is subtracted from unity in Eq. 3 to model CPU availability. In the following, ξ represents the count of CPU core hyper-threading.

$$\left(\frac{n-1}{n+r}\right) \times \left(\frac{L}{(n+r)\times\xi}\right) \quad \text{if } L \leq r \tag{1}$$

In the second scenario, when the value of $L > r$, the best CPU availability for running threads is considered as r/L. This scenario is based on interleaved work phase of running threads. This model provides an optimistic estimation, but when the value of L increases, contention is more likely due to relatively wider work portions than idle portions. That is, increased CPU load increases more context switching overhead. The following estimated context switching overhead is subtracted from the second scenario of the upper-bound model (in Eq. 3) to reflect the observed behavior.

$$\left(\frac{1}{1+\left(\frac{n-1}{n}\right)\times\left(\frac{L}{r}\right)\times(r\times\xi+\xi)}\right) \quad \text{if } L > r \tag{2}$$

The following mathematical model derived from the empirical observation provides an explanation of thread assignment in multi-core processor. The model of Eq. 3 depends on the aggregate CPU load (sum total) of the set of tasks, number of threads, number of processor cores, and number of hyper-threading in cores. For a multi-core machine, the following prediction model estimates the average CPU availability for a set of tasks:

$$c_{avg} = \begin{cases} 1 - \left(\frac{n-1}{n+r}\right) \times \left(\frac{L}{(n+r)\times\xi}\right), & \text{if } L \leq r \\ \frac{r}{L} - \left(\frac{1}{1+\left(\frac{n-1}{n}\right)\times\left(\frac{L}{r}\right)\times(r\times\xi+\xi)}\right), & \text{if } L > r. \end{cases} \tag{3}$$

The model of Eq. 3 is derived from Eqs. 1 and 2 that depends on whether the aggregate CPU load is less than the number of cores or more than the number of cores. In the first situation, CPU resources are lightly loaded resulting in low context switching overhead and better efficiency. In the second situation, threads are moderate to highly loaded (i.e., aggregate CPU load is more than the number of cores), resulting in more context switching overhead and reduced efficiency. In general, thread with more CPU load incurs more contention for resources and context switching overhead. Therefore, an estimated context switching overhead is subtracted from the efficiency value in both cases (i.e., when for $L \leq r$ or $L > r$) to match the average efficiency.

IV. Empirical Studies

A. Overview

Depending on the process variation (CPU versus I/O bound), the Linux scheduler version 2.6 and above dynamically modifies this measured value to assign a priority penalty to CPU-bound processes and boost I/O-bound processes [11]. This Dynamic Priority Scheme is controlled with the sleep avg variable for each process. When an I/O-bound process is awakened from a sleep interval, its total sleep time is added to this variable. In addition, when a process leaves the processor, the time it has been executing on it is subtracted from this variable. The higher the sleep avg value is, the higher the dynamic priority will be. In addition, the time slice of a process is computed with its priority value, always maintaining its length between the minimum and maximum values [11].

B. Empirical Environment

The system used for evaluating the multi-core test cases is equipped with an Intel(R) Xeon(R) Quad-core W3520 processor, 2.67GHz clock speed, 1.333 MHz bus speed, and 6.0 GB of RAM. This machine also has Linux kernel version 3.2.36. The average CPU load (represents the average system load over a period of time) was 0.0161302 per core in a scale of 1.0 (in a fifteen minute period) before running test cases, which indicates that the machines were lightly loaded (essentially unloaded). The Linux Shell command *top* and *sysinfo* is called and output is redirected and parsed to extract required data from the system.

The C programming language in the Linux environment is used to implement the analytical model framework and benchmark programs. The *gcc* compiler version used is 4.6.3. Threads deployed here are independent tasks, meaning there are no interdependencies among threads such as message passing. Threads are spawned concurrently. When the batch of threads complete assigned work and terminates, an execution report is produced, which contains start time, work time, idle time, end time, number of phases, and others for statistical analysis.

ISBN: 978-988-14047-3-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

C. Benchmarking

The benchmark programs used for conducting empirical study in the Linux environment consists of real-world CPU-intensive programs like Super prime number generator, Monte-Carlo estimation of π along with a similar synthetic benchmark program. Table I consists of a complete list of programs used as benchmark program for conducting empirical case studies.

TABLE I
BENCHMARK PROGRAMS USED FOR VALIDATING INTRODUCED CPU AVAILABILITY PREDICTION MODEL

Name	Description
piEstimation	Uses Monte Carlo method to estimate the value of π
supPrime	Generates High Order Prime Numbers
linSearch	Linear search program reads from a large data file to search a key
margeSort	Merge sort program to arrange unordered numbers from a large text file
iRender	Image Rendering of Large BMP files using Image Smoothing Algorithm
smvm	Sparse Matrix Vector Multiplication
swk	Benchmark program with synthetic work load.

