A New Composite CPU/Memory Model for Predicting Efficiency of Multi-core Processing

Khondker S. Hasan, John K. Antonio, Sridhar Radhakrishnan
School of Computer Science
University of Oklahoma
Norman, OK, USA
Email: {shajadul, antonio, sridhar}@ou.edu

Abstract—Techniques for predicting the efficiency of multi-core processing associated with a set of tasks with varied CPU and main memory requirements are introduced. Given a set of tasks each with different CPU and main memory requirements, and a multi-core system (which generally has fewer cores than the number of tasks), our goal is to derive equations for upper- and lower-bounds to estimate the efficiency with which the tasks are executed. Prediction of execution efficiency of processes due to CPU and required memory availability is important in the context of making process assignment, load balancing, and scheduling decisions in distributed systems. Input parameters to models include: number of cores, number of threads, CPU usage factor of threads, available memory frames, required amount of memory for each thread, and others. Additionally, a CPU availability average prediction model is introduced from the empirical study for the set of applications that require a single predicted value instead of bounds. Extensive experimental studies and statistical analysis are performed and observed that the proposed efficiency bounds are consistently tight. The model provides a basis of an empirical model for predicting execution efficiency of threads while CPU and memory resources are uncertain. To facilitate scientific and controlled empirical evaluation, real-world benchmark programs with dynamic behavior are employed on UNIX systems that are parameterized by their CPU usage factor and memory requirement.

Index Terms—Composite prediction model, CPU availability, Execution Efficiency, Memory availability, Multi-core processors, Modeling and prediction.

I. INTRODUCTION

Multi-threading is a common technique used for exploiting performance from multi-core processors. 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%.

System efficiency prediction involves estimating the system’s behavior for the set of tasks to be executed on it. The prediction of resource availability in a system is important in the context of making task assignment, load balancing, and scheduling decisions in distributed systems. Making such predictions is complicated by the dynamic nature of the system and its workload, which can vary drastically in a short span of time [1]. 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 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. It is also useful to know the CPU requirement of the task, which is the fraction of time a task requires the CPU.

While priori information such as memory and CPU requirements of tasks are useful, it is easier to obtain in some cases and more difficult in others. For example, in the case of Merge sort we can approximately determine the memory requirement based on the number of elements to be sorted. For tasks such as generating prime numbers, computing Fast Fourier Transforms, and related others require significant use of the CPU. One can also know the CPU and memory requirements of a task based on the information gathered from its earlier executions. The execution of many scientific models (economic, meteorological, numerical, and others) fall into this category, where the program remains the same and the data on which it operates changes over time.

Given the CPU requirements of tasks in a run queue, Beltrán et al. [1] 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 these information can be used to schedule the execution of tasks in such a way that the completion time of all the tasks is minimized [1]. Khondker et al. [4] extended the work of Beltrán et al. [1] as follows. 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 \textit{a priori}. To address this challenge, the analytical model in Khondker et al. [4] provided tight upper- and lower-bounds on CPU availability. The bounds are necessary since the actual CPU availability depends on the order of execution of tasks in the batch. Thus the analytical model in Khondker et al. [4] is oblivious of the CPU scheduler.

In the present paper, we have further improved the analytical model in [4] in several ways. First, we have introduced a \textit{memory model} that determines the \textit{execution efficiency}, which is the amount of CPU time that a task requires to complete its execution when adequate free memory frames are available in main memory divided by the total time it takes when only part of memory frames are available in the main memory. The amount of main memory available for a task is determined by the main memory requirements of each of the tasks in the batch of tasks and the order in which they are scheduled to be executed by the CPU. Second, the CPU availability model in [4] combined non-trivially with the memory model developed in this paper to derive a \textit{composite model}. The composite prediction model consists of analytically derived upper- and lower-efficiency bounds for execution of tasks in the batch. Third, we have applied an empirical approach based on thread assignment observation in [4] to introduce a \textit{CPU availability average} prediction model. This empirical model provides a single prediction value (instead of upper- and lower-bounds) of CPU availability for a set of tasks prior to their execution without explicit knowledge of the mapping between available cores and tasks.

