2009

Dependence-driven techniques in system design

Ningfang Mi
College of William & Mary - Arts & Sciences

Dependence-driven Techniques In System Design

Ningfang Mi

Nanjing, Jiangsu, China

Bachelor of Science, Nanjing University, 2000
Master of Science, University of Texas at Dallas, 2004

A Dissertation presented to the Graduate Faculty
of the College of William and Mary in Candidacy for the Degree of
Doctor of Philocophy

Department of Computer Science

The College of William and Mary
August, 2009

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Ningfang Mi

Approved by the Committee, June, 2009

Committee Chair
Dr. Evgenia Smirni
The College of William and Mary

Dr. Phil Kearns
The College of William and Mary

Dr. Peter Kemper
The College of William and Mary

Dr. Weizhen Mao
The College of William and Mary

Dr. Ludmila Cherkasova
Hewlett-Packard Labs

ABSTRACT PAGE

Burstiness in workloads is often found in multi-tier architectures, storage systems, and communication networks. This feature is extremely important in system design because it can significantly degrade system performance and availability. This dissertation focuses on how to use knowledge of burstiness to develop new techniques and tools for performance prediction, scheduling, and resource allocation under bursty workload conditions.

For multi-tier enterprise systems, burstiness in the service times is catastrophic for performance. Via detailed experimentation, we identify the cause of performance degradation on the persistent bottleneck switch among various servers. This results in an unstable behavior that cannot be captured by existing capacity planning models. In this dissertation, beyond identifying the cause and effects of bottleneck switch in multi-tier systems, we also propose modifications to the classic TPC-W benchmark to emulate bursty arrivals in multi-tier systems.

This dissertation also demonstrates how burstiness can be used to improve system performance. Two dependence-driven scheduling policies, SWAP and ALoC, are developed. These general scheduling policies counteract burstiness in workloads and maintain high availability by delaying selected requests that contribute to burstiness. Extensive experiments show that both SWAP and ALoC achieve good estimates of service times based on the knowledge of burstiness in the service process. As a result, SWAP successfully approximates the shortest job first (SJF) scheduling without requiring a priori information of job service times. ALoC adaptively controls system load by infinitely delaying only a small fraction of the incoming requests.

The knowledge of burstiness can also be used to forecast the length of idle intervals in storage systems. In practice, background activities are scheduled during system idle times. The scheduling of background jobs is crucial in terms of the performance degradation of foreground jobs and the utilization of idle times. In this dissertation, new background scheduling schemes are designed to determine when and for how long idle times can be used for serving background jobs, without violating predefined performance targets of foreground jobs. Extensive trace-driven simulation results illustrate that the proposed schemes are effective and robust in a wide range of system conditions. Furthermore, if there is burstiness within idle times, then maintenance features like disk scrubbing and intra-disk data redundancy can be successfully scheduled as background activities during idle times.

Table of Contents

To my parents, my husband Bo, and my son Anray.

ACKNOWLEDGMENTS

This dissertation is the result of the support of many people over the years. First and foremost, I would like to thank my thesis advisor Dr. Evgenia Smirni. She has provided continuous assistance in numerous ways whenever I needed it. Her advice and guidance at crucial points have been invaluable in helping me grow and mature as a researcher. I feel fortunate to have her as my advisor, and I hope to one day be as good an advisor as she is.

I would also like to thank my committee members, Dr. Phil Kearns, Dr. Peter Kemper, Dr. Weizhen Mao, and Dr. Ludmila Cherkasova. Their thoughtful feedback have been extremely helpful in improving this thesis.

I am especially grateful to Dr. Alma Riska, my mentor at Seagate Research, and Dr. Ludmila Cherkasova, my mentor at HP Labs. I could not have asked for better mentors to show me the ropes in the "real world". Their patient advice as well as their rich experience and knowledge have made my internships truly productive and memorable.

My research has benefited from collaborating with other talented researchers. I would like to thank Dr. Qi Zhang for her help and encouragement when I first started my research. My smooth transition to research would not have been possible without Qi's help. I would also like to say a special thank you to Dr. Giuliano Casale for his help in the later part of my research. During his 2-year stay at William and Mary, we had many productive discussions and fruitful collaborations on several topics. Working closely with Giuliano was a rewarding experience. He has been a role model for me with his wide breadth of knowledge, diligence, and work ethic.

In addition, I would like to thank the academic staff in the Computer Science Department, especially Vanessa Godwin and Jacqulyn Johnson, for their assistance over the years. My PhD journey would not have been this smooth without their help.

Finally, I would like to express my deepest gratitude to my family for their unconditional love and support all these years. I am indebted to my parents, my grandparents, and my sister for encouraging me and being patient for so many years. Special thanks to my husband Bo Sheng who has helped and supported me through this journey. I would like to dedicate this thesis to my lovely son Anray Sheng, who has been the source of my energy in the final stage of writing this thesis.

List of Tables

List of Figures

Dependence-driven Techniques In System Design

Chapter 1

Introduction

Burstiness has been widely observed in different levels of real systems, such as the service demands in e-commerce Web servers [64], the arrivals at storage systems [78], and the arrivals in grid services [51]. Under bursty conditions, it is more likely that large service times progressively aggregate in bursts, resulting in the reduction of system throughput for a period. Similarly, burstiness in the arrivals may cause more persistent flash crowds in the system, where periods of continuous peak arrivals significantly deviate from the normal traffic intensity. As a result, burstiness in both the arrival and the service processes significantly reduces system performance and availability.

In networking, a lot of studies have been done to counteract the performance effect of burstiness and to maintain high service availability, e.g., the development of accurate models of bursty traffic processes [39, 100], and measurement-based admission control schemes for network availability under rapidly changing flows [37]. Unfortunately, these models and schemes cannot be easily applied to systems due to the systematic violation of the underlying assumptions. In this dissertation, we focus on identifying the characteristics

of workload burstiness in *systems* and on understanding their performance implications. Such an understanding is critical for developing new techniques and tools to maintain performance and availability in systems with bursty workloads.

One particular target area is multi-tier enterprise systems which have risen in popularity during the past decade. It is critical and difficult to ensure performance and availability of such enterprise systems, especially when their complexity increases. Sizing enterprise systems to meet the needs of future workloads is a very challenging task. To address this problem, practitioners use capacity planning (i.e., predict future system performance using mathematical models) in order to decide on software and/or hardware. However, if workload flows in enterprise systems are bursty, then no capacity planning methods exist that can capture the effects of burstiness in system performance. Thus, there is a clear need for new capacity planning models and methods. Scheduling, another important component in system design, can also use burstiness to improve system performance. Deriving information on the future workload from its burstiness profile can be used to design more efficient scheduling techniques.

An additional area of interest is to maintain data availability and reliability at disk drives. Nowadays, the needs for hard disk drives are not only from traditional computer systems but also from a wide range of consumer electronic devices. As digital storage of commercial and personal data becomes mainstream, high data availability and reliability become imminently critical. As a result, maintenance tasks are developed to gain reliable disk-based storage systems, such as disk scrubbing to detect sector errors via background media scans [84] and intra-disk data redundancy to recover from sector errors [23, 45]. Since most computer systems operate 24 hours a day, 7 days a week, they must complete

these maintenance tasks while in operation. This additional work, although labeled as low priority, still affects the performance of foreground tasks, especially if it is non-preemptive. Therefore, developing efficient scheduling of maintenance tasks becomes an important issue in storage systems. Additionally, as disk drives operate under a wide range of applications exhibiting high variability and strong burstiness [78], reliable storage devices must be driven by policies that incorporate burstiness.

In general, burstiness in workloads processed by multi-tier architectures, storage systems, and communication networks, significantly reduces system performance, thus it is necessary to consider burstiness in performance models and system design. Capturing burstiness within performance models can be used to implicitly model caching, context switching overhead, and contention for memory or locks, while keeping the model surprisingly simple. Additionally, burstiness, as a form of temporal dependence, provides information on the upcoming workload arrivals and service demands. Therefore, by taking advantage of burstiness, one may effectively predict the immediate future, e.g., estimate service times in multi-tier systems and forecast idle interval lengths in storage systems. This dissertation focuses on how to use burstiness to develop new techniques and tools for improving performance prediction, scheduling, and resource allocation in enterprise systems and storage systems.

1.1 Contributions

The contributions of this dissertation are summarized as follows.

- effective *capacity planning models* that capture workload burstiness are developed

for multi-tier enterprise systems. Model parameterization is achieved via coarse measurements in real systems, see Section 1.1.1;

- new *general scheduling policies* are designed for systems with bursty workloads to estimate service times of upcoming requests and improve system performance and availability, see Section 1.1.2;

- a model for evaluating the performance of *foreground/background jobs at the disk drive level* is proposed and new background scheduling schemes that efficiently manage the idle times in storage systems are developed, see Section 1.1.3.

1.1.1 Capacity Planning Models

Capacity planning is a critical area in IT management and aims at quality of service support and decision making. Building effective models of complex enterprise systems is a priority for capacity planning and resource provisioning. In this dissertation, we build simple capacity planning models that can predict the performance of systems with bursty workloads.

- We observe that the bursty workloads often result in dramatic degradation of the perceived user performance in an e-commerce system that is built according to the TPC-W benchmark. We show that existing capacity planning models, e.g., Mean Value Analysis (MVA) models, cannot capture burstiness in the service process, and thus yield large inaccuracies in performance prediction if the system operates under bursty conditions.

- We propose to use the *index of dispersion* to capture burstiness. The index of dispersion can jointly capture service *variability* and *burstiness* in a single number but without identifying the low-level *exact* cause of burstiness as traditional models would require. We first find a simple and practical approach to measure the index of dispersion of the service process at a server, which is inferred by observing the number of completions within the concatenated busy periods of that server. Then, we integrate the index of dispersion into performance models by using it together with other measurements (i.e., the "estimated" mean and the 95th percentile of service times) to derive a Markov-modulated process. We show that these parameterized models accurately predict the system performance, despite inevitable inaccuracies that result from inexact and limited measurements.

- We develop a simple and powerful approach to incorporate burstiness into benchmarking of client-server systems. Benchmarking is a critical step for capacity planning and resource provisioning. An effective benchmark should take account of the system behavior under bursty conditions. However, the traditional client-server benchmarks, e.g., the standard TPC-W benchmark, do not provide any mechanisms for injecting burstiness into the workload. In this dissertation, we rectify this deficiency of TPC-W by generating workloads that emulate the traffic bursty phenomenon in a controllable way, and thus provide a mechanism that enables testing and evaluation of client-server system performance under reproducible bursty workloads. This new approach injects different amounts of burstiness into the arrival stream using the index of dispersion, a single parameter that can be used as a turnable knob.

1.1.2 General Scheduling Policies

Recent work in Web systems, such as Internet servers [97], multi-tier architectures [18], and online Data Stream Management Systems (DSMSs) [28], has drawn attention to the problem of defining effective scheduling techniques to keep a system responsive under a variety of overload conditions. In this dissertation, we show how to counteract burstiness and maintain high availability by delaying selected requests that contribute to temporal locality.

- We show that by selectively delaying requests that contribute to burstiness, delay-based scheduling can achieve significant performance gains and high system availability. We observe that delaying *selected* requests significantly improves system throughput across the network, which allows delay-based scheduling to increase the amount of requests that a server can process at a given time, therefore avoids harmful congestion conditions.

- We design a new delay-based scheduling policy, called SWAP. This policy classifies (i.e., "predicts") requests as short or long based on the knowledge of burstiness in service times and approximates the behavior of the shortest job first (SJF) scheduling by delaying the predicted long requests. We show that SWAP significantly improves system performance and availability, where the system capacity under SWAP is largely increased compared to the first-come first-served (FCFS) scheduling and is highly-competitive with SJF, but without requiring a priori information of job service times.

- We also define two scheduling policies, call ALoC and D_ALoC, which extend

SWAP by *infinitely* delaying the selected requests in order to meet pre-defined quality-of-service levels. Both policies are easy to implement and rely on minimal assumptions. In particular, D_ALoC is a fully no-knowledge measurement-based policy that self-adjusts its scheduling parameters based on policy targets and statistical information of requests served in the past. We show that if these policies are employed in the server with a bursty service process, then the overall system performance (e.g., end-to-end response time across all servers) can be significantly improved by only infinitely delaying a small fraction of the incoming requests.

1.1.3 Idleness Management in Storage Systems

An additional area of focus is to use the knowledge of burstiness in storage systems for efficiently scheduling maintenance tasks at disk drive levels and thus improving system reliability, availability and performance. These maintenance tasks are considered as additional work and scheduled during idle intervals. However, they still affect the performance of foreground tasks, especially if they are non-preemptive. In this dissertation, we develop algorithms to schedule background jobs in storage systems that can work effectively under both bursty and non-bursty conditions.

- We propose an analytic model to evaluate the performance trade-offs of the amount of maintenance (or background) work that a storage system can sustain. The proposed model results in a quasi-birth-death (QBD) process that is analytically tractable. We show that under bursty arrivals both foreground and background performance strongly depends on system load. In contrast, if arrivals of foreground jobs are not bursty, performance sensitivity to load is reduced. The model identifies

burstiness in the arrivals of foreground jobs as an important characteristic that controls the decision of how much background load the system can accept to maintain high availability and performance gains.

- A common approach in system design is to be non-work-conserving by "idle waiting", i.e., delay scheduling of a background job to avoid slowing down upcoming foreground tasks. We show that "idle waiting" is insufficient as a "standalone" technique for idleness management, because it may cause background starvation while trying to meet the foreground performance targets. We propose to complement "idle waiting" with the "estimation" of background work to be served in every idle interval. This new scheduling scheme can well balance the trade-offs between the performance of foreground and background tasks. If burstiness exists in idle intervals, then this information can be used to predict the length of the upcoming idle intervals. Predicting that the next idle interval is long given that the current interval is also long is of particular interest, because scheduling of background jobs can become more aggressive but without affecting more on foreground performance.

- We show that two known techniques to detect and/or recover from latent sector errors, namely scrubbing and intra-disk data redundancy, can be treated as background activities and be effectively served by the proposed background scheduling scheme without affecting foreground task performance, while reducing the window of vulnerability for data loss.

- We develop a new algorithmic framework to determine the schedulability of non-preemptable background tasks, i.e., estimating *when* and *for how long* idle times

can be used to serve background work without violating pre-defined foreground performance targets. The estimation is based on monitored system information that includes the histogram of idle times. This histogram captures accurately important statistical characteristics of the complex demands of the foreground activity.

1.2 Organization

The dissertation is organized as follows. In Chapter 2, we present an overview of basic concepts and terminology that are used in this dissertation. In Chapter 3, we present new capacity planning models for effectively capturing burstiness in workloads and for explicitly introducing burstiness in a client-server benchmark. In Chapter 4, we present new general scheduling policies for improving the performance and availability in systems with bursty workloads. In Chapter 5, we propose an analytic model for the evaluation of disk drives or storage systems with background jobs. In Chapters 6, we show how to efficiently manage the idleness in storage systems for serving background jobs without violating pre-defined performance targets of foreground jobs. Finally, a summary of this dissertation and future work are outlined in Chapter 7.

Chapter 2

Background

In this chapter, we introduce basic concepts and models that are used in the entire dissertation to quantify, identify, and model burstiness in workloads. We also present the performance impact of burstiness that motivates this dissertation.

2.1 Introduction to Burstiness

Burstiness captures the order of a sequence in the time series. For example, with burstiness in arrival streams, we can observe a burst of requests arriving during some periods but very few requests during some other periods. Similarly, when the service times are bursty, requests with long service times are frequently clustered together while short ones are next to each other as well.

Figure 2.1 gives an example to better understand this intuition. In this example, we use two different service processes, where one is independent without burstiness in service times (see Figure 2.1(I)) and the other one exhibits strong burstiness (see Figure 2.1(II)). For each of the two service processes, the two subfigures (i.e., Figure 2.1(a)-(b) and Fig-

ure 2.1(c)-(d)) compare the probability of occurrence of (small,small)$_k$ (white part of the bar) and (small,large)$_k$ (black part of the bar) pairs with the similarly defined probability of occurrence of (large,small)$_k$ (white part of the bar) and (large,large)$_k$ (black part of the bar) pairs, as a function of the lag k, $1 \leq k \leq 9$, i.e., their relative distance in the sequence. Figure 2.1(I) shows that without burstiness the probability of the next observation being small or large neither depends on the lag k nor on the current observation. The opposite is observed in Figure 2.1(II), where the lag-k probability of the next observation being small or large strongly depends on the current observation. We see that the probability of having large observations within the next nine lags after a large one is significant and ranges from $\sim$ 65% to $\sim$ 35%, see Figure 2.1(d). Similarly, the probability of having a small observation after another small one is very large, see Figure 2.1(c).

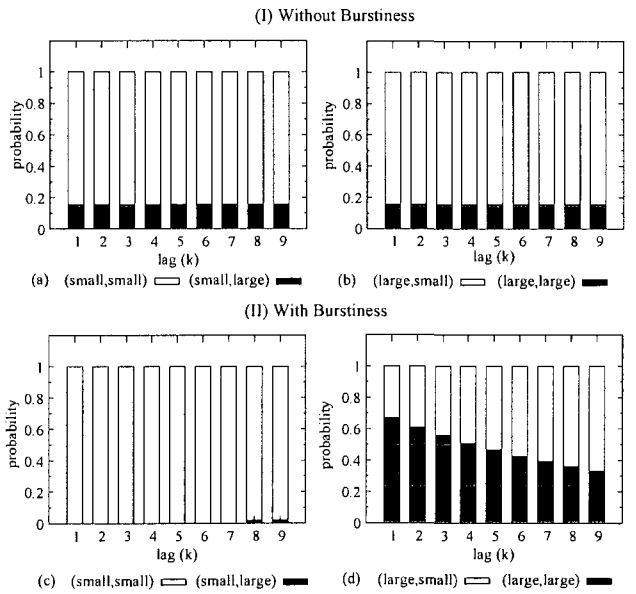


Figure 2.1: The probabilities of pairs (small,small)$_k$, (small,large)$_k$, (large,small)$_k$ and (large,large)$_k$ as a function of lags k. Plots (a) and (b) give results for a sequence without burstiness. Plots (c) and (d) give results for a sequence with burstiness.

One can take advantage of the above information to predict the near future based on the recent past. Meanwhile, this also implies that burstiness may have significant impact on system performance, which will be shown later in Section 2.5. Therefore, it is important to find some measures to capture burstiness in flows. In the following sections, we describe two statistical measures that identify burstiness.

2.2 Autocorrelation

The *autocorrelation function* (ACF) is used to quantitatively describe burstiness in flows [13]. Let $\{X_t\}$ be a stationary time series of identically distributed random variables, where $t = 0, 1, 2, \ldots, \infty$. The autocorrelation function ρ_k is the sequence of correlation coefficients:

$$\rho_k = E[(X_t - \mu^{-1})(X_{t+k} - \mu^{-1})]/\sigma^2, \tag{2.1}$$

where μ^{-1} is the mean and σ^2 is the variance of $\{X_t\}$; the subscript k is called the lag and denotes the number of observations that separate X_t and X_{t+k}. The values of ρ_k are in the range $[-1, 1]$. In most cases, ρ_k approaches zero as k increases. If $\rho_k = 0$ at all lags, then we say the time series $\{X_t\}$ is not autocorrelated.

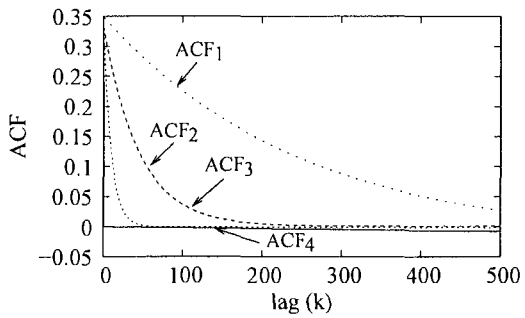


Figure 2.2: Illustrating the ACF of the four service processes with different autocorrelation profiles ACF_1, ACF_2, ACF_3, and ACF_4, respectively.

Figure 2.2 shows the examples of the ACF of four service processes with different autocorrelation profiles:

- ACF_1: $\rho_1 = 0.35$ decays to zero beyond lag $k = 680$;
- ACF_2: $\rho_1 = 0.32$ decays to zero beyond lag $k = 210$;
- ACF_3: $\rho_1 = 0.30$ decays to zero beyond lag $k = 37$;
- ACF_4: $\rho_k = 0$ for all lags $k \geq 1$.

Here, ACF_4 in Figure 2.2 gives an example of the service process without burstiness, where ACF values are equal to 0 at all lags. ACF_1 has the highest values across all lags and illustrates the service process with the strongest burstiness.

Intuitively, higher ACF values indicate stronger burstiness within the workload. If a time series is not bursty, then samples in this time series are drawn in a *random* way without any temporal locality in the distribution space, i.e., given the current sampled value, any value of the distribution space is equally likely to occur in the next sampling. We say that such a stochastic process is memoryless and not bursty. In contrast, an autocorrelated stochastic process generates random variables within a certain range for some time before moving into another range. This way of sampling can keep the same distribution function for overall values but does create temporal locality, i.e., very large or very small values are sampled close to one another. Such a stochastic process exhibits burstiness.

2.3 Index of Dispersion

The autocorrelation function describes burstiness as a *function* of lags. However, this form of description may not be suitable for effective use in practice. In this section, we consider an alternative simple measure, called the *index of dispersion* I, to capture burstiness. The main advantage of I is that it can qualitatively identify burstiness in a single number and thus can provide a simple and effective way to infer different burstiness profiles in flows.

The index of dispersion has a broad applicability and wide popularity in stochastic analysis and engineering [20]. From a mathematical perspective, the index of dispersion of a stochastic process is a measure defined on the squared coefficient-of-variation SCV and on the lag-k autocorrelations ρ_k, $k \geq 1$, of the samples in the time series as follows:

$$I = SCV\left(1 + 2\sum_{k=1}^{\infty} \rho_k\right). \tag{2.2}$$

The joint presence of SCV and autocorrelations in I captures burstiness in flows. For example, we have examined the 1998 FIFA World Cup website trace available at [5] over a period of ten days and found significant burstiness in the arrivals to server 0, see Figure 2.3. By using the theoretical formulas in [38] Eq.(6), we estimate that the arrival stream has I slightly larger than 6300. This implies that a parameterization of I spanning a range from single to multiple digits can give a good sense of scalability between workloads with "no burstiness" and workloads with "dramatic burstiness".

If the stochastic process is exponential, then $I = 1$. Thus, the index of dispersion may be interpreted qualitatively as the ratio of the observed burstiness with respect to a Poisson process. Values of I of the order of hundreds or more indicate a clear departure

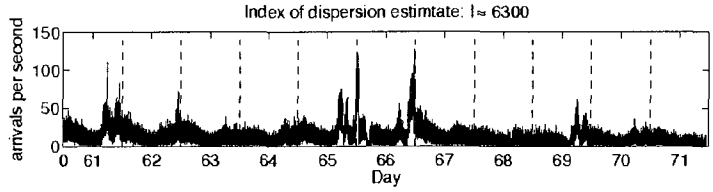


Figure 2.3: Burstiness of arrivals to server 0 in the 1998 FIFA World Cup trace over ten consecutive days. This figure focuses on the server with label "0" from day 61 to day 71. The index of dispersion I is reported on the top of the figure.

from the exponentiality assumptions and, unless the real SCV is anomalously high, I can be used as a good indicator of burstiness.

2.4 Markovian Arrival Processes (MAPs)

After measuring workload burstiness, a mathematical model is needed to integrate burstiness into a stochastic process. In this dissertation, we use Markovian Arrival Processes (MAPs) to express the arrival and/or service process in queueing networks. MAPs, introduced by Neuts [67], can easily model general distributions and nonrenewal features such as autocorrelation of the stochastic process. Previous work in [69, 43, 16] has developed efficient fitting schemes for MAP parameterization from measurements and the resulting MAP can approximate effectively long-range and short-range dependence [69].

The MAP is a generalization of the Poisson process by having non-exponential distributed sojourn times between arrivals. Guided by the transitions of an underlying Markov chain, the MAP can signify real events to generate a single random variable. The MAP is formally described by two square matrices $\mathbf{D}_0$ and $\mathbf{D}_1$, with dimensions equal to the number of states in the Markov chain. Matrix $\mathbf{D}_1$ captures all transitions that are associated with real events in the MAP while matrix $\mathbf{D}_0$ only captures the tran-

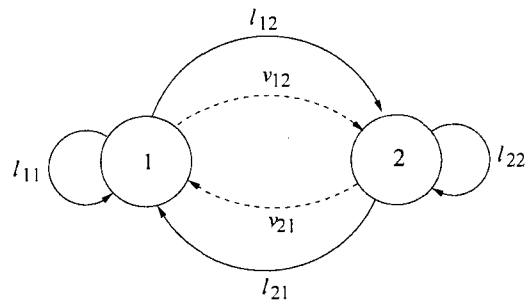


Figure 2.4: State transitions of MAP(2). Transitions shown in solid arrows are associated with the events in MAP and transitions shown in dashed arrows are associated with the changes between states only.

sitions between states without signifying any real events. All off-diagonal entries of $\mathbf{D}_0$ and all entries in $\mathbf{D}_1$ are non-negative.

Let π_{MAP} be the stationary probability vector of the underlying Markov chain for MAP, i.e., $\pi_{\text{MAP}}(\mathbf{D}_1 + \mathbf{D}_0) = \mathbf{0}$, $\pi_{\text{MAP}}\mathbf{e} = 1$, where $\mathbf{0}$ and $\mathbf{e}$ are vectors of zeros and ones of the appropriate dimension. A variety of performance measures are computed using π_{MAP}, $\mathbf{D}_0$, and $\mathbf{D}_1$, such as the mean arrival rate, the squared coefficient of variation, the n-th moments, and the lag-k of its autocorrelation function ACF [68]:

$$\lambda = \pi_{\text{MAP}}\mathbf{D}_1\mathbf{e}, \tag{2.3}$$

$$\begin{aligned} CV^2 &= \frac{E[X^2]}{(E[X])^2} - 1 \\ &= 2\lambda\pi_{\text{MAP}}(-\mathbf{D}_0)^{-1}\mathbf{e} - 1, \end{aligned} \tag{2.4}$$

$$E[X^n] = n!\pi_{\text{MAP}}(-\mathbf{D}_0)^{-n}\mathbf{e}, \tag{2.5}$$

$$\begin{aligned} ACF(k) &= \frac{E[(X_0 - E[X])(X_k - E[X])]}{\text{Var}[X]} \\ &= \frac{\lambda\pi_{\text{MAP}}((-\mathbf{D}_0)^{-1}\mathbf{D}_1)^k(-\mathbf{D}_0)^{-1}\mathbf{e} - 1}{2\lambda\pi_{\text{MAP}}(-\mathbf{D}_0)^{-1}\mathbf{e} - 1}, \end{aligned} \tag{2.6}$$

where X_0 and X_k denote two inter-event times with k lags apart.

As an example, Figure 2.4 illustrates the state transitions of a 2-state MAP with the following 2×2 matrices $\mathbf{D}_0$ and $\mathbf{D}_1$.

$$\begin{aligned} \mathbf{D}_0 &= \begin{bmatrix} -(v_{12} + l_{11} + l_{12}) & v_{12} \\ v_{21} & -(v_{21} + l_{21} + l_{22}) \end{bmatrix}, \\ \mathbf{D}_1 &= \begin{bmatrix} l_{11} & l_{12} \\ l_{21} & l_{22} \end{bmatrix} . \end{aligned} \tag{2.7}$$

As shown in Figure 2.4, there are three kinds of transitions: (1) the transitions that *only* change the state from "1" (resp., "2") to "2" (resp., "1") but do not correspond to any real events, see the dashed arrows with mean rate of v_{12} (resp., v_{21}) in the figure; (2) the transitions that *only* signify real events but do not change the state "1" (resp., "2"), see the solid arrows with mean rate of l_{11} (resp., l_{21}) in the figure, and (3) the transitions that change the state from "1" (resp., "2") to "2" (resp., "1") *and* signify real events, see the solid arrows with mean rate of l_{12} (resp., l_{21}) in the figure. Based on this description, we provide in Appendix A the generation of a MAP process in the form of a pseudo code.

A special case of MAP(2) is a 2-state Markov-Modulated Poisson Process (MMPP(2)) [42, 57, 49]. The matrix representation of the MMPP(2) is defined as follows.

$$\begin{aligned} \mathbf{D}_0 &= \begin{bmatrix} -(l_{11} + v_{12}) & v_{12} \\ v_{21} & -(l_{22} + v_{21}) \end{bmatrix}, \\ \mathbf{D}_1 &= \begin{bmatrix} l_{11} & 0 \\ 0 & l_{22} \end{bmatrix} . \end{aligned} \tag{2.8}$$

As shown in Eq.(2.8), matrix $\mathbf{D}_1$ is diagonal. That is, there is no solid arrows with mean rate of l_{12} and l_{21} in Figure 2.4. All the transitions that are associated with real events do not change the state. MMPP(2) is the type of process that is popular because it has only four parameters and can be easily parameterized. One can set any of the four parameters

(e.g., v_{12}) as the free parameter and calculate the remaining three parameters by using the Eqs.(2.3), (2.4), and (2.6) to match the pre-defined mean, squared coefficient of variation and ACF(1) of a process.