Algorithm 1 presents major parts of the experimental system. The algorithm takes the number of concurrent threads and test runs as input and outputs the measured execution efficiency for the batch. Uniform sampling of data across the values of possible aggregate CPU loading was implemented. A test run of each batch of threads provides one measurement (the minimum efficiency is considered for accuracy) of CPU availability.

Algorithm 1 Aggregate load distribution and measurement of execution efficiency.

Input: Number of threads (n), and number of test run (tr)
for count $\leftarrow 1 \ \ldots \ tr$ **do**
  Select a random aggregate CPU load between $[\epsilon \ \ldots \ n]$
  for i $\leftarrow 1 \ \ldots \ (n-1)$ **do**
    $L_i = Max\left(L - \sum_{j=1}^{i-1} T_j - (n-i)), \epsilon\right)$
    $U_i = Min\left(L - \sum_{j=1}^{i-1} T_j - (n-i) \times \epsilon), 1.0\right)$
    Select T_i randomly so that $T_i \in [L_i, U_i]$
    $T_i \leftarrow (U_i - L_i) \times T_i + L_i$
  end for
  $T_n \leftarrow L - \sum_{i=1}^{n-1} T_i$
  Compute sleep phase length for each thread
  Compute total work amount for each thread
  Assign phase shift values of all threads
  Spawn *benchmark* threads concurrently into the system
  Wait for threads to complete assigned work
  Collect and Persist data in respective CSV files
end for
Close all files and connections
Output: Thread execution report

To ensure a uniform sampling of data across the values of possible aggregate loadings, a random value of aggregate loading between ($\varepsilon \times n$) and n is chosen first. The value of ε is 0.05, denoting a 5% CPU-load, is used to represent the extreme lower CPU load value for a thread. A thread cannot have CPU load value of zero (0.0), else it would never complete its assigned work. The selected aggregate load (sum-total) is then distributed among threads using expressions inside the inner for loop. For example, if a thread batch contains 8-threads then the minimum-limit is 0.4 which is (0.05 × 8) and the maximum-limit is 8.0 (i.e., all CPU-bound threads). Then a CPU load value between 0.4 and 8.0 is chosen and distributed among 8 threads using *Algorithm 1*.

D. Empirical CPU Availability Case Studies

For measuring the CPU availability of the quad-core processor, three case studies were conducted in which 8, 12, and 16 threads were spawned concurrently to observe the execution efficiency in quad-core systems. A moving average with a window size of 0.1% is calculated and plotted in the Figure 1. A 90% confidence interval value is computed and presented along with measured execution efficiency. CPU availability graphs for 8 and 16 threads are shown in Figures 1 and 2 respectively.

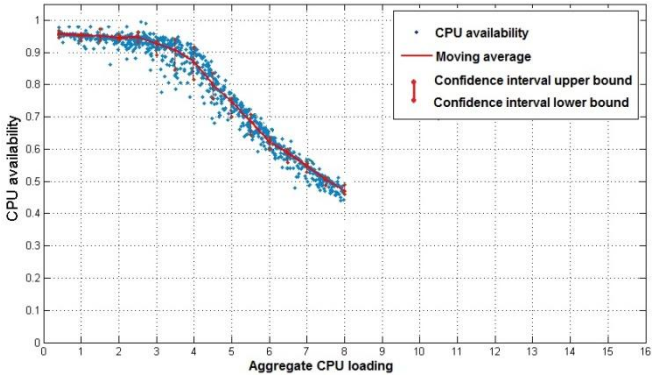

Fig. 1. CPU availability of 8 concurrent processes in a quad-core machine. Results of 4,000 independent test cases with average efficiency and 90% confidence interval bars.

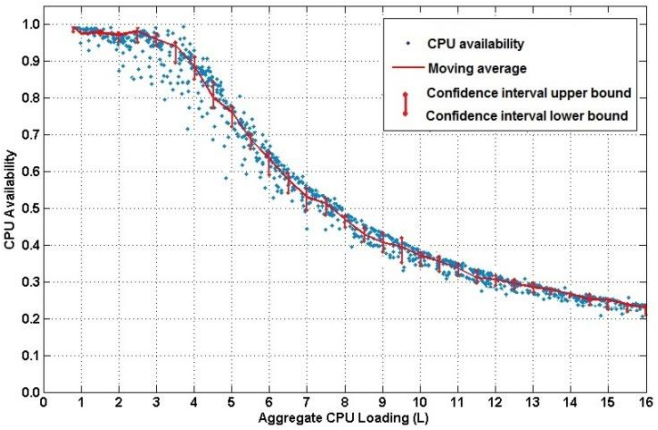

Fig. 2. CPU availability of 16 concurrent processes in a quad-core machine. Results of 4,000 independent test cases with average efficiency and 90% confidence interval bars.