Given a set of tasks (with known CPU and memory requirements) and set of compute nodes, we can use the composite model to determine the best node (in terms of thread execution efficiency) and assign the task to that node. An extensive empirical work using real-world benchmark programs with dynamic behavior are carried out to measure the accuracy of introduced prediction models.

The rest of the paper is organized in the following manner. Section II discusses relevant background related to the resource availability prediction models, and motivates the importance of predicting resource availability. Section III introduces the composite prediction model and derives the upper- and lower-bounds from the CPU availability and memory models. Section IV provides information on the empirical environment, provides case studies for prediction models, and shows the statistical analysis of the extensive empirical work to validate introduced models. Finally, Section V contains concluding remarks, application areas for introduced models, and future work.

## II. Background

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 [1], [5]. 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 [6]. The introduced models overcome this limitation and suitable for both CPU and I/O bound processes.

Federova et al. [8] also worked on operating system scheduling on heterogeneous 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 [7] introduced a practical new method for estimating performance degradation on multi-core processors, and it’s application to workloads of clusters nodes.

When a processor accesses memory, it spends a significant amount of time waiting for the data to become available because of cache misses which may result in up to 50% of stall time. This situation generates huge overhead when the frequency of memory access increases. To overcome this situation, most of the recent hardware designs have implemented multi-threaded processor cores in which two or more hardware threads are assigned to each core [5]. That way, if one thread stalls while waiting for memory, the core can switch to another thread [5]. To maintain its own architectural state, each core has its independent register set and thus appears to the operating system to be a separate physical processor. From an operating system perspective, each hardware thread (when hyper-threading is enabled) 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. We have incorporated the effect of hyper-threading in our new CPU availability and memory models for accuracy.

## III. Composite CPU Availability and Memory Model

The primary focus of this section is to derive a composite prediction model from proposed CPU availability and memory models for estimating the efficiency of thread execution on multi-core systems. It is necessary to have accurate models for estimating CPU and memory resources because of the dynamic nature of computer systems and their workload. As new processes are assigned or existing processes complete execution, the CPU and memory availability of a given compute node can change significantly in a short interval of time. Therefore, existence of composite prediction models are important because there exists a wide range of applications and scientific models (e.g., geological, meteorological, economical
and others) that requires extensive use of both CPU and memory resources, repeatedly. Assigning a set of batch tasks in a distributed environment (distributed schedulers) can utilize the composite model 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.

A. Bounds of Composite CPU Availability and Memory Model

A composite analytical framework consisting of upper- and lower-bounds are derived for estimating the overall efficiency for a batch of tasks in multi-core systems. The composite upper-bound is derived from the upper-bounds of CPU availability and memory models. Similarly, the composite lower bound model is derived from the lower bounds of CPU availability and memory models. The composite prediction model’s upper-bound, denoted by \( \bar{\tau} \), represents the best case efficiency value of a compute node for concurrent thread execution can be represented by the product of two models because CPU and memory are the two primary factors used to characterize compute nodes:

\[
\bar{\tau} = \overline{\tau} \times \overline{m}.
\]  

The values of \( \overline{\tau} \) and \( \overline{m} \) represents the relative impact of a compute nodes’s overall efficiency due to loading of nodes’s CPU and memory resources, respectively.

The CPU availability upper-bound model (\( \overline{\tau} \)), represents the best case CPU availability wherein none of the threads uses the CPU resource concurrently [4]. If the sum of the usage factors of the threads is less than unity, then it is possible that the CPU availability could be as high as unity (i.e., 100%). When the sum of the CPU usage factors is greater than unity, then the best possible value for CPU availability is \( 1/L \) where \( L \) is the aggregate loading factor. For a multi-core node with \( r \) cores and \( n \) threads, the following models define upper-bounds for CPU availability.

\[
\overline{\tau} = \frac{1}{\max(1, \frac{L}{r})} = \begin{cases} 1, & \text{if } L/r < 1 \\ r/L, & \text{if } L/r \geq 1 \end{cases}
\]  

