Department of Computer Science
George Mason University Technical Reports

4400 University Drive MS#4A5
Fairfax, VA 22030-4444 USA
http://cs.gmu.edu/ 703-993-1530

Epochs: Trace-Driven Analytical Modeling of Job Execution Times

Daniel A. Menascé
menasce@gmu.edu

Shouvik Bardhan
sbardhan@masonlive.gmu.edu

Technical Report GMU-CS-TR-2014-01

Abstract

Queuing theory has extensively studied the problem of estimating job execution times in steady state conditions both in the case of single queues and queuing networks. This paper discusses the use of closed Queuing Network (QN) models during finite time intervals to estimate the execution time of jobs submitted to a computer system. More specifically, the paper presents the *Epochs* algorithm that allows job traces and randomly generated job inter-arrival times to be used as input to analytic models of computer systems. In other words, the paper combines methods used in discrete event simulation for the characterization of job arrivals with efficient analytic methods to model contention for resources in a computer system. The Epochs algorithm was validated against experimental results using jobs derived from a micro-benchmark and real jobs. The validation shows that the absolute relative error between measurements and execution time predictions obtained with the Epochs algorithm is below 10% in most cases and is at most 15%.

1 Introduction

This paper discusses the use of closed Queuing Network (QN) models during finite intervals to estimate the execution time of jobs submitted to a computer system. Queuing theory has extensively studied the problem of estimating job execution times in steady state conditions both in the case of single queues or queuing networks (see e.g., [2, 8, 10, 13]). Early results on queuing theory were derived assuming certain stochastic assumptions (e.g., steady-state equilibrium, Poisson arrivals, exponentially distributed service times). Some of these results were later generalized to more general distributions [2], but still the steady-state equilibrium assumption was required in these cases.

However, the results obtained under stochastic assumptions proved to be quite robust even when these assumptions were violated. Buzen explained why in his formulation of operational analysis of computer system performance [3]. Operational analysis establishes mathematical relationships between variables that can be measured during a finite time interval. If the relationships are always true they are called operational laws and if they require some assumptions they are called operational theorems. The validity of the assumptions in operational theorems can also be assessed by taking measurements during the same finite interval during which the relationships are established. Examples of operational assumptions are: (1) Flow Balance (i.e., the number of arrivals is equal to the number of departures during a given time interval, (2) Homogeneous Arrivals (i.e., the arrival rate does not depend on the queue size, (3) Homogeneous Service Times (i.e., the mean time between completions does not depend on the queue size), and (4) One-Step Behavior (i.e., a queue length can only vary by increments of $\pm$ 1) [3]. Note that steady-state implies flow balance but the converse is not true [4].

Buzen and Denning have derived in [4, 5] the operational counterpart of the Mean Value Analysis (MVA) [13] equations for solving closed QN models. They showed that the MVA equations are valid for finite time intervals if flow balance, one-step behavior, and homogeneous service times are met. As will be seen later in this paper, this is a key aspect on which we rely on.

The steady state treatment of queuing systems considers open or closed systems. In the case of open queuing systems, the job arrival process is characterized by an inter-arrival time distribution and in the case of closed systems, the workload is characterized by a job population vector that indicates the steady state number of concurrent jobs of each class. Contention for resources among jobs leads to waiting times that are used to determine the steady state execution time of jobs (either average and in some cases higher moments or distributions).

This paper considers a different kind of problem, the understanding of which is better illustrated with the help of Fig. 1. The figure shows a computer system (top right) and a model of that system (bottom right), which represents the processors, I/O devices and their respective queues. The system model can be solved using simulation techniques [14] or analytic models such as analytic queuing networks [2, 10, 13].

The left-hand side of Fig. 1 shows three typical methods for characterizing the arrival process of jobs to a computer

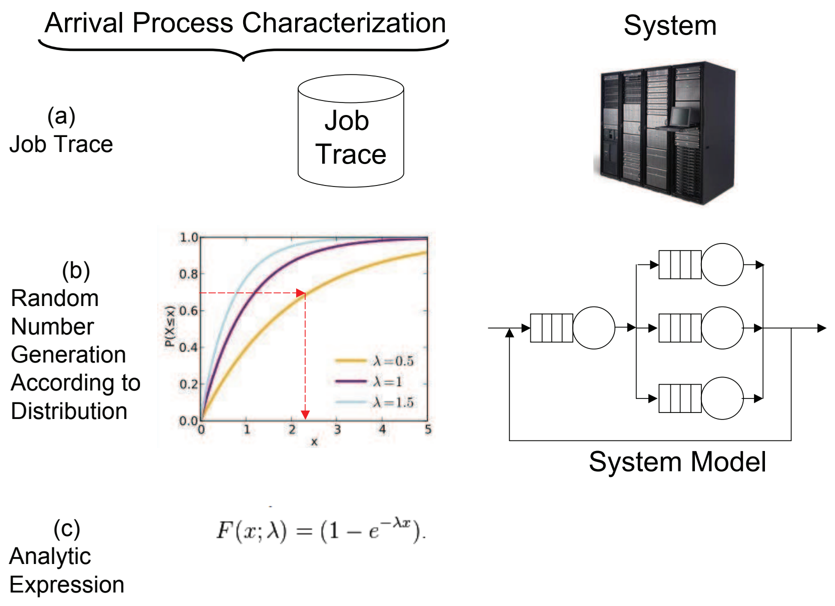
Arrival Process Characterization
System
(a) Job Trace
Job Trace
(b) Random Number Generation According to Distribution
P(X≤x)
x
λ=0.5
λ=1
λ=1.5
System Model
(c) Analytic Expression
$F(x;\lambda) = (1 - e^{-\lambda x})$.

Figure 1: Methods for characterizing arrival processes in modeling.

system. Method (a) consists of a job trace that can be either replayed against the real system in order to derive actual measurements or used as input in trace-driven simulation studies. Method (b) consists of generating job inter-arrival times (and their features) as random numbers that follow a desired probability density function. This is a typical method used in discrete event simulation. Finally, method (c) considers the mathematical expression that characterizes job inter-arrival time distributions. This method is used in deriving solutions for analytical models of computer systems.