Figures 1 and 2 show measured CPU availability scatter graphs for 8 and 16 concurrent processes executing on a quad-core processor, superimposed with the plots of the moving average and a 90% confidence interval bars. In these figures, the horizontal axis represents aggregated CPU load and the vertical axis represents CPU availability. Each small dot in these graphs is an independent test case

ISBN: 978-988-14047-3-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

measurement of CPU availability. There are 4,000 dots in each figure representing the measured CPU availability value among the concurrent threads. A moving average line is also drawn through the data on the graphs for helping to visualize the average measured performance. A window size of 0.1 aggregate CPU load and incremental value of 0.01 was used to calculate the moving average values. A similar sliding window approach was employed to calculate the 90% confidence interval upper and lower limits. The empirical results for the quad-core processor under Linux environment show that the CPU availability prediction is fairly accurate in all cases when the CPU is lightly, moderately (similar to real-world environment), and heavily loaded.

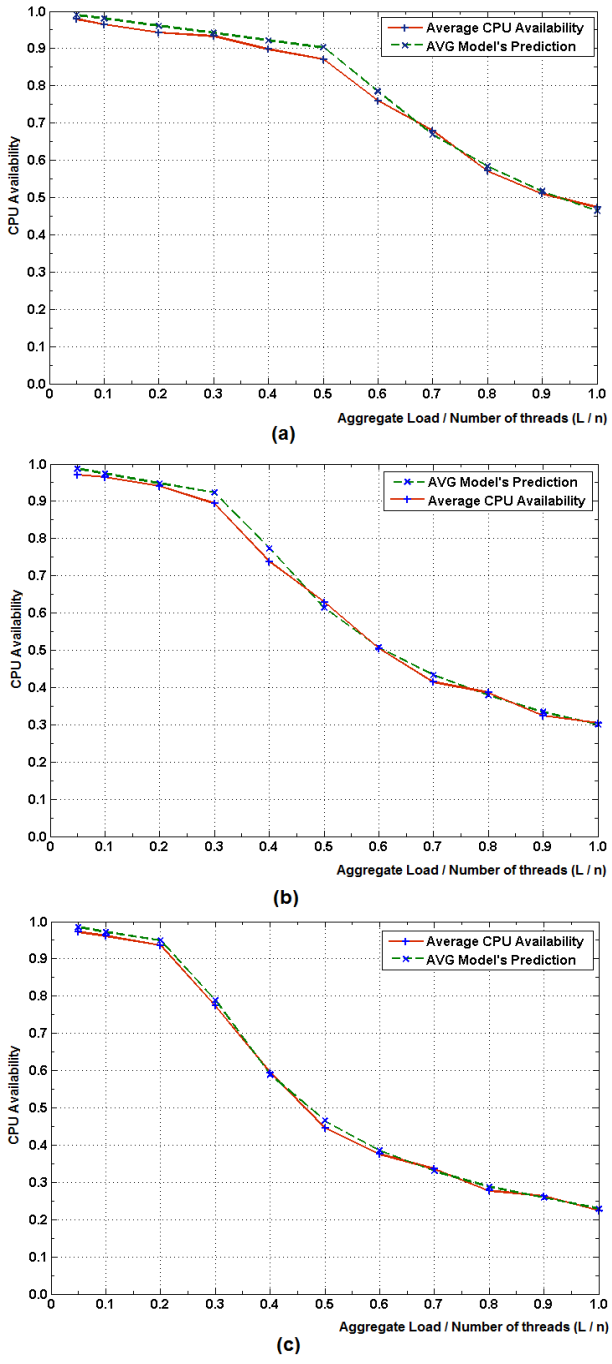

Fig. 3. The measured average CPU availability data and predicted average data associated with Eq. 3 for (a) multi-core systems with r = 4 and n = 8. (b) multi-core systems with r = 4 and n = 12. (c) multi-core systems with r = 4 and n = 16.