It is customary to keep several processes running in time-shared systems. Predicting the execution efficiency of tasks when the available memory is less than required memory is critical in making task assignment and scheduling decisions. The upper-bound model for execution efficiency depending on memory availability, \( \overline{m} \), can be represented as:

\[
\overline{m} = \frac{1}{\tau + (A_p + A_v) \times ((\sum_{i=1}^{n} R_i) - M_a) + (B_t + B_s)}.
\]  

Table I summarizes the notation and definitions of required parameters of memory bounds. The memory upper-bound model consists of two scenarios. When \( M_a \) is greater than \( R \) and cache memory hit ratio is 100% (all page entries are in translation look-aside buffer), there will be a single look-up in the page table and no additional virtual memory access overhead due to page fault service time. The second scenario is where \( M_a \) is less than \( R \); due to shortage of required number of memory frames, pages will be swapped out to virtual memory. For deriving the upper-bound efficiency model, the ideal thread execution time, denoted by \( \tau \) (derived in Eq. 4), is divided by an expression that represents the execution time including the virtual memory access time and backing store overhead for the swapped pages. The ideal thread execution time (\( \tau \)) can be expressed as:

\[
\tau = \left( A_p + A_v \right) \times \sum_{i=1}^{n} R_i + \rho, \quad \text{if } M_a \geq R.
\]

The ideal thread execution time (\( \tau \)) represents the best possible execution environment in which threads receive the required amount of primary memory in all situations and the cache memory hit ratio is 100%. That is, for each page, there is only one look-up in page table and a single access in primary memory and data are available in the primary memory. Thus, the memory access time and the data process time, \( \rho \), represents the ideal thread execution time.

Figure 1 shows surfaces of the theoretically derived upper- and lower-bound model in which horizontal axes represents

<table>
<thead>
<tr>
<th>Terms</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_i \geq 0 )</td>
<td>( R_i ) is the required number of memory frames by the process ( i ) where ( i = (1, 2, \ldots, n) ).</td>
</tr>
<tr>
<td>( R &gt; 0 )</td>
<td>( R ) is the total required number of memory frames by all processes ( (R = R_1 + R_2 + \ldots + R_n) ).</td>
</tr>
<tr>
<td>( M_a \geq 0 )</td>
<td>( M_a ) is the total number of free memory frames available in the system.</td>
</tr>
<tr>
<td>( \overline{\tau} )</td>
<td>( \overline{\tau} ) is the access time of translation look-aside buffer in the cache memory.</td>
</tr>
<tr>
<td>( \overline{m} )</td>
<td>( \overline{m} ) is the access time of primary memory.</td>
</tr>
<tr>
<td>( \rho &gt; 0 )</td>
<td>( \rho ) is the access time of virtual memory (backing store).</td>
</tr>
</tbody>
</table>

Fig. 1. Composite upper- and lower-bound surfaces for \( n = 16, r = 4, \overline{\tau} = 100 \), and required memory availability variation from 0% - 100%.
aggregate CPU loading (of the set of tasks) and required memory availability percentage (required memory by the set of tasks with respect to the available main memory), and the vertical axis represents the overall efficiency of the compute node for executing the batch of tasks. It can be observed from the Figure 1 that for CPU loading values greater than the number of cores (here, \( r = 4 \)), the idealized function for \( \tau \) decreases according to the ratio of the number of cores to the total CPU loading. The significant degradation of efficiency value can also be observed when the availability of required memory for threads decreases from 100% down to 0%. Ideally, if a node’s CPU and memory resources are both lightly loaded, then the efficiency of the node is at or near its maximum value.