2.5 Performance Impacts

In this section, we use a simple example to exemplify the performance impact of burstiness. Consider the four workloads shown in Figure 2.5, where each plot represents a sample of $20,000$ service times generated from the same MAP distribution with mean $\mu^{-1} = 1$ and squared coefficient-of-variation $SCV = 3$. The only difference is that we impose to each service trace a unique burstiness profile. Here, for the details on the trace generation, we refer the reader to Section 2.4. In Figure 2.5(b)-(d), the large service times progressively aggregate in bursts, while in Figure 2.5(a) they appear in random points of the trace. In particular, Figure 2.5(d) shows the extreme case where all large requests are compressed into a single large burst. Additionally, the values of I for these four traces are also shown in Figure 2.5: for the trace in Figure 2.5(a), the correlations are statically negligible, thus the value of I is exactly equal to SCV; however, for the trace in Figure 2.5(d), consecutive samples tend to assume similar values, therefore the sum of autocorrelation in Eq.(2.2) is maximal in Figure 2.5(d), which gives the largest value of I among the four traces.

What is the performance implication on systems of the different burstiness profiles in Figure 2.5(a)-(d)? Assume that the request arrival times to the server follow an exponential distribution with mean $\lambda^{-1} = 2$ and 1.25. A simulation analysis of the $M/Trace/1$ queue at 50% and 80% utilization provides the response times, i.e., the service time plus waiting/queueing times in a server, shown in Table 2.1.

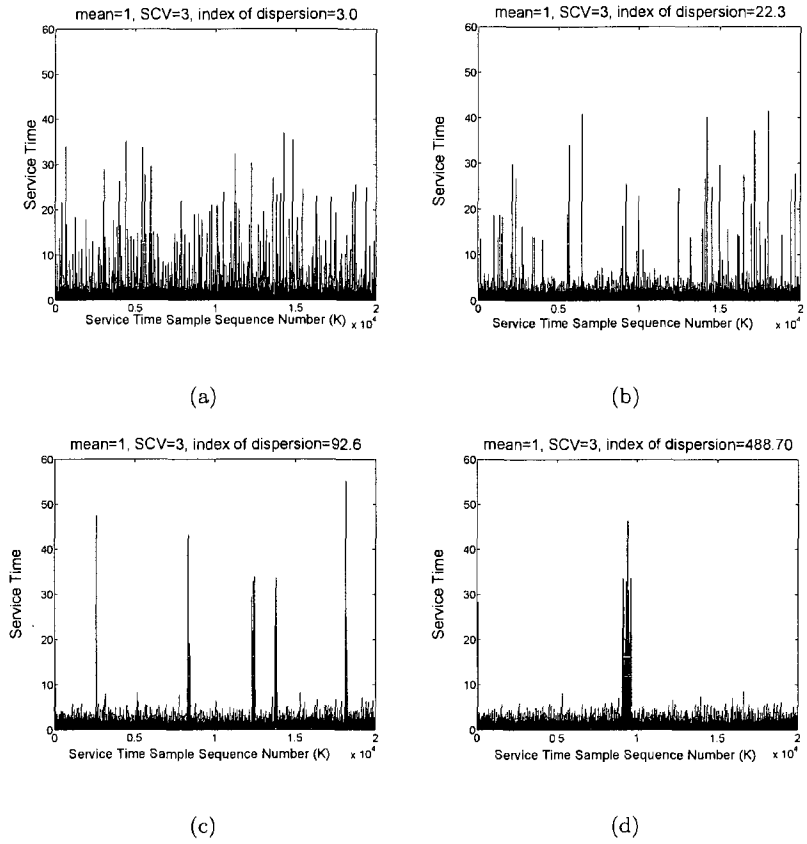


Figure 2.5: Four workload traces with identical MAP distribution (mean $\mu^{-1} = 1$, $SCV = 3$), but different burstiness profiles. Given the identical variability, trace (d) represents the case of maximum burstiness where all large service times appear consecutively in a large burst. The index of dispersion I, reported on top of each plot, is able to capture the significantly different burstiness of the four workloads.

Irrespectively of the identical service time distribution, burstiness clearly has paramount impact on system performance, in terms of both response time mean and tail. For instance, at 50% utilization the mean response time for the trace in Figure 2.5(d) is approximately 40 times slower than the service times in Figure 2.5(a) and the 95th percentile of the response times is nearly 80 times longer. In general, the performance degradation is monotonically increasing with burstiness; therefore it is important to consider the performance effect of burstiness in system design.

	Response Time (util=0.5)		*Response Time (util=0.8)*	
Workload	mean	95th percentile	mean	95th percentile
Fig. 2.5(a)	3.02	14.42	8.70	33.26
Fig. 2.5(b)	11.00	83.35	43.35	211.76
Fig. 2.5(c)	26.69	252.18	72.31	485.42
Fig. 2.5(d)	120.49	1132.40	150.32	1346.53

Table 2.1: Response time of the $M/Trace/1$ queue relatively to the service times traces shown in Figure 2.5. The server is evaluated for utilizations $\rho = 0.5$ and $\rho = 0.8$.

2.6 Chapter Summary

In this chapter, we give an overview of basic concepts and terminology. We focus on burstiness, the main topic in this dissertation, as well as the related concepts and models, including autocorrelation function, index of dispersion and Markovian-Arrival Process. In particular, the index of dispersion is exploited as a metric to capture burstiness in arrival streams and service times, see Chapter 3. The autocorrelation function is used to predict the immediate future, e.g., to estimate service times in multi-tier systems (see Chapter 4) and to forecast idle interval lengths in storage systems, see Chapter 6. This dissertation also use the MAP models to represent the arrival and/or service process to regulate bursty flows in systems.

Chapter 3

Capacity Planning in Multi-tier Enterprise Systems

The performance of a multi-tier system is determined by the interactions between the incoming requests and the different hardware architectures and software systems that serve them. In order to model these interactions for capacity planning, a detailed characterization of the workloads and of the application is needed, but such "customized" analysis and modeling may be very time consuming, error-prone, and inefficient in practice. An alternative approach is to rely on live system measurements and to assume that the performance of each software or hardware resource is completely characterized by its *mean* service time, a quantity that is easy to obtain with simple measurement procedures. The mean service times of different classes of transaction requests together with the transaction mix can be used as inputs to the widely-used Mean Value Analysis (MVA) models [50, 92, 101] to predict the overall system performance under various load conditions. The popularity of MVA-based models is due to their simplicity and to their ability to capture complex

systems and workloads in a straightforward manner. In this chapter, we present strong evidence that MVA models of multi-tier architectures can be unacceptably inaccurate if the processed workloads exhibit *burstiness*.

Burstiness in the *service process* is often found in multi-tier systems. The source of burstiness can be located in the application server [64] or in the back-end database [63] and is an effect of the hardware/software configuration of the system. In such multi-tier systems, this congestion may arise from the super-position of several events including database locks, variability in service time of software operations, memory contention, and/or characteristics of the scheduling algorithms. The above events interact in a complex way with the underlying hardware/software systems and with the incoming requests, often resulting in burstiness in service processes and creating congestion periods where and the entire system is significantly slowed down. For example, even for multi-tier systems where the database server is highly-efficient, a locking condition on a database table may slow down the service of multiple requests that try to access the same data and make the database the bottleneck server for a time period. During that period of time, the database performance dominates the performance of the overall system, while most of the time another resource, e.g., the application server, may be the primary cause of delays in the system. Thus, the performance of the multi-tier system can vary in time depending on which is the current bottleneck resource and can be significantly conditioned by dependencies between servers. However, *this time-varying bottleneck switch, as a symptom of burstiness in service processes, cannot by captured by MVA models.*

Motivated by this problem, we define a new methodology for effective capacity planning under systems with bursty service demands. This new approach integrates burstiness

in performance models, by relying on server busy periods (they are immediately obtained from server utilization measurements across time) and measurements of request completions within the busy periods. All measurements are collected with coarse granularity by existing commercial tools, e.g., the HP Diagnostic tool. After giving quantitative examples of the importance of integrating burstiness in performance models, we analyze a real three-tier architecture subject to TPC-W workloads with different burstiness profiles. We show that burstiness in the service process can be inferred effectively from traces using the index of dispersion and the accuracy of the model prediction can be increased by up to 30% compared to standard queueing models parameterized only with mean service demands [74].

Later in this chapter, we propose a methodology that can introduce burstiness into the *arrival process* of a benchmark. The methodology complements the existing benchmarks which only present burstiness in the service process but not in the arrival process. An effective benchmark should evaluate the system responsiveness under a wide range of client traffic. The existing benchmarks, e.g., the standard TPC-W benchmark, are designed to assess the system responsiveness only under a steady/normal traffic. However, the system behavior under bursty traffic may actually be very different from that under the steady one. Due to its tremendous performance implications, burstiness must be accounted in capacity planning and must be incorporated into benchmarking of client-server systems. Therefore, we present a new methodology for generating workloads that emulate burstiness in a controllable way by introducing it into the arrival stream, and thus providing a mechanism to test and evaluate the system performance under reproducible bursty workloads. We exemplify this new methodology to create bursty workloads within the TPC-W

benchmark.

3.1 Related Work

Capacity planning of multi-tier systems is a critical part of the architecture design process and requires reliable quantitative methods, see [61] for an introduction. Queueing models are popular for predicting system performance and answering what-if capacity planning questions [61, 94, 93, 92]. Single-tier queueing models focus on capturing the performance of the most-congested resource only (i.e., bottleneck tier): [94] describes the application tier of an e-commerce system as a M/GI/1/PS queue; [73] abstracts the application tier of a N-node cluster as a multi-server G/G/N queue.

Mean Value Analysis (MVA) queueing models that capture all the multi-tier architecture performance have been validated in [93, 92] using synthetic workloads running on real systems. The parameterization of these MVA-based models requires only the mean service demand placed by requests at the different resources. In [81], the authors use linear regression techniques for estimating from utilization measurements the mean service demands of applications in a single-threaded software server. In [55], Liu et al. calibrate queueing model parameters using inference techniques based on end-to-end response time measurements. A traffic model for Web traffic has been proposed in [54], which fits real data using mixtures of distributions.

The observations in [64, 63] show that burstiness does exist in the *service process* of multi-tier systems, which can cause the phenomenon of bottleneck switch between the tiers. Indeed, [11] shows that burstiness in the World Wide Web and its related applications increases the load of the Web server beyond its capacity, which results in significant

degradation of the server performance. The class of MAP queueing networks has been first introduced in [15] together with a bounding technique to approximate the model solution of queueing network models with bursty service processes. Here, we propose a parameterization of MAP queueing networks based on the coarse measurements from real systems.

Several studies have shown that the *arrival process* in a Web-based system is self-similar [21, 58]. Self-similar workloads exhibit significant request correlations or bursts over multiple timescales [3]. If a system is not able to support bursts at some timescale, significant queuing delays may occur [73]. Several commonly used workload generators have been developed for testing Web servers [12, 47, 65, 48]. For example, SURGE [12] uses an offline trace generation engine to create a trace of HTTP requests, but this approach is difficult to apply for controlling or enforcing the aggregate traffic characteristics, especially the network impact on the individual user arrival process. The GEIST tool [47] attempts to match the aggregate workload characteristics and models attributes of the request arrival process at the system level. The Httperf [65] tool provides a flexible facility to generate various HTTP workloads for measuring Web server performance. However, none of these tools provide a special, controlled way of enforcing burstiness into the generated workload. In this dissertation, we present a hybrid approach which can generate bursty request arrivals for Web and e-commerce workloads. We show that this approach supports session-based workloads and in addition enables a fine control over the aggregate request arrival process in the system.

3.2 Burstiness in the Service Process of Multi-Tier Applications

The multi-tier architecture is now the industry standard for implementing scalable client-server enterprise applications. In our experiments, we use a testbed of a multi-tier e-commerce site that is built according to the TPC-W specifications. This allows to conduct experiments under different settings in a controlled environment, which then allows to evaluate the proposed modeling methodology that is based on the index of dispersion.

3.2.1 Experimental Environment

TPC-W is a widely used e-commerce benchmark that simulates the operation of an on-line bookstore [32]. Typically, this multi-tier application uses a three-tier architecture paradigm, which consists of a web server, an application server, and a back-end database. A client communicates with this web service via a web interface, where the unit of activity at the client-side corresponds to a webpage download. In general, a web page is composed by an HTML file and several embedded objects such as images. In a production environment, it is common that the web and the application servers reside on the same hardware, and shared resources are used by the application and web servers to generate main HTML files as well as to retrieve page embedded objects. We opt to put both the web server and the application server on the same machine called the front server. Note, we use terms "front server" and "application server" interchangeably in this chapter. A high-level overview of the experimental set-up is illustrated in Figure 3.1 and specifics of the software/hardware used are given in Table 3.1.

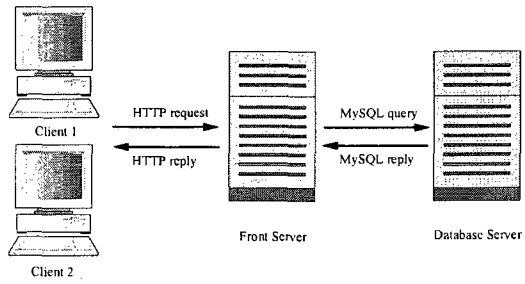


Figure 3.1: E-commerce experimental environment.

	Processor	RAM	OS
Clients (Emulated-Browsers)	Pentium D, 2-way x 3.2GHz	4GB	Linux Redhat 9.0
Front Server (Apache/Tomcat5.5)	Pentium D, 1-way x 3.2GHz	4GB	Linux Redhat 9.0
Database Server (MySQL5.0)	Pentium D, 2-way x 3.2GHz	4GB	Linux Redhat 9.0

Table 3.1: Hardware/software components of the TPC-W testbed.

Since the HTTP protocol does not provide any means to delimit the beginning or the end of a web page, it is very difficult to accurately measure the aggregate resources consumed due to web page processing at the server side. Accurate CPU consumption estimates are required for building an effective application provisioning model but there is no practical way to effectively measure the service times for *all* page objects. To address this problem, we define a *client transaction* as a combination of *all* processing activities that deliver an entire web page requested by a client, i.e., generate the main HTML file as well as retrieve embedded objects and perform related database queries.

Typically, a continuous period of time during which a client accesses a Web service is referred to as a *User Session* which consists of a sequence of consecutive individual transaction requests. According to the TPC-W specification, the number of concurrent sessions (i.e., customers) or emulated browsers (EBs) is kept constant throughout the

experiment. For each EB, the TPC-W benchmark defines the user session length, the user think time, and the queries that are generated by the session. In our experimental environment, two Pentium D machines are used to simulate the EBs. If there are m EBs in the system, then each machine emulates $m/2$ EBs. One Pentium D machine is used as the back-end database server, which is installed with MySQL 5.0 having a database of 10,000 items in inventory.

There are 14 different transactions defined by TPC-W. In general, these transactions can be roughly classified of "Browsing" or "Ordering" type, as shown in Table 3.2. Furthermore, TPC-W defines three standard transaction mixes based on the weight of

Browsing Type	**Ordering** Type
Home	Shopping Cart
New Products	Customer Registration
Best Sellers	Buy Request
Product detail	Buy Confirm
Search Request	Order Inquiry
Execute Search	Order Display
	Admin Request
	Admin Confirm

Table 3.2: The 14 transactions defined in TPC-W.

each type (i.e., browsing or ordering) in the particular transaction mix:

- the *browsing mix* with 95% browsing and 5% ordering;
- the *shopping mix* with 80% browsing and 20% ordering;
- the *ordering mix* with 50% browsing and 50% ordering.

One way to capture the navigation pattern within a session is through the *Customer Behavior Model Graph (CBMG)* [60], which describes patterns of user behavior, i.e., how users navigate through the site, and where arcs connecting states (transactions) reflect the probability of the next transaction type. TPC-W is parameterized by the set of probabilities that drive user behavior from one state to another at the user session level. During a session, each EB cycles through a process of sending a transaction request, receiving the response web page, and selecting the next transaction request. Typically, a user session starts with a Home transaction request.

Transaction Latency Monitoring

The TPC-W implementation is based on the J2EE standard – a Java platform which is used for web application development and designed to meet the computing needs of large enterprises. For transaction monitoring we use the HP (Mercury) Diagnostics [98] tool which offers a monitoring solution for J2EE applications. The Diagnostics tool consists of two components: the Diagnostics Probe and the Diagnostics Server as shown in Figure 3.2.

The Diagnostics tool collects performance and diagnostic data from applications without the need for application source code modification or recompilation. It uses bytecode instrumentation and industry standards for collecting system and JMX metrics. Instrumentation refers to bytecode that the Probe inserts into the class files of the application as they are loaded by the class loader of the virtual machine. Instrumentation enables a Probe to measure execution time, count invocations, retrieve arguments, catch exceptions and correlate method calls and threads.

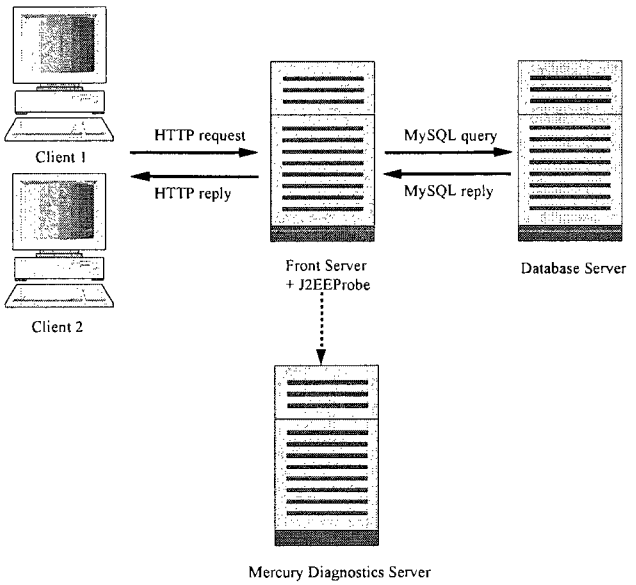


Figure 3.2: TPC-W experimental configuration with the Diagnostics tool.

The J2EE Probe shown in Figure 3.2 is responsible for capturing events from the application, aggregating the performance metrics, and sending these captured performance metrics to the Diagnostics Server. In a monitoring window, Diagnostics provides the following information for each transaction type:

- a transaction count;
- an average overall transaction latency for observed transactions. This overall latency includes transaction processing time at the application server as well as all related query processing at the database server, i.e., latency is measured from the moment of the request arrival at the application server to the time when a prepared reply is sent back by the application server, see Figure 3.3;
- a count of outbound (database) calls of different types;
- an average latency of observed outbound calls (of different types). The average

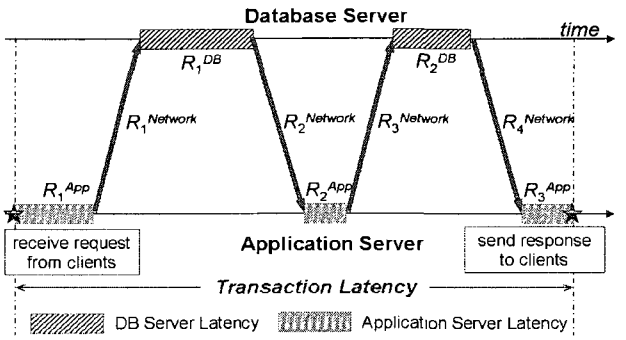


Figure 3.3: The transaction latency measured by the Diagnostics tool.

latency of an outbound call is measured from the moment the database request is issued by the application server to the time when a prepared reply is returned back to the application server, i.e., the average latency of the outbound call includes database processing and communication latency.

Currently, the Diagnostics server reports the measured metrics via a GUI interface and stores them in a time series database. A Java-based processing utility has been implemented for extracting performance data from the Diagnostics server in real-time and creating a so-called "application log" that provides a complete information on all transactions processed during the monitoring window, such as their transaction counts, overall latencies, and outbound calls. While in this work, we use only a subset of the extracted fields, we believe that the proposed application log format enables many value-added services such as anomaly detection and application behavior diagnosis.

3.2.2 Bottleneck Switch in TPC-W

For each transaction mix, we run a set of experiments with different numbers of EBs ranging from 25 to 150. Each experiment runs for 3 hours, where the first 5 minutes and the last 5 minutes are considered as warm-up and cool-down periods and thus omitted in the analysis. User think times are exponentially distributed with mean $E[Z] = 0.5s$. Figure 3.4 presents the overall system throughput, the mean system utilization at the front server and the mean system utilization at the database server as a function of EBs. Figure 3.4(a) shows that the system becomes overloaded when the number of EBs reaches 75, 100, and 150 under the browsing mix, the shopping mix, and the ordering mix, respectively. Beyond these EB values, the system throughput remains asymptotically flat. This is due to the "closed loop" aspect of the system, i.e., the fixed number of EBs (customers), that is effectively an upper bound on the number of jobs that circulate in the system at all times.

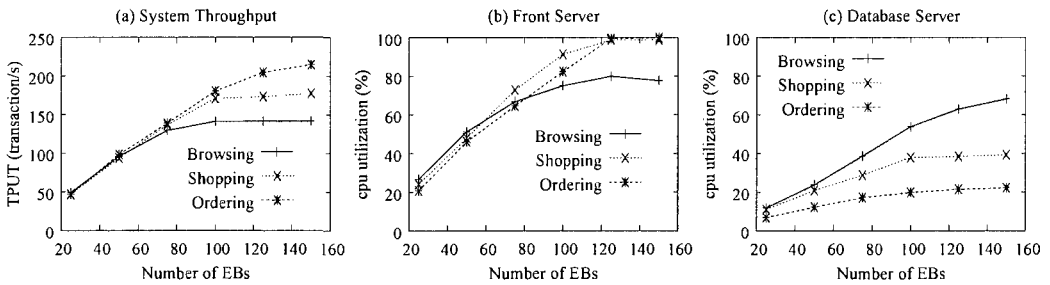


Figure 3.4: Illustrating a) system overall throughput, b) average CPU utilization of the front server, and c) average CPU utilization of the database server for three TPC-W transaction mixes. The mean think time is set to $E[Z] = 0.5s$.

The results from Figure 3.4(b) and (c) show that under the shopping and the ordering mixes, the front server is a bottleneck, where the CPU utilizations are almost 100% at the front tier but only 20-40% at the database tier. For the browsing mix, we see that the

CPU utilization of the front server increases very slowly as the number of EBs increases beyond 75, which is consistent with the very slow growth of throughput. For example, when the front server is already 100% utilized under the shopping and the ordering mixes, the front server for the browsing mix is just around 80%. Meanwhile, for the browsing mix, the CPU utilization of the database server increases quickly as the number of EBs increases. When the number of EBs is beyond 100, it is not obvious which server is responsible for the bottleneck: the average CPU utilizations of two servers are about the same, differing by a statistically insignificant margin. In presence of burstiness in the service times, this may suggest that the phenomenon of *bottleneck switch* occurs between the front and the database servers *across time*. This phenomenon is not specific to the testbed described in the current work. In an earlier work [102], a similar situation was observed for a different TPC-W testbed. That is, a server may become the bottleneck while processing consecutively large requests, but be lightly loaded during other periods. In general, additional investigation to determine the existence of bottleneck switch is required when the average utilizations are relatively close or when the workloads are known to be highly variable.

To confirm our conjecture about the existence of bottleneck switch in the browsing mix experiment, we present CPU utilizations of the front and the database servers across time for the browsing mix, as well as for the shopping and the ordering mixes with 100 EBs, see Figure 3.5. A bottleneck switch occurs when the database server utilization becomes significantly higher than the front server utilization, as clearly visible in Figure 3.5(a) under the browsing mix workload. Such bottleneck switch is a characteristic effect of burstiness in the service times. This unstable behavior is extremely hard to model. In

contrast, as shown in Figure 3.5(b) and (c), there is no bottleneck switch for the shopping and the ordering mixes, although these two workloads are also highly variable.

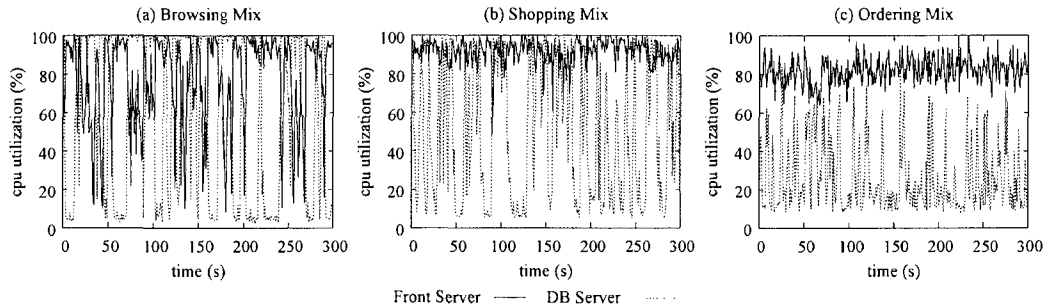


Figure 3.5: The CPU utilization of the front server and the database server across time with 1 second granularity for (a) the browsing mix, (b) the shopping mix, and (c) the ordering mix under 100 EBs. The monitoring window is 300 seconds.

3.2.3 The Analysis of Bottleneck Switch

Now, we focus on the burstiness in a multi-tier application to further analyze the symptoms and possible causes of the bottleneck switch. Indeed, for a typical request-reply transaction, the application server may issue multiple database calls while preparing the reply of a web page. This cascading effect of various tasks breaks down the overall transaction service time into several parts, including the transaction processing time at the application server as well as all related query processing times at the database server. Therefore, the application characteristics and the high variability in database server may cause burstiness in the overall transaction service times.

To verify the above conjecture, we record the queue length at the database server at each instance that the database request is issued by the application server and a prepared reply is returned back to the application server. Figure 3.6 presents the queue length across time at the database server (see solid lines in the figure) as well as the CPU utilizations

of the database server (see dashed lines in the figure) for all three transaction mixes.

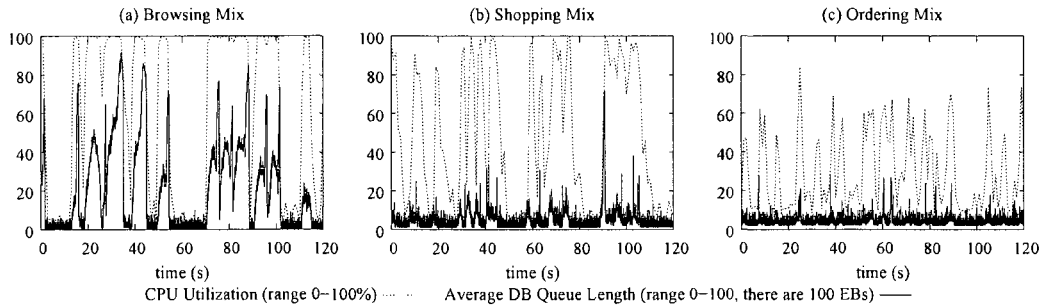


Figure 3.6: The CPU utilization of the database server (dashed lines) and average queue length at the database server (solid lines) across time for (a) the browsing mix, (b) the shopping mix, and (c) the ordering mix. In this figure, the y-axis range of both performance metrics is the same because there are 100 EBs (clients) in the system. The monitoring window is 120 seconds.

Here, in order to make the figure easy to read, we show the case with 100 EBs such that the y-axis range for both performance metrics (i.e., queue length and utilization) is the same. First of all, the results for the browsing mix in Figure 3.6(a) verify that burstiness does exist in the queue length at the database server, where the queue holds less than 10 jobs for some periods, while sharply increases to as high as 90 jobs during other periods. More importantly, the burstiness in the database queue length exactly matches the burstiness in the CPU utilizations of the database server. Thus, at some periods almost all the transaction processing happens either at the application server (with the application server being a bottleneck) or at the database server (with the database server being a respective bottleneck). This leads to the alternated bottleneck between the application vs the database servers.

In contrast, no burstiness can be observed in the queue length for the shopping and the ordering mixes, although these two workloads have also high variability in their utilizations, see Figure 3.6(b) and (c). These results are consistent with those shown in

Figure 3.5(b) and (c), where the application server is the main system bottleneck.

According to the TPC-W specification, different transaction types may have different number of outbound database queries. For example, the *Home* transaction has two database queries in maximum and one in minimum for each transaction request while the *Best Seller* transaction always has two outbound database queries per transaction request. To analyze whether burstiness in the database queue length originates from some particular transaction types, we measure the number of current requests for each transaction type over time. After revisiting all 14 transaction types, we find that the sources of this burstiness are indeed due to specific transaction types. Figures 3.7 and 3.8 show the results for two representative transaction types, the *Best Seller* transaction and the *Home* transaction, under three transaction mixes.

In Figure 3.7, the overall database queue length across time is also plotted as a base line. As shown in Figure 3.7(a), although in the browsing mix only 11% of requests belongs to the *Best Seller* transaction type, the number of these requests dominates the overall database queue length: the spikes in the overall queue length in the database clearly originate from this particular transaction type. Furthermore, there is burstiness in the number of requests for this transaction type and this burstiness "matches" well the overall queue length in the database server. In addition, for some extremely high spikes, e.g., at timestamp 40 in Figure 3.7(a), the requests of another popular transaction type, the *Home* transaction, also contribute to burstiness (see Figure 3.8(a)). These figures indicate that *Best Seller* and *Home* transactions share some resources required for their processing at the database server, which leads to extreme burstiness during such time periods.