TABLE II
MEASURED PREDICTION ERROR FOR THE CASE STUDIES OF 8, 12 AND 16 CONCURRENT THREADS IN A QUAD-CORE MACHINE

L/n	8 Thread CPU Availability			12 Thread CPU Availability			16 Thread CPU Availability		
	Avg	Model (c_{avg})	Error (%)	Avg	Model (c_{avg})	Error (%)	Avg	Model (c_{avg})	Error (%)
0.05	0.979	0.990	1.08	0.9714	0.987	1.56	0.9716	0.985	1.34
0.10	0.965	0.981	1.64	0.9653	0.974	0.87	0.9607	0.970	0.93
0.20	0.943	0.961	1.82	0.9408	0.948	0.72	0.9364	0.940	0.36
0.30	0.933	0.942	0.89	0.8942	0.923	2.88	0.7741	0.786	1.19
0.40	0.898	0.922	2.36	0.7377	0.774	3.63	0.5939	0.585	0.89
0.50	0.871	0.903	3.24	0.6288	0.614	1.48	0.4458	0.465	1.92
0.60	0.760	0.785	2.53	0.5054	0.508	0.26	0.3753	0.386	1.07
0.70	0.679	0.669	0.96	0.4137	0.433	1.93	0.3358	0.330	0.58
0.80	0.571	0.583	1.19	0.3857	0.378	0.77	0.2767	0.288	1.13
0.90	0.510	0.517	0.72	0.3235	0.334	1.05	0.2627	0.255	0.77
1.00	0.473	0.464	0.85	0.3038	0.300	0.38	0.2242	0.229	0.48

Figure 3 (a, b, c) corresponds to the experimental data for 8, 12, and 16 batches of threads in a quad-core machine. In this figure, the horizontal axis represents the normalized aggregate load, *L/n*, and the vertical axis represents the CPU availability for running threads. The average CPU availability line is plotted from Table II, which is from empirical multi-core test data discussed in Section IV. It can be observed that the CPU availability model lines are smooth and follow similar pattern compared with the average lines obtained empirically (Fig. 1 and 2). As the number of threads in a batch increases, the shape and pattern of the line of prediction model become almost identical to average measured line.

Table II contains empirical data to validate the proposed model. It can be observed from the table that for a set of 8-threads, the maximum prediction error is 3.24% when normalized aggregate loading is 0.5. For 12-threads, the maximum prediction error is 3.63% when normalized aggregate loading is 0.4. Finally, for 16-threads, the maximum error is 1.92% when normalized aggregate loading is 0.5.

This analysis shows that the introduced average CPU availability model is consistent with the measured values. The maximum predicted error is only 3.63%. Thus, the model depicts accuracy and reliability and can be deployed in real-world applications for scheduling.

V. CONCLUSION

In this paper, a new multi-core CPU availability prediction model is introduced. Given a set of processes, aggregate CPU load of the set of processes, number of CPU cores, and hyper-threading of a machine, we can predict the CPU availability for the set of processes with very small error. A mathematical prediction model is introduced and validated. A wide range of test cases have been conducted in Linux systems using real-world benchmark programs along with synthetic benchmark programs for verifying the accuracy of the new CPU availability prediction model. All benchmark programs are implemented using the C language and deployed in Linux systems concurrently. The results of empirical studies are presented in this paper in the form of scatter plots, and tables. Thread availability scatter plots provide clear visualization of measured efficiency based on the density of the dots in the plots. The empirical studies performed for validating the average CPU availability model shows that the model values follow the same shape and

ISBN: 978-988-14047-3-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

pattern of the experimentally measured average CPU availability lines with a maximum error rate less than 4.0%. This model is suitable for applications that require a single prediction value instead of upper- and lower bounds for dispatching tasks. Using this reliable information one might be able to determine the order in which tasks should be assigned to the system so that the completion time of all the tasks is minimized.

The introduced prediction model can be used as a building block for distributed task scheduler to determine the order (or find sub sets) in which tasks should be assigned to compute nodes for minimizing the total execution time prior to its placement in the run queue. All the obtained results justify the strength of the introduced model for predicting the CPU availability of multi-core systems while executing threads. Hence, the ability of this model to predict the CPU availability and multi-core processor efficiency for process execution while the CPU resource availability is uncertain has been demonstrated. Finally, the usefulness of the introduced prediction model in real-world applications for estimating the execution efficiency of processes before they are placed into the run-queue by a scheduler has been motivated, and is the topic of future studies.

ACKNOWLEDGMENT

The authors would like to thank Jonathan Mullen, System Administrator, School of Computer Science, University of Oklahoma, for his time and coordinated support. We would also like to extend our gratitude to Dr. Mahendran Veeramani for his valuable feedback towards this work.

REFERENCES

[1] Khondker S. Hasan, John K. Antonio, and Sridhar Radhakrishnan, "A New Composite CPU/Memory Model for Predicting Efficiency of Multi-core Processing," *The 20th IEEE International Symposium On High Performance Computer Architecture (HPCA-2014) workshop*, Sponsored by: IEEE Computer Society, Orlando, FL, USA, 2014.