This paper shows how to combine methods (a) and (b), typically used in simulation studies, with analytic system models. The advantages of this approach are:

- The system model can be solved through efficient analytic methods (e.g., Mean Value Analysis) instead of more complex and time-consuming simulations.

- The analytic models can be employed in situations for which they were not designed for (e.g., using a job trace as input). Note that the conventional approach for dealing with job traces as input to analytic models requires processing the trace in order to fit a known distribution for the job inter-arrival times. A difficulty may arise if this distribution does not meet the assumptions required by the analytic model.

The main contribution of this paper is an algorithm, which we call the *Epochs* algorithm, that estimates the execution times of jobs in a job stream. These jobs are executed by a computer system and contend for its resources. The algorithm was validated against experimental results using both jobs derived from a micro-benchmark and well-known benchmarks. The validation shows that the absolute relative error between measurements and predictions by the Epochs algorithm is below 10% most of the time and is at most 15%. These errors are considered to be acceptable for execution time prediction.

The rest of this paper is organized as follows. Section 2 describes the notation and formalizes the problem description. Section 3 presents the Epochs algorithm. The next section discusses validation results with a micro-benchmark and with three UNIX benchmark programs. Section 6 discusses some related work. Finally, section 7 presents some concluding remarks.

2 Notation and Problem Description

Consider a known stream $\mathcal{S} = \{(J_1, t_1), \cdots, (J_n, t_n), \cdots, (J_N, t_N)\}$ of jobs J_n that arrive at times $t_n, (n = 1, \cdots, N)$. Let $t_1 \leq t_2 \leq \cdots t_N$ without loss of generality. Each job J_n is characterized by a vector $\vec{D}_n = (D_{1,n}, \cdots, D_{K,n})$ of service demands at resources $1, \cdots, K$. The service demands of a given type of job are not deterministic but are average values obtained during multiple executions of each type of job. The differences in service demands for each job type in the stream may be attributed to differences in the execution path due to differences in the input data and/or due to a variability in the conditions in which measurements were taken.

The number of jobs in the stream is finite and the job arrival instants are assumed to be known as we first present the Epochs algorithm. We then relax this latter assumption. There are no steady state considerations. However, similarly to the steady state analysis, there is contention for the use of resources and this contention has to be considered in order to compute the execution times of the jobs in the stream.

The problem described here could be addressed by using trace-driven discrete simulation in which $\mathcal{S}$ is the input trace.

In this approach, arriving jobs join queues, receive service at the various simulated system resources, and leave after receiving all the required service at the system resources. The service demands at each resource are drawn from some distribution at each job arrival. The approach presented in this paper replaces the stochastic simulation of job behavior at the computer system by analytic models. However, job arrivals is still "trace-driven" or obtained through random number generation.

Figure 2 illustrates the concepts presented here. We divide time into time intervals of finite duration called *epochs*. The first epoch starts by definition at t_1, the time at which the first job(s) arrive. The end of an epoch is characterized by one of two events: (1) the arrival of a new job or jobs (if more than one job arrives at the same time) or (2) the completion of a job. The last epoch ends when the last job in execution ends. Therefore, each epoch has a constant workload mix, i.e., a set of jobs running concurrently.

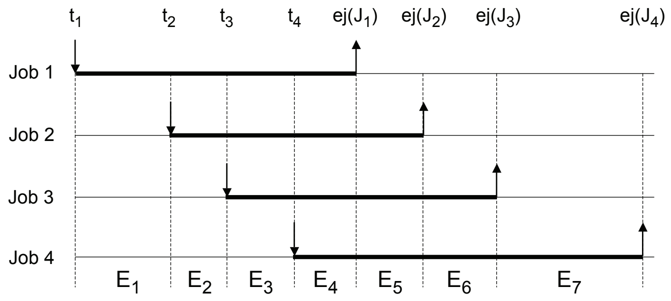

Figure 2: Concept of Epochs.

Let us denote the subsequent epochs as $E_1, \cdots, E_i, \cdots, E_M$ and let $e(E_i)$ be the end time of epoch E_i $(i = 1, \cdots, M)$ such that $e(E_1) < e(E_2) < \cdots < e(E_M)$. Let $e(E_0)$ be defined as t_1. The end time of epoch i is the start time of epoch $i+1$ for $i = 1, \cdots, M-1$. The time it takes to execute all jobs in the stream $\mathcal{S}$ is $e(E_M) - e(E_0)$.

With respect to Fig. 2, jobs 1-4, arrive at times t_1, t_2, t_3, and t_4, respectively, and end at times $ej(J_1), ej(J_2), ej(J3)$, and $ej(J_4)$, respectively. There are seven epochs: $E_1, \cdots, E_7$.

Let $\mathcal{W}_i = \{J^i_{i_1}, J^i_{i_2}, \cdots, J^i_{j_i}\}$ be the workload mix during epoch E_i. The jobs in $\mathcal{W}_i$ compete for the use of the K resources. Therefore, during each epoch, the jobs in that epoch's workload mix spend some time using the system resources (i.e., spending some of their service demands at these resources) and waiting to use these resources. The fraction of its service demand that a job is able to use during a given epoch is a function of the contention it finds from the other jobs in that workload mix.

The workload mixes at the various epochs of Fig. 2 are: $\mathcal{W}_1 = \{J_1\}; \mathcal{W}_2 = \{J_1, J_2\}; \mathcal{W}_3 = \{J_1, J_2, J_3\}; \mathcal{W}_4 = \{J_1, J_2, J_3, J_4\}; \mathcal{W}_5 = \{J_2, J_3, J_4\}; \mathcal{W}_6 = \{J_3, J_4\}$; and $\mathcal{W}_7 = \{J_4\}$. Thus, the concurrency levels at the various epochs of Fig. 2 are 1, 2, 3, 4, 3, 2, and 1, respectively.