For the shopping and the ordering mixes, there is no visible burstiness in either the

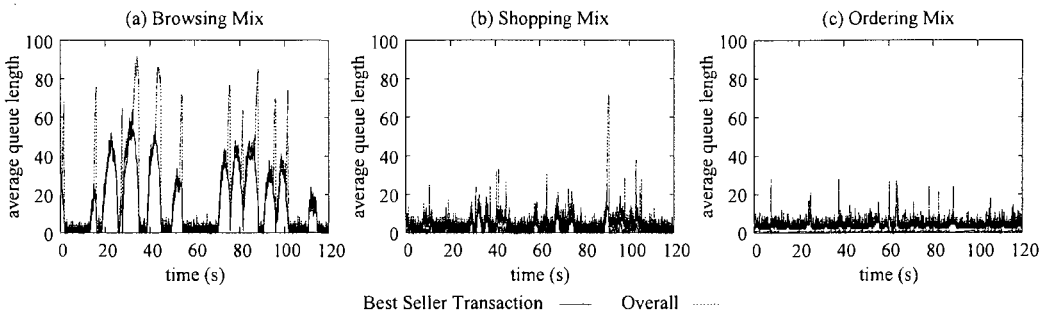


Figure 3.7: The overall queue length at the database server (dashed lines) and the number of current requests in system for the *Best Seller* transaction (solid lines) across time for (a) the browsing mix, (b) the shopping mix, and (c) the ordering mix, with 100 EBs and mean think time equal to $0.5s$. The monitoring window is 120 seconds.

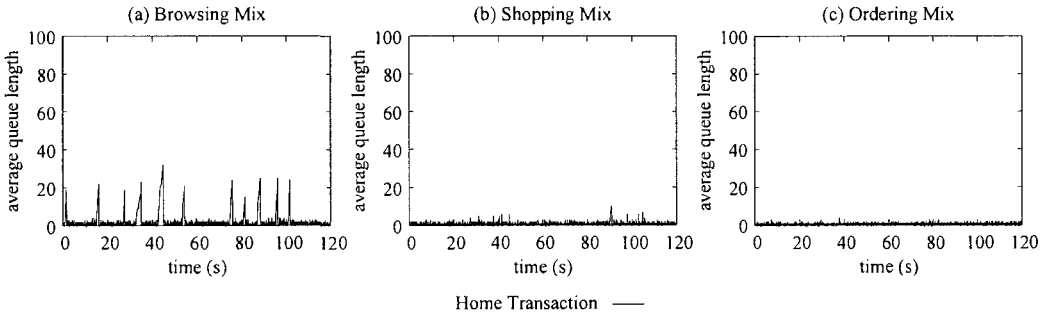


Figure 3.8: The number of current requests in system for the *Home* transaction across time for (a) the browsing mix, (b) the shopping mix, and (c) the ordering mix, with 100 EBs and mean think time equal to $0.5s$. The monitoring window is 120 seconds.

queue length at the database server or the number of current requests for each transaction type, as shown in Figure 3.7(b)-(c) and Figure 3.8(b)-(c), respectively.

In summary, we showed that

- burstiness in the service times can be a result of a certain workload combination (mix) in the multi-tier applications (e.g., burstiness in the service times may exist under the browsing mix in the TPC-W testbed);
- burstiness in the service times can be caused by a bottleneck switch between the tiers, and can be a result of "hidden" resource contention between the transactions

of different types and across different tiers.

Systems with burstiness result in unstable behavior that is extremely hard to express and model. The super-position of several events, such as database locking conditions, variability in service time of software operations, memory contention, and/or characteristics of the scheduling algorithms, may interact in a complex way, resulting in burstiness in the system. The challenging is instead of identifying the low-level *exact* causes of burstiness as traditional models would require, whether one can provide an effective way to infer this information using live system measurements in order to capture burstiness into new capacity planning models.

3.2.4 Traditional MVA Performance Models Do not Work

In this section, we use standard performance evaluation methodologies to define an analytical model of the multi-tier architecture presented in Section 3.2.1. Our goal is to show that existing queueing models can be largely inaccurate in performance prediction if the system is subject to bottleneck switches. We show in Section 3.3 how performance models can be generalized to correctly account for burstiness and bottleneck switches based on the index of dispersion.

We model the multi-tier architecture studied in our experiments by a closed queueing network composed of two queues and a delay center as shown in Figure 3.9. Closed queueing networks (see [50] for an introduction) are established as the standard capacity planning models for predicting the performance of distributed architectures using inexpensive algorithms, e.g., Mean Value Analysis (MVA) [75].

In the MVA model shown in Figure 3.9, the two queues are used to abstract perfor-

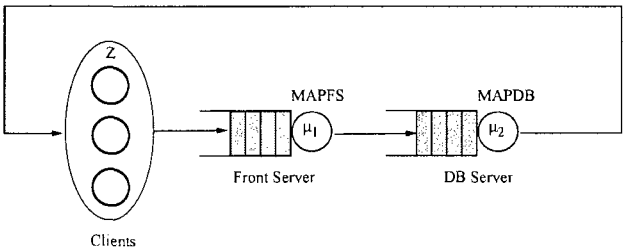


Figure 3.9: The closed queueing network for modeling the multi-tier system.

mance of the front server and of the database server, respectively. The delay center is instead representative of the average user think time $E[Z]$ between receiving a Web page and submitting a new page download request. The main difference between a queue and a delay server is that the mean response time at the latter is *independent* of the number of requests present. The two queues serve jobs according to a processor-sharing scheduling discipline. In the real application, the servlet code is a mix of instructions at the front server and the database server: without an expensive analysis of the source code, it is truly difficult to characterize the switch of the execution from the front server to the database server and back, we thus make a simplification by assuming that requests first execute at the front server without any interruption and then the residual service time is processed at the database server. Consequently, with this simplification, the two queues in Figure 3.9 are connected in series. In the following sections, we consider the burstiness associated to the execution of these requests at the front server and at the database server. We stress that our abstraction ignores the order of execution of portions of the servlet code and has no impact on the burstiness estimates because the requests complete *faster* than the monitoring window of the measurement tool. Thus, for an external observer, it would be impossible to distinguish between samples collected from the real system and those of the abstracted system where the code first executes only at the front server and then

completes at the database server.

The proposed MVA model can be immediately parameterized by the following values:

- the mean service time S_{FS} of the front server;
- the mean service time S_{DB} of the database server;
- the average user think time $E[Z]$;
- the number of emulated browsers (EBs).

Note that the arrival process at the multi-tier system, which is in the real system the arrival of new TPC-W sessions, is fully reproduced by the $E[Z]$ parameter. In fact, a new TPC-W session is generated in $E[Z]$ seconds after completion of a previously-running user session: thus, the feedback-loop aspect of TPC-W is fully captured by the closed nature of the queueing network and the user think time $E[Z]$ completes the model of the TPC-W arrival process.

The values of S_{FS} and S_{DB} can be determined with linear regression methods from the CPU utilization samples measured across time at the two servers [101]. Instead, $E[Z]$ and the number of EBs are imposed to set a specific scenario. For example, in Figure 3.10, we evaluate an increase of the number of EBs under the fixed think time $E[Z] = 0.5s$; other choices of the delay are possible, see Section 3.3.3 for a discussion. Indeed, increasing the EB number is a typical way in capacity planning to explore the impact of increasingly larger traffic intensities on system performance. Figure 3.10 shows the results of the MVA model predictions versus the actual measured throughputs (TPUTs) of the system as a function of the number of EBs.

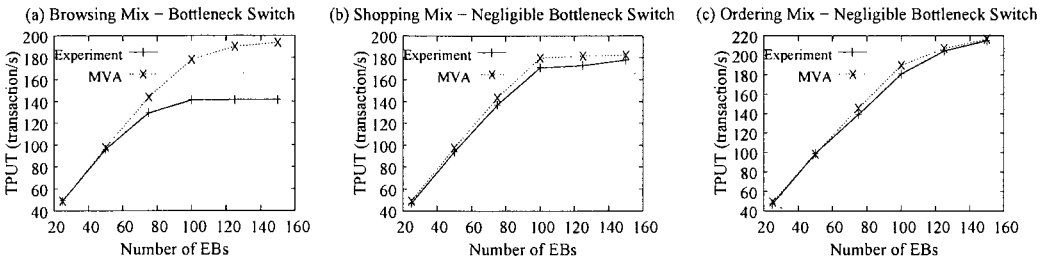


Figure 3.10: MVA model predictions versus measured throughput.

The three plots in the figure illustrate the accuracy of the MVA model under the browsing, the shopping, and the ordering mixes. The results show that the MVA model prediction is quite accurate for the shopping and ordering mixes, but there exists a large error up to 36% between the predicted and the measured throughputs for the browsing mix, see Figure 3.10(a). This indicates that MVA models can deal very well with systems *without* burstiness (e.g., the ordering mix in Figure 3.10(c)) and with systems where burstiness does *not* result in a bottleneck switch (e.g., the shopping mix in Figure 3.10(b)). However, the fundamental and most challenging case of burstiness that causes bottleneck switches reveals the limitation of the MVA modeling technique, see Figure 3.10(a). This is consistent with established theoretical results for MVA models, which rule out the possibility of capturing the bottleneck switching phenomenon [10].

3.3 Integrating Burstiness in Performance Models

In this section, we consider the *index of dispersion* I for counts to characterize the burstiness of service times [20, 38]. As shown in Chapter 2, I can be used as a good indicator of burstiness and can be jointly defined by the squared coefficient-of-variation SCV and the lag-k autocorrelations ρ_k of the samples in the time series. Although this mathematical

definition of I in Eq.(2.2) is simple, this formulation is not practical for estimation because of the infinite summation involved and its sensitivity to noise. In the following subsection, we describe a simple alternative way of estimating I.

3.3.1 Measuring the Index of Dispersion

Instead of Eq.(2.2), we provide an alternative definition of the index of dispersion for a service process as follows. Let N_t be the number of requests completed in a time window of t seconds, where the t seconds are counted *ignoring* the server's idle time (that is, by conditioning on the period where the system is busy, N_t is a property of the service process which is independent of queueing or arrival characteristics). If we regard N_t as a random variable, that is, if we perform several experiments by varying the time window placement in the trace and obtain different values of N_t, then the index of dispersion I is the limit [20]:

$$I = \lim_{t \to +\infty} \frac{Var(N_t)}{E[N_t]}, \tag{3.1}$$

where $Var(N_t)$ is the variance of the number of completed requests and $E[N_t]$ is the mean service rate during busy periods. Since the value of I depends on the number of completed requests in an asymptotically large observation period, an approximation of this index can be also computed if the measurements are obtained with coarse granularity. For example, suppose that the sampling resolution is $T = 60s$, and assume to approximate $t \to +\infty$ as $t \approx 2$ hours, then N_t is computed by summing the number of completed requests in 120 consecutive samples. Repeating the evaluation for different positions of the time window of length t, we compute $Var(N_t)$ and $E[N_t]$. Here, we use the pseudo-code in Figure 3.11 to estimate I directly from Eq.(3.1). The pseudo-code is a straight-forward evaluation of

$Var(N_t)/E[N_t]$ for different values of t. Intuitively, the algorithm in Figure 3.11 calculates I of the service process by observing the completions of jobs in concatenated busy period samples. Because of this concatenation, queueing is masked out and the index of dispersion of job completions serves as a good approximation of the index of dispersion of the service process.

Here, we use a measure of burstiness for the parameterization of the performance model presented in Figure 3.9. In Section 3.3.2, we first present the methodology for integrating the burstiness in queueing models and then discuss the impact of measurement granularity in Section 3.3.3. The experimental results that validate the proposed model are given in Section 3.3.4.

Input

T, the sampling resolution (e.g., $60s$)

K, total number of samples, assume $K > 100$

U_k, utilization in the kth period, $1 \leq k \leq K$

n_k, number of completed requests in the kth period, $1 \leq k \leq K$

tol, convergence tolerance (e.g., 0.20)

Estimation of the Index of Dispersion I

1. get the busy time in the kth period $B_k := U_k \cdot T$, $1 \leq k \leq K$;
2. initialize $t = T$ and $Y(0) = 0$;
3. do
   a. for each $A_k = (B_k, B_{k+1}, \ldots, B_{k+j})$, $\sum_{i=0}^{j} B_{k+i} \approx t$,
      aa. compute $N_t^k = \sum_{i=0}^{j} n_{k+i}$;
   b. if the set of values N_t^k has less than 100 elements,
      bb. stop and collect new measures because the trace is too short;
   c. $Y(t) = Var(N_t^k)/E[N_t^k]$;
   d. increase t by T;

   until $|1 - (Y(t)/Y(t-T))| \leq tol$, i.e., the values of $Y(t)$ converge.
4. return the last computed value of $Y(t)$ as estimate of I.

Figure 3.11: Estimation of I from utilization samples.

3.3.2 Integrating I in Performance Models

In order to integrate the index of dispersion in queueing models, we model service times as a two-phase Markovian Arrival Process (MAP(2)) [67, 80, 15]. As shown in Chapter 2, a MAP(2) can be fitted with closed-form formulas (see Eqs.(2.3), (2.4), (2.5), and (2.6)) given the mean, SCV, skewness, and lag-1 autocorrelation coefficient ρ_1 of the measured service times [30, 17]. We use these closed-form formulas to define the MAP(2) as follows. After estimating the mean service time and the index of dispersion I of the trace, we also estimate the 95th percentile of the service times as we describe at the end of this subsection. Given the mean, the index of dispersion I, and the 95th percentile of service times, we generate a set of MAP(2)s that have $\pm 20\%$ maximal error on I, see [40, 4] for computational formulas of I in MAP(2)s. Among this set of MAP(2)s, we choose the one with its 95th percentile closest to the trace. Overall, the computational cost of fitting the MAP(2)s is negligible both in time and space requirements. For instance, the fitting of the MAP(2)s has been performed in MATLAB in less than five minutes for the experiments in this work. Occasionally, and only for certain combinations of I and 95th percentile, there may exist more than one MAP(2) with identical mean, I, and 95th percentile. We have not found this case during the experiments in this work, but in general we recommend to choose the MAP(2) with largest lag-1 autocorrelation since this results in a slightly more aggressive burstiness profile that provides conservative capacity planning estimates.

We conclude by explaining how to estimate the 95th percentile of the service times from the measured trace. We compute the 95th percentile of the measured busy times B_k in Figure 3.11 and scale it by the median number of requests processed in the busy periods. If the trace has high dispersion (e.g., $I >> 100$), this estimate is very accurate

because the n_k jobs that are served in the kth busy period receive a similar service time S_k and the busy time is therefore $B_k \approx n_k S_k$. This approximation consists in assuming that n_k is always constant and equal to its median value $med(n_k)$. Under this hypothesis the 95th percentile of B_k is simply $med(n_k)$ times the 95th percentile of S_k. Conversely, if the trace has low dispersion (e.g., $I < 100$), the estimation is inaccurate. Nevertheless, we observe that we can still use this simplification, because under low-burstiness conditions the queueing performance is dominated by the mean and the SCV of the distribution, and therefore a biased estimate of the 95th percentile does not have any appreciable effect on accuracy. In practice, we have found this estimation approach to be highly satisfactory for system modeling as shown by the experimental results reported in the next sections.

3.3.3 Impact of Measurement Granularity and Monitoring Windows

Starting from the MAP-based model defined in the previous section, we validate the accuracy of the new analytic model using the same experimental setup as in Section 3.2.4. We denote by $E[Z_{qn}]$ the think time used in the capacity planning queueing network model that represents the system presented in Section 3.2.4. For validation, we always compare the predictions of this model with a real experiment where the TPC-W has think time $E[Z]_{qn}$. The notation $E[Z_{estim}]$ denotes the TPC-W think time used in experiments to generate the traces from which we estimate I and the MAP(2)s. In general, $E[Z_{estim}]$ can differ from $E[Z_{qn}]$, e.g., if we want to explore the sensitivity of the system to different think times we may consider models with different $E[Z]_{qn}$, but the MAP(2)s are parameterized from the same experimental trace obtained for a certain $E[Z_{estim}] \neq E[Z_{qn}]$. A robust modeling methodology could predict well the performance of the system also for $E[Z_{qn}] \neq$

$E[Z_{estim}]$ and we are seeking for a robust characterization of the service processes which is insensitive to the value $E[Z_{estim}]$ that describes a characteristic of the arrival process to the multi-tier system, rather than a property of the servers.

In all validations, we set $E[Z_{qn}] = 0.5s$ and evaluate throughput and an increase of the number of EBs. The default think time value for the TPC-W benchmark is $7s$, but setting $E[Z_{qn}] = 7s$ we would need to set the number of EBs as high as 1200 to reach heavy-load. Unfortunately, no existing numerical approach can solve the model for exact solutions when the system has such a large number of EBs. Since in this work we are interested in validating models with respect to their exact accuracy, we have explored exact solutions in Section 3.2.4 by reducing the user think time to $E[Z_{qn}] = 0.5s$, such that the system becomes overloaded when the number of EBs is around $100 - 150$. Models with larger number of EBs should be evaluated with approximations, e.g., with the class of performance bounds presented in [15]. In the rest of chapter, we only consider queueing network models with $E[Z_{qn}] = 0.5s$. By building the underlying Markov chain and solving the system of linear equations, we solve the new analytic model and get the analytic results, see [15] for a description of the Markov chain underlying a MAP queueing network.

Here, we first present validation results on the browsing mix for different values of the measurement granularity $E[Z_{estim}]$. Since measurements should not interfere with normal server operations, we have set the monitoring window resolution of the Diagnostics tool to a standard $W = 5s$, which means that hundreds of requests may be served between the collection of two consecutive utilization samples. For instance, when the user think time in TPC-W is set to $E[Z_{estim}] = 0.5s$ and the number of EBs is 50, there are on average 465 requests completed in a monitoring window of $W = 5s$. A reduction of

the frequency of sampling makes it difficult to collect a large number of samples (e.g., tens of thousands), and this significantly reduces the statistical robustness of the index of dispersion estimates. Conversely, we have found that decreasing the mean throughput of the system by an increase of $E[Z_{estim}]$ can have beneficial effects on the quality of the index of dispersion estimation without having to modify the monitoring window resolution.

Figure 3.12 compares the analytic results with the experimental measurements of the real system for the browsing mix. A summary of the think time values used in the two models is given in Table 3.3. In all models, we set the mean user think time to $E[Z_{qn}] = 0.5s$ and vary the system loads with different EBs. To evaluate the effect of the measurement granularity on the analytic model, we have estimated two sets of MAP(2)s by using the measured traces from the experiments with 50 EBs and two different levels of measurement granularity, i.e., the user think time $E[Z_{estim}] = 0.5s$, and $E[Z_{estim}] = 7s$, respectively. As $E[Z_{estim}]$ increases, we are getting monitoring data of finer granularity, because in the same monitoring window W a smaller number of requests is completed. This makes the estimation of the variance of N_t in the algorithm in Figure 3.11 more accurate as the finer granularity reveals better the nature of the service times. This is intuitive, e.g., in the extreme case where $E[Z_{estim}]$ is so large that only a single request is completed during a single monitoring window W, then our measurement corresponds to a direct measure of the request service time and the estimation becomes optimal. Indeed, a large increase of $E[Z_{estim}]$ to this level would be unrealistic because it would hide possible slowdowns in service times that become evident only when several requests are served simultaneously, e.g., increased memory access times in algorithms due to an increase in size of shared data structures. For this reason, it is always advisable to increase $E[Z_{estim}]$

such that there are some tens of requests completed in a time window W during the experiment.

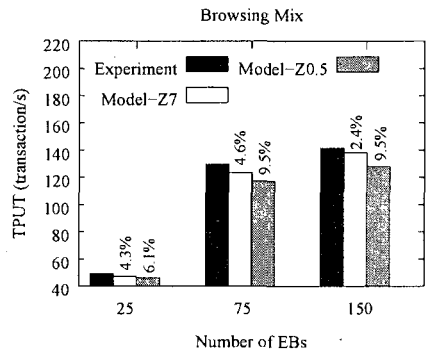


Figure 3.12: Comparing the results for the model which fits MAPs with different $E[Z_{estim}] = 0.5s$ and $E[Z_{estim}] = 7s$. On each bar, the relative error with respect to the experimental data is also reported.

	Queueing Network	*MAP(2) Estimation*
Model-Z0.5	$E[Z_{qn}] = 0.5s$	$E[Z_{estim}] = 0.5s$
Model-Z7	$E[Z_{qn}] = 0.5s$	$E[Z_{estim}] = 7s$

Table 3.3: Think time values considered in the accuracy validation experiments.

In Figure 3.12, the corresponding relative prediction error, which is the ratio of the absolute difference between the analytic result over the measured result, is shown on each bar. The figure shows that precision increases non-negligibly when a finer granularity of monitoring data is used. As the system becomes heavily loaded, the model with finer granularity (i.e., $E[Z_{estim}]$ as high as $7s$) dramatically reduces the relative prediction error to 2.4%.

3.3.4 Validation of Prediction Accuracy on Different Transaction Mixes

Figure 3.13 compares the analytical results with the experimental measurements of the real system for the three transaction mixes. The values of the index of dispersion for the front and the database service processes are also shown in the figure. Throughout all experiments, the mean user think time is set to $E[Z_{qn}] = 0.5s$; the MAP(2)s are obtained from experimental data collected with $E[Z_{estim}] = 7s$.

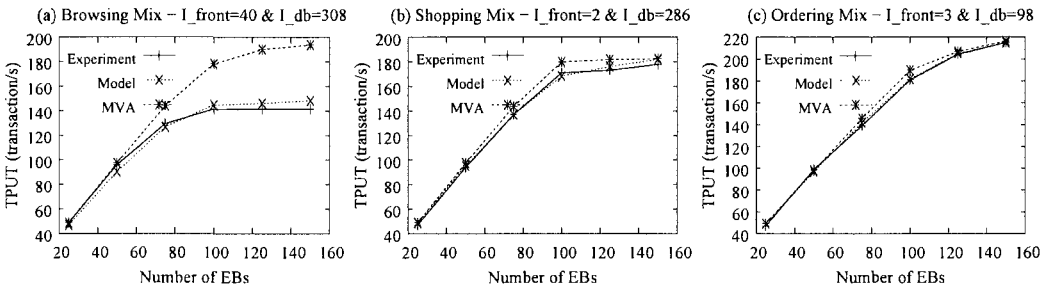


Figure 3.13: Modeling results for three transaction mixes as a function of the number of EBs.

Figure 3.13 gives evidence that the new analytic model based on the index of dispersion achieves gains in the prediction accuracy with respect to the MVA model on *all* workload mixes, showing that it is reliable also when the workloads are not bursty. In the browsing mix, the index of dispersion enables the queueing model to effectively capture *both* burstiness and bottleneck switch. The results of the proposed analytic model match closely the experimental results for the browsing mix, while remaining robust in all other cases.

The shopping mix presents an interesting case: as already observed in Section 3.2.4, the MVA model performs well on the shopping mix despite the existing burstiness because, regardless of the variation of the workload at the database server, the front server remains the major source of congestion for the system and the model behaves similarly to a MVA

model (i.e., there is no bottleneck switch).

In the ordering mix, the feature of workload burstiness is almost negligible and the phenomenon of bottleneck switch between the front and the database servers cannot be easily observed, see Section 3.2.2. For this case, MVA yields prediction errors up to 5%. Yet, as shown in Figure 3.13(b) and (c), our analytic model further improves MVA's prediction accuracy. This happens because the index of dispersion I is able to capture detailed properties of the service time process, which can not be captured by the MVA model.

All results shown in Figure 3.13 validate the analytic model based on the index of dispersion: its performance results are in excellent agreement with the experimental values in the system, and it remains robust in systems *with* and *without* the feature of workload burstiness and bottleneck switch.

3.4 Injecting Burstiness in the Arrival Process of Multi-tier Benchmarks

In this section, we propose a robust methodology to inject burstiness into the *arrival process* of TPC-W. In our method, we use the *index of dispersion* as a simple "turnable knob" to regulate the intensity of traffic burstiness in workload flows. Extensive research has been carried out in recent years on mechanisms to neutralize the impact of burstiness on web architectures. However, little research has been carried out on workload benchmarks that emulate the phenomenon of bursty traffic and that are also easily reproducible, scalable, and representative of real workloads. Here, we provide a new extension to the standard

TPC-W benchmark. This new extension enables testing and evaluation of system performance under reproducible and controllable bursty workloads, validation of efficiency of the corresponding management/provisioning solution, and comparison across different management solutions in a reproducible way.

3.4.1 Limitations of Standard TPC-W

Indeed, fluctuations of the number of jobs in TPC-W is regulated by the average user think time $E[Z]$, which represents the time between receiving a Web page and the following page download request. In this dissertation, we propose to inject burstiness into the incoming traffic by modifying the way think times are generated in the client machines. Think times in the standard TPC-W benchmark are drawn randomly from an exponential distribution that is identical for all clients [32]. Because of the memoryless property of the exponential distribution, this is equivalent to imposing that clients operate independently of their past actions. However, exponential think times are incompatible with the notion of burstiness for several reasons:

Temporal locality: intuitively, under conditions of burstiness, arrivals from different customers cannot happen at random instants of time, but they are instead condensed in short periods across time. Therefore, the probability of sending a request inside this period is much larger than outside it. This behavior is inconsistent with classic distributions considered in performance engineering of web architectures, such as Poisson, hyper-exponential, Zipf, and Pareto, which all miss the ability of describing temporal locality within a process.

Variability of different time scales: Variability within a traffic burst is a relevant characteristic for testing peak performance degradation. Therefore, a benchmarking model for

burstiness should not only create bursts of variable intensity and duration, but also create fluctuations within a burst. This implies a hierarchy of variability levels that cannot be described by a simple exponential distribution and instead requires a more structured arrival process.

Lack of aggregation: in the standard TPC-W, each thread on the client machines uses a dedicated stream of random numbers, thus think times of different users are always independent. This is representative of normal traffic, but fails in capturing the essential property of traffic burstiness: users act in an aggregated fashion which is mostly incompatible with independence assumptions. Here, we do not assume that users explicitly coordinate their submission of requests. Instead, we impose a loose synchronization which leaves large room for fluctuations within a traffic burst. Yet, this is a common problem to many request generation techniques based on the user-equivalents approach [12].

In order to address all above points, we propose to regulate the arrival rate of requests to the system using a class of Markov-modulated processes known as Markovian Arrival Processes (MAPs) [67], which have the ability of providing variability at different levels as well as temporal locality effects. Here, we depart from the traditional approach to model increased load in the systems by simply increasing the fixed number of jobs (connections) in the system. Instead, burstiness can occur now in a system with few or many connections by simply handling the duration of user think time. In particular, we propose a new module that creates a set of identical MAPs which are replicated over the different client machines and here *shared* for generation of think times by all clients running on that particular client machine. We show the fluctuation of loads in client-server systems via this new module in our experiments, see Figures 3.20 and 3.23. We stress that this new

module can be added to any benchmark with a closed loop structure.

3.4.2 Using MAP to Model Traffic Bursts

A MAP can be seen as a simple mathematical model of a sequence of user think times, for which we can accurately shape distribution and correlations between successive values. Here, we refer the reader to Chapter 2 for the detailed properties of the MAP. Correlations among consecutive think times are instrumental to capture periods of the time series where think times are consecutively small and thus a burst occurs, as well as to determine the burst duration.

We use a class of MAPs with two states only, one responsible for the generation of "short" think times implying that users produce closely spaced arrivals, possibly resulting in bursts, while the other is responsible for the generation of "long" think times associated to periods of normal traffic. In the "short" state, think times are generated with mean rate λ_{short}, similarly they have mean rate $\lambda_{long} < \lambda_{short}$ in the "long" state. We explain in Section 3.4.3 how to assign values for λ_{short} and λ_{long} starting from standard TPC-W measurements. In order to create correlation between different events, after the generation of a new think time sample, our model has a probability $p_{s,s}$ that two consecutive think times are short and a different probability $p_{l,l}$ of two consecutive think times being both long. The probability $p_{s,l} = 1 - p_{s,s}$ (resp., $p_{l,s} = 1 - p_{l,l}$) determines the frequency of jump from the short (resp., long) state to the long (resp., short) state. Thus, the values of $p_{s,s}$, $p_{s,l}$, $p_{l,s}$ and $p_{l,l}$ shape the correlations between consecutive think times and are instrumental to determine the duration of the traffic burst, see the next subsection for further details. Henceforth, we focus only on the independent values $p_{l,s}$ and $p_{s,l}$.

Figure 3.14 summarizes the traffic burst model described above. Note from the pseudo code that the problem of variability of different time scales is solved effectively in MAPs: if the MAP is in a state i, then samples are generated by an exponential distribution with rate λ_i associated to state i. This creates fluctuations within the traffic burst. It is also compatible with the observations in Section 3.4.1 against the exponential think times because the probability of arrival inside the traffic burst is larger than outside it, thanks to the state change mechanism that alters the rate of arrival from λ_{long} to λ_{short}.

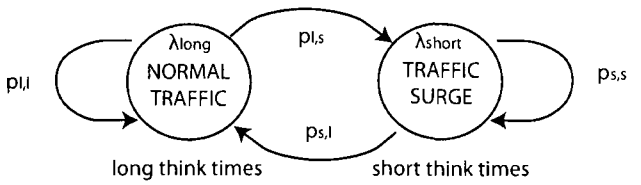


Figure 3.14: Model of traffic bursts based on regulation of think times

3.4.3 Integrating Burstiness in TPC-W

To avoid inter-machine communication and keep the modifications to TPC-W simple, we propose to use a *shared* MAP process to draw think times for all users emulated on the same client machine[1]. This solves immediately the problem of independence between requests of different users and is a paradigm change, because *we no longer model in the TPC-W benchmark the individual think times; instead we shape directly the behavior of all clients.*

The most complex aspect of this new approach is the parameterization of the MAP process: how should we define the arrival stream in order to stress effectively a system?

[1]Often, TPC-W setup involves multiple client machines to generate enough user requests to load the benchmarked system.