The composite prediction model’s lower bound, denoted by \( c_l \), represents the worst case efficiency value of a compute node for concurrent thread execution can be represented by the product of CPU and memory lower-bounds as:

\[
\xi = c_l \times m_l.
\] (5)

The lower bound model for CPU availability is associated with a situation in which the threads’ usage of the CPU resource has maximum overlap and threads always use the CPU resource concurrently [4]. For a multi-core node with \( r \) cores, lower-bound for CPU availability can be defined as:

\[
\xi = \frac{1}{1 + \left( \frac{r - 1}{r} \right) \times \left( \frac{\nu}{\nu} \right)}.
\] (6)

The lower bound model for execution efficiency depending on memory availability, denoted by \( m_l \), that additionally includes the backing store overhead expressed in Eq. 8 and cache miss-hit overhead can be represented as:

\[
m_l = \begin{cases} 
\frac{\tau + A_p \times \sum_{i=1}^{n} R_i}{\tau + (A_p \times M_a) + (A_s + A_p + A_c) \times ((\sum_{i=1}^{n} R_i) - M_a) + \eta}, & \text{if } M_a \geq R, \\
\frac{\tau + A_p \times \sum_{i=1}^{n} R_i}{\tau}, & \text{else,}
\end{cases}
\] (7)

The lower bound model represents the worst case execution time that includes a memory access time because of cache misses, the backing store seek time for individual pages (virtual memory access by multiple concurrent threads will result in storage of pages in non-consecutive order), and the I/O queuing delay. The backing store overhead, denoted by \( \eta \), can be expressed as:

\[
\eta = \left( B_l + \sum_{M_a=1}^{R} (B_s) + \kappa \right).
\] (8)

Note that the difference in upper- and lower-bound might be significant, when the initial page fault starts, due to the uncertainty of the backing store seek time, latency, and command queue delay, and others. The number of disk commands waiting in the queue is normally the factor that slows down the disk performance by increasing the average disk queue time [4], [13].

Figure 2 (a, b, c) shows surfaces of the theoretically derived upper-bound model in which horizontal axis represents CPU loading and required memory availability (in percentages), and the vertical axis represents the overall efficiency of the compute node. Purpose of this figure is to show the effect of ideal thread execution time \( \tau \) when the value changes from 1 to 100 sec. It can be observed among surfaces that when the value of \( \tau \) increases from 1 to 100, the overall efficiency of the node increases considering the memory requirement is same in all three cases. The efficiency value increases because when the program runs for longer period of time, the page fault overhead (which is same for all the three cases) has less impact compared to a program which runs for a shorter period of time.

B. Average CPU Availability Prediction

This section introduces a model for predicting the expected (on average) availability of CPU (instead of calculating the upper- and lower-bounds). As illustrated in previous subsection, exact values of CPU availability are difficult to predict because of dependencies on many factors, including context switching overhead, memory speed, 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 approach is employed here to estimate expected CPU availability. Eq. 9, provides a model/explanation

![Fig. 2. Composite model upper-bound surfaces showing the effect of ideal thread execution time, \( \tau \), for a quad-core system (\( r = 4 \)). (a) Surface diagram of upper-bound model for \( \tau = 1 \). (b) Surface diagram of upper-bound model for \( \tau = 10 \). (c) Surface diagram of upper-bound model for \( \tau = 100 \).](Image)
of what has been observed for thread assignment in processor on the average in [4].

The model of Eq. 9 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, denoted by $\xi$. For a multi-core machine, the following prediction model estimates the average CPU availability for a set of tasks:

$$c_{avg} = \begin{cases} 1 - \left(\frac{n-1}{n+r}\right) \times \left(\frac{L}{(n+r)\times \xi}\right), & \text{if } L \leq r \\ \frac{r}{L} - \left(\frac{n-1}{1+(\frac{n-1}{r})\times (r\times \xi+\xi)}\right), & \text{if } L > r. \end{cases} \quad (9)$$

The model of Eq. 9 considers 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 less 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; the usage of CPU resource for threads has maximum overlap. In general, more thread 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 $L \leq r$ or $L > r$) to best-fit the average efficiency plot in [4].