The problem we want to solve is: Given a stream of jobs $\mathcal{S}$ and their corresponding service demands, find the execution time of each job in $\mathcal{S}$. The execution time of a job J_n is the difference between its end time $ej(J_n)$ and its start time $sj(J_n)$. Note that both $ej(J_n)$ and $sj(J_n)$ must coincide with an epoch transition time because a new epoch starts when a job arrives or completes. Clearly, the execution time of a job depends on how much service at each device the job can accomplish within each epoch. The accomplished service demand within an epoch depends on the contention at various resources caused by the jobs that are part of the workload mix of that epoch.

Some additional notation is in order:

- $d(E_i)$: duration of epoch i.
- $se(J_n)$: index of start epoch of job J_n. For example, $se(J_2) = 2$ in Fig.2. Note that $sj(J_n) = e(E_{se(J_n)-1})$.
- $ee(J_n)$: index of end epoch of job J_n. For example, $se(J_2) = 5$ in Fig.2. Note that $ej(J_n) = e(E_{ee(J_n)})$.
- $et(J_n)$: execution time of job J_n. $et(J_n) = ej(J_n) - sj(J_n)$.
- $D^i_{k,n}$: portion of service demand at resource k for job J_n accomplished during epoch E_i. Then,

$$D_{k,n} = \sum_{i=se(J_n)}^{ee(J_n)} D^i_{k,n} \tag{1}$$

- $R^i_{k,n}$: residual service demand at resource k for job J_n at the beginning of epoch E_i

3 The Epochs Algorithm

We now explain how we estimate $ej(J_n)$ using the *Epochs* algorithm described in this section. The process consists of estimating $D^i_{k,n}$ for every job J_n in the workload $\mathcal{W}_i$ for epoch E_i. This value can be obtained by solving a multi-class closed QN model for epoch E_i. The parameters for this model are as follows:

- R_i: number of classes for the model in epoch E_i. $R_i = |\mathcal{W}_i|$. Each class corresponds to exactly one job in $\mathcal{W}_i$.
- $\vec{N}_i = (N_1, \cdots, N_{R_i})$: population vector for the QN model for epoch E_i. By definition, this vector is equal to $(1, \cdots, 1)$ because each class corresponds to exactly one job in $\mathcal{W}_i$.
- $\vec{R}^i_n$: vector of residual service demands for job J_n at the beginning of epoch i. This vector can be computed as the difference between the original vector of service demands for that job and what the job has already accomplished in terms of its service demands from its starting epoch until the epoch preceding epoch E_i.

$$\vec{R}^i_n = \vec{D}_n - \sum_{v=1}^{i-1} (D^v_{1,n}, \cdots, D^v_{K,n}) \tag{2}$$

- $\mathbf{R}_i$: matrix of residual service demands for the model at epoch i. Each column corresponds to a job in $\mathcal{W}_i$ and each row corresponds to each of the K resources. The values in column n that correspond to job J_n come from the vector $\vec{R}_n^i$.

- $\mathcal{M}(R_i, \vec{N}_i, \mathbf{R}_i)$: closed QN model with parameters R_i, $\vec{N}_i$ and $\mathbf{R}_i$. The solution to this model returns the execution times $T_{i,r}$ of class r $(r = 1, \cdots, R_i)$ (i.e., the execution time of the job corresponding to class r).

The execution times $ej(J_n) - sj(J_n)$ for all jobs in $\mathcal{S}$ are computed through Algorithm 1, the Epochs algorithm. The inputs to the algorithm are the job stream, $\mathcal{S}$, and the vector of service demands $\vec{D}_n$ for each job J_n. Lines 5-6 initialize the epoch count i, the workload mix for epoch E_1, the value of the variable LastArrival to the earliest job arriving time (i.e., t_1), and the residual service demand values as equal to the service demands for all jobs in epoch 1.

Then, the algorithm loops (lines 7-50) while the set of jobs in the workload for epoch i is not empty. Line 13 invokes an AMVA solver (see e.g., [10]) during epoch i, a finite time interval, to compute the execution times of the jobs present at the beginning of that epoch. As indicated above, the MVA equations are valid for finite intervals if flow balance, one-step behavior, and homogenous service times are satisfied. According to Buzen and Denning, many real systems satisfy service homogeneity assumptions [4]. One step-behavior is satisfied by our definition of an epoch, which starts when either a new job arrives or a job leaves. Flow balance is also satisfied by definition because the workload mix is constant during any epoch, which implies that no jobs arrive or leave during an epoch.

In line 15, the algorithm computes the minimum execution time (MinEnd) of the jobs executing in the current epoch and, in line 16, the NextArr function is used to return the next job arrival time (NextArrival) by inspecting the job stream $\mathcal{S}$ after instant LastArrival.

The algorithm then determines if the current epoch ends due to a job completion (line 19) or due to a job arrival (line 33) by comparing the values of MinEnd and LastArrival. In the former case, the end time of the job(s) completing at the end of the epoch is computed (line 24) and the accomplished demand for all jobs in the workload mix during the current epoch is computed in line 26 as:

$$D_{k,n}^i = R_{k,n}^i \cdot \frac{e(E_i) - e(E_{i-1})}{T_{i,n}} \quad \forall\, k = 1, \cdots, K \;\; \forall J_n \in \mathcal{W}_i \tag{3}$$

Equation (3) says that the fraction of service demand accrued by a job during an epoch is proportional to the proportion of the total execution time of that job with respect to the epoch duration.

The workload for the next epoch is computed by removing the job(s) that completed from the workload of the current epoch (line 31). In the latter case, the end time of the current epoch is set to the arrival time of next job(s) to arrive in the job stream (line 34). As before, the accomplished demand for all jobs in the workload mix during the current epoch is computed (line 38) as in line 26 and the workload for the next epoch is computed by adding to the current workload the arriving job(s) (line 43). Lines 45-49 take care of the case in which there is an idle period in the job stream.