The fundamental problem is how to determine a parameterization of (λ_{long}, λ_{short}, $p_{l,s}$, $p_{s,l}$) that produces a sequence of bursts in the incoming traffic. Further, this parameterization must remain representative of a realistic (i.e., probabilistic, non DDoS-like) scenario. Henceforth, we assume that the user gives to the modified TPC-W benchmark the desired values of the mean think time $E[Z]$ and of the index of dispersion I which specifies the burstiness level. The benchmark automatically generates a parameterization of (λ_{long}, λ_{short}, $p_{l,s}$, $p_{s,l}$) capable of stressing the system. We also assume that the standard TPC-W benchmark has been previously run on the architecture and that the mean service demand $E[D_i]$ of each server i has been estimated from utilization measurements, e.g., using linear regression methods [102, 14].

The mean think time $E[Z]$ can be parameterized as in the standard TPC-W benchmark, i.e., $E[Z] = 7$ seconds, while the index of dispersion I, is the additional parameter that can be used to tune the level of burstiness in workloads. To fully define the properties of MAP think times other than the mean $E[Z]$, our approach starts by the following parameterization equations:

$$\lambda_{short}^{-1} = (\textstyle\sum_i E[D_i])/f, \tag{3.2}$$

$$\lambda_{long}^{-1} = f \max(N(\textstyle\sum_i E[D_i]), E[Z]). \tag{3.3}$$

Here, $f \geq 1$ is a free parameter, N is the maximum number of client connections considered in the benchmarking experiment, $\sum_i E[D_i]$ is the minimum time taken by a request to complete at all servers, and $N(\sum_i E[D_i])$ provides an upper bound to the time required by the system to respond to all requests. Eq.(3.2) states that, in order to create bursts, the think times should be shorter than the time required by the system to respond to

requests. Thus, assuming that all N clients are simultaneously waiting to submit a new request, one may reasonably expect that after a few multiples of λ_{short}^{-1} all clients have submitted requests and the architecture has been yet unable to cope with the traffic burst. Conversely, Eq.(3.3) defines think times that on average give to the system enough time to cope with any request, i.e., the normal traffic regime. Note that the condition $\lambda_{long}^{-1} \geq fE[Z]$ is imposed to ensure that the mean think time can be $E[Z]$, which would not be possible if both $\lambda_{short}^{-1} > \lambda_{long}^{-1} > E[Z]$ since $f > 1$ and in MAPs the moments $E[Z], E[Z^2], \ldots$ are:

$$E[Z^k] = k! \left(\frac{p_{l,s}}{p_{l,s} + p_{s,l}} \lambda_{short}^{-k} + \frac{p_{s,l}}{p_{l,s} + p_{s,l}} \lambda_{long}^{-k} \right). \tag{3.4}$$

The above formula for $k = 1$ implies that $E[Z]$ has a value between λ_{short}^{-1} and λ_{long}^{-1}, which is not compatible with $\lambda_{short}^{-1} \geq \lambda_{long}^{-1} \geq fE[Z]$. According to Eq.(3.4), the MAP parameterization can always impose the user-defined $E[Z]$ if

$$p_{l,s} = p_{s,l} \left(\frac{\lambda_{long}^{-1} - E[Z]}{E[Z] - \lambda_{short}^{-1}} \right), \tag{3.5}$$

and we use this condition in the modified TPC-W benchmark to impose the mean think time.

In order to fix the values of $p_{s,l}$ and f in the above equations, we first do a simple search on the space ($0 \leq p_{s,l} \leq 1$, $f \geq 1$) where at each iteration we check the value of the index of dispersion I and lag-1 autocorrelation coefficient ρ_1 from the current values of $p_{s,l}$ and f. We stop searching when we find a MAP with an I that is within 1% of the target user-specified index of dispersion and the lag-1 autocorrelation is at least $\rho_1 \geq 0.4$ in order to have consistent probability of formation of bursts within short time periods.

We remark that the threshold 0.4 has been chosen since it is the closest round value to the maximum autocorrelation that can be obtained by a two-state MAP. The index of dispersion of the MAP can be evaluated at each iteration as [16, 67]:

$$I = 1 + \frac{2\, p_{s,l} p_{l,s} (\lambda_{short} - \lambda_{long})^2}{(p_{s,l} + p_{l,s})(\lambda_{short} p_{s,l} + \lambda_{long} p_{l,s})^2}, \tag{3.6}$$

while the lag-1 autocorrelation coefficient is computed as

$$\rho_1 = \frac{1}{2}(1 - p_{l,s} - p_{s,l}) \left(1 - \frac{E[Z]^2}{E[Z^2] - E[Z]^2}\right), \tag{3.7}$$

where $E[Z^2]$ is obtained from Eq.(3.4) for $k = 2$. We remark that if no MAP exists with at least $\rho_1 \geq 0.4$, then the benchmark should search for the MAP with largest ρ_1 in order to facilitate the formation of bursts which persists over several units of time.

3.4.4 Case Study: TPC-W

We exemplify the effectiveness of this new methodology by introducing a new module into the TPC-W, a benchmark that is routinely used for capacity planning of e-commerce systems. We define a modified TPC-W benchmark where sequences of bursts with different intensities and durations are created.

For each transaction mix, we run a set of experiments with different number of maximum client connections (fixed within each experiment) ranging from 200 to 1200. As a result, we evaluate the new methodology under various system loads with utilization levels at the front and the database servers within the range of 12%-98% and 6%-74%, respectively. In all experiments, the mean user think time is set to $E[Z] = 7$ seconds, which is the default value for the TPC-W benchmark. We use a 2-state MAP to generate

the user think times as described in the previous subsection. Our experiments are done with two different MAPs that result in the index of dispersion equal to $I = 400$ (mild burstiness) and $I = 4000$ (severe burstiness).

For comparison, we also do experiments with the standard configuration, i.e., think times are exponentially distributed with mean $E[Z] = 7$ seconds and squared coefficient-of-variation $SCV = 1$. All experiments run for 3 hours each, where the first 5 minutes and the last 5 minutes are considered as warm-up and cool-down periods and thus omitted in the measurements.

Figure 3.15 demonstrates the arrival processes to the system under the shopping mix. The results for the browsing and the ordering mixes are qualitatively the same. In this figure, we depict the number of arriving clients to the system (i.e., the front server) in monitoring windows of 1 second. In the standard TPC-W experiment, there is no burstiness in the number of arriving clients, which remains stable around 150, see Figure 3.15(a). When we adopt two-state MAPs in think times, bursts are generated in the arrivals as shown by periods of continuous peak arrival rates, see Figure 3.15(b) and Figure 3.15(c). We stress that all three arrival processes have the same mean. As the index of dispersion increases from $I = 400$ to $I = 4000$, there are sharp bursts in the number of active clients, consistently with our purpose to "create" bursty conditions.

Average Performance

Figure 3.16 presents the *average latency* for a client transaction, which is the interval from the moment when the client sends an HTTP request to the moment when an entire HTTP web page (including embedded objects) is retrieved. We first direct the reader's attention

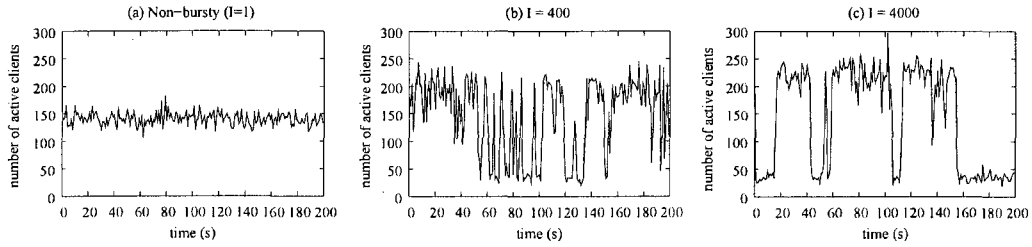


Figure 3.15: Arriving clients to the system (front server) for the shopping mix with (a) non-bursty (standard TPC-W), (b) $I = 400$, and (c) $I = 4000$ in user think times, where the maximum number of client connections is set to $N = 1000$.

to the system performance under the standard TPC-W experiment (i.e., exponential think times, labeled *non-bursty* in Figure 3.16, see all solid curves). As shown in Figure 3.16 across all workloads, average latencies increase as the maximum number of client connections increases. Especially for the browsing mix, the latency becomes two orders of magnitude larger when N is increased from 200 to 1200. This is due to the presence of burstiness in the service times at the database server, which dramatically degrades the overall system performance, see more details in [63]. For the shopping and the ordering mixes, there is no burstiness in neither the front nor the database service processes, although these two workload mixes are highly variable. Consequently, a large number of clients does not deteriorate performance as severely as in the browsing mix.

When burstiness is injected into the arrival flows, the overall system performance becomes significantly worse for all three transaction mixes. For instance, for the shopping and the ordering mixes, when the index of dispersion in the two-state MAP for user think times is $I = 4000$ and the maximum number of client connections is beyond 600, the average latency is increased by at least 13 times and 35 times, respectively, compared to the non-bursty case. As the index of dispersion decreases, e.g., $I = 400$, the degradation caused by burstiness on the overall system performance becomes weaker yet visible as

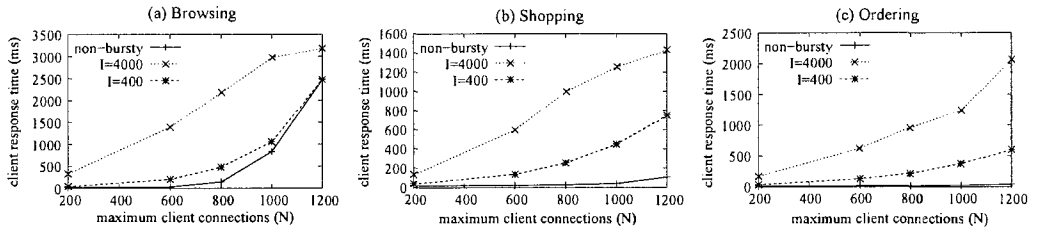


Figure 3.16: Average latencies as a function of the number of maximum client connections N for (a) browsing mix, (b) shopping mix, and (c) ordering mix with non-bursty and bursty of $I = 4000$ and 400 in the user think times.

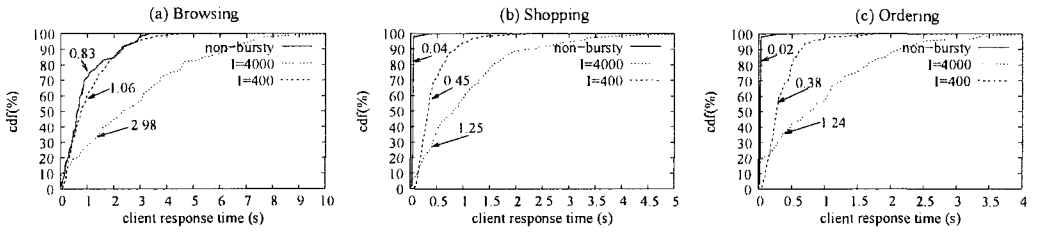


Figure 3.17: CDFs of latencies for (a) browsing mix, (b) shopping mix, and (c) ordering mix with non-bursty and bursty of $I = 4000$ and 400 in user think times, where $N = 1000$ and the corresponding average latencies are also marked.

latencies remain at least 6 times slower. For the browsing mix, the newly injected burstiness in arrivals further deteriorates average latencies. Yet, as the maximum number of client connections reaches 1200, the system performance under $I = 400$ is similar to the non-bursty case. This happens because the system is already overloaded, regardless of burstiness.

In addition to average latency values, we also evaluate the distribution of latencies. Figure 3.17 shows the cumulative distribution function (CDF) of the latencies of the three transaction mixes when $N = 1000$. The corresponding average latencies are also marked in the figure. With bursty arrivals, the mass of clients experience significantly worse performance and much longer tails in the latency distributions. This essentially argues that QoS guarantees cannot be given for significant percentiles of the workload and further

highlights the pressing need to evaluate client-server systems under bursty conditions.

Transient Performance

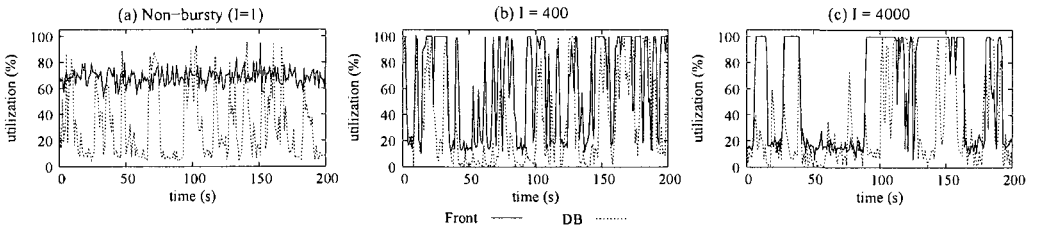


Figure 3.18: Shopping mix: transient utilizations at the front server and the database server for (a) non-bursty, (b) $I = 400$, and (c) $I = 4000$ in user think times, where $N = 1000$.

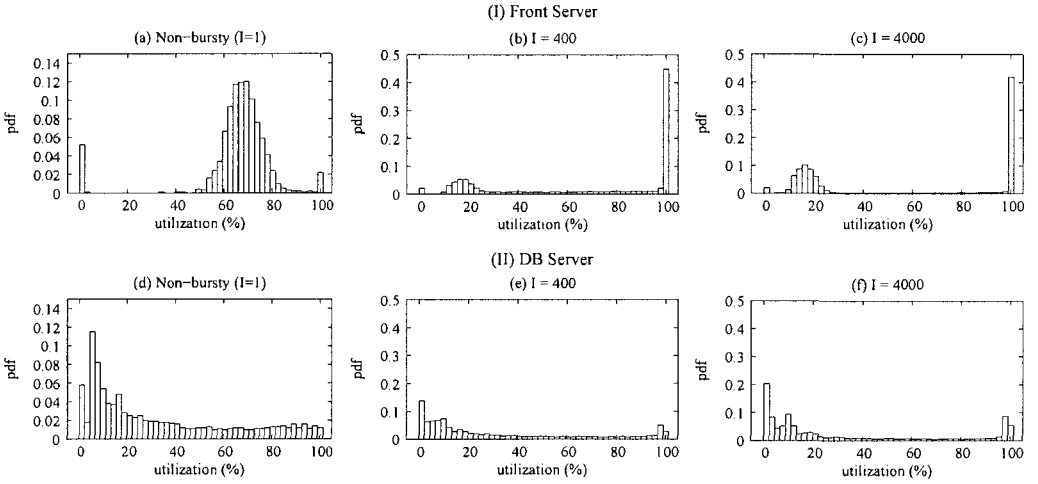


Figure 3.19: Shopping mix: PDFs of utilizations at (I) the front server and (II) the database server for non-bursty, $I = 400$, and $I = 4000$ in user think times, where $N = 1000$.

Here, we examine the performance metrics including the transient CPU utilizations of the front and the database servers, the empirical frequencies of CPU utilizations, and the transient number of active clients in the system as given by the summation of queue lengths at the front server and at the back-end database. The maximum number of client connections in the system is fixed to $N = 1000$.

The Shopping Mix. We first present CPU utilizations of the front and the database

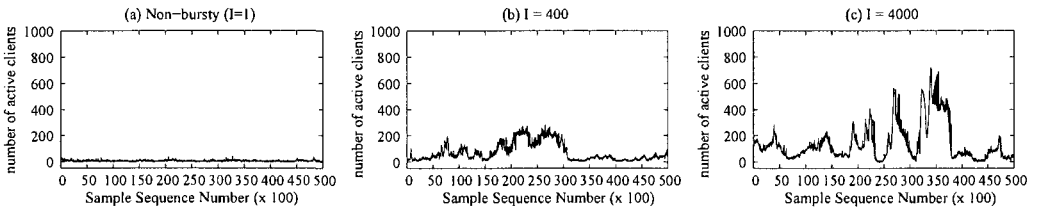


Figure 3.20: Shopping mix: transient number of active clients in the system, i.e., summation of queue length at the front and the database servers, for (a) non-bursty, (b) $I = 400$, and (c) $I = 4000$ in user think times, where $N = 1000$.

servers across time for the shopping mix. We remark that the results for the ordering mix have qualitatively the same trends. In this workload mix, there is no burstiness in either the front or the database service processes. Therefore, if burstiness in CPU utilizations exists, then this must be a direct result of bursts. As shown in Figure 3.18(a), when there are no traffic bursts, the utilization at the front server remains stable around 70% while for the database server the utilization levels vary from 10% to 80%, due to high variability in its service times. When bursts are generated, the phenomenon of stable utilizations at the front server disappears. Instead, we observe very bursty CPU utilizations at the front server, where the server remains fully utilized (i.e., 100%) for some periods, but then it sharply drops to only 20% during other periods, see Figure 3.18(b). Meanwhile, the range for the utilizations at the database server is further enlarged up to even 100%. As the intensity of traffic bursts increases, the trend for the front server being either overloaded or lightly loaded becomes more evident, see Figure 3.18(c).

Figure 3.19 illustrates the empirical frequencies (i.e., empirical PDF) of CPU utilizations at both the front server (see the first row in the figure) and the back-end database (see the second row in the figure). If the arrival process to the system is not bursty, then there is a large mass around 60%-80% in the distribution of utilizations at the front

server, which is consistent with the transient results shown in Figure 3.18(a). For the two cases with burstiness in the arrival process, the distributions are bimodal, an effect that is further accentuated as burstiness increases, see Figure 3.19(c) and Figure 3.19(f).

To better understand how traffic bursts are generated by using two-state MAPs in user think times, we present the number of active clients (i.e., summation of queue lengths at the front and the database servers) across time for the shopping mix, see Figure 3.20. This performance metric directly indicates how many active clients are in the system waiting for service. First, as shown in Figure 3.20(a), we cannot observe any burstiness in the overall queue length, despite the fact that the shopping mix workload is highly variable. When the two-state MAPs with $I = 400$ and $I = 4000$ are adopted for user think times, the number of active clients in the system fluctuates dramatically. When $I = 4000$, the system is congested with more than 700 clients for some periods, while it sharply drops to as low as 10 clients during other periods. This exactly matches the burstiness in the CPU utilizations of the front and the database servers.

The Browsing Mix. We now turn to investigate the browsing mix. The distinct difference of this browsing versus the shopping or the ordering is that there is burstiness in the flows which originates in the database service process. We direct the reader to [63] for detailed discussion on this phenomenon.

In the browsing mix, even if no additional burstiness is injected into the system (i.e., think times are exponential), there does exist burstiness in the CPU utilizations of the front and the database servers, see Figure 3.21(a). If there is burstiness in think times as well, the burstiness in CPU utilizations becomes more prominent.

We depict the empirical PDF of the CPU utilizations for the browsing mix in Fig-

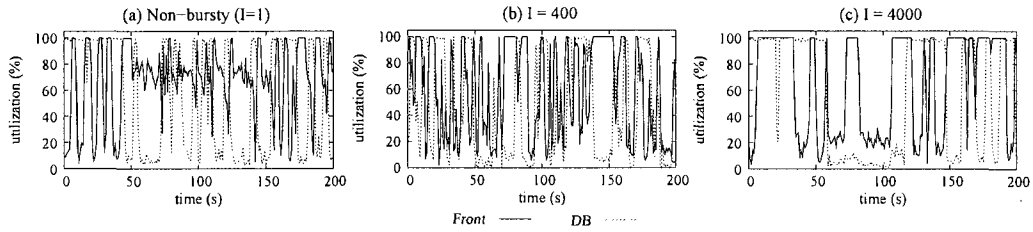


Figure 3.21: Browsing mix: transient utilizations at the front server and the database server for (a) non-bursty, (b) $I = 400$, and (c) $I = 4000$ in user think times, where $N = 1000$.

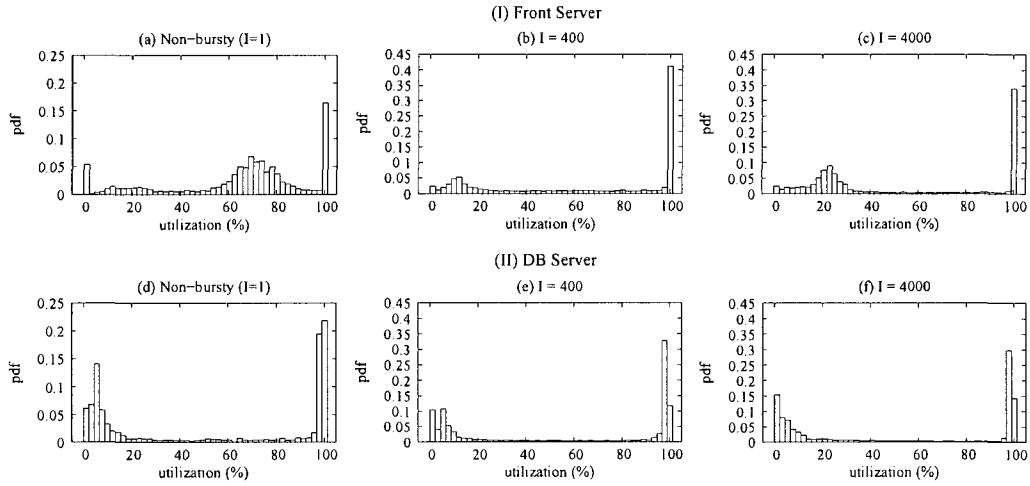


Figure 3.22: Browsing mix: PDFs of utilizations at (I) the front server and (II) the database server for non-bursty, $I = 400$, and $I = 4000$ in user think times, where $N = 1000$.

ure 3.22 for $N = 1000$. Different from the shopping mix, the database utilizations have a bimodal distribution with two peaks around 8% and 100%, this is due to the database correlated service process, see Figure 3.22(d). For the front server, although most of CPU utilizations are still gathered around 60%-80%, the probabilities of having the front server fully utilized (100%) and fully idle (0%) are as high as 0.16 and 0.06, respectively. When traffic bursts are generated, CPU utilizations at the front server and the back-end database become extreme, i.e., either very high or very low.

Figure 3.23 illustrates the number of active clients in the system across time for the

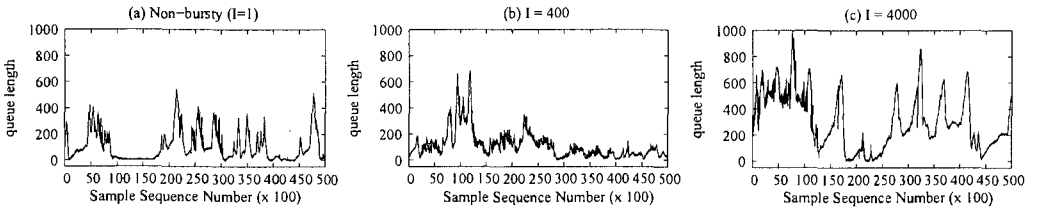


Figure 3.23: Browsing mix: transient number of active clients in the system, i.e., summation of queue length at the front and the database servers, for (a) non-bursty, (b) $I = 400$, and (c) $I = 4000$ in user think times, where $N = 1000$.

browsing mix in the network with $N = 1000$. The observation of the transient number of active clients is consistent with the transient CPU utilizations: under the non-bursty case, the curve of the number of active clients is no longer flat but contains a lot of spikes caused by the burstiness in the database service process; while the additional burstiness in the arrival process continuously increases the spikes in the number of active clients, making the system performance erratic and extremely variable.

3.5 Chapter Summary

Today's IT and Services departments are faced with the difficult task of ensuring that enterprise business-critical applications are always available and provide adequate performance. Predicting and controlling the issues surrounding system performance is a difficult and overwhelming task for IT administrators. With complexity of enterprise systems increasing over time and customer requirements for QoS growing, effective models for quick and automatic evaluation of required system resources in production systems become a priority item on the service provider's "wish list".

In this chapter, we have presented a solution that models burstiness in the *service process* of multi-tier systems. Most importantly, the model parameterization is done by

inferring essential process information from *coarse* measurements in a real system. After giving quantitative examples of the importance of integrating burstiness in performance models and pointing out its role relatively to the bottleneck switching phenomenon, we show that coarse measurements can still be used to parameterize queueing models that effectively capture burstiness and variability of the service process. The parameterized queueing model can thus be used to closely predict performance in systems even in the very difficult case where there is persistent bottleneck switch among the various servers. Detailed experimentation on a multi-tier system using the TPC-W benchmark validates that the proposed technique offers a robust solution to predict performance of systems subject to burstiness and bottleneck switching conditions.

We have also provided a robust methodology to inject burstiness into the *arrival process* of the traditional client-server benchmarks, e.g., TPC-W, that can be of great practical use for assessing the effectiveness of mechanisms that counteract burstiness. We integrate the methodology in the well established TPC-W benchmark. Our methodology injects burstiness into the arrival process of the server in a controllable way using simple parameterization. Extensive experimentation in a real testbed demonstrates the effectiveness and robustness of the proposed methodology and further demonstrates the importance of evaluating the system under bursty conditions as its performance decidedly worsens as burstiness increases. The code of the proposed extensions to TPC-W benchmark are presented in Appendix B.

Chapter 4

General Dependence-driven Scheduling Policies

In this chapter, we leverage on the temporal dependence in service times and show how temporal dependence can be exploited to forecast future service requirements of requests. If the dependence structure is strong, then it is likely that a large request is followed by another large one, that is, requests with large service demands appear clustered together. This information can be used as an additional criterion for resource allocation.

In this chapter, we show that significant performance gains that reduce the probability of having the system unavailable can be obtained by delaying selected requests Request delaying in a server may yet result in slower response times at that resource, but significant improvement is observed throughout the rest of the network, allowing delay-based scheduling to reduce the overall end-to-end response times. Furthermore, the lower response times let the system sustain more customers, therefore improving its overall availability.

In this chapter, we first focus on the difficult case where workload processing is manda-

tory and work reduction techniques such as request drop cannot be applied. In Section 4.2, we propose a new delay-based scheduling policy, called SWAP. SWAP is a fully measurement-based policy that classifies (i.e., "predicts") requests as short or long based on the temporal dependence of the workload service process and approximates the shortest job first (SJF) scheduling without requiring any knowledge of job service times. Then, we develop two extensions of SWAP that infinitely delay (i.e., drop) a portion of the workload in order to meet pre-defined quality-of-service levels, see Section 4.3.

4.1 Related Work

There is a large body of literature on scheduling policies that has been developed over the years (see [31] and [29] and references therein). Recently, Friedman and Henderson introduce a preemptive scheduling policy for Web servers in [31]. This new policy called Fair Sojourn Protocol (FSP) provides both efficiency and fairness for the sojourn time of the jobs. The Priority-based Blind Scheduling (PBS) policy approximates the existing standard blind scheduling policies, e.g., FCFS, PS, and LAS, by tuning a single parameter [29]. The Generalized Processor Sharing (GPS) policy is studied in the literature [52]. For a two-class GPS system, the admission region is selected for the general Gaussian traffic sources which contain the service processes with both long-range dependence and short-range dependence. However, to our best knowledge, no existing policy considers the structure of temporal locality in scheduling for systems.

Several works have investigated the idea of using measured temporal dependence in capacity control policies for the networking. In [37], a general framework for measurement-based admission control is introduced. Admission decisions are taken by means of an

approximate Gaussian model of the aggregated traffic which is parameterized by the measured mean, variance and correlation of the superposed flows. Similar approaches appear frequently in the networking literature, e.g., for bandwidth allocation in VBR traffic [19], for data communications over CDMA mediums [100], and for general self-similar multiplexed traffic modeled as fractional Brownian motion (fBm) [95]. However, these works differ substantially in the scope and approach of this dissertation for several reasons. First, network flows can have highly-variable bandwidth requirements that are non-stationary and difficult to model outside heavy traffic or asymptotic regimes; instead service in systems typically shows consistent functional forms which are easier to model and can be exploited effectively to control system load. Another important difference is that network traffic is often modeled as a superposition of flows which share the available bandwidth according to a discriminatory or generalized process-sharing policy; this assumption is instead often unrealistic in systems, e.g., when the scheduling discipline is approximately first-come first-served (FCFS). FCFS scheduling is also found in networks, e.g., in ATM communication, but the service time distributions are here usually deterministic or Erlang, whereas high job size variability in systems is a fundamental factor of congestion.

4.2 Delay-Based Scheduling Policy: SWAP

In this section, we introduce SWAP, a new delay-based scheduling policy that improves performance and availability in systems with temporal dependent workloads. The basic idea behind SWAP can be summarized as follows. Consider a system processing jobs with a first-come first-served (FCFS) scheduling policy. Assume initially that job size information is available to the scheduler. If we want to maximize performance given that

the future instants of new job arrivals are unknown, then the optimal scheduling is shortest job first (SJF) as it is well-know from classic scheduling theory [82]. That is, if the resource has K enqueued jobs having ordered service times S_k, $1 \leq k \leq K$, being S_1 the service time required by the job at the head of the queue, then the total completion time $C(T)$ under the FCFS discipline is

$$C(T) = KS_1 + (K-1)S_2 + \ldots + S_K,$$

which is immediately minimized if $S_k \leq S_{k+1}$, i.e., when short jobs are served first.

Outside the above assumptions, SJF is not in general optimal, yet provides significant gains with respect to simpler scheduling policies such as FCFS. The well-know problem of SJF is that it requires information on the job service times, which in practice may not be available. We therefore investigate how the performance of SJF could be approximated with an online policy that does not require a priori knowledge of job duration. The basic idea behind SWAP is to use the measured autocorrelation of the service times to estimate this missing information. Once these reliable estimates of the job service times are available, we delay large jobs up to a fixed number of times by putting them at the tail of the queue. In such a way, long jobs are more likely to be served after most short jobs have been completed. Estimated-short jobs are not delayed by SWAP.

Summarizing, the basic ideas of SWAP are as follows:

1. approximate the behavior of the SJF scheduling discipline by proper use of job delaying;

2. estimate from the process temporal dependence, as modeled by the correlation between successive service time values, the expected service times of the jobs waiting

in queue.