IV. EMPIRICAL STUDIES

A. Overview

The purpose of the experimental study is to empirically measure the efficiency of the machine as a function of aggregate CPU loading and memory availability factors. The following benchmark programs are used for measuring the overall efficiency of a multi-core machine:

- supPrime: High Order Prime Number Generator
- mcp: Monte Carlo Estimation of $\pi$
- smvm: Sparse Matrix Vector Multiplication, and
- tridiSolver: Tridiagonal Solver (using Gaussian elimination)

The utilized benchmark programs consist of expressions that are mix of CPU and memory related operations and thus ideal for the composite model empirical case studies. Aggregate CPU loading and memory requirement values are selected randomly and distributed among threads by using Algorithm 1 and 2, respectively. Uniform sampling of data across the values of possible aggregate CPU loading and memory availability has been ensured. For implementing the benchmark and case study programs, most of the modules of the CPU and memory availability programs are reused.

Algorithm 1 presents major parts of the CPU availability experimental system. It can be observed from Algorithm 1 that for ensuring uniform sampling of data across the values of possible aggregate loadings, a random value of aggregate loading between $(\epsilon \times n)$ and $n$ is chosen first. The aggregate load is then distributed among benchmark threads using expressions inside the inner for loop. The expressions of $U_i$ and $L_i$ is introduced to provide an upper- and lower-limit of available CPU load for the $i^{th}$ thread. A random CPU load value is selected from the range $(L_i, U_i)$ and assigned to the $i^{th}$ thread. The CPU load for the thread is then scaled and placed into $T_i$. For example, for a scenario having two threads, a value of aggregate loading is chosen between a small value ($\epsilon \times 2$) and 2.0; denote this value as $L$. A small value for $\epsilon$ (0.005) is used because a thread can not have CPU load value of 0.0, else it would never complete the defined non-zero work. Then a random value is chosen between $\max\{\epsilon, (L - 1.0)\}$ and $\min\{1.0, L\}$, which defines the CPU usage factor of the first thread, say $T_1$; the CPU usage factor of the second thread is then defined as $T_2 = L - T_1$. In general, for $n$ threads, Algorithm 1 is used to randomly assign the CPU usage factors for a given value of aggregate loading $L$.

Algorithm 2 presents major parts of the memory experimental system. Based on CPU load of each thread, total amount of work (upper range) and sleep phase length values are derived. During each work phase, threads accomplish a fixed amount of work. Threads needs to run several phases to complete the total amount of work. Memory availability percentage (MAP) is the percentage of available free primary memory with respect to the total memory requirements by all threads. MAP is computed using memory requirements of threads in a batch.

1) Empirical Environment: The systems used for evaluating the task assignment models are three Intel(R) Xeon(R) Quad-core CPU W3520 with 2.67GHz clock speed, 1.333 MHz bus speed and 6.0 GB of RAM and one Intel dual core with 3.06GHz clock speed, 1.333 MHZ bus speed and 4.0 GB of RAM. These nodes are equipped with Linux kernel version

---

**Algorithm 1** Aggregate load distribution and measurement of execution efficiency.

**Input:** Number of threads ($n$), and number of test run ($tr$)

for count ← 1 ... $tr$ do

Select a random aggregate CPU load between $[\epsilon ... n]$

for $i ← 1$ ... $(n - 1)$ do

$L_i = \text{Max} \left( L - \sum_{j=1}^{i-1} T_j - (n - i), \epsilon \right) $

$U_i = \text{Min} \left( L - \sum_{j=1}^{i-1} T_j - (n - i) \times \epsilon, 1.0 \right) $

Select $T_i$ randomly so that $T_i \in [L_i, U_i]$

$T_i ← (U_i - L_i) \times T_i + L_i$

end for

$T_n ← L - \sum_{i=1}^{n-1} T_i$

Compute sleep phase length for each thread (Eq. 5)

Compute total work amount for each thread (Eq. 6)

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

---
Algorithm 2 Measuring aggregate pages required by threads and allocating memory.