Algorithm 1 Epochs Algorithm: Compute Execution Times for a Job Stream

```
    Inputs: $\mathcal{S} = \{(J_1, t_1), \cdots, (J_n, t_n), \cdots, (J_N, t_N)\}$ and
    $\vec{D}_n = (D_{1,n}, \cdots, D_{K,n})\ \forall\ J_n$
    Output: $et(J_n)\ \forall J_n \in \mathcal{S}$
    /* Initialization */
5:  $i \leftarrow 1; \mathcal{W}_i = \{J_n \mid (J_n, t_1) \in \mathcal{S}\}$; LastArrival $\leftarrow t_1$
    $R_{k,n}^1 \leftarrow D_{k,n} \quad \forall\ k = 1, \cdots, K\ \forall J_n \in \mathcal{W}_i$
    while $\mathcal{W}_i \neq \emptyset$ do
        /* Build matrix of service demands for epoch $i$ */
        $\vec{D}_n^i \leftarrow \vec{D}_n - \sum_{v=1}^{i-1}(D_{1,n}^v, \cdots, D_{K,n}^v)\ \forall\ J_n \in \mathcal{W}_i$
10:     $\mathbf{D}_i \leftarrow$ BuildDemands($\vec{D}_n^i\ \forall\ J_n \in \mathcal{W}_i$)
        /* Solve the closed QN model for epoch $i$ */
        $R_i \leftarrow \mid \mathcal{W}_i \mid$
        $(T_{i,1}, \cdots, T_{i,R_i}) \leftarrow \mathcal{M}(R_i, \vec{N}_i, \mathbf{D}_i)$
        /* Determine end of epoch $i$ */
15:     MinEnd $\leftarrow \min_{r=1}^{R_i} T_{i,r}$ /* minimum execution time */
        NextArrival $\leftarrow$ NextArr (LastArrival, $\mathcal{S}$) /* next job arrival time */
        LastArrival $\leftarrow$ NextArrival
        if NextArrival $>$ MinEnd then
            /* epoch $i$ ends due to job completion */
20:         $e(E_i) \leftarrow e(E_{i-1}) +$ MinEnd
            /* Find set of completing jobs */
            EndingJobs $\leftarrow \{J_n \mid J_n \in \mathcal{W}_i\ \wedge T_{i,n} =$ MinEnd$\}$
            /* Compute end time of ending jobs */
            $et(J_n) \leftarrow e(E_i) - t_n\ \forall J_n \in$ EndingJobs
25:         /* Compute accomplished demand for all jobs in $\mathcal{W}_i$ */
            $D_{k,n}^i \leftarrow R_{k,n}^i \times [e(E_i) - e(E_{i-1})]/T_{i,n}$
                    $\forall\ k = 1, \cdots, K\ \forall J_n \in \mathcal{W}_i$
            /* Update residual service demands */
            $R_{k,n}^{i+1} \leftarrow R_{k,n}^i - D_{k,n}^i \quad \forall\ k = 1, \cdots, K\ \forall J_n \in \mathcal{W}_i$
30:         /* Adjust workload mix for next epoch */
            $\mathcal{W}_{i+1} \leftarrow \mathcal{W}_i -$ EndingJobs
        else
            /* Epoch $i$ ends due to new jobs arrivals */
            $e(E_i) \leftarrow$ NextArrival
35:         /* Find set of next jobs to arrive */
            ArrivingJobs $\leftarrow \{J_n \mid (J_n, t_n) \in \mathcal{S} \wedge t_n =$ NextArrival$\}$
            /* Compute accomplished demand for all jobs in $\mathcal{W}_i$ */
            $D_{k,n}^i \leftarrow R_{k,n}^i \times [e(E_i) - e(E_{i-1})]/T_{i,n}$
                    $\forall\ k = 1, \cdots, K\ \forall J_n \in \mathcal{W}_i$
40:         /* Update residual service demands */
            $R_{k,n}^{i+1} \leftarrow R_{k,n}^i - D_{k,n}^i \quad \forall\ k = 1, \cdots, K\ \forall J_n \in \mathcal{W}_i$
            /* Adjust workload mix for next epoch */
            $\mathcal{W}_{i+1} \leftarrow \mathcal{W}_i \bigcup$ ArrivingJobs
        end if
45:     $i \leftarrow i + 1$ /* increment epoch count */
        if $(\mathcal{W}_i = \emptyset) \wedge$ (NextArrival $\leq t_N$) then
            /* there is an inactive period before the next epoch */
            $\mathcal{W}_i \leftarrow \{J_n \mid (J_n,$ NextArrival$) \in \mathcal{S}\}$
        end if
50: end while
```

Table 1 illustrates the execution of the algorithm on a simple example wit two jobs J_1 and J_2 and three epochs. Job J_1 has starting service demands at the CPU and disk equal to 2 sec and 4 sec, respectively. The starting service demands at the CPU and disk for job J_2 are 3 sec and 5 sec, respectively. Row 2 of the table indicates the event that triggers the start of an epoch. Row 3 indicates the workload mix at each epoch.

Row 4 indicates the duration of each epoch. Row 5 shows the end time of each epoch. Rows 6 and 7 and 11 and 12 show the residual service demands at the CPU and disk for jobs J_1 and J_2, respectively, at the start of each epoch. Rows 6 and 12 illustrate the execution times of jobs J_1 and J_2, respectively, as if no other event were to start a new epoch and these jobs would continuously execute as indicated by the workload mix for that epoch. Lines 9 and 10 and 14 and 15 indicate the accomplished service demands for jobs J_1 and J_2, respectively at each epoch.

Epoch E_1 ends due to the arrival of job J_2 at time equal to 3 sec. Epoch 2 now starts and jobs J_1 and J_2 execute simultaneously during a period of time. During epoch E_1, job J_1 was able to accomplish half of its service demands at the CPU and disk. The residual service demands for that job at the beginning of epoch E_2 are 1 sec and 2 sec, respectively. The solution of the AMVA model for epoch E_2 indicates that the execution times for jobs J_1 and J_2 are 4.69 sec and 12.44 sec, respectively. Because no other job arrives, epoch E_2 ends when job J_1 ends at time 3 + 4.69 = 7.69 sec. The duration of epoch E_2 is then 4.69 sec. The residual service demands at the CPU and disk for job J_2 are 1.87 sec and 3.11 sec respectively at the start of epoch E_3. During that epoch, job J_2 runs by itself and takes 4.98 sec to complete. Thus, the execution time of job J_2 is 4.69 + 4.98 = 9.67 sec.