SWAP does *not* assume any a priori knowledge of the length of any of the enqueued jobs. The system knows the exact service time received by a job only *after* the job completes execution. Estimation of service times for the remaining jobs is based only on the past history of the system. We also stress that we provide mechanisms to avoid job starvation. In the next subsections we detail the implementation of the SWAP policy.

4.2.1 Forecasting Job Service Times

The effectiveness of the new proposed policy depends on the accuracy of forecasting job service times. If prediction is done effectively, long jobs to be delayed can be accurately identified and SWAP performs optimally.

Exploiting Service Time Variability

Our service time forecasting relies on two system aspects: service time variability and temporal dependence of workloads. Concerning the former, we leverage on the fact that service time distributions found in systems are typically characterized by high variance [6, 78], therefore the discrimination between small and large service times can be performed effectively and can be used to improve performance. In particular, SWAP uses a large-job threshold

$$LT = \mu^{-1}(1 + k \cdot CV), \tag{4.1}$$

where μ^{-1} is the mean service time at the resource, CV is the coefficient of variation of service times, and $k \geq 1$ is a constant determined online. If a job service time is greater than LT, then SWAP regards the job as "long" (also referred throughout the chapter as

"large"). Otherwise, SWAP classifies it as "short". Note that the policy can successfully measure the parameters for computing LT in an online fashion, i.e., the mean μ^{-1} and the coefficient of variation CV of the service times are continuously updated in SWAP using Welford's one-pass algorithm [96].

Exploiting Temporal Dependence

Given a classification into large and short jobs, the next step to effective forecasting is to exploit the structure of temporal dependence in order to "guess" if a job in the queue is long or short. This is the critical information needed to approximate the behavior of SJF scheduling. We assume that the scheduler is able to measure correctly the service times of jobs completed by the server, which can be easily implemented in most systems. Let T be the time instant in which a forecasting decision is needed, which in SWAP always corresponds to the departure instant of a *long* job departing from the queue. Also assume that during the period $[T - T_W, T]$, where T_W, $0 \leq T_W \leq T$, is an update window monitoring past history, the system has completed n jobs with service times $S_1, S_2, \ldots, S_n$. Given the sequence $\{S_i\}$, $1 \leq i \leq n$, our forecasting is based on the estimates of the conditional probabilities as follows:

$$P[L|L]_j = P[S_{t+j} \geq LT | S_t \geq LT],$$

$$P[S|L]_j = P[S_{t+j} < LT | S_t \geq LT] = 1 - P[L|L]_j,$$

which are computed using the service times $S_t \in \{S_i\}$ for $t = 1, \ldots, n-j$. Here j is called the lag of the conditional probability and denotes the distance between the service completions considered in the conditional probabilities. Given that the last completed job

is long, $P[L|L]_j$ measures the fraction of times that the j-th job that had arrived after it is also long; similarly, $P[S|L]_j$ estimates how many times the lag-j arrival is instead short. Using these estimates, we forecast that the lag-j arrival after the last completed job is going to receive large service time if the following condition holds:

$$P[L|L]_j \geq P[S|L]_j, \tag{4.2}$$

i.e., there is higher probability that the j-th arrival is going to be long than to be short. SWAP is triggered only when the last finished job is long; therefore, since we focus on systems with finite buffers, i.e., systems with constant population N, we *only* make use of the conditional probabilities $P[L|L]_j$, for $1 \leq j < N$.

An example that builds intuition on the tight relation between our forecasting approach and temporal dependence in service times has been shown in Figure 2.1 of Chapter 2. This example illustrates that workloads, for which service times are independent and no temporal locality exists, cannot be used to forecast future service requirements. Conversely, dependent service processes found in systems are best-fit to predict future service requirements. We exploit in SWAP this property to approximate the behavior of SJF scheduling.

4.2.2 The Delaying Algorithm: SWAP

We now describe SWAP in detail. For presentation simplicity, we assume here that the large threshold LT that is fundamental for forecasting is given; in the next subsection, we present how SWAP self-adjusts LT on-the-fly, i.e., no a priori knowledge of LT is required and SWAP becomes truly autonomic.

Upon the completion of a long job, the entire queue is scanned and the size of the j-th queued job is predicted by using the conditional probabilities as described in the previous subsection. If the j-th job is estimated as large, SWAP marks it as such. All jobs that are marked long are delayed by moving them at the end of the queue. After all jobs in the queue have been examined and long jobs have been delayed, SWAP admits for service the first job in the queue. Delaying is not triggered again before completion of another long job.

We point out that jobs are "reshuffled" in the queue based on their anticipated service times; the order of the jobs in the service process is therefore altered (attempting to approximate SJF scheduling) and this modifies both the throughput at the queue and the autocorrelation of the process. Concerning the latter, we point to [4] for an accurate analysis of the effects of shuffling in stochastic processes that can be modeled using Markovian methods.

SWAP does not re-forecast the length of a job whose service time has been already forecasted to be long. This is done by recording an absolute arrival index A_i for each job. That is, once a job has been marked as long it remains as such for all the duration of its stay in the queue and is never forecasted as short in successive activations of SWAP; the same property holds also for short jobs. We apply the conditional probabilities on the sequence of jobs in the queue obtained by ordering the jobs according to the arrival indexes only.

To avoid starvation of long jobs, we introduce the *delay limit* D, i.e., the maximum number of times a single job can be delayed. When the number of times a job has been delayed is more than D, the policy does not delay this job any longer and allows it to

wait for service in its current position in the queue. Figure 4.1 summarizes the above discussion and gives the pseudocode of SWAP.

4.2.3 Self-Adjusting the Threshold LT

Now, we discuss how SWAP adjusts the threshold LT for large values, aiming at controlling the strength of delaying to strike a good balance between being too aggressive or too conservative. Intuitively, when the threshold LT is too large, the policy becomes conservative by delaying few long jobs, the performance improvement is then negligible. Conversely, when LT becomes too small, more jobs (even short ones) are delayed and therefore throughput is reduced. As a result of this, performance may be improved very little. Therefore, the choice of an appropriate large threshold LT is critical for the effectiveness of SWAP.

As observed in Section 4.2.1, the computation of LT is a function of the updating window T_W used by SWAP. We express T_W as the maximal time period in which the system has completed exactly W requests; in the experiments presented here, we set $W = 100,000$. The algorithm in Figure 4.2 describes how the threshold LT is dynamically adjusted every W requests. At the end of a period of length T_W, we update LT while keeping as upper and lower bounds for its value the 90th and the 50th percentiles of the observed service times in T_W. Indeed, whenever specific information on the workload processed by a system is available, these values can be increased or decreased according to the characteristics of the workload. The threshold LT is updated by assuming that the value of the conditional probability $P[L|L]_j$ at some large lag j is representative of the overall tendency of the system to delay jobs.

In the implementation, we adjust the parameter k which defines $LT = \mu^{-1}(1 + k \cdot CV)$ with step adj according to the following scheme. Let QT be the current queue-length at the server with SWAP scheduling. We evaluate $P[L|L]_j$ for the large lag $j = \lfloor QT/2 \rfloor$ and if $P[L|L]_j \geq P[S|L]_j$, then SWAP is assumed to be too aggressive, since it may delay at the next round up to $\lfloor QT/2 \rfloor$ jobs. Here, we implicitly assume that the conditional probabilities $P[L|L]$ are decreasing in j which indeed is the typical case for workloads where large service times are a minority compared to the small service times. In this case we set $k = k + adj$, which reduces the number of jobs identified as long. As a result, we can avoid half of total requests waiting in the queue to be delayed. A similar procedure is done for the case $j = \lfloor QT/10 \rfloor$, where if $P[L|L]_j \geq P[S|L]_j$, we conventionally assume that SWAP is too conservative; in this case we set $k = k - adj$ which increases the number of jobs estimated as long. Since delaying jobs in an aggressive way may achieve worse performance than in a conservative way, we here set $j = \lfloor QT/10 \rfloor$ instead of $\lfloor QT/2 \rfloor$ to guarantee at least 10% of queued requests to be delayed. Throughout experiments we have always observed that the LT online algorithm does not show instability problems and always provides effective choices of LT which lead to consistent performance gains as discussed in the next subsection.

1. initialize:

a. maximum allowable delay limit D

b. arrival index $i \leftarrow 0$

c. large threshold $LT \leftarrow \mu^{-1}(1 + k \cdot CV)$

2. upon each job arriving at queue

a. $i \leftarrow i + 1$

b. record that job's arrival index: $A_i \leftarrow i$

c. initialize that job's predicted result as UnCheck

d. initialize that job's num. of delays $d \leftarrow 0$

3. upon each job completion at queue

a. measure conditional probabilities $P[L|L]_j$, $1 \leq j < N$

b. if its service time is greater than LT

c. then trigger one round of the delaying

I. initialize $j \leftarrow 1$

II. if predicted result of the j-th job is not UnCheck

then keep using its predicted result

else calculate the lag apart the two jobs: $lag \leftarrow j$-th job's A_i - completed job's A_i

III. if $P[L|L]_{lag} \geq P[S|L]_{lag}$

then set that job's predicted result as large

else set that job's predicted result as small

IV. $j \leftarrow j + 1$

V. if reaching the end of the queue

then delay all large jobs with num. of delays $d \leq D$ to the end of the queue and

increase $d \leftarrow d + 1$

else, go to step 3-c-II

d. else, go to step 3

Figure 4.1: Description of SWAP.

1. initialize: $k \leftarrow 1$ & $adj \leftarrow 0.5$

2. set $LT \leftarrow \mu^{-1}(1 + k \cdot CV)$

3. for each request in T_W do

a. upon each job completion at the autocorrelated server

I. compute observed conditional probabilities: $P[L|L]_j$, for $1 \leq j < N$;

II. update μ^{-1} and CV by Welford's algorithm

III. update the mean queue length QL

b. at the end of T_W

I. if $P[L|L]_{\lfloor QT/2 \rfloor} \geq P[S|L]_{\lfloor QT/2 \rfloor}$, then $k \leftarrow k + adj$

else if $P[L|L]_{\lfloor QT/10 \rfloor} < P[S|L]_{\lfloor QT/10 \rfloor}$, then $k \leftarrow k - adj$

II. set maximum and minimum large thresholds:

$LT\_max \leftarrow$ 90 percentile of observed service times

$LT\_min \leftarrow$ 50 percentile of observed service times

III. recalculate $LT \leftarrow \mu^{-1}(1 + k \cdot CV)$

IV. if $LT > LT_{max}$, then $LT \leftarrow LT_{max}$

V. if $LT < LT_{min}$, then $LT \leftarrow LT_{min}$

Figure 4.2: Description of how to self-adjust the large threshold LT.

4.2.4 Performance Evaluation of SWAP

In this subsection, we present representative case studies illustrating the effectiveness and the robustness of SWAP. All simulations refer to a 10 million sample space and the reported results are within 98% confidence intervals.

We use simulation to evaluate the performance improvement of SWAP in a network with M first-come-first-served (FCFS) servers in series. We assume that there is only one server with temporal dependence in its service process and denote that queue as Q_{ACF}. Throughout experiments, the service process at Q_{ACF} is always a two-state Markov-Modulated Poisson Process (MMPP(2)) [67] with identical distribution having mean rate $\mu = 1$ and squared coefficient of variation $CV^2 = 20$. Let ρ_j be the lag-j autocorrelation coefficient. For the MMPP(2), we consider three different autocorrelation profiles:

- ACF_1: $\rho_1 = 0.47$ decays to zero beyond lag $j = 1400$;
- ACF_2: $\rho_1 = 0.46$ decays to zero beyond lag $j = 240$;
- ACF_3: $\rho_1 = 0.45$ decays to zero beyond lag $j = 100$.

Figure 4.3 shows the ACF for the three profiles. The other $M - 1$ queues, denoted as Q^i_{Exp}, have exponentially distributed service times with mean rate λ_i, $1 \leq i < M$. We focus on the case where a constant workload of N requests circulates in the network, i.e., the model is a closed queueing network. Simple networks of this type are often used to model real systems of large diffusion, e.g., multi-tier architectures [59, 92].

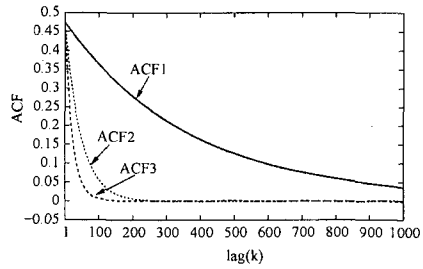


Figure 4.3: The ACF of the service process that generates the autocorrelated flows in the system, where the service times are drawn from MMPP(2)s with ACF_1, ACF_2 and ACF_3, respectively.

Performance Improvement

We first simulate a network with two queues: the exponential queue Q^1_{Exp} has mean service rate $\lambda_1 = 2$; the autocorrelated queue Q_{ACF} uses the MMPP(2) described above with autocorrelation structure ACF_1. The model population is set to $N = 500$, the delay limit is $D = 100$. Sensitivity to the most important experiment parameters is explored later.

We compare system capacity under SWAP as measured by the system throughput with the throughputs observed when Q_{ACF} uses FCFS or SJF scheduling. Indeed, larger throughput means that the system can sustain more load. Therefore, it is protected from the degradation of sudden bursts of requests, which improves the overall availability of the system. FCFS performance is used as baseline in comparisons. We recall that our stated goal is to show that SWAP performance is competitive with that of SJF which would prove that the knowledge required by SJF can be inferred effectively from the temporal dependence of workloads.

Table 4.1 shows the mean throughput of the difference policies and the relative improvement with respect to FCFS. Throughput is measured at an arbitrary point of the network, since for the topology under consideration throughput at steady state must be

identical everywhere [22]. The table shows that, although we are not reducing the overall amount of work processed by the system, both with SJF and SWAP the capacity is significantly better than with FCFS. Noticeably, SJF and SWAP perform closely, thus suggesting that the SWAP approximation of SJF is very effective.

	FCFS	SWAP	SJF
TPUT	0.71 job/sec	0.92 job/sec	1.01 job/sec
% improv.	baseline	29.6%	40.8%

Table 4.1: Mean system throughput (TPUT) and relative improvement over FCFS for a network with $M = 2$ queues, $N = 500$ jobs, $\lambda_1 = 2$ and autocorrelation profile ACF_1. SWAP achieves a performance improvement similar to SJF, but without requiring a priori knowledge of service times.

Further confirmation of this intuition comes from Figure 4.4(a), which shows the complementary cumulative distribution function (CDDF) of the round trip times, i.e., the probability that the round trip times experienced by individual jobs are greater than the value on the horizontal axis. The plot shows that the largest part of job experiences the lowest round trip times when the scheduling is SJF or SWAP. Indeed, the part of the workload whose execution is delayed at Q_{ACF} receives increased response times, but the number of penalized requests amounts to less than 3% of the total. We observe also in this case that the performance of SJF and SWAP is extremely close, the only significant difference being that in SJF a small fraction of jobs (less than 0.5%) receives much worse round trip times than in SWAP. We attribute such difference to the unavoidable forecasting errors in SWAP, which may occasionally fail in identifying jobs as long also if their actual service requirement is large, thus resulting in a smaller CDDF tail than SJF.

Other interesting observations arise from Figures 4.4(b) and Figures 4.4(c). Figure

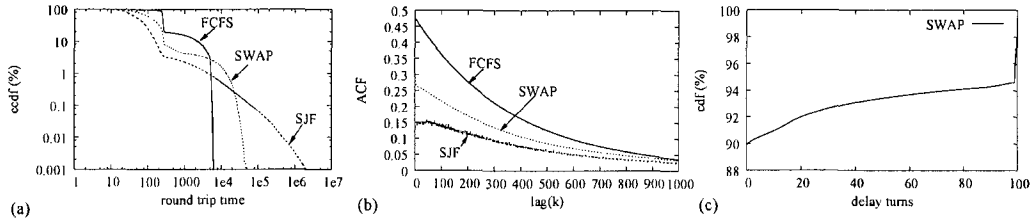


Figure 4.4: Comparative evaluation of SWAP, SJF and FCFS: (a) CCDF of round trip times, (b) autocorrelation (ACF) of service times at Q_{ACF}, and (c) CDF of the number of times jobs are delayed at Q_{ACF}.

4.4(b) shows the autocorrelation of the service times at Q_{ACF} under the different scheduling disciplines. It is immediate to observe that the temporal dependence is much less pronounced under SJF and SWAP, thus suggesting that both techniques are able to break the strong temporal locality of the original process. Also in this case, the results of SJF and SWAP are very close to each other. Figure 4.4(c) shows the cumulative distribution function (CDF) of the number of times that a job is consecutively delayed at Q_{ACF} (here delay turns denotes the number of received delays by a job). Indeed, approximately 90% of the jobs never suffer a delay, while for the rest of the population the delay is often much less than the limit $D = 100$.

Sensitivity to Device Relative Speeds

Here, we investigate the robustness of SWAP performance to changes in the experimental parameters. We first focus on evaluating networks with varying processing speeds, i.e., we consider the same model but vary the service rate at the exponential queue Q^1_{Exp} while keeping fixed the speed at Q_{ACF}. Figure 4.5 presents the average system throughput for three experiments, labeled $Exp1$, $Exp2$, and $Exp5$, where we set $\lambda = 1, 2$, and 5 job/sec, respectively. As the service rate at Q_{ACF} is $\mu = 1$ job/sec, in $Exp1$ the two queues have

identical speed, while in both $Exp2$ and $Exp5$, Q_{ACF} is the system bottleneck and in $Exp5$ the relative speed at Q_{ACF} becomes even slower. The relative capacity improvement with respect to FCFS scheduling is marked above each bar in the figure. The interpretation of the experimental results leads to the following observations.

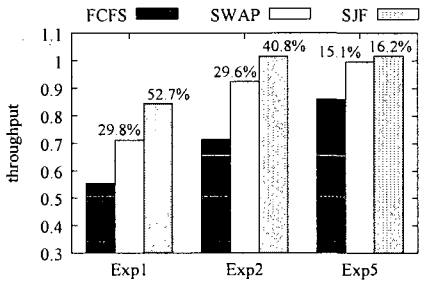


Figure 4.5: Sensitivity to service process ratio in a network with $M = 2$, $N = 500$, and ACF_1.

First, SWAP improves the system throughput across all experiments and is better for smaller values of λ. The intuition behind this result is that as λ decreases, more jobs are enqueued at the resource Q^1_{Exp}, and then delaying a job produces less overhead because a job put in the tail of Q_{ACF} can yet reach the head of the queue quite rapidly. Therefore, the cost of delaying becomes negligible and the network can benefit more of the reordering of jobs sizes.

A second important observation is that, as λ increases, the SWAP performance converges to that of SJF. This suggests that SWAP forecasting is very accurate since in $Exp5$ almost all population in the network is queueing at Q_{ACF} and SJF sorts nearly perfectly a large population close to N jobs according to their exact size. The fact that SWAP achieves similar performance indicates that the same accurate ordering is obtained if forecasting is based on temporal dependence.

As a final remark, it is interesting to observe that SWAP can be more effective than

hardware upgrades. For instance, the throughput under SWAP in $Exp2$ (white bar, $Exp2$) is more than the expected throughout with FCFS in $Exp5$ (black bar, $Exp5$). That is, under temporal dependent workloads, it can be more effective to adopt SWAP than doubling the hardware speed of Q^1_{Exp}.

We conclude the experiment showing in Figure 4.6 the CCDF of round trip times for the previous experiments. The CCDF tail behavior observed in the previous subsection persists for $Exp1$, $Exp2$, and $Exp5$, where again SWAP degrades the performance of only 3% of the total number of requests.

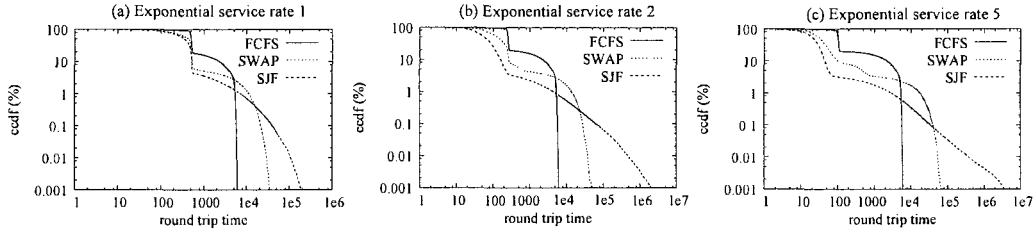


Figure 4.6: Illustrating the CCDF of round trip times in a network with $M = 2$, $N = 500$, and ACF_1. The service rate λ_1 of the exponential queue is equal to (a) 1, (b) 2, and (c) 5.

Sensitivity to Temporal Dependence

In order to analyze the effect of temporal dependence on policy performance, we conduct experiments with various autocorrelation profiles at Q_{ACF}, but always keeping the same mean and CV of the job sizes. We use three service processes with autocorrelation ACF_1, ACF_2, and ACF_3 shown in Figure 4.3.

Figure 4.7 shows the system throughput under FCFS, SWAP and SJF policies for the same model but for different autocorrelations. In general, we expect that strong ACF degrades overall system performance more than weak ACF, as it is clearly confirmed by the experimental results. Yet, SWAP under the stronger ACF improves more than

under the weaker ACF. This is because the stronger the ACF, the higher the conditional probabilities for having large-large pairs in the service time series and the delaying is more aggressive. For instance, for ACF_1, we have $P[L|L]_j \geq P[S|L]_j$ for all $j < 69$.

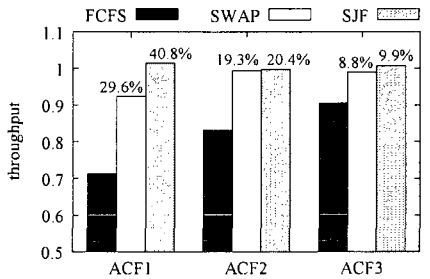


Figure 4.7: Sensitivity to temporal dependence in a network with $M = 2$, $N = 500$, and $\lambda_1 = 2$, where the relative improvement over the FCFS policy is indicated on each bar.

When the service process has the two weaker ACFs, i.e., ACF_2 and ACF_3, the margin for performance improvement of SWAP and SJF is much reduced. In this case, only the conditional probabilities with lags up to $j = 30$ for ACF_2 and up to $j = 14$ for ACF_3 satisfy $P[L|L]_j \geq P[S|L]_j$. This implies that weaker ACFs make SWAP more conservative in delaying long jobs, but SWAP still achieves performance very close to the target behavior of SJF.

The plots in Figure 4.8 present the effect of different temporal dependence on the tail of round trip times under SWAP. Strong temporal dependence in the service process makes SWAP to delay long jobs more effectively, and thus helps almost 97% of requests be served up to seven times faster than under the FCFS policy, see Figure 4.8(a). As temporal dependence becomes weaker in Figure 4.8(b), the policy delays long jobs less aggressively and a few requests show worse performance. That is, SWAP becomes less effective, resulting in a longer tail of the round trip times distribution. With low autocorrelation, see Figure 4.8(c), SWAP becomes more conservative in delaying jobs, which is reflected by a

small fraction of affected jobs. Consistently with the results presented in the previous case studies, SJF gives a long tail in the distribution of round trip times across all experiments and as the strength of ACF decreases, the tail becomes longer.

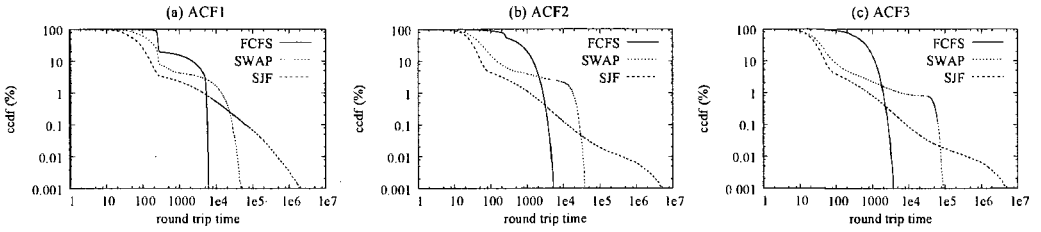


Figure 4.8: Illustrating the CCDF of round trip time in a network with $M = 2$, $N = 500$, and $\lambda_1 = 2$. The service process of Q_{ACF} has temporal dependence (a) ACF_1, (b) ACF_2, and (c) ACF_3.

Sensitivity to System Load

Now we investigate the sensitivity of SWAP to an increased number of requests in the system. This is extremely important to understand the performance benefit of the technique as the system reaches critical congestion. In order to evaluate how SWAP improves system availability, we conduct experiments with three different network populations $N = 500$, $N = 800$, and $N = 1000$, while keeping fixed the other parameters as the previous experiments. The system throughput for these three experiments is illustrated in Figure 4.9 and the CCDFs of the round trip times experienced by individual requests are plotted in Figure 4.10. In the experiment with the highest load $N = 1000$, SWAP improves throughput by 33% compared to the baseline case and achieves performance close to the target SJF performance. The improvement is clear also for lower loads, i.e., $N = 500$ and 800, but performance gains are maximal under the most congested case $N = 1000$.

Regarding availability, SWAP enables the system to sustain higher loads compared

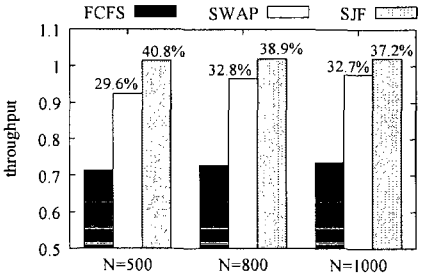


Figure 4.9: Sensitivity to network population in the system with $M = 2$, $\lambda_1 = 2$, and ACF_1, where the relative improvement over the FCFS policy is indicated on each bar.

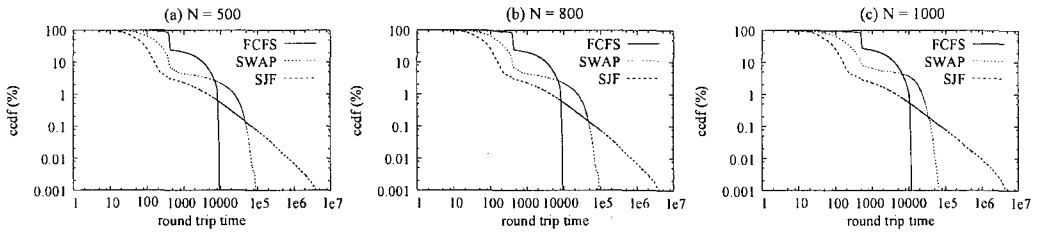


Figure 4.10: Illustrating the CCDF of round trip time in a network with $M = 2$, $\lambda_1 = 2$, ACF_1. The network population is (a) $N = 500$, (b) $N = 800$, and (c) $N = 1000$.

to the FCFS policy. For instance, for $N = 800$ and FCFS scheduling, 80% of requests experience round trip times less than 1146 when no delaying of jobs occurs, see the solid curve in Figure 4.11. However, even for $N = 1000$ requests, the fraction of requests having round trip times less than 1146 becomes 95% with SWAP (see the dashed curve in Figure 4.11). That is, SWAP is able to give a remarkably better performance to most jobs than with FCFS even if the overall population is increased by 200 requests. In this sense, it is immediately clear that SWAP can be very effective in addressing request bursts that threaten system availability. Overall, these results imply that SWAP dramatically improves system availability by providing high percentiles of jobs having round trip times less than a predefined target.

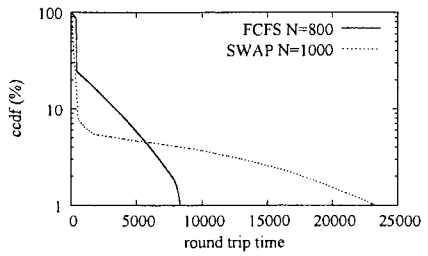


Figure 4.11: Illustrating the CCDF of round trip time in a network with $M = 2$, $\lambda_1 = 2$, and ACF_1. The solid curve shows the results in the experiment with $N = 800$ under the FCFS policy and the dashed curve presents the results in the experiment with $N = 1000$ under SWAP.

Sensitivity to Network Size

We investigate the sensitivity of SWAP to the network size by evaluating throughput improvement for $M = 2, 3, 4$. Except for the autocorrelated queue Q_{ACF}, the remaining $M - 1$ resources are queues with exponential service times. In order to evaluate the different impact of service times that are balanced or unbalanced with respect to the service at Q_{ACF}, we consider the rates shown in Table 4.2, see the initial part of this subsection for related notation.

M	Q_{ACF}	Q^1_{Exp}	Q^2_{Exp}	Q^3_{Exp}
2	$\mu = 1$	$\lambda_1 = 1$	N/A	N/A
3	$\mu = 1$	$\lambda_1 = 1$	$\lambda_2 = 0.25$	N/A
4	$\mu = 1$	$\lambda_1 = 1$	$\lambda_2 = 0.25$	$\lambda_3 = 1$

Table 4.2: Queue service rates in the three experiments used to study SWAP sensitivity to different network sizes.

Figure 4.12 shows throughput improvement provided by the three scheduling disciplines. Note that the first experiment is different from the conditions of Table 4.1, since here the two queues are balanced. We observe that as the number of queues in the net-

work increases, the relative improvement over the FCFS policy decreases. We interpret this effect by observing that since there are more exponential servers in the network, the temporal dependence of the successive requests at the queues are much weaker than in the experiments considered before. That is, throughout its path, each request is served multiple times by exponential service processes without temporal dependence and therefore the temporal locality effects in the network are reduced. Therefore, the reduced gain in this experiment is rather a consequence of the more limited margin for improvements on these network rather than a limit of SWAP. In fact, we see that also SJF improves modestly with respect to the FCFS case.