**Input:** Number of threads \((n)\), Number of test runs \((tr)\), Memory requirements of each thread \((ts)\) in KB, and required memory availability percentage \((RMAP)\).

for count ← 1 ... tr do
    Select a random RMAP between 0 ... 120.
    \[ R \leftarrow \frac{\sum_{i=1}^{n} ts_i}{\text{total memory - used memory}} \] sizes are in KB.
    \[ M_a \leftarrow \frac{\text{total memory}}{\text{page size}} \]
    \[ cm \leftarrow M_a - (R \times RMAP) \times 0.01 \]
    Create child threads to carryout the following tasks:
    for k ← 1 ... cm do
        Allocate memory, \(m = (\text{void} *) \text{malloc(pagesize)}\)
        Initialize pages, \(\text{memset}(m, 0, \text{pagesize})\)
    end for
    Occupy measured amount of memory while threads run
    Generate and spawn threads concurrently for measuring execution efficiency
end for

**Output:** \(R\), \(M_a\), and allocated memory.

3.2.0-36. The average CPU load (represents the average system load over a period of time) was 0.018136 per core in a scale of 1.0 (in a fifteen minute period) before running test cases which indicates that the nodes were lightly loaded (essentially unloaded). The C programming language was used to implement the prediction based resource management framework (analytical prediction and task assignment models) with the gcc compiler version 4.6.3.

Threads deployed for composite prediction model validity are independent tasks, meaning there are no interdependencies among threads such as message passing. Threads deployed for validating models are real-world benchmark programs like high order prime number generator, Monte Carlo Estimation of \(\pi\), sparse matrix vector multiplication, and tridiagonal solver. Depending on test case requirements, threads are spawned concurrently in a multi-core machine for estimating the execution efficiency. When a threads finishes its work, an execution report is produced, which contains start time, execution CPU time, idle time, end time, CPU availability for the task, and others.

2) Composite Prediction Model Case Studies: The major objective of this section is to verify whether the introduced upper- and lower-bound models can bind the efficiency of thread execution, when both CPU availability and required memory are varied. The utilized set of benchmark programs represents the classes of real-world applications that depicts dynamic behavior.

For measuring the overall efficiency value of thread execution, three independent case studies using benchmark threads are conducted on a quad-core machine. About 18,000 test runs are performed for three independent sets of test cases in which 8, 12, and 16 threads are spawned concurrently. This vast number of test runs are conducted to empirically cover all possible scenarios of thread execution on multi-core systems. As the focus of this empirical study is to measure the effect in overall efficiency of machines when both CPU and memory usage are varied, the number of threads deployed is always above the number of CPU cores of the machine and the memory availability is varied from 120% down to 0%.

Figure 3 (a) shows measured efficiency surface for the execution of 8 threads in a quad-core machine superimposed with theoretically derived composite upper- and lower-bounds. About 4,000 independent test cases are carried out to capture all possible execution efficiency scenarios due to CPU and memory availability variation. A moving average is taken from these test results with a sliding window size of 0.10 aggregate CPU loading and 0.5% of memory availability, and incremental value of 0.01 CPU loading and 0.5% memory availability. The data is then converted to a two dimensional matrix format for plotting a 3D surface diagram.

It can be observed from Figure 3 (a) is that the efficiency value decreases significantly when the aggregate CPU load
reaches beyond the CPU cores (here, \( r = 4 \)) because of the increased CPU contention among running threads. Figure 3 (a) also illustrates the decrease of node efficiency because of the decrease of memory availability. When the total memory requirement is higher than the total available memory, the relative performance has a significant impact on execution time due to page fault service time. In addition to memory availability, total amount of memory required by concurrent threads is a factor in predicting the thread execution efficiency. More threads require more memory and triggers more page swap out, which results in degraded performance. Moreover, the I/O is serial and it suffers from queuing delay. The efficiency decrease due to memory availability is moderate in these empirical studies as the value of \( \tau = 30 \text{sec} \) for test runs (refer to Figure 2).

The main purpose of Figures 3 (a) and (b) is to illustrate that the efficiency surface of 8-threads spawned concurrently in a multi-core machine can be bound using the introduced composite upper- and lower-bound models derived in Eq 1 and Eq 5. Figure 3 (b) shows the same surface diagrams in a different perspective to illustrate that there are no overlap among efficiency, upper, and lower limit surfaces. From the empirical results and measured efficiency surface plot in Figure 3 (a) and (b), it is apparent that theoretically derived upper- and lower-bounds introduced in this section do bound the actual measured efficiency surface very well.