The Epochs algorithm was described as requiring that the entire job stream $\mathcal{S}$ be known in advance, as is the case of a trace in a trace-driven simulation. Minor modifications in the algorithm allow the job arrival process to be determined through stochastic generation of job types and job inter-arrival times from a given distribution as would be done in a typical discrete event simulation. The modifications are as follows:

- Remove the requirement that the job stream $\mathcal{S}$ be known as input, but continue to require that the job types be known and randomly generated and that their service demands be known.
- Replace line 16 of the Epochs algorithm by NextArrival $\leftarrow$ NextArr (LastArrival, IntArrivalTimeDistrib) where the function NextArr determines the next arrival instant by generating a job inter-arrival time from a given distribution IntArrivalTimeDistrib and adding the generated value to LastArrival.

4 Experimental Validation

As a first validation of the Epochs algorithm, we built a micro-benchmark program in C to provide us control and flexibility in designing jobs with varying characteristics (see pseudo-code in Algorithm 2). The program alternates between writing to a file and performing CPU operations (in this case, computing π using the Monte Carlo method). The main loop is repeated RepeatCount times and within each loop 50% of the time I/O is done and the other 50% a CPU-Intensive computation takes place.

Algorithm 2 Micro-benchmark Pseudo Code

```
Input: RepeatCount
/* Open a temp file in direct and truncate mode */
f ← OpenTempFile();
for $i = 0$ to RepeatCount do
    /* Compute random number between 0 and 1 */
    r ← GenerateRandomNumber(0,1);
    if r > .5 then
        /* write block of size 2048 bytes five times */
        performDiskIO;
    else
        /* CPU activity */
        r ← GenerateRandomNumber (0, RepeatCount/100)
        for all $i = 1$ to $r$ do
            Calculate $\pi$ using Monte Carlo with iteration count equal to RepeatCount
        end for
    end if
end for
CloseFile (f);
printTimingInfo;
```

The micro benchmark was parameterized to generate four different types of jobs. Each job was run in isolation 100 times and average CPU and I/O service demands were computed.We used two hardware configurations to run the experiments with the benchmark:

1. *Virtual Machine*: A RHEL 2.6+ kernel based CentOS VM running on one core of a i7-3740QM processor running at 2.7GHz. This VM has 4GB of memory.
2. *Physical Machine*: A machine running CentOS Linux 2.6+ kernel. This machine has a 32-core Xeon(R) CPU E5-2665 running at 2.4 GHz and is organized as 2 NUMA nodes with a total of 132 GB memory. Only one of the 32 cores was used to run the benchmark.

The service demands obtained by running the four types of jobs at the VM and physical machine environments are shown in Tables 2 and 3, respectively.

Table 2: Service Demands for Jobs on the Virtual Machine

Job Id →	Job1	Job2	Job3	Job4
CPU	53.4	18.7	6.3	26.96
Disk	6.6	4.5	3.4	5.14
RepeatCount	10000	7000	5000	8000

Table 3: Service Demands for Jobs on the Physical Machine

Job Id →	Job1	Job2	Job3	Job4
CPU	60.38	28.5	4.87	11.05
Disk	9.02	8.3	3.24	5.45
RepeatCount	9000	7000	4000	5000

Table 1: Example of the operation of the Epochs Algorithm

	Epoch 1	Epoch 2	Epoch 3
Start Event	Arr J1	Arr J2	End J1
$\mathcal{W}_i$	J1	J1, J2	J2
$d(E_i)$	3	4.69	4.98
$e(E_i)$	3	3+ 4.69 = 7.69	7.69 + 4.98 = 12.67
$R^i_{\text{cpu},1}$	2	$2 - 1 = 1$	-
$R^i_{\text{disk},1}$	4	$4 - 2 = 2$	-
$T_{i,1}$	6	4.69	-
$D^i_{\text{cpu},1}$	$2 \times 3/6 = 1$	$1 \times 4.69/4.69 = 1$	-
$D^i_{\text{disk},1}$	$4 \times 3/6 = 2$	$2 \times 4.69/4.69 = 2$	-
$R^i_{\text{cpu},2}$	-	3	3 - 1.13 = 1.87
$R^i_{\text{disk},2}$	-	5	5 - 1.89 = 3.11
$T_{i,2}$	-	12.44	4.98
$D^i_{\text{cpu},2}$	-	$3 \times 4.69/12.44 = 1.13$	$1.87 \times 4.98/4.98 = 1.87$
$D^i_{\text{disk},2}$	-	$5 \times 4.69/12.44 = 1.89$	$3.11 \times 4.98/4.98 = 3.11$

The measured execution times reported in the tables that follow represent averages over 10 runs for the virtual machine configuration and over 15 runs for the physical machine one. The tables also report 95% confidence intervals for these averages.

We ran two scenarios on the physical machine configuration using our micro-benchmark and the four jobs derived from it. The epoch data for the first scenario, which has seven epochs, is shown in Table 4. The table also shows the start time and end time of each epoch, their duration, the event that triggered the start of the epoch, and the workload mix in each epoch. The arrival times of jobs J_1-J_4 were predetermined. The other values in the table were determined through measurements obtained by the execution of the four jobs. The values of Table 4 represent a single run of Scenario 1 for illustration purposes.

Table 5 shows the average measured execution times for the four jobs for scenario 1 on the physical machine and their corresponding 95% confidence intervals. Column 3 shows the predicted execution times computed with the Epochs algorithm. The last column of the table shows the percent relative error, whose absolute value varies between 1.7% and 8.4%.