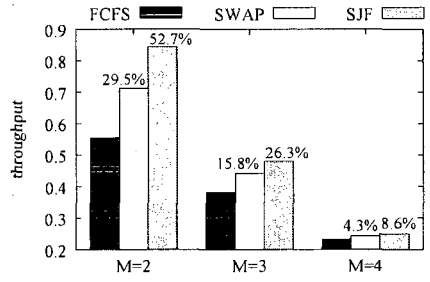


Figure 4.12: Sensitivity to network size in a network with $N = 500$ and ACF_1, where the relative improvement over the FCFS policy is indicated on each bar.

We complete the analysis in this subsection with Figure 4.13 that plots the CCDF of job round trip times for the three experiments. Consistently with the results presented in the previous cases, SWAP only sacrifices 2-3% of requests due to delaying but achieves better performance for most requests. The results are consistent with the properties of SWAP observed in the previous experiments, and the results are almost indistinguishable across the three experiments.

In summary, the extensive experimentation carried out in this subsection has revealed that SWAP can effectively approximate the performance of SJF without the need of

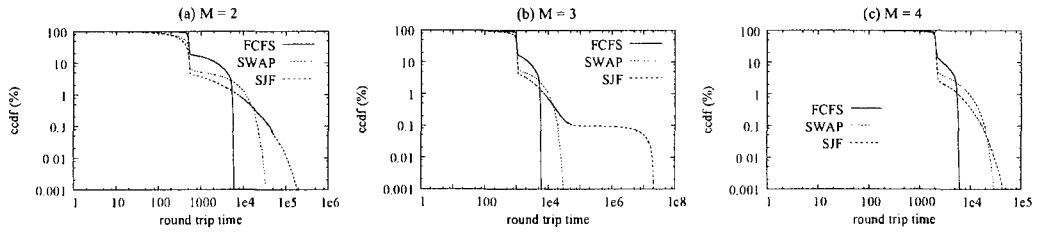


Figure 4.13: Illustrating the CCDF of round trip time in a network with $N = 500$ and ACF_1. The number of queues in the network is (a) $M = 2$, (b) $M = 3$, and (c) $M = 4$.

additional information about job service times. The sensitivity results on the various autocorrelation (burstiness) profiles have proved that the gains are more pronounced in presence of higher temporal dependent workloads. This suggests that SWAP is an effective solution to increase performance in systems processing this type of workloads. Sensitivity analysis to the number of queues in the network and system load show that the gains of SWAP are visible in a variety of different conditions.

4.3 Autocorrelation-Guided Load Control Policy

In this section, we extend SWAP by *infinitely* delaying requests to control the system load and thus improve the overall system performance. We assume that load control by infinitely delaying (i.e., "dropping") requests is an acceptable practice for the application under consideration. For example, the MPEG video coding schemes store the necessary information to decode the video redundantly in multiple frames. Consequently, under heavy load, some of the frames can be dropped, up to a certain percentage, without compromising the overall quality of service perceived by the user. Furthermore, the quality of service perceived by the user depends on the device that is playing the digital video. If it is a low resolution device (such as a handheld), then the percentage of video frames

that can be dropped without affecting the quality of viewing is higher than if the video is watched on a high-definition television.

The proposed load control policy is driven by autocorrelation, another form of temporal dependence, to reduce the average durations of "slow" service periods. The duration of slow periods can be reduced by dropping longer (relatively to other) requests from the queue of the autocorrelated server. Note that reducing the duration associated to short requests also reduces autocorrelation and may thus improve performance. However, the performance impact of short jobs is typically small compared to that of long jobs and this makes the practice of dropping small jobs less interesting than for large jobs. Our scheme is more effective than other methods, e.g., random drop, since we effectively forecast which jobs in the queue are long, thus become good candidates to drop.

First, we present a static load control policy, called ALoC, for the autocorrelation-driven load control in autonomic systems. The static version of this policy assumes no knowledge of the length of queued jobs, but requires a priori knowledge of the autocorrelation function of the service process. Then, we present a related dynamic version, called D_ALoC, which is truly a *no knowledge* policy, i.e., it does not assume any a priori knowledge and dynamically adapts its load control parameter based on online measurement, policy targets, and statistical information of past workloads.

4.3.1 ACF-Guided Dropping

Dropping effectively the most harmful requests for performance depends on the prediction accuracy of future job sizes. Henceforth, we do *not* assume a priori knowledge of any job size, i.e., the system knows the size of a job only *after* it completes execution. The

aim of the prediction is as follows: if size prediction is done effectively, long jobs can be accurately identified and removed from the queue so that the duration of the slow state decreases, yielding an improvement of the average response times.

We now describe how we can use the autocorrelation function to predict job sizes. Let us assume that the last served job was long, i.e., its size was greater than $\mu^{-1}(1 + k \cdot CV)$ for a given k, and suppose that we wish to forecast the size of the j-th job in queue (the job $j = 1$ is the one immediately entering service after the long job just completed). If the service process has a positive ACF ρ_j, then there is similarity in size between the completed long job and the j-th job in queue. Therefore, we cast a random number with uniform distribution in $[0, 1]$, if the result is less than or equal to ρ_j, then we assume that the j-th job is long, otherwise it is short. That is, we assume that ρ_j is a measure of the conditional probability for a job to be large given that the last served job was large. A negative or zero ρ_j implies high probability that the j-th job significantly differs in size from the long job and therefore it is likely to be forecasted as short. Since in our analysis negative and zero autocorrelations lead to identical forecasting, we set $\rho_j = 0$ whenever the measured lag-j autocorrelation is negative.

4.3.2 ALoC: Static Version

We propose a load control policy driven by autocorrelation that reduces the average durations of "slow" service periods in order to improve the overall system performance. For the rest of this section, we assume that load control by dropping requests is an acceptable practice for the application under consideration, as in the case, e.g., of media workloads. For instance, selectively dropping *redundant* MPEG video packets does not compromise

the overall quality of service perceived by the end users. Furthermore, dropping a small number of *non-redundant* MPEG video packets may not be noticed by the end users due to low device resolution.

First, we present a static load control policy, called ALoC, which requires a priori knowledge of the autocorrelation function at the server process and of a user-defined parameter for controlling the load at the autocorrelated server. ALoC does *not* require a priori knowledge of the request size. The high level idea of ALoC is as follows. After a long job is completed, the queue is scanned to find other long jobs probabilistically, according to their position j in the queue and the value of the corresponding autocorrelation coefficient ρ_j. All jobs that have been estimated as long are then dropped from the queue; indeed, if some jobs are known to be indispensable for the application then they can be tagged as "undroppable" and be left in queue; we discuss this issue in Section 4.3.4.

In order to control and maintain load reduction at a minimum, we introduce a queue length threshold Q for dropping requests, where $0 < Q/N \leq 1$. Thus, ALoC starts dropping requests only when the last completed request is long *and* the queue length at the autocorrelated server is higher than Q. Therefore, if the system is under-utilized and Q is not reached, then no request is dropped and all long requests are still served. Note also that since the policy is triggered only after a long job is executed, the policy avoids starvation of long jobs.

We use an example to describe ALoC. In the example, $Q = 5$ and there are 9 jobs waiting in the queue as shown in Figure 4.14. Upon completion of a long job, ALoC is triggered because the current queue length is greater than Q. ALoC starts to probabilistically predict the size of the jth waiting job for $j = 1, 2, \ldots, 9$. For instance, if $\rho_1 = 0.40$,

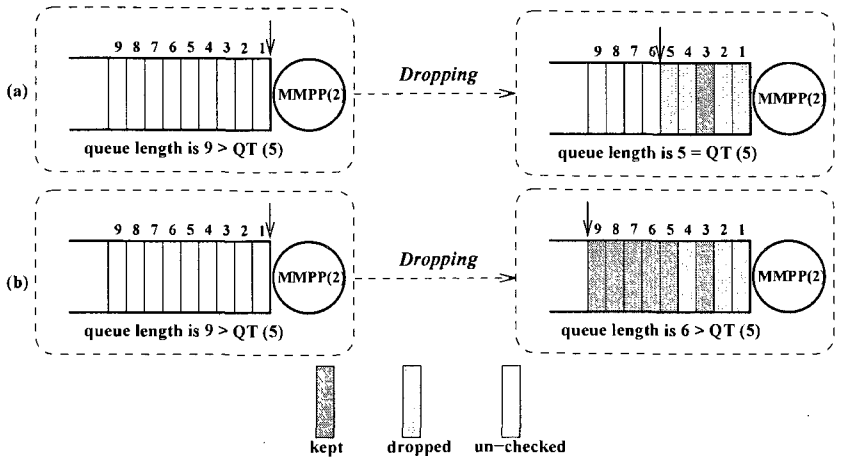


Figure 4.14: Illustration of ALoC's operations. The most-recent served job is larger than $\frac{1}{\mu}(1 + k \cdot CV)$ and the current queue length exceeds $Q = 5$. Dark bars represent requests to be kept in the queue, light gray bars represent requests to be dropped, and blank bars represent requests yet to be considered by the policy. Figure 4.14(a) shows one possibility that the policy stops dropping any job when four jobs have been dropped from the queue and the current queue length reaches Q. Figure 4.14(b) shows another possibility that the policy scans the entire waiting queue and only three jobs are estimated long ones and dropped.

then we interpret this value as a 40% probability that the job in position 1 is similar to the last completed job, i.e., it is a long job. We therefore cast a random number in $[0, 1]$ and if the result is less than or equal to 0.40, then the job in position 1 is dropped from the queue. A similar approach can be used to estimate the job service requirement for the j-th job in queue using the ρ_j autocorrelation coefficient, see light gray and dark bars in Figure 4.14 for an example of possible outcome of the forecasting. The policy has two stopping conditions: a first case is when enough estimated-long jobs have been dropped from the queue, and the current queue length has been reduced to the threshold value Q, see Figure 4.14(a). Alternatively, the policy may exhaust the waiting queue predicting that only estimated-short jobs wait in the queue with the current queue length still exceeding Q, see Figure 4.14(b). At the end of one round, the first job waiting in the current queue is admitted for service. ALoC is not triggered again before completion of another

long job.

Figure 4.15 gives the pseudo-code for ALoC, which assumes a priori knowledge of the autocorrelation coefficients ρ_j, for $1 \leq j < N$, of the service process, and the queue length threshold Q. In order to specify a fully autonomic load control policy, the controller must be able to estimate these values online. We introduce in Section 4.3.3, D_ALoC, the dynamic version of ALoC, which is able to do online estimation of all parameters.

1. initialize variables

a. initialize the index of the ACF queue: $i \leftarrow I$

b. initialize the ACF values of the service stream at queue i: ρ_j for all $1 \leq j \leq N$

c. initialize the threshold $Q \leftarrow R \cdot N$, for a given $0 < R \leq 1$

2. for every job completion at queue i do

a. check if service time of current request is long *and* the current queue length exceeds Q

b. if yes, start dropping

I. initialize the index of jobs: $j \leftarrow 1$

II. for job j, generate a random number $s \in [0, 1]$

if $s < \rho_j$ then assume the job is long and drop it

else assume the job is short and keep it

III. if the current queue length reaches Q then go to step **2**

else $j \leftarrow j + 1$ and go to step **2-b-II**

c. else, go to step **2**

Figure 4.15: Description of ALoC. All input parameters are determined off-line.

Performance of ALoC

We use simulation to evaluate the performance of ALoC in a system described by two first-come-first-served (FCFS) queues Q_{EXP} and Q_{ACF} with mean service rates λ and μ, respectively. This simple abstraction can be used to model a simple consumer electronic system, e.g., a personal video recorder, a game console, or a MP3 player. The observations given here readily apply to systems with several queues. In all simulations, we use a 10 million sample space and the reported results are within 98% confidence intervals.

Q_{ACF} is the device with an autocorrelated service process, which is drawn from a MMPP(2) with mean rate $\mu = 1$ and squared coefficient of variation $CV^2 = 20$. Q_{EXP} is evaluated in two configurations:

- *Experiment 1*: Q_{EXP} is one order of magnitude faster than Q_{ACF};
- *Experiment 2*: Q_{EXP} is two orders of magnitude faster than Q_{ACF}.

That is, the service times of Q_{EXP} are exponentially distributed with mean rates $\lambda = 10$ and $\lambda = 100$ in *Experiment 1* and *Experiment 2*, respectively. We remark that experiments with varying relative speed up to three and four orders of magnitude of the two devices yield qualitatively similar results. Differences of orders of magnitude in the service of this entity are often encountered when the modeled resources are CPU and disks. Furthermore, in order to qualitatively analyze the effect of autocorrelation on policy performance, we also conduct experiments with the same three MMPPs as shown in Section 4.2.4, having different autocorrelation profiles (i.e., ACF_1, ACF_2, and ACF_3) at Q_{ACF} for both *Experiment 1* and *Experiment 2*, but always such that they have the same mean, CV^2, and higher moments of the job sizes.

Comparison with Random Dropping

To evaluate the effectiveness of ALoC, we compare it with a policy where request drop is done *randomly*. The random policy continuously drops from the head of the waiting queue with probability set as same as the overall dropping ratio of ALoC. For ALoC, we set $N = 500$ and $Q = 490$, i.e., 98% of N. Figure 4.16 presents the average response times for *Experiment 1* and *Experiment 2*. The relative *improvement* in round trip times is marked above each bar in the figure. Round trip times when all jobs are admitted without load control are plotted as a baseline comparison (NoDrop bars).

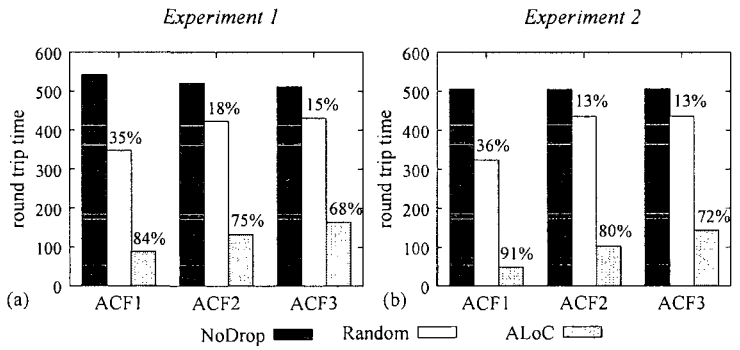


Figure 4.16: Average response times of ALoC for *Experiment 1* and *Experiment 2*, with $N = 500$ and $Q = 490$ (98% of N). The drop ratios in the random and ALoC policies are 0.08, 0.10, and 0.13 for ACF_1, ACF_2, and ACF_3, respectively. The numbers above bars indicate the relative improvement over the NoDrop case.

In the two experiments, drop ratios of ALoC are equal to 0.08, 0.10, and 0.13 for ACF_1, ACF_2, and ACF_3, respectively. Counter-intuitively, the drop ratio for strong autocorrelated service process (e.g., ACF_1) is lower than that for the weak one, e.g., ACF_3. This is because with strong ACF, the prediction of long jobs becomes more accurate, which increases the throughput of the autocorrelated queue and thus decreases the queue length. As the result of this, the trigger condition, i.e., the queue length being beyond Q, occurs less often. Consequently, the policy drops less jobs but improves the performance

more with stronger ACF.

Figure 4.16 shows that across all experiments, ALoC dramatically improves expected response times compared to a random drop policy. Parsimonious selection of the request to be denied service results in significant improvements when compared to the random policy. In *Experiment 1* with ACF_1, the random policy results in a response reduction of about 35% from the baseline case. ALoC further reduces average response time by 84% relative to the baseline case. Performance trends persist for ACF_2 and ACF_3, but performance gains slightly reduce as the strength of ACF decreases. This can be explained by the fact that forecasting becomes less effective when the autocorrelations are smaller, and job size is thus harder to predict. Similar trends persist in *Experiment 2* where ALoC presents additional performance improvements as the speeds of the two devices differ now by two orders of magnitude. The higher the difference in the devices speed, the better the performance improvement of ALoC in comparison to dropping randomly.

Sensitivity to Queue Length Threshold Q

We quantify the performance effect of the pre-defined threshold Q used to trigger request dropping at the Q_{ACF} queue. We investigate the effectiveness of a choice of Q by computing the related average drop ratio and the relative improvement of response time. Reported statistics are only for those requests that complete work in both queues.

Figure 4.17 presents performance measures as a function of Q for *Experiment 1* and *Experiment 2* by using ALoC. The population in the model is set to $N = 500$. Q ranges from 100% of N, i.e., no drop since ALoC is never triggered, to 10% of N, i.e., we drop requests when the queue length in Q_{ACF} is equal to 50. From the figures we see that as

Q decreases, the drop ratio increases quickly (see Figure 4.17 (a) and (c)), but there is a point beyond which the drop ratio stabilizes. This happens because the smaller the Q, the larger the proportion of large jobs that are denied service at Q_{ACF}. When Q gets smaller than a certain value, the policy becomes very aggressive: most long jobs are dropped and only few long jobs remain in the queue to be dropped. Therefore, the drop ratio stabilizes because the queue is almost empty of long jobs. Across both experiments, the position of the knee of the drop ratio curve depends on the strength of the autocorrelation function. The stronger the autocorrelation, the lower the value of Q for which the knee appears.

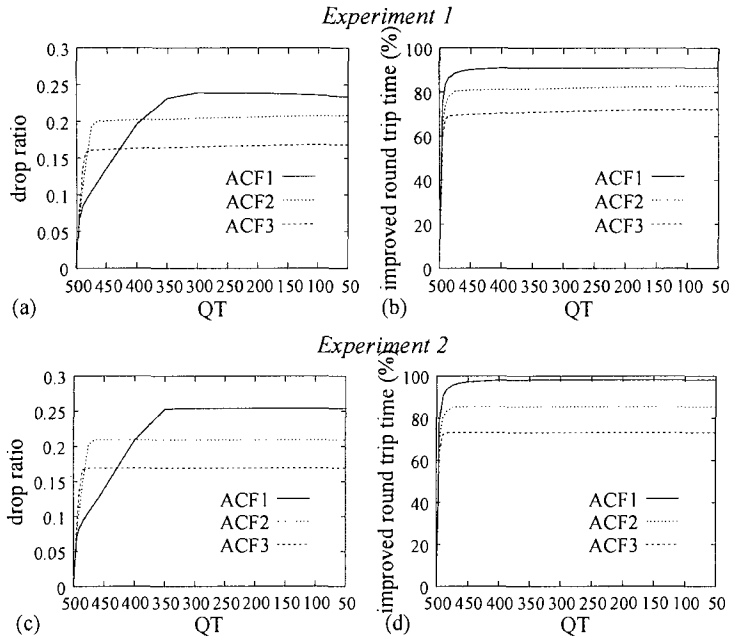


Figure 4.17: Average drop ratio and average response time reduction achieved by ALoC as a function of Q for *Experiment 1* and *Experiment 2* with three MMPP(2) processes (i.e., ACF_1, ACF_2, and ACF_3) at Q_{ACF}. $N = 500$.

The performance effect as a function of Q values is illustrated in Figure 4.17(b) and Figure 4.17(d). The plots show that excellent performance improvements can be achieved by triggering ALoC infrequently with large Q values, which also results in desirable

smaller drop ratios. In Figure 4.17(b) and Figure 4.17(d), a large Q equal to 490 results in dramatic performance improvements while containing the drop ratio at a minimum across all experiments.

For completeness, we have also conducted sensitivity analysis under different job populations, e.g., $N = 100$ and $N = 300$. Our results can be summarized as follows. Drop ratios and relative performance gains with different populations are qualitatively the same as those for $N = 500$. Drop ratios are lower in less populated models while relative gains in response times remain high.

Round Trip Time Distribution

We analyze the tail performance and plot in Figure 4.18 the complementary cumulative distribution function (CCDF) of round trip times and of response times at Q_{EXP} and Q_{ACF} for *Experiment 1* with ACF_2 . Results for ACF_1 and ACF_3 are remarkably similar to those reported in this figure. The figure shows that ALoC significantly improves the tail of the response times at Q_{ACF} and consequently the response times. The tails of response times at Q_{EXP} of all three policies are almost identical.

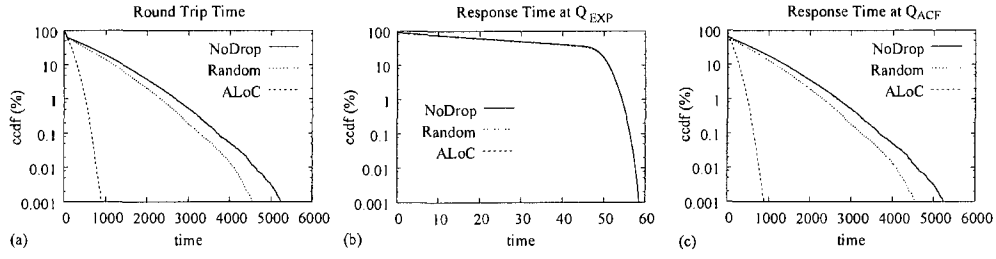


Figure 4.18: CCDFs of response times for *Experiment 1* for the random and ALoC policies. Service times of Q_{ACF} have ACF_2, $N = 500$ and $Q = 490$. The drop ratio of both Random and ALoC is equal to 0.10.

Figure 4.19 also depicts the ACFs in the departure process of Q_{ACF} (i.e., arrivals to

Q_{EXP}). The random policy achieves a small reduction in the autocorrelation function compared to the original one (labeled as "NoDrop" in the figure). ALoC's ability to selectively deny service of jobs in the queue that cause autocorrelation is shown in the figure: the departure process curve that corresponds to this policy shows autocorrelation that is significantly reduced.

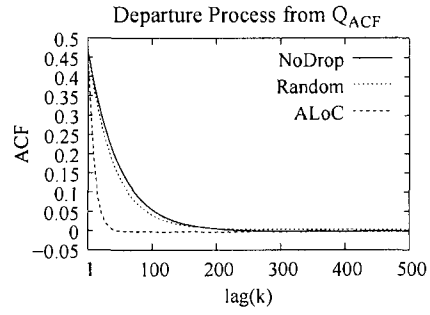


Figure 4.19: ACF of the departure process of Q_{ACF} for *Experiment 1*. $N = 500$, the autocorrelation of the service process is ACF_2, and $Q = 490$. The drop ratio of random and ALoC is 0.10.

4.3.3 D_ALoC: Dynamic Version

This version of the policy does not require any a priori knowledge of either autocorrelation coefficients or queue length threshold Q. D_ALoC computes the autocorrelation coefficients online, allowing for changes in the workload characteristics over time and self-adjusts Q such that target performance parameters are met.

For each server, D_ALoC evaluates its mean service time, its coefficient of variation, and the autocorrelation coefficients of the service process when a job is completed at that particular server, using a modified version of Welford's one-pass algorithm [96]. The definition of ACF at lag j given in Eq.(2.1) can be rewritten as follows:

$$\rho_j = (E[X_t X_{t+j}] - E[X_t + X_{t+j}] \cdot \mu^{-1} + \mu^{-2})/\sigma^2, \tag{4.3}$$

where μ^{-1} and σ^2 are respectively mean and variance of the sequence and

$$\begin{aligned} E[X_t X_{t+j}] &= E[X_{t-1} X_{t+j-1}] \\ &\quad + \frac{X_t X_{t+j} - E[X_{t-1} X_{t+j-1}]}{t} \\ E[X_t + X_{t+j}] &= E[X_{t-1} + X_{t+j-1}] \\ &\quad + \frac{X_t + X_{t+j} - E[X_{t-1} + X_{t+j-1}]}{t}. \end{aligned}$$

If the autocorrelation coefficients in a specific server are positive, then D_ALoC determines that this server is the source of autocorrelation in the traffic flows of the entire system. Consequently, the load reduction is triggered at that server.

1. initialize threshold $Q \leftarrow N$

2. initialize the maximum allowable drop ratio D

3. for every C requests do

a. upon each job completion at queue i,

I. calculate the ACFs using Eq. (4.3): ρ_j for all $1 \leq j \leq N-1$

II. if the service process at queue i is autocorrelated

then drop using the same scheme of ALoC

b. at the end of an updating window of C requests

I. calculate current drop ratio d and compare with D

II. adjust Q using Eq. (4.4)

Figure 4.20: Description of D_ALoC. *All* policy parameters are computed on-line.

To dynamically adjust Q, we use an updating window of C requests that have been served. In the experiments presented here C is set to 3000. The value of Q is initialized to N. For every batch of C requests, D_ALoC compares the current request drop ratio

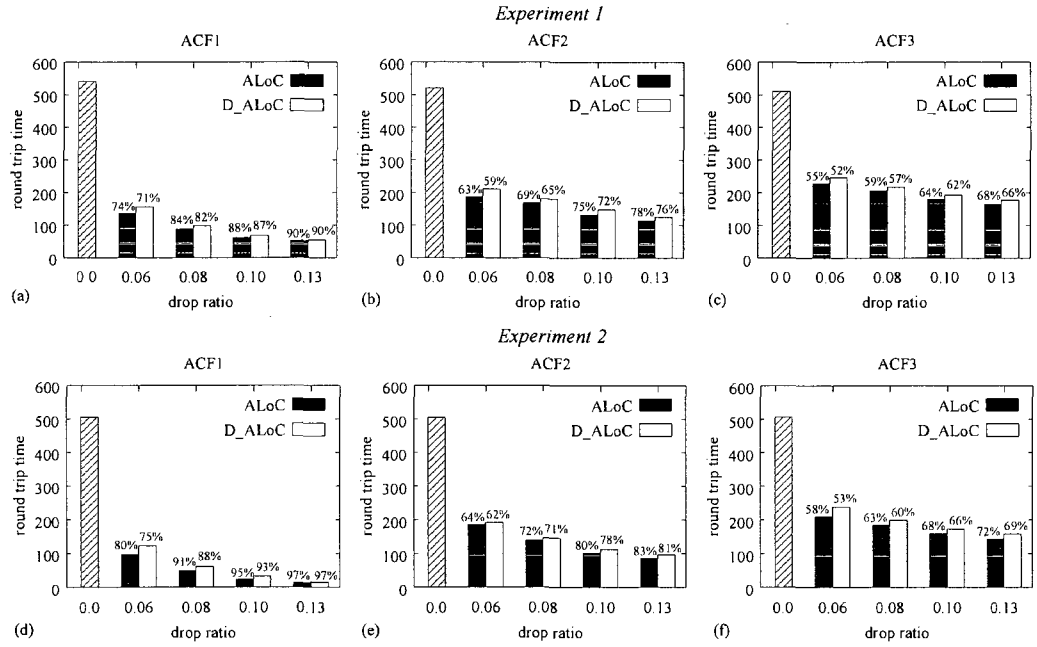


Figure 4.21: Average response times under drop ratios of 0.0, 0.06, 0.08, 0.10, and 0.13, for *Experiment 1* and *Experiment 2*. Service times of Q_{ACF} are drawn from three different MMPP(2)s (i.e., ACF_1, ACF_2, and ACF_3). $N = 500$. The number above bar are the relative improvements over the NoDrop case.

d with the maximum allowable drop ratio D. If the current drop ratio exceeds D, then Q is increased to reduce the frequency of dropping requests. If the drop ratio d is below D, then Q is reduced to drop requests more aggressively. The following equation illustrates how Q changes by a value that is proportional to the difference between the drop ratio d and the allowable drop ratio D:

$$Q = \begin{cases} Q + (N - Q) \cdot \frac{d-D}{1.0-D} & \text{if } d > D \\ Q - (Q - 0) \cdot \frac{D-d}{d-0.0} & \text{if } d \leq D \end{cases} \tag{4.4}$$

Upon updating Q, the new threshold for the next C requests in the autocorrelated server. Figure 4.20 gives the pseudo-code for D_ALoC.

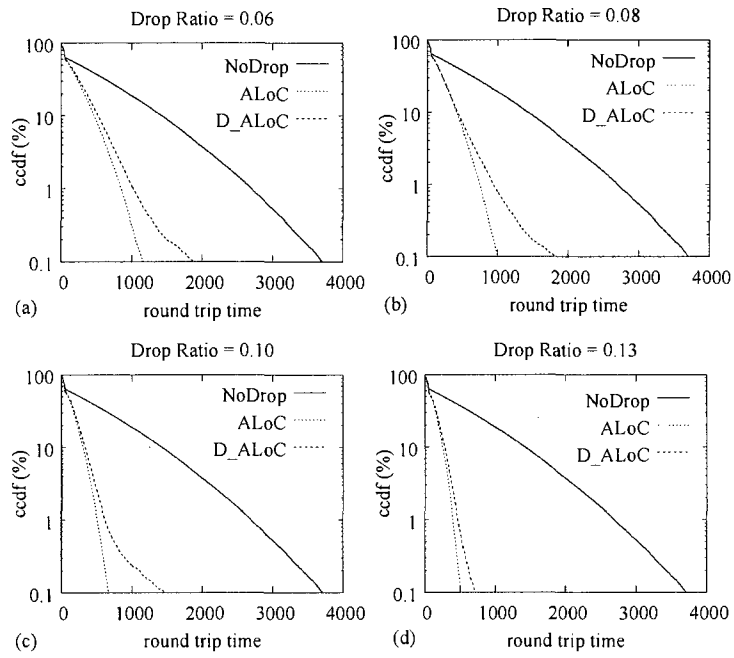


Figure 4.22: CCDFs of response times under drop ratios of 0.0 (i.e., no drop), 0.06, 0.08, 0.10, and 0.13, for *Experiment 1*, where the service times of Q_{ACF} are drawn from ACF_2. $N = 500$.