[2] Martha Beltrán, Antonio Guzmán and Jose Luis Bosque, "A new CPU Availability Prediction Model for Time-Shared Systems," *IEEE Computer*, Vol 57, July 2008.

[3] 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.

[4] Khondker S. Hasan, Sridhar Radhakrishnan, and John K. Antonio, "Composite Prediction Model and Task Distribution on a Cloud of Multi-core Processors," *The 20th IEEE International Conference on High Performance Computing (HiPC) Workshop on Cloud Computing Applications (IWCA-13)*, Bangalore, India, 18-21 Dec. 2013.

[5] Khondker Shajadul Hasan, Nicolas G. Grounds, John K. Antonio, "Predicting CPU Availability of a Multi-core Processor Executing Concurrent Java Threads," *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 11)*, sponsor: World Academy of Science and Computer Science Research, Education, and Applications (CSREA), Las Vegas, NV, July 2011.

[6] Silberschatz Avi, Galvin B. Peter, Gagne Greg, Operating System Concepts, Eighth Edition, John Wiley and Sons, 2009.

[7] M. Beltrán and Antonio Guzmán, "How to Balance the Load on Heterogeneous Clusters," *International Journal of High Performance Computing Applications*, Volume 23, No. 1, pp. 99118, Spring 2009.

[8] Tyler Dwyer, Alexandra Fedorova, Sergey Blagodurov, "A Practical Method for Estimating Performance Degradation on Multicore Processors, and its Application to HPC Workloads," *International Conference on High Performance Computing, Networking, Storage and Analysis*, Article No. 83, ISBN: 978-1-4673-0804-5, IEEE Computer Society Press, CA, 2012.

[9] Alexandra Fedorova, David Vengerov, Daniel Doucette, "Operating System Scheduling On Heterogeneous Core Systems," Sun Microsystems Technical Report, http://www.techrepublic.com /whitepapers /operatingsystem-scheduling-on-heterogeneous-core-systems/314436, July 2007.

[10] Jen-Chieh Yeh; Chi-Hung Lin; Chun-Nan Liu, "Multi-core system performance prediction and analysis at the ESL," *Int. J. of Computational Science and Engineering*, DOI: 10.1504/ IJCSE.2014.058700, Vol. 9 No. 1/2, pp. 86-94, 2014.

[11] Andrew S. Tanenbaum, Modern Operating Systems, Third Edition, ISBN13: 978-0136006633, Prentice Hall Inc., 2008.

[12] Shannon Cepeda, "Intel Hyper-Threading Technology: Your Questions Answered," http://software.intel.com/en-us/articles/intel-hyper-threadingtechnology- your-questions-answered 27 January, 2012.

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

[14] R. Wolski, N. Spring, and J. Hayes, "Predicting the CPU Availability of Time-Shared Unix Systems on the Computational Grid," *Proc. 8th International Symposium on High Performance Distributed Computing*, pp. 105-112, ISBN: 0-7803-5681-0, August 2002.

[15] Khondker Shajadul Hasan "A Distributed Chess Playing Software System Model Using Dynamic CPU Availability Prediction," *International Conference on Software Engineering Research and Practice (SERP-11)*, Las Vegas, Nevada, July 2011.

[16] Brian K. Tanaka, "Monitoring Virtual Memory with vmstat," Linux Journal, http://www.linuxjournal.com/article/8178, Oct 31, 2005.

[17] M. Tim Jones, "Inside the Linux scheduler 2.6", *The Journal of A Linux Sysadmin*, http://www.ducea.com/2006/07/08/inside-the-linux-scheduler, July 2006.

[18] M. Tim Jones, "Inside the Linux scheduler," The latest version of this all-important kernel component improves scalability, *IBM Technical Report*, http://www.ibm.com/developerworks/linux/library/lscheduler/ June 2006.

[19] Zhang, Y., Owens, J.D., "A quantitative performance analysis model for GPU architectures," *IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 382–393 (2011).

[20] Paessler Knowledge Base, "Monitoring Running Processes in Linux," http://www.paessler.com/knowledge-base/en/topic/29403-monitoringprocesses-in-linux, Dec 2011.

[21] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, Keith I. Farkas, "Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance," *31st Annual International Sympo-sium on Computer Architecture (ISCA)*, June 2004.

[22] Sim, Jaewoong and Dasgupta, Aniruddha and Kim, Hyesoon and Vuduc, Richard, "A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications," *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*, pp. 11-22, New Orleans, Louisiana, 2012.