Table 5: Execution times for the physical machine - Micro-benchmark - Scenario 1

Job Num	Measured Execution Time	Predicted Execution Time	% Relative Error
Job 1	116.9 ± 2.3	112.6	3.7
Job 2	69.1 ± 0.8	68.4	1.0
Job 3	24.9 ± 0.07	27.0	-8.4
Job 4	17.2 ± 0.8	17.5	-1.7

The epoch data for the second scenario on the physical machine is shown in Table 6. One of the differences between this scenario and the previous is that two instances of each of the four job types were used. The second instance is indicated in the table with a “-2” suffix (e.g., $J1$-2). This execution leads to 16 epochs and a larger concurrency level than the previous scenario. For example, there are eight concurrent jobs in execution in epoch 8.

Similarly to Table 5, Table 7 shows experimental and predicted execution times for each of the jobs. As it can be seen, most of the absolute values of the percent relative error are below 3.4%. Only one value has an error of 15%.

Table 7: Execution times for the physical machine - Micro-benchmark - Scenario 2

Job Number	Measured Exec. Time	Predicted Exec. Time	% Relative Error
Job 1	207.2 ± 2.24	204.4	1.4
Job 1-2	198.7 ± 1.03	195.7	1.5
Job 2	142.2 ± 2.60	137.4	3.4
Job 2-2	142.0 ± 2.37	138.3	2.6
Job 3	24.9 ± 0.44	25.5	-2.4
Job 3-2	43.3 ± 0.75	36.8	15
Job 4	71.4 ± 0.43	71.6	-0.3
Job 4-2	68.8 ± 0.39	70.8	-2.9

Table 8 shows epoch data for a virtual machine scenario with jobs from the micro-benchmark. There are seven epochs in this scenario. The measured execution times and execution times predicted by the Epochs algorithm for the data in Table 8 are shown in Table 9. As the table indicates, the absolute relative error varies from 4 to 11.6%.

We then validated the Epochs algorithm using jobs from real Unix benchmarks. In particular, we selected three jobs from the benchmark: Nbench [18], Bonnie++ [16], and Dbench [17]. Nbench is a synthetic computing benchmark program intended to measure a computer’s CPU, FPU, and

Table 4: Epoch data for the physical machine - Micro-benchmark - Scenario1

Epoch No.	Start Time	End Time	Duration	Start Event	Workload Mix
1	0	5	5	Arr J1	J1
2	5	10	5	Arr J2	J1,J2
3	10	15	5	Arr J3	J1,J2,J3
4	15	32.5	17.5	End J3	J1,J2
5	32.5	77.3	44.8	End J2, Arr J4	J1,J4
6	77.3	104.3	27.0	End J4	J1
7	104.3	118.0	13.7	End J1	J1

Table 6: Epoch data for the physical machine - Micro-benchmark - Scenario 2

Epoch No.	Start Time	End Time	Duration	Start Event	Workload Mix
1	0	5	5	Arr J1	J1
2	5	10	5	Arr J2	J1, J2
3	10	15	5	Arr J3	J1, J2, J3
4	15	20	5	Arr J4	J1, J2, J3, J4
5	20	25	5	Arr J1	J1, J2, J3, J4, J1-2
6	25	30	5	Arr J2	J1, J2, J3, J4, J1-2, J2-2
7	30	35	5	Arr J3	J1, J2, J3, J4, J1-2, J2-2, J3-2
8	35	40	5	Arr J4	J1, J2, J3, J4, J1-2, J2-2, J3-2, J4-2
9	40	40.5	.5	End J3	J1, J2, J4, J1-2, J2-2, J3-2, J4-2
10	40.5	71.8	31.3	End J3-2	J1, J2, J4, J1-2, J2-2, J4-2
11	71.8	91.6	19.8	End J4	J1, J2, J1-2, J2-2, J4-2, J2-2, J4-2
12	91.6	110.8	19.2	End J4-2	J1, J2, J1-2, J2-2
13	110.8	147.3	36.5	End J2	J1, J1-2, J2-2
14	147.3	168.2	20.9	End J2-2	J1, J1-2
15	168.2	209.3	41.1	End J1	J1-2
16	209.3	220.6	11.3	End J1-2	J1-2

Table 8: Epoch data for the virtual machine environment - Micro-benchmark Scenario

Epoch No.	Start Time	End Time	Duration	Start Event	Workload Mix
1	0	5	5	Arr J1	J1
2	5	10	5	Arr J2	J1,J2
3	10	15	5	Arr J3	J1,J2,J3
4	15	36.9	21.9	End J3	J1,J2
5	36.9	57.2	20.3	End J2,Arr J4	J1,J4
6	57.2	114.8	57.6	End J4	J1
7	114.8	115.5	0.7	End J1	J1

Table 9: Execution times for the virtual machine environment - Micro-benchmark Scenario

Job Number	Measured Execution Time	Predicted Execution Time	% Relative Error Error
Job 1	114.8 ± 0.6	110.2	4.0
Job 2	61.8 ± 0.88	57.6	6.6
Job 3	50.6 ± 0.52	47.3	6.5
Job 4	24.8 ± 1.02	21.9	11.6

Table 10: Service Demands for UNIX benchmark jobs on the Virtual Machine Environment

Job Names →	Nbench (J1)	Bonnie++ (J2)	Dbench (J3)
CPU	25.0	8.2	5.5
Disk	0.0	9.8	4.5

memory system speeds, and includes various types of sorts, bit manipulation, compression and encryption algorithms, LU decomposition, floating-point emulation, neural network, and a task allocation algorithm. Bonnie++ is a benchmark suite aimed at performing a number of simple tests of hard drive and file system performance. Dbench is a tool to generate I/O workloads to either a filesystem or to a networked CIFS or NFS server.

We ran each of these three benchmark programs in isolation a large number of times and computed the average service demands at the CPU and disk on the virtual machine configuration. The results are shown in Table 10, which also indicates that these three jobs will be referred heretofore as J1, J2, and J3.