Performance of D_ALoC

Now, we evaluate the effectiveness of D_ALoC. The simulation environment is the same as in Section 4.3.2. For all experiments presented here, we set the maximum allowable drop ratio equal to 0.06, 0.08, 0.10, or 0.13. Figure 4.21 presents the average response times as a function of drop ratio in *Experiment 1* and *Experiment 2* under both D_ALoC and ALoC policies. Round trip times when all jobs are admitted are also plotted as a baseline comparison, i.e., drop ratio is equal to 0. Here, ALoC is parameterized such that the ideal Q is selected to achieve the pre-defined drop ratio while achieving best performance. At the beginning of the simulation, D_ALoC initializes $Q = 500$ (i.e., no drop), but it gradually changes this value such that the average drop ratios are maintained below the corresponding pre-defined allowable drop ratio.

The experiments depicted in Figure 4.21 indicate that D_ALoC's performance is very close to ALoC's. This means that D_ALoC is truly effective, especially because for each ALoC bar in Figure 4.21, the value of Q is selected by exhaustive searching so that ALoC achieves the best response values.

Figure 4.22 illustrates the CCDF of response times under drop ratios of 0.0 (i.e., no drop), 0.06, 0.08, 0.10, and 0.13 for *Experiment 1* with ACF_2 in the service stream at Q_{ACF}. The figure clearly shows that both ALoC and D_ALoC significantly improve the tail of response times. The tail of D_ALoC is close to that of ALoC, and the gap between these two tails diminishes as the drop ratio increases. In summary, Figures 4.21 and 4.22 argue for D_ALoC's effectiveness and robustness with respect to different autocorrelation strengths in the service process of Q_{ACF}, different target drop ratios, and relative speeds of Q_{EXP} and Q_{ACF}.

4.3.4 Trace Driven Evaluation

The majority of Internet-based media streaming systems can be modeled as a closed queueing system. In such a model, the first queue represents the device which receives the streaming media, e.g., a personal computer and other consumer electronic devices and the second queue represents the server that has stored the media content, e.g., movies, songs and games.

Here, we use actual traces measured at the disk level of a streaming system to evaluate how ALoC and D_ALoC perform in a practical setting. The traces record, in high resolution, both arrival and departure times of each request. Further details on these traces and their representativeness can be found in [78].

The mean service time recorded in the trace is 1.09 ms and CV is equal to 2.47. Figure 4.23 presents the autocorrelation function for the disk service process in this trace. At the first queue in our model, service times are drawn from an exponential distribution with mean service time equal to 0.01 ms, i.e., the first server is two orders of magnitude faster to view the content than the server that reads the content from the disk. The population N in the system is set to 200 and the sample space is equal to 1,043,259 requests. We remark that, for increased values of the population N, the autocorrelation of disk request sizes would be the same since in queueing models the service process is commonly assumed to be independent of N.

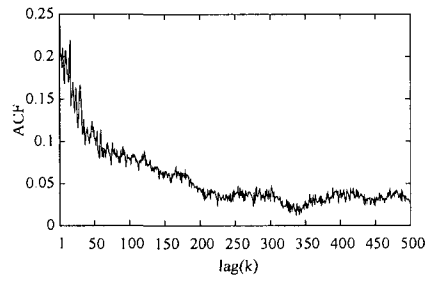


Figure 4.23: The ACF of the service times at a streaming device.

To investigate policy robustness, we add an additional restriction by marking some requests as "undroppable". In particular, we focus on trace data where the transmitted files are MPEG video streams. MPEG video streams compress raw frames specifically into three kinds of pictures: (1) I(ntra-coded)-pictures, which are independent of others, (2) P(redictive-coded)-pictures, which depend on the previous I- or P- pictures for being displayed correctly, and (3) B(idirectionally predictive-coded)-pictures, which need the information from the previous and the following I- or P- pictures for motion compensation [1]. I-pictures are the most important pictures and thus cannot be dropped, while a

limited drop of a P- or B- pictures is acceptable. We therefore investigate the effectiveness of D_ALoC under the restriction that some requests are undroppable (i.e., I-pictures). This variation of D_ALoC is called "Dyn-Mark".

We show response times using the actual traces as a service process of the streaming server for random, ALoC, D_ALoC, and "Dyn-Mark" policies. Figure 4.24 plots the average response time for the various policies when 7%, 12%, and 16% of the total requests are dropped. Additionally, ALoC is tuned such that it achieves its best performance for the target drop ratio. The relative performance improvement of the various policies compared to a no drop policy is consistent with the previous results: as the drop ratio increases, the response times significantly decrease. D_ALoC self-adjusts its configuration parameters and achieves closely as good response times as the carefully tuned ALoC. The restriction of dropping certain requests in Dyn-Mark results in a slight degradation in performance improvements, but nevertheless significant gains in comparison to no drop.

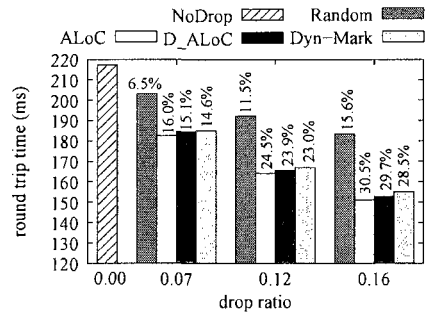


Figure 4.24: Average performance response times when 0% (no dropping), 7%, 12%, and 16% of the work in the second server (disk) is dropped. $N = 200$. The numbers above bars indicate the relative improvement over the NoDrop case.

Figure 4.25 plots the tails of the response time distribution for all policies. Results are consistent with those reported in the synthetic trace, further arguing for the robustness of D_ALoC even with drop restrictions.

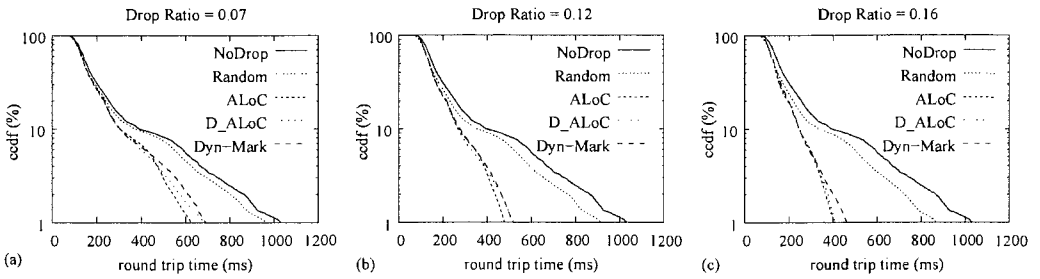


Figure 4.25: CCDFs of response times using the real traces. $N = 200$.

4.4 Chapter Summary

In this chapter, we proposed SWAP, a no-knowledge scheduling technique for increasing the performance of systems processing temporal dependent workloads. Temporal locality has been observed in several practical settings [78, 51, 64, 83], arguing for significant applicability of SWAP in real systems. Using simulation, we have shown that SWAP consistently improves performance, as quantified by the system mean throughput and by the distribution of round-trip times experienced by requests under temporal dependent conditions. We have shown that SWAP is able to effectively approximate SJF but without requiring additional knowledge on job service times.

We also proposed ALoC and D_ALoC, two autonomic load control policies that extend SWAP by selectively infinitely delaying or dropping queued requests and control load. Using temporal dependence, both policies are able to effectively guess the future service requirements of incoming jobs at a server and drop the load according to this forecasting information. Using extensive simulations, we have shown that ALoC and D_ALoC are able to reduce system response times for different workload intensities, levels of dependence, and target drop ratios. Experiments on synthetic traces show that the response time improvement of ALoC and D_ALoC typically varies between 50% and

80%. On a real trace where some requests are marked as "undroppable", both policies are still very effective, with a response time improvement between 15% and 30%. These results promote ALoC and D_ALoC as simple-to-implement policies for load control in autonomic systems.

Chapter 5

Performability of Systems with Background Jobs

Nowadays, computer systems are rarely taken off-line for maintenance. Even simple workstations are in operation 24 hours a day, 7 days a week. Consequently, most systems schedule necessary maintenance that intends to address system reliability and availability, as background tasks [2, 7, 62, 84] and serve them during idle times. Very often, background activity is also associated with approaches that aim at enhancing system performance [33, 90, 26].

Although background activity is critical to system operation, it often has lower priority than foreground work, i.e., the work requested by the system users. Therefore, it is of paramount importance for system designers to better understand the trade-offs between minimizing the foreground performance degradation and maximizing completion of background tasks so that system reliability, availability, and performance are improved in the long-run but without compromising the short-term performance of foreground work.

While facilitating and supporting background activity in a system is a general concept [88], its applicability differs among systems, i.e., distributed and clustered systems, storage systems, and communication systems. Consequently, efforts for utilizing idle time to improve reliability or performance are often system specific and are either based on prototyping and measurements [26, 2, 33, 90] or analytic models [7, 62, 66, 70].

In this chapter, we propose an analytic model that addresses performance trade-offs between foreground and background work at the disk drive level of a storage system. The analytic model consists of an infinite Markov chain with repetitive structure. This model captures the disk or storage system behavior under the background activity whose service demands are similar to the foreground activity. It differs from similar models proposed for storage systems [7] because it allows for modeling the effect of bursty arrivals, which are the case in storage systems [76]. The solution of the proposed model is tractable and can be solved using the well-known matrix-geometric method [49]. The model establishes that the relative performance of foreground and background jobs is similar for either independent or bursty arrivals. However, the saturation under bursty arrivals is very fast (for small changes in foreground workload), which actually effects more the completion rate of background jobs rather than the latency of the foreground ones.

5.1 Related Work

Multiple sources [33, 26, 76] indicate that computer system resources operate under bursty arrivals and while they have periods of high utilization, they may also have long stretches of idleness. For example, in average disk drives are only 20% utilized [76]. Given that a system operates in low utilization, a myriad of approaches have been proposed aiming at

utilizing idle times to improve performance [33, 26], fault tolerance [2], and reliability [84]. The goal is to schedule performance/availability enhancing activities as low priority and minimize their impact on user performance [26].

The motivation here stems from storage systems, where traditionally a variety of tasks, mostly aiming at enhancing data reliability, are treated as a background activity [7]. In a storage system, background functions that address reliability, availability, and consistency typically include data reconstruction [62], data replication [66], disk scrubbing [84], and WRITE verification [7]. Background jobs may also address storage performance issues including data replication in a cluster to improve throughput or data reorganization to minimize disk arm movement [33, 90].

Because background activity has often low priority, its service is completed only when there is no foreground activity in the system, i.e., at the end of a busy period. Vacation models have been proposed for the general performance analysis of systems where foreground/background jobs coexist [88, 70, 70, 91, 99]. To the best of our knowledge, vacation models that are applied in storage systems or disk drives have been considered only in [7]. The models in [7] attempt to model a system whose arrival process is strictly exponential and the background task results from sequential scanning of the data on a disk or part of it. In this chapter, we explicitly model the performance effects of dependence in the arrival process of background/foreground jobs on the disk, which has been detected in [36, 76, 34]. We examine the effects of both variability and dependence in the arrivals. We further assume that background and foreground jobs are drawn from the same distribution because we are interested in the set of background activities such as WRITE verification that have the same service demands as the user requests.

5.2 Storage System

In this section, we first identify the salient characteristics of IO workloads and we give an overview of the operation of the system with foreground and background tasks.

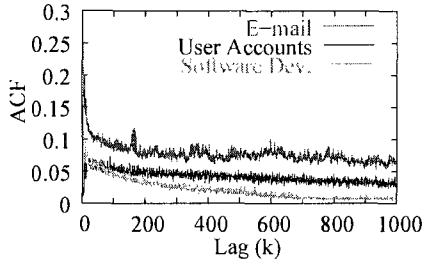
5.2.1 Workload Parameterization

In storage systems and disk drives, the arrival process is bursty [36, 76]. Here, we look at traces measured in different storage systems [76] that show high burstiness in the arrival streams of requests.

Figure 5.1 presents the autocorrelation function (ACF) of the inter-arrival times of three traces that have been collected in three different systems, each supporting an e-mail server, a software development server, and user accounts server, respectively. These traces consist of a few hundred thousands entries each and are measured over a period of 12 to 24 hours. As expected, for different applications the dependence structure of the arrivals is different and it is a result of multiple factors including the architecture of the storage system, the file system running on top of the storage system, and the I/O path hierarchy together with the resource managing policies at all levels of the I/O path. Nonetheless, independently of all these factors, all measurements show that arrivals at the storage system exhibit some amount of autocorrelation.

The table in Figure 5.1 shows the mean and coefficient of variation (CV) for the inter-arrival times and the service times of all requests in the trace. The three traces represent systems under different loads. Specifically, the "User Accounts" trace comes from a lightly loaded system (only 2% utilized), while the "E-mail" and "Software Development" traces come from systems with modest utilizations also ("E-mail' is 8% utilized and "Software

Development" 6% utilized). These cases of underutilized systems naturally indicate that an opportunity exists for scheduling low priority jobs in the system and treating them as background work. Additionally, the low utilization levels in the above measurement traces allow to assume that the measured job response times are a close approximation of the workload service times. Because all storage systems in Figure 5.1 consist of similar hardware, the service process is similar across all traces and it actually has low variability, i.e., CV values are less than 1.

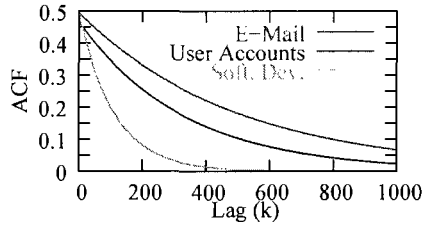


	Inter arrival times		Service times		Utili-
	mean	CV	mean	CV	zation
E-mail	56.93	9.01	5.59	0.75	8%
Soft. Dev.	88.06	12.38	6.34	0.84	6%
User Accs,	246.65	3.85	6.1	0.74	2%

Figure 5.1: ACF of inter-arrival times of three traces, the respective mean (in ms) and CV of the inter-arrival and service times.

We propose models of the arrival and service processes in a storage system that reflect the characteristics of the various traces illustrated in Figure 5.1. We model the service process via an exponential distribution with mean service time of 6 ms. For the arrival process, we use a two-state Markovian Modulated Poisson Process (MMPP) [42, 57], see

Chapter 2. We parameterize three different MMPPs to model separately the three different arrival processes of our traces. The MMPPs are labeled as "E-mail", "User Accounts", and "Software Development" and are used as input to the analytic model that we develop here. We stress that the MMPP models used here do not represent an exact fitting of the traces in Figure 5.1, they only match the first two moments of the trace and provide a range of different autocorrelation functions. Workload fitting such that the ACF is matched exactly, is outside the scope of this dissertation. In Figure 5.2, we show the ACF of the three MMPPs used here and their full parameterization.



	v_1	v_2	l_1	l_2
E-mail	0.31e-5	0.69e-6	*0.09*	0.35e-3
Soft. Dev.	0.90e-6	0.19e-5	*0.10e-3*	0.35e-1
User Accs,	0.36e-4	0.13e-5	*0.10e-1*	0.49e-3

Figure 5.2: ACF of our 2-state MMPP models for the interarrival times of the three traces and their parameterization.

5.2.2 Background Tasks in Storage Systems

There are numerous cases where storage systems and disk drives deal with background jobs[1]. One widely accepted background task is data integrity check or media scrubbing in

[1]The terms "task" and "job" are used interchangeably.

disk drives [84]. Disk scrubbing is a periodic checking of disk media to detect unaccessible sectors. If a sector is not accessible then it is reported up to the file system for data recovery and it is remapped elsewhere on the disk. Another background activity in disk drives is the RAID rebuild process [87, 62], which happens when one disk in a RAID array fails and its data is reconstructed in a spare disk using the data in the remaining disks of the array. Other examples of background activities include flushing of write-back caches, prefetching, and replication [87].

Background tasks in a storage system may be periodic such as disk scrubbing, or may span over a long period of time, such as the RAID rebuild. Yet, there are background tasks which have the same service demands as the foreground ones. For example, disk WRITE verification incurs one extra READ to detect any disk WRITE error. This process, known as READ-after-WRITE, degrades disk performance substantially and is not feasible if running in foreground, but is attractive as a low priority background activity. Nevertheless, its successful completion is tightly related to the reliability and consistency of the data.

We model a simple storage system with one service center, where foreground jobs are served in a first-come first-serve (FCFS) fashion. We assume that the amount of available buffer space is *always* large enough to store all data associated with waiting foreground tasks in the queue. Therefore, the above system is approximated by an infinite-buffer queue.

Foreground jobs consist of user arrivals *only*. Upon completion, a foreground job may either leave the system with probability $(1 - p)$, or generate a new background job with probability p, i.e., background tasks are only a portion of foreground tasks and have service demands with the same stochastic characteristics as the foreground jobs. Think of

WRITE verification; only a portion of all user requests are WRITEs and they need to be verified once they are serviced by the disk. Background tasks are served in a "best-effort" manner: a background job will get served only if there is no foreground job waiting in the queue, i.e., during idle periods. Consequently, background tasks will ordinarily have longer waiting times than foreground tasks.

Neither foreground nor background tasks are preemptive, which is consistent with the nature of work in disk drives, where the service process consist of three distinct operations, i.e., seek to the correct disk track, position to the correct sector, and transfer data. The "seek" portion accounts in average for 50% of the service time and is a non-preemptive operation [46, 85]. Because of the non-preemptive nature of seeks, background activity inevitably impacts foreground work performance: if a background task starts service, then this precludes the existence of any foreground task in the system, but if a foreground job arrives during the service of a background job, it will have to wait in the queue and on the average experience longer delay than the delay it would have experienced if the system was not serving background tasks. To minimize this effect, background tasks do not start service immediately after the end of a foreground busy period, but after the system has been *idle* for some pre-specified period of time, which we refer to as "idle wait".

Background jobs, similarly to the foreground ones, require buffer space. Because the buffer is reserved for foreground jobs, background buffer is limited. As a result, some of background tasks will be dropped because the buffer is full. A practical setting in a disk drive would be to allocate 0.5-1MB of buffer space for background activity, which corresponds to approximately 50 background jobs of average size. Throughout the chapter, we assume a buffer that stores a maximum of 50 background jobs. We also examined buffer

sizes for up to 250 background jobs and the results are qualitatively same as those with buffer size 50.

5.3 The Markov Chain

In this section, we describe a Markov chain that models the queueing system with foreground/background activity as described in the previous section. To simplify the definition of the state space as well as transitions among states, we first assume exponential inter-arrival and service times with mean rates λ and μ, respectively. Later, we show how the exponential inter-arrival process is replaced by the 2-stage MMPP process. The Markov chain of the foreground/background activity is depicted in Figure 5.3. Because foreground jobs use an infinite buffer and the background jobs use only a finite one, the Markov chain is infinite in one dimension only. For presentation simplicity, Figure 5.3 shows the instance where the background buffer can store up to 2 background jobs only.

The state space is defined by a 2-tuple (x, y), where x indicates the number of background tasks in the system (waiting or in service) and y indicates the number of foreground tasks in the system (waiting or in service). There are two sets of 2-tuples in Figure 5.3: (x, y) and (x', y). States (x, y) indicate that a foreground job is being served. States (x', y) show that a background job is being served. The "idle wait" is represented by states $(x, 0)$, where $x > 0$ means that the background jobs wait for a time period, which is exponentially distributed with mean $1/\alpha$, before starting.

We define levels in this Markov chain such that level j consists of the set of states $\mathcal{S}^{(j)}$

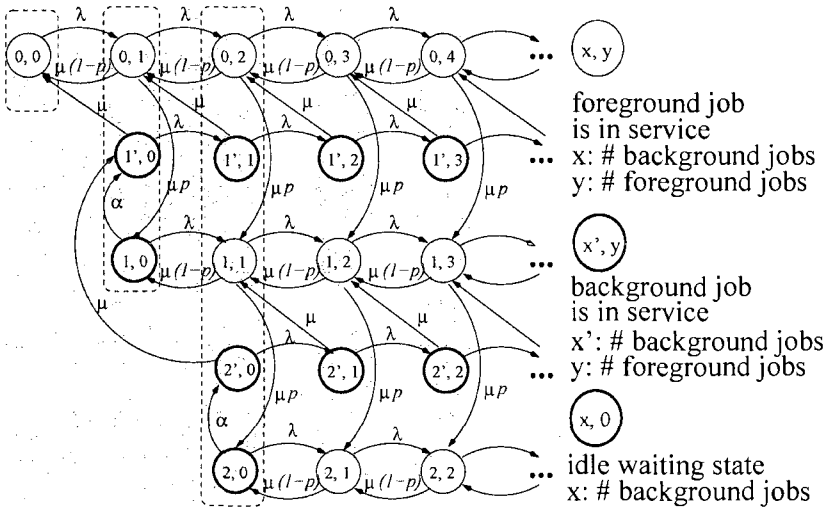


Figure 5.3: The Markov chain of the queueing system with infinite buffer size for foreground tasks and a buffer size of 2 for background tasks.

defined as

$$
\begin{aligned}
\mathcal{S}^{(j)} \;=\; \{(x,y) \;\text{ and }\; (x',y) \;\mid\; & \\
& 0 \le x \le j, \;\; 0 \le y \le j, \;\; x+y=j \;\text{ and} \\
& 0 \le x' \le j, \;\; 0 \le y \le j, \;\; x'+y=j\}. \qquad (5.1)
\end{aligned}
$$

Let the maximum buffer size of the background jobs be X. Until there are X tasks in the system, the Markov chain has a tree-like structure. Beyond that point, the background buffer could be full and the levels of the Markov chain form a repetitive pattern. The form of the chain is that of a Quasi-Birth-Death process (QBD) which can be solved using matrix-analytic methods [49].

5.3.1 Modeling Dependence in the Arrival Process

Here, we enhance the simple Markov chain model to capture arrival streams with high variability and various degrees of dependence in their inter-arrival structure using a 2-state Markov Modulated Poisson Process (MMPP). Each state in the Markov chain of Figure 5.3 is now replaced by a set of sub-states, and scalars λ, μ and α are replaced by matrices $\mathbf{F}$, $\mathbf{B}$, and $\mathbf{W}$, respectively. An additional matrix $\mathbf{L_0}$ is used to describe transitions within a set of sub-states. Assume that $\mathbf{D}_0^{(A)}$ and $\mathbf{D}_1^{(A)}$ describe an A-state MMPP. Then $\mathbf{L_0}$, $\mathbf{F}$, $\mathbf{B}$, and $\mathbf{W}$ are $A \times A$ matrices computed by the following equations.

$$\mathbf{F} = \mathbf{D}_1^{(A)},\ \mathbf{B} = \mathbf{I}_A \times \mu,\ \mathbf{W} = \mathbf{I}_A \times \alpha,\ \mathbf{L_0} = (\mathbf{D}_0^{(A)})^{(*)}, \tag{5.2}$$

where $\mathbf{I}_A$ is an $A \times A$ unit matrix and $(\mathbf{D}_0^{(A)})^{(*)}$ is equal to $\mathbf{D}_0^{(A)}$ except that diagonal elements are all 0. Note that the service time and the idle waiting time are exponentially distributed in our model. However, a similar method and Kronecker products can be used to generate the auxiliary matrices $\mathbf{F}$, $\mathbf{B}$, $\mathbf{W}$, and $\mathbf{L_0}$ when use a MMPP (or MAP) for the service and idle waiting processes. Therefore, we construct a new Markov chain and its corresponding infinitesimal generator $\mathbf{Q}$ by replacing each state in Figure 5.3 with a set of A sub-states and use $\mathbf{F}$, $\mathbf{B}$, $\mathbf{W}$, and $\mathbf{L_0}$ to describe its state transitions. The resulting Markov chain is also a QBD process.

Figure 5.4(A) illustrates the transitions between the sub-states corresponding to states (x, y), $(x, y+1)$ and $(x+1', y)$ in Figure 5.3. If we do not draw the detailed state transitions, but simply substitute λ, μ and α in Figure 5.3 with matrices $\mathbf{F}$, $\mathbf{B}$ and $\mathbf{W}$, and add $\mathbf{L_0}$ to describe the local state transitions, we obtain the matrix-based transitions in Figure 5.4(B). According to Eq.(5.2), $\mathbf{F}$, $\mathbf{B}$, $\mathbf{W}$, and $\mathbf{L_0}$ of a system with two-state

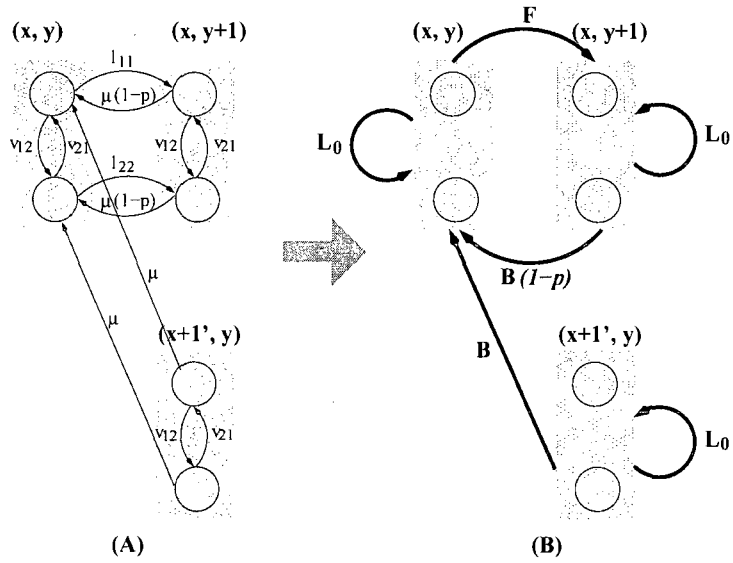


Figure 5.4: Changes in the Markov chain of Figure 5.3 when the arrival process is a 2-state MMPP.

MMPP arrivals are computed as follows:

$$\mathbf{F} = \begin{bmatrix} l_1 & 0 \\ 0 & l_2 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mu & 0 \\ 0 & \mu \end{bmatrix},$$
$$\mathbf{W} = \begin{bmatrix} \alpha & 0 \\ 0 & \alpha \end{bmatrix}, \quad \mathbf{L_0} = \begin{bmatrix} 0 & v_1 \\ v_2 & 0 \end{bmatrix}, \tag{5.3}$$

where v_{12}, v_{21}, l_{11}, and l_{22} are the parameters of the 2-state MMPP model in Eq.(2.8). One can easily show the equivalence of state transitions in Figure 5.4(A) and Figure 5.4(B).

The infinitesimal generator $\mathbf{Q}$ of this new Markov chain can be obtained from Figure 5.3. For each level i corresponding to the $(i+1)^{th}$ column in Figure 5.3, the stationary

state probabilities are given by the following vectors:

$$\begin{aligned}
\boldsymbol{\pi}^{(i)} &= [\boldsymbol{\pi}^{(i)}_{(0,i)}, \boldsymbol{\pi}^{(i)}_{(1',i-1)}, \boldsymbol{\pi}^{(i)}_{(1,i-1)}, \cdots, \boldsymbol{\pi}^{(i)}_{(i',0)}, \boldsymbol{\pi}^{(i)}_{(i,0)}], \\
&\quad \text{for } 0 \leq i \leq X, \\
\boldsymbol{\pi}^{(i)} &= [\boldsymbol{\pi}^{(i)}_{(0,i)}, \boldsymbol{\pi}^{(i)}_{(1',i-1)}, \boldsymbol{\pi}^{(i)}_{(1,i-1)}, \cdots, \boldsymbol{\pi}^{(i)}_{(X',i-X)}, \boldsymbol{\pi}^{(i)}_{(X,i-X)}], \\
&\quad \text{for } i > X.
\end{aligned}$$

$\boldsymbol{\pi}^{(i)}_{(x,y)}$ or $\boldsymbol{\pi}^{(i)}_{(x',y)}$ is a row vector of size A that corresponds to a set of sub-states under the MMPP arrival process. Then, $\boldsymbol{\pi}\mathbf{Q} = \mathbf{0}$ and $\boldsymbol{\pi}\mathbf{e} = 1$ where $\boldsymbol{\pi} = (\boldsymbol{\pi}^{(0)}, \boldsymbol{\pi}^{(1)}, \cdots, \boldsymbol{\pi}^{(X)}, \boldsymbol{\pi}^{(X+1)}, \cdots)$.

We solve the QBD using the matrix geometric solution [49]. The state space is partitioned into boundary states and repetitive states. Boundary states in the QBD of Figure 5.3 are the union of all levels i for $0 \leq i \leq X$. We use $\boldsymbol{\pi}^{[0]}$ to denote the stationary probability vector of these states, i.e., $\boldsymbol{\pi}^{[0]} = (\boldsymbol{\pi}^{(0)}, \boldsymbol{\pi}^{(1)}, \cdots, \boldsymbol{\pi}^{(X)})$. Each level i for $i > X$ represents a repetitive set of states. Key to the matrix geometric solution is that a geometric relation holds among the stationary probabilities of the repetitive states, i.e.,

$$\forall i > X, \quad \boldsymbol{\pi}^{(i)} = \boldsymbol{\pi}^{(X+1)} \cdot \mathbf{R}^{i-1}. \tag{5.4}$$

Here the matrix $\mathbf{R}$ is a squared matrix of dimension equal to the cardinality of repetitive levels, and can be computed using an iterative numerical algorithm [49]. By computing $\boldsymbol{\pi}^{[0]}$ and $\boldsymbol{\pi}^{(X+1)}$ as in [49] one can easily generate the entire infinite stationary probability vector for the QBD. Thanks to the geometric relationship in Eq.(5.4), several metrics can be computed in closed form formulas.