In the second set of test cases, 12-threads are spawned concurrently in the same quad-core machine for measuring the CPU availability and the effect of memory availability in concurrent thread execution. A similar approach has been taken, like 8-threads, for conducting the empirical case studies. About 6,000 independent test cases are carried out in which CPU and memory availability values were selected randomly to capture all possible scenarios. A similar performance degradation is observed when the aggregate CPU load reaches beyond the CPU cores and when the memory availability decreases below 100% (page swap out starts to backing store). Theoretically derived upper and lower limit binds the actual measured efficiency surface of 12 thread very well.

In the final set of empirical studies to validate the composite bound, 16-threads are spawned concurrently in multi-core nodes and efficiency surface is plotted from the resulting data. About 8,000 independent test cases were conducted in which CPU and memory availability were selected randomly and distributed among threads to cover all possible scenarios. It can also be seen from Figure 4 (a) and (b) that theoretically derived upper- and lower-limits do bound the actual measured efficiency surface very well for 16 concurrent threads.

In further reporting the results of the studies, it is convenient to define the normalized aggregate load, \( L/n \), which is the aggregate load \( L \) normalized by the number of threads \( n \). For sample values of normalized aggregate load, Table II shows the average measured CPU availability (Avg.), the difference in the upper- and lower-bound models (Bnd Diff) and the difference in the 90% confidence interval limits (CI Diff) for 8, 12 and 16 concurrent threads on a quad-core processor.

Table II shows that the difference between the upper- and lower-bounds that can reach as high as 0.466, for 8-threads and a normalized aggregate loading of 0.50. However, the measured 90% confidence interval difference for the case is much smaller, around 0.113. The difference of the formula-based bound is more precise when the CPU is lightly or heavily loaded for quad-core processors. Additionally, the empirically-based values for CI Diff can be used as a basis for creating sharper estimates of CPU availability.

This empirical results justifies the validity of the introduced composite prediction model bounds in multi-core environment. These empirical results shows the accuracy and reliability of the composite prediction model which is used as the building blocks of both task assignment models.

V. Conclusion

This paper has introduced composite prediction model derived from the proposed CPU availability and memory models. CPU availability and memory models have been introduced for predicting (and measuring) the overall efficiency of machines for concurrent thread execution on a time-shared system. The
composite prediction model was validated empirically by an extensive set of case studies, which demonstrates that the introduced upper- and lower-bound models can bind the thread execution efficiency very well for the cases considered where the CPU and memory requirements of threads were varied from high to low levels. As would be expected, degradation in CPU availability occurs when total CPU loading is greater than the total capacity of all CPU cores. In addition to total CPU loading, the total number of concurrent threads and the amount of memory required by threads are also factors in predicting the thread execution efficiency. More threads incur more context switching overhead, which results in degraded efficiency.

Our proposed new composite prediction model shows that the derived bounds processing efficiency are consistently tight. Using this information one might be able to determine the order in which tasks have to be assigned to the system so that the completion time of all the tasks is minimized. Furthermore, the introduced composite prediction model can be used as the building block for task scheduler for assigning tasks to proper computing nodes in a distributed environment for minimizing task execution time. The execution efficiency of a batch of tasks can be predicted before placing the tasks into the run-queue.

All the obtained results justify the strength of introduced models for predicting the efficiency of a compute node while executing threads. The ability of introduced models to predict the resource availability and efficiency for thread execution while the resource (CPU and memory) availability is uncertain in dynamic environment has been demonstrated. Thus, the usefulness of using the introduced models in real-world applications for task assignment has been motivated. Besides, the introduced prediction models can be further enhanced in the GPU environment (GPU architecture are way different than the CPU architecture) for estimating the execution efficiency of kernels before launching. The prediction results can be utilized to reschedule blocks and threads to enhance execution efficiency and save power and is the topic of future studies.

### ACKNOWLEDGMENT

Authors would like to thank Jonathan Mullen, System Administrator, School of Computer Science, University of Oklahoma, for his time and coordinated support.

### REFERENCES