Table 11 shows epoch data for an experiment using the three UNIX benchmark jobs described above. There are 12 epochs in this scenario and two instances of jobs J1, J2, and J3 arriving at difference time instants. During epoch 6, both instances of the three jobs are running concurrently.

Table 12 shows the average execution times and their 95% confidence intervals as well as the job execution times predicted by the Epochs algorithm. As it can be seen, the absolute relative error varied from 2.7% to 11.3%.

Table 12: Execution times for Unix Benchmark Jobs in the Virtual Machine Environment

Job Number	Measured Exec. Time	Predicted Exec. Time	% Relative Error
Job 1	67.6 ± 1.08	69.4	-2.7
Job 1-2	67.8 ± 1.24	64.6	4.7
Job 2	51.3 ± 2.04	45.5	11.3
Job 2-2	53.0 ± 1.43	47.3	10.75
Job 3	27.8 ± 0.83	29.9	-7.5
Job 3-2	30.3 ± 1.2	32.2	-6.2

5 Applications of the Epochs Algorithm to Scheduling

One of the applications of the Epochs algorithm is on the performance evaluation of server clusters that receive a stream of jobs that are scheduled according to some scheduling policy. Figure 3 shows several servers each receiving a sub-stream of the global job stream. The stream of jobs received by each server is determined by a scheduler.

Modeling the scheduler using analytic models is very difficult in general. There is a vast body of literature on analytic modeling of schedulers for single queues or for server clusters in which servers are modeled as single queues. See [7] for a good survey on the topic. However, one can either simulate the scheduler or implement the scheduler in a way that generates the local sub-streams for each server. Then, the Epochs algorithm can be applied to each server as was done in [1]. For example, Fig. 4 shows the makespan (i.e., the time to complete all the jobs in a job stream) for 10 different job streams randomly generated from jobs chosen from Nbench, Bonnie++, and Dbench, for an average arrival rate of 0.167 jobs/sec.

The four scheduling policies used in Fig. 4 are:

- Round Robin (RR): The scheduler chooses the servers in the cluster in a round robin fashion. This scheduling scheme is oblivious to the utilization of any server resource (either CPU or disk).
- Least Response Time (LRT): This scheduling algorithm finds the machine on which the incoming job is predicted to have the least response time. Since the scheduler has the exact states of all the jobs running on all the servers, it can estimate the response time of an incoming job if it were added to any node in the cluster.
- Least Maximum Utilization First (LMUF): The server with the minimum utilization for the resource with the

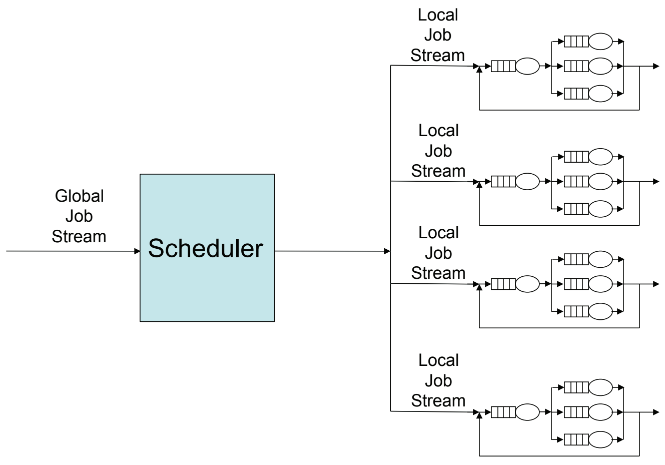

Figure 3: Use of the Epochs Algorithm for Assessing Schedulers in Server Clusters.

Table 11: Epoch data for Unix Benchmark Jobs in the Virtual Machine Environment

Epoch No.	Start Time	End Time	Duration	Start Event	Workload Mix
1	0	5	5	Arr J1	J1
2	5	10	5	Arr J2	J1, J2
3	10	15	5	Arr J3	J1, J2, J3
4	15	20	5	Arr J1-2	J1, J2, J3, J1-2
5	20	25	5	Arr J2-2	J1, J2, J3, J1-2, J2-2
6	25	39.97	14.97	Arr J3-2	J1, J2, J3, J1-2, J2-2, J3-2
7	39.97	50.47	10.5	End J3	J1, J2, J1-2, J2-2, J3-2
8	50.47	57.23	6.76	End J2	J1, J1-2, J2-2, J3-2
9	57.23	67.33	10.1	End J3-2	J1, J1-2, J2-2
10	67.33	69.38	2.05	End J2-2	J1, J1-2
11	69.38	79.59	10.21	End J1	J1-2
12	79.59	79.59	0	End J1-2	None

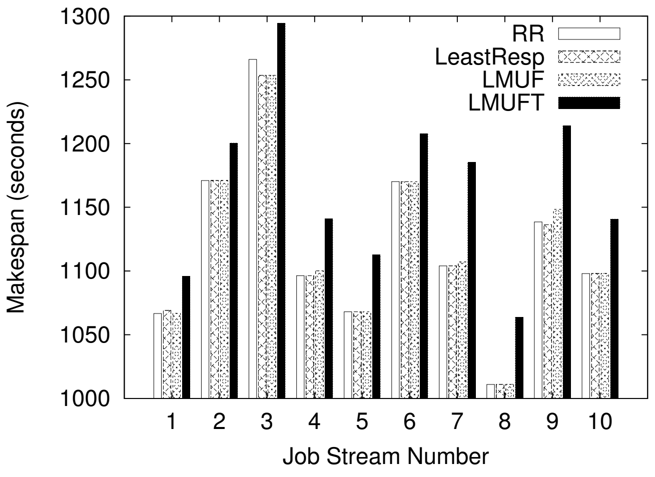

Figure 4: Makespan vs. jobs stream for various schedulers.

highest utilization is the one that receives an incoming job. The utilization of the resources at each server is calculated as a snapshot at the time the new job arrives to be scheduled.