Let $\mathbf{e}^{(i)}$ be a column vector of 0's with appropriate dimension except the $(2i \cdot A + 1)^{\text{th}}$ to the $(2i \cdot A + A)^{\text{th}}$ elements that are equal to 1, and let $\mathbf{e}^{(i')}$ be another column vector

of 0's except the $((2i-1)\cdot A+1)^{\text{th}}$ to the $((2i-1)\cdot A+A)^{\text{th}}$ elements that are equal to 1, for $i \geq 0$. Note that all the elements of $\mathbf{e}^{(0')}$ are equal to 0. Both $\mathbf{e}^{(i)}$ and $\mathbf{e}^{(i')}$ are of size A, where A is the order of the arrival MMPP process. The average queue length of the foreground jobs $QLEN_{FG}$, the completion rate (or admission rate) of the background jobs $Comp_{BG}$, and the percentage of foreground jobs waiting behind background jobs $WaitP_{FG}$ can be calculated as follows.

$$
\begin{aligned}
QLEN_{FG} &= \sum_{i=1}^{X}\sum_{j=0}^{i-1}((i-j)*(\pi^{(i)}_{(j,i-j)}+\pi^{(i)}_{(j',i-j)})\mathbf{e}) \\
&+ \sum_{i=0}^{X}(X+1-i)\pi^{(X+1)}(\mathbf{I}-\mathbf{R})^{-2}(\mathbf{e}^{(i)}+\mathbf{e}^{(i')}) \ , \\
Comp_{BG} &= 1-\frac{\pi^{(X+1)}(\mathbf{I}-\mathbf{R})^{-1}\mathbf{e}^{(X)}}{1-\sum_{i=0}^{X}\pi^{(i)}_{(0,i)}\mathbf{e}-\pi^{(X+1)}(\mathbf{I}-\mathbf{R})^{-1}\mathbf{e}^{(0)}} \ , \\
WaitP_{FG} &= \frac{\sum_{i=2}^{X}\sum_{j=1}^{i-1}\pi^{(i)}_{(j',i-j)}+\sum_{i=1}^{X}\pi^{(X+1)}(\mathbf{I}-\mathbf{R})^{-1}\mathbf{e}^{(i')}}{1-\sum_{i=0}^{X}(\pi^{(i)}_{(i,0)}+\pi^{(i)}_{(i',0)})\mathbf{e}} \ .
\end{aligned}
$$

5.4 Performance Evaluation Results

Here, we use the analytic model to analyze the performance of a storage system that serves foreground and background jobs, as described in the previous section. The model is parameterized using the E-mail and Software Development traces (see Figure 5.1). This parameterization results in the MMPPs of Figure 5.2 which have different mean, CV, and dependence structure, and we consider representative. The User Account trace performs qualitatively the same as the E-mail trace because of its strong ACF structure.

We evaluate the general performance of the system as a function of system load. In this section we use interchangeably the terms "load" and "utilization". Foreground load is a function of the mean of the arrival process in the system (i.e., the mean of the MMPPs in Figure 5.2) while background load is a function of p, i.e., the probability that a foreground generates a background job upon its completion. We scale the mean of the two MMPPs in Figure 5.2 to obtain different foreground utilizations. We also scale the value of p between 0.1 and 0.9 to obtain different background loads. The mean "idle wait" time for a background job before starting service during an idle period is equal to the mean service time, unless otherwise stated. The background buffer size is 50.

5.4.1 Performance of Foreground Jobs

First, we report on the performance of foreground jobs. Figure 5.5 presents the average queue length of foreground jobs, which sharply increases as a function of foreground load. This increase is nearly insensitive to different p values, showing that foreground load determines overall system performance. Note that for strong dependent arrivals ("E-mail" MMPP) the saturation is reached much faster than for arrivals with weak dependence structure ("Software Development" MMPP). We return to the question of how intensity in the dependence structure of the arrival process affects system performance later in this section.

Figure 5.6 shows the percentage of foreground jobs that are delayed because of background jobs. As background load increases, the portion of foreground jobs that are delayed increases, but as foreground load increases, the portion of foreground jobs that are delayed decreases. In the worst case scenario that we present here, i.e., for $p = 0.9$, only

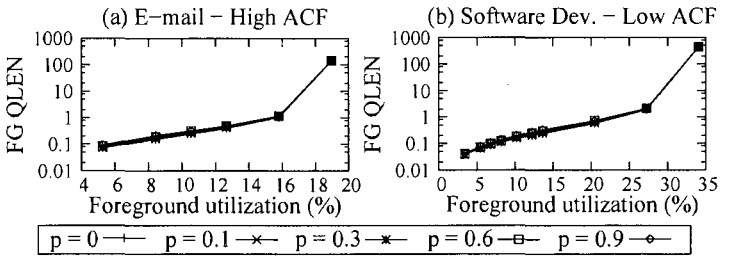


Figure 5.5: Average queue length of foreground jobs for (a) the Email and (b) the Software Dev. traces as a function of foreground load.

20% of foreground jobs are delayed, which shows that most foreground jobs maintain their expected performance. The most interesting point in Figure 5.6 is that when the (total) load increases beyond a certain point then the portion of foreground jobs that are affected decreases dramatically, which is explained by background jobs performance in the next subsection.

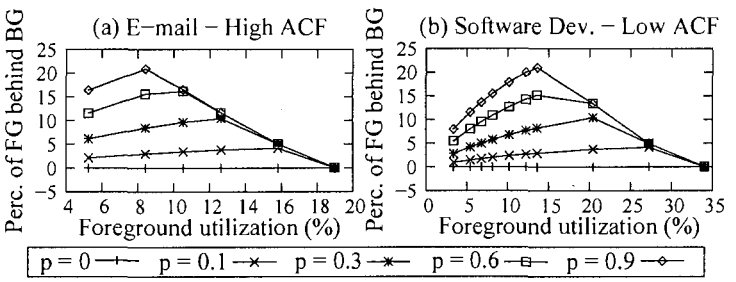


Figure 5.6: Portion of foreground jobs delayed by a background job for (a) the Email and (b) the Software Dev. traces as a function of foreground load.

5.4.2 Performance of Background Jobs

We measure the performance of background jobs by the portion of background tasks that complete. This metric is directly related to reliability (or long term performance benefits) of background activity. Results are given in Figure 5.7, which shows that as load increases, the completion rate decreases to zero, independent of load or dependence structure. For

arrivals with a strong dependence structure, (i.e., of "E-mail"), this point comes sooner than for arrivals with weak dependence structure, (i.e., the "Software Development"), see the range of the x-axis in Figure 5.7. Note that the completion rate of the background activity relates to the probability of the background buffer being full, which supports the observation that the strong dependence structure in arrivals increases the queue length of background jobs, as illustrated in Figure 5.8.

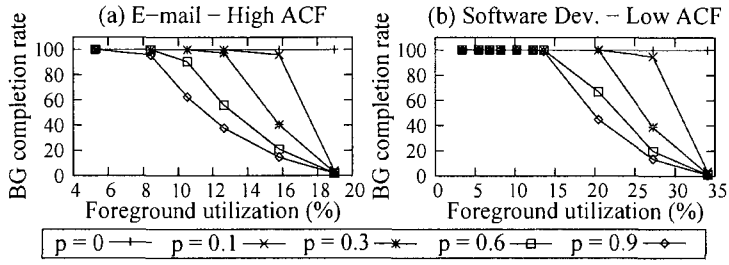


Figure 5.7: Completion rate for background jobs for (a) the Email and (b) the Software Dev. traces as a function of foreground load.

Figure 5.8 shows the average queue length of background jobs. Consistent with results in Figure 5.7, Figure 5.8 shows a similar qualitative behavior across the two workloads. Quantitatively, the average queue length of the strong dependent workload is smaller than that of the weak dependent workload because more background jobs are dropped.

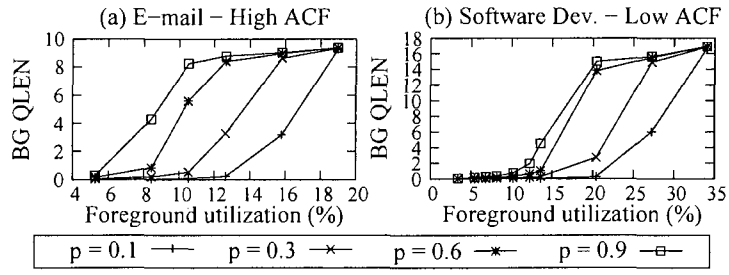


Figure 5.8: Average queue length of background jobs in the workloads (a) Email and (b) Software Dev. as a function of foreground load.

5.4.3 Effect of "Idle Wait" Duration

An important design issue in a storage system that serves foreground and background jobs is the length of the "idle wait" period, i.e., the time that the system operates in non-work-conserving mode. The shorter the duration of "idle wait", the higher is the performance degradation of foreground jobs.

In Figure 5.9, we show how the length of "idle wait" affects the average queue length of foreground jobs under different background loads. These experiments are conducted for the parameterization of the actual traces given in Figure 5.2. Increase in "idle wait" does improve foreground performance, because it reduces the number of foreground jobs delayed by serving background jobs.

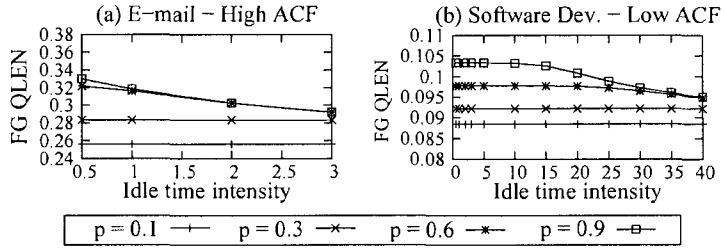


Figure 5.9: Foreground jobs average queue length for (a) the Email and (b) the Software Dev. traces as a function of idle wait (in multiples of service time).

However, improvement of foreground performance does come due to a considerable drop in background completion rate, as shown in Figure 5.10. For example, in the case of "E-mail" parameterization under an "idle wait" of twice the service time and $p = 0.6$, the completion rate of background jobs drops by 20% compared to the completion rate when the idle wait is half of service time, but the foreground performance gains are as low as 6.5% (the average foreground queue lengths are 0.32 and 0.30 when idle wait is twice of the service time and when idle wait is half of the service time, respectively). Given the

long-term benefits of background activity, maintaining a small "idle wait" period, close to the average service time, is beneficial for sustaining foreground job performance and high background completion rate.

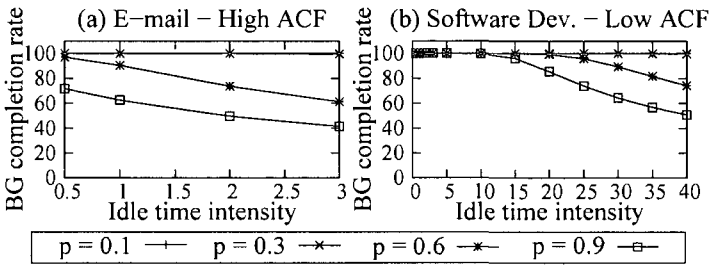


Figure 5.10: Completion rate for background jobs in the workloads (a) Email and (b) Software Dev. as a function of idle wait (in multiples of service time).

5.4.4 The Impact of Dependence in the Arrival Process

In this subsection, we analyze the effect that the arrival process has on a system with background jobs. Using only the "E-mail" workload parameterization. We remark that qualitatively similar results can be obtained using the other two workloads. we examine the performance effects of Poisson arrivals, of an Interrupted Poisson Process (IPP) (a process with high variability but no correlation [35]) and of two MMPP processes with low and high dependence structure. All these processes have the same mean and CV as the measured in the arrival process of "E-mail" trace, with the only exception of the Poisson arrival process that maintains the same mean only. Results show that the dependence structure of the arrival process determines the sensitivity of system performance toward load changes, that is, the stronger the dependence structure, the higher the sensitivity toward system load.

Figure 5.11 shows the average queue length for foreground jobs under two different

loads of background jobs, i.e., p equal to 0.3, and 0.9. There is a dramatic queue length increase under autocorrelated arrivals, that is orders of magnitude higher than the queue length increase with exponential inter-arrivals. Even at 19% foreground utilization under the strong correlated arrivals the foreground queue length reaches 100. Such queue length is reached only under 95% foreground utilization for the Poisson arrivals. For comparative purposes, we plot the results using different scales on the x-axis, separated by a vertical line. Consistent with the results in Figure 5.5, high foreground load rather than foreground load determines overall foreground performance.

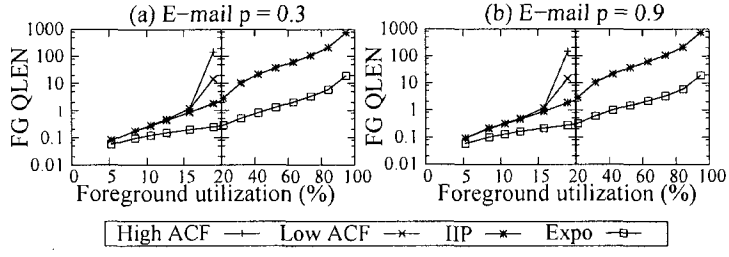


Figure 5.11: Average queue length for foreground jobs for the "E-mail" workload as a function of foreground load in the system.

In Figure 5.12, we show completion rates for background jobs as a function of foreground load. There are cases when under high foreground load, there is nearly a 100% difference in performance between exponential and correlated arrivals. The system simply saturates faster under correlated arrivals and does not have the capacity to serve background tasks. Therefore, under correlated arrivals light background load should be sustained to ensure acceptable background completion rates.

Finally, Figure 5.13 shows the percentage of foreground jobs delayed by background jobs as a function of foreground load. Interestingly, the figure shows that the worst impact on foreground jobs is contained within a limited range which is reached faster under highly

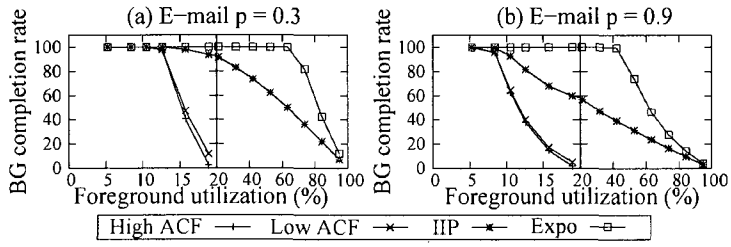


Figure 5.12: Completion rate of background jobs for the "E-mail" workload as a function of foreground load in the system.

correlated arrivals than independent arrivals. In a dynamically changing environment with correlated arrivals, the system regulates itself faster to sustain foreground job performance than under independent arrivals.

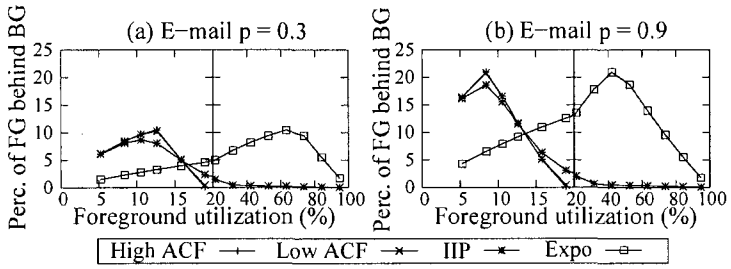


Figure 5.13: Portion of foreground jobs delayed by a background job for the "E-mail" workload as a function of foreground load in the system.

5.5 Chapter Summary

In this chapter, we presented an analytic model for the evaluation of disk drives or storage systems with background jobs. Because of the non-preemptive nature of work in disks (e.g., seeks), background work inevitably affects performance of foreground work. The proposed model allows to evaluate the trade-offs between foreground and background activities. Our model incorporates most important characteristics in storage systems workloads, including burstiness and dependence in the arrival process. The model results in a Markov chain of

a QBD form that is solved using the matrix-geometric method.

Experiments show that, independent of workload characteristics, the non-preemptive background jobs minimally impact performance of foreground jobs. However, sustained foreground performance under worst case scenarios is a result of low background completion rates, which suggests that background load must be kept modest to benefit system reliability or performance in the long-term. Via sensitivity analysis of the "idle wait" duration, we show that increasing the length of "idle wait" reduces the degradation on foreground performance but also decreases the background completion rate. The results suggest that it is critical to maintain a small or moderate "idle wait" period, e.g., close to the average service time, for gaining sustained foreground job performance and high background completion rate. We have also shown that under bursty arrivals, both foreground and background performance strongly depends on system load. In particular, the background completion rate becomes significantly sensitive to system load for autocorrelated arrivals, which indicates that workload burstiness is an important factor to determine the amount of background work in the system.

Chapter 6

Background Scheduling in Storage Systems

In this chapter, we focus on how to utilize the idleness resource for serving background jobs without degrading the foreground performance beyond the predefined target. Because foreground tasks are of high priority, background tasks are served only when there are no foreground jobs in the system, i.e., during system idle times. The non-preemptive nature of background tasks coupled with the stochastic nature of the system makes serving background tasks challenging if delays on foreground jobs are limited within predefined targets. Therefore, serving background tasks must meet two conflicting goals:

1. foreground performance degradation should be contained within predefined targets;
2. background work should not be starved and its throughput should be maximized.

Efficient use of system idle times to serve background jobs is key to meet the above goals.

System specific efforts to use idle times for improving performance and availability in the context of a *specific* system feature have been presented in the literature [2, 90, 41,

24, 7, 62, 70] and are usually evaluated based on prototyping, measurements, or analytic models. Here, we define a general *schedulability* algorithm that determines *when* and for *how long* the system can serve background jobs during idle times such that background throughput is maximized and performance degradation of foreground jobs remains within a pre-defined range. Here, we do not focus on the problem of background job scheduling. The specific scheduling of background jobs, i.e., their service order, is outside the scope of this work.

6.1 Related Work

Various studies have shown that in systems, periods of high utilization may be interleaved with long stretches of idleness [53, 33, 26, 77]. A myriad of approaches have been proposed to best utilize idle times in order to enhance system performance, reliability, and consistency. System idleness may be exploited locally (i.e., within the same system), or remotely (i.e., busy systems may offload part of their work in idle ones).

Systems that serve locally both foreground and background tasks are considered as system that serve different priority tasks [88]. However, as it becomes more common for systems to operate 24/7, idle times offer the only time window to complete maintenance work [33, 44, 90, 2, 84, 7]. Consequently, the general problem of idle-time scheduling has recently regained attention [33, 24, 26] as a distinct problem within the larger and well studied problem of priority scheduling [88].

Utilization of remote idleness is often exploited in distributed or peer-to-peer systems and focuses on identifying idle remote systems to complete some work remotely. V-system [89] and Condor [53] are examples of such systems. Other studies on utilizing

remote idleness are presented in [71, 56]. On the analytic side, several models have been developed for analysis of systems where foreground/background jobs coexist, including vacation models [70, 91, 99] and queueing models of cycle stealing [88, 71].

The main performance pitfall of scheduling background tasks during idle times relates to cases where background jobs cannot be preempted instantaneously and foreground performance may be significantly affected. If tasks are non-preemptive, effective scheduling of background tasks is more challenging. [26] focuses on managing idle intervals under a wide range of characteristics for background and foreground tasks and first defines the notion of the preemption interval or idle wait period that delays execution of background jobs in idle periods. This technique avoids using short idle intervals to schedule long background jobs. Efforts to adaptively determine the amount of time that the system should idle wait are proposed in [25, 41] for power saving in mobile devices by spinning-down their disks.

The closest to the work presented here is the one presented in [26]. In this dissertation, we depart from previous work by presenting a methodology to maintain foreground performance while avoiding background starvation, by estimating the amount of work to be completed in any idle interval, in addition to the estimation of the idle wait. Furthermore, the estimation of the idle wait and per-interval background work is based not only on the characteristics and performance of the foreground and background jobs, but also on the characteristics of idle intervals. We identify when idle wait is effective and when it is not (i.e., delay should be set to zero) based on the histogram of the observed idle times. More importantly, we propose ways to exploit burstiness in idle times (if burstiness exists) to best utilize pairs of long idle intervals, resulting in superior system performance.

6.2 Characterizing Idleness

Viewing system idleness as a resource, we first develop an understanding of the significance of the system idle times characteristics. Our goal is to develop policies that sustain system's foreground performance without starving background work. Via idle intervals characterization, we aim at estimating as accurately as possible how much background activity can be packed into an idle time period.

The stochastic characteristics of idle times are a result of the complex interaction of arrival and service processes in the system. Instead of deriving characteristics of idle times through analysis of the arrival and service processes, we concentrate on idle intervals themselves, which capture the interaction of the arrival and service processes. Idle intervals are viewed here as a separate stochastic process.

The characterization is based on two dimensions. First, variability in idle times can provide a lot of useful information for scheduling. Second, it is critical to find out whether there is any burstiness in the sequence of idle intervals (i.e., if the lengths of consecutive idle times are correlated) or whether consecutive intervals are independent from one another.

6.2.1 Independent Idle Intervals

First, we focus on independent idle intervals. If the sequence of idle intervals is independent, then short past history does not determine the short future. Therefore, all information about the idle intervals process can be extracted by its empirical distribution function, which can be easily constructed via on-line monitoring. After computing the first two moments of the monitored idle times sample, i.e., mean and variance, the coefficient of variation (CV) is calculated. The CV gives an indication about the existence of tails in

the empirical distribution of idle times.

Figure 6.1 depicts the cumulative distribution function (CDF) of two stochastic processes, one with low CV (left plot) and the other one with high CV (right plot). For simplicity, the x-axis in Figure 6.1 gives values that are normalized by the mean, e.g., "0.5" on the x-axis corresponds to a value that is half of the mean of the empirical distribution.

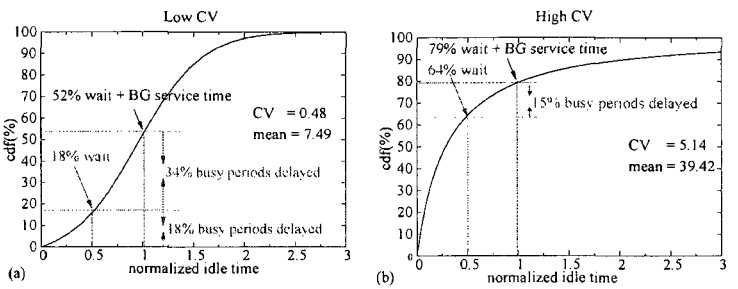


Figure 6.1: CDF of idle times with (a) low CV and (b) high CV. The x-axis gives idle times normalized by their mean.

Observing the long tail in the right plot of Figure 6.1 is straight-forward: the CDF line goes toward 100% with a much lower pace than the CDF of the left plot. The CDFs can therefore provide important information about the majority of upcoming intervals. For idle intervals with low CVs, the mean of the empirical distribution provides a good guess about the idle interval length. For idle intervals with high CVs, there is a large percentage of intervals that are much shorter than their mean and a small percentage of intervals that are much longer than their mean. This useful information on the anticipated length of future idle intervals is embedded on the CDF and proves to be tremendously useful for efficient background scheduling, particularly in determining the idle-wait length and the amount of background work to be scheduled in any given idle interval.

The intuition behind idle waiting relates to the relative lengths of foreground and

background jobs. Determining the delay parameter should depend on the length of the background job to be scheduled and the length of inter-arrival times [26], but this is not equally effective for idle intervals of low CV or high CV. We illustrate this via a simple example.

Following the assumptions in [26], in a system where the expected background service time is half of the expected idle times mean, the idle wait may also be equal to the expected background service. If this idle-wait policy were used for the case where the idle intervals are of low variability (see Figure 6.1, left plot), then 18% of the idle intervals would not be utilized by background jobs. Since in a system every idle period is followed by a busy period that starts when a foreground job arrives and finds the system idle, the same amount (i.e., 18%) of the busy periods would not be affected by background service either. If only one background job is scheduled after an idle wait period equal to the average background service time (i.e., a normalized idle time of 0.5), it is expected that 34% of all idle periods (52% of idle intervals with normalized idle time = 1, minus 18% of idle intervals with normalized idle time = 0.5, which equals to 34%) would be serving a background job, while the next foreground job arrives (which causes undesired delays on foreground work). A more conservative idle wait policy that waits longer than half the mean value would affect still a sizable but smaller percentage of foreground jobs. In contrast, if idle wait is zero and average background duration is half of the mean of idle intervals length, then only 18% of the busy periods in the system are affected if only one background job is served. Consequently, we conclude that no idle wait is necessary for systems with idle intervals of low variability.

The policy of idle waiting for half of the expected value of idle times affects foreground

jobs very differently if idle intervals are of high variability. The high CV plot of Figure 6.1 shows that by idle waiting for a period equal to the expected background service time, 64% of all idle intervals are not used to service background jobs. If the expected length of a background job is equal to half of the mean of the idle times and one background job is served in one idle interval, then only 15% of busy periods are affected by the background work. If idle intervals have high CV, then idle waiting helps exploiting the "longer" intervals at the tail of the distribution.

The above two examples of low and high CV highlight the disadvantage of a scheduling policy that uses a "fixed" idle waiting period as in [26]. The idle waiting period should be adapted according to the characteristics of foreground and background service demands, and idle times.

6.2.2 Bursty Idle Intervals

We now turn to bursty idle intervals. All discussion of the previous subsection applies here as well. In addition, burstiness in the sequence of idle times provides extra information which can be used for the prediction of the (short) future [33]. Burstiness in a sequence implies that among all the observed values in the sequence, the *order* of their occurrence is not random as it is in the independent case. In a sequence that is characterized as bursty, very large (multiple times larger than the mean) or very small (multiple times smaller than the mean) values are sampled close to one another.

Figure 2.1 in Chapter 2 shows this effect in a bursty sequence of observations. In the independent case (see Figures 2.1 (a) and (b)), the probability of the next observation (idle time) being small or large does not depend on the value of the current observation, as

expected. However, in the bursty case (see Figures 2.1 (c) and (d)) the effect is different. Figure 2.1 (c) shows that if the current idle interval is small, then the probability to have a small idle interval in the next 9 lags is very high. Figure 2.1 (d) equivalently shows how the current large idle interval determines with high probability that the next observed idle interval would be large as well. This information, in addition to the one provided by the CDF, can be used to improve scheduling of background activity by allowing to complete more background work (during long idle intervals) without imposing additional delays on foreground work.

6.3 Background Scheduling Policy

Efforts on utilizing idle times to improve performance or reliability are often *system specific* and are evaluated in the context of a specific feature based on prototyping and measurements [33, 2, 90, 41] or analytic models [7, 62, 66, 70, 71]. In addition to the above system specific solutions, there are also efforts to evaluate the general concept of managing idle times in a system, by viewing idleness as an additional resource [24, 26]. ¿From the theoretical perspective, the performance of systems with foreground and background jobs (also viewed as systems with jobs of high and low priority) have been extensively analyzed via queueing theory (see [88] and references within).

Motivated by the above storage system examples, our focus is on the general problem of scheduling non-preemptive background jobs during system idle times. Our approach to this problem reflects the fact that storage system idleness is expected to be highly dependent on system workload. For example, variability and burstiness dominate workload and idleness characteristics in general purpose servers, including web servers, and file

servers. On the other hand, video streaming servers are expected to work under more deterministic workloads, and consequently the idleness reflects that characteristic. Here, idleness is considered as an additional system resource but its management is driven by the performance trade-off between maintaining desired levels of foreground performance while maximizing the completions of non-preemptive background tasks.

In contrast to previous work, which sustains foreground performance by letting the system to "idle wait" before a background job is scheduled causing sometimes background work starvation, we propose to complement "idle waiting" with the "estimation" of the amount of background work to be served in any given idle interval. Such an approach does not compromise the foreground performance and avoids starvation (if any) among background jobs by allowing background jobs to be served in as many idle intervals as possible. The end result is that the overall system is better utilized, while foreground performance targets are met.

We achieve such balance in the system by monitoring characteristics of foreground and background jobs, similarly to other works in the literature that focus on the same problem [26, 33]. In addition, we also collect measurements of the empirical distribution of idle times. Resource management of idle times is now done in a dynamic way, using statistical information not only on the foreground and background job demands, but also on the idle intervals of the system. All statistical information is collected online while the system is in operation, and is incorporated into scheduling policies.

Detailed analysis of various systems with different statistical characteristics of foreground/background jobs and idle times shows that the effectiveness of idle wait strongly depends on the variability of the empirical distribution of idle times. In systems with low

variability of idle times, idle waiting is not effective. The opposite holds for idle times of high variability. In both cases, the cumulative data histogram of idle times is used to dynamically determine the length of idle wait.

In addition, we show how to take advantage of the burstiness (if any) in idle intervals to improve background scheduling. Specifically, if burstiness in idle times exists, then additional information on the idle intervals can be derived which allows for more accurate prediction of upcoming idle interval lengths. This additional knowledge becomes extremely effective, in particular when the background non-preemption penalty on foreground performance could become severe (as it is the case when background jobs are long). Validation of our methodology using actual disk drive traces shows that monitoring stochastic characteristics of idle times, in addition to the characteristics of foreground and background tasks, is an effective way to manage idleness.

6.3.1 Background Activity in Independent Idle Intervals

Idle waiting is used as a technique to ensure that a desired level of foreground performance is sustained while serving non-preemptive background work. Because idle-waiting is a non-work conserving strategy, background jobs may suffer from starvation [26]. We avoid background starvation by coupling idle wait with the amount of work that can effectively complete within an idle interval. These two scheduling parameters are determined using the foreground and background service demands as well as the distribution of idle times themselves. We consider the following policies.

Mean-based: This policy serves as a base-line comparison [26]. When an idle interval occurs, no background job is scheduled during a delay period which is defined as the

mean service time of background jobs. After the delay period elapses, the system starts serving background jobs until a foreground job arrives.

CDF-based: Similar to mean-based, this policy starts serving background jobs after an idle wait until a foreground job arrives. Different from the mean-based policy, the CDF-based policy continuously monitors the empirical distribution (i.e., cumulative histogram) of idle intervals and the mean of background service times to dynamically calculate the idle wait time. Based on