- Least Maximum Utilization First-Threshold (LMUF-T): Similar to LMUF except that a job is not sent to a server if the utilization for the resource with the highest utilization exceeds a certain threshold. In that case, the job is queued at the scheduler. When a job completes at any machine, the scheduler attempts to send the queued job again to one of the servers. The goal of LMUF-T is to bound the contention at each node. However, jobs will wait at the scheduler. It may be more advantageous to wait a bit at the scheduler and then be assigned to a less loaded machine.

Each of the four schedulers was implemented taking as input a global job stream that arrives at a server cluster. Then, a scheduler uses its scheduling policy to decide where the arriving jobs should be sent, generating the job streams for each server.

6 Related Work

Combining different modeling techniques to evaluate the performance of a computer system has been done by many before with the goal of obtaining the benefits of more than one technique. For example, Mehdipour et al. have combined simulation and analytic models for processor design [9]. Norton has introduced the Simalytic technique, which combines simulation methods with analytic models [12]. Menascé has shown how to combine Generalized Stochastic Petri Nets (GSPN) and Queuing Networks (QN) to reduce the size of the state space of GSPNs [11].

The idea of hybrid simulation/analytic models is not new. In fact, an interesting taxonomy on the types of hybrid models is presented in [15]. In that paper the authors present some examples of hybrid models. All examples except for one are for single-queue systems. The exception is for a computer system with admission control due to a limited multiprogramming degree. The authors in [15] suggest simulating the computer system for different value of the degree of multiprogramming n, and modeling the entire system as a load dependent single queue in which the service rate is a function of n.

The current paper differs from previous work in the sense that it takes inputs that are typical in simulation modeling and uses them as inputs for analytic QN models.

7 Concluding Remarks

The traditional approach for specifying the workload in analytic queuing network models uses the following methods for specifying the workload: (a) average job arrival rates in the case of open job classes and (b) job population in the case of closed classes [2, 10, 13]. Besides these two types of workload intensity parameters, the service demands for each job class have to be specified. This paper presented a novel technique for driving analytic queuing network models with job traces, which specify job arrival instants within the trace and the types of the arriving jobs. Service demands are associated with job types.

The method presented here, called the Epochs algorithm, is based on scanning a job trace and determining finite-duration time intervals called epochs in which a certain workload mix is active. The duration of each epoch is estimated using the Mean Value Analysis equations in each epoch in order to determine when the current epoch terminates due to the completion of a job. The basis of the approach used in the Epochs algorithm lies in the fact that the MVA equations are valid for finite duration intervals if certain operational assumptions such as flow balance, one-step behavior, and homogenous service times are satisfied [4]. The advantage of the method presented in this paper over traditional methods for specifying the workload in analytic models is that any distribution-independent job trace can be used as input as long as the operational assumptions are met. These assumptions are much more general and easy to verify than the stochastic assumptions used in traditional analytic models.

The job execution time predictions were validated experimentally using a micro-benchmark developed by the authors and with real programs from well-known Unix benchmarks. The results indicated that the relative absolute error stays below 10% in most cases and is at most 15% for the cases examined.

References

[1] S. Bardhan and D.A. Menascé, *Trace-Driven Analytic Modeling for Scheduler Assessment*, Technical Report GMU-CS-TR-2014-02, Computer Science Department, George Mason University, March 2014, available at http://cs.gmu.edu.

[2] F. Baskett, K.M. Chandy, R.R. Muntz, F. Palacios-Gomez, *Open, closed and mixed networks of queues with different classes of customers*, J. ACM 22 (2), 1975, pp. 248260.

[3] J.P. Buzen, *Fundamental Operational Laws of Computer System Performance*, Acta Informatica, Vol. 7, No. 2, 1976, pp. 167-182.

[4] J.P. Buzen and P.J. Denning, *Operational Treatment of Queue Distributions and Mean-Value Analysis*, Computer Performance, IPC Press, Vol. 1, No. 1, June 1980, pp. 6-15.

[5] J.P. Buzen and P.J. Denning, *Measuring and Calculating Queue Length Distributions*, IEEE Computer, April 1980, pp. 33-44.

[6] P.J. Denning and J.P. Buzen, *The Operational Analysis of Queuing Network Models*, ACM Comp. Surveys, Vol. 10, No. 3, September 1978, pp. 225-261.

[7] Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queuing Theory in Action*, Cambridge University Press, 2013.

[8] L. Kleinrock, Queueing Systems. Volume 1: Theory, John Wiley & Sons, 1975.

[9] F. Mehdipour, H. Noori, B. Javadi, H. Honda, K. Inoue, K. Murakami, and J. Kazuaki, *A combined analytical and simulation-based model for performance evaluation of a reconfigurable instruction set processor*, Proc. 2009 Asia and South Pacific Design Automation Conference, January 19-22, 2009.

[10] D.A. Menascé, V.A.F. Almeida, and L.W. Dowdy, *Performance by Design: Computer Capacity Planning by Example*, Prentice Hall, Upper Saddle River, 2004.

[11] D.A. Menascé, *A Methodology for Combining GSPNs and QNs*, 2011 International Computer Measurement Group Conference, Washington, D.C., December 5-9, 2011.

[12] T.R. Norton, *The Simalytic Modeling Technique*, in Performance Engineering, State of the Art and Current Trends, eds. Reiner Dumke, Claus Rautenstrauch, and Andre Scholz, 2001, Lecture Notes in Computer Science, Vol. 2047, 2001, Springer-Verlag Berlin/Heidelberg, pp. 222–238.

[13] M. Reiser and S. Lavenberg, *Mean-Value Analysis of Closed Multichain Queuing Networks*, J. ACM 27 (2), 1980.

[14] F.L. Severance, *System Modeling and Simulation: An Introduction*, John Wiley & Sons, Inc., 2001.

[15] J.G. Shanthikumar and R.G. Sargent, *A Unifying View of Hybrid Simulation/Analytic Models and Modeling*, Operations Research, Vol. 31, No. 6, 1983, INFORMS, pp. 1030–1052.

[16] http://www.coker.com.au/bonnie++/

[17] http://dbench.samba.org/

[18] http://www.tux.org/ mayer/linux/bmark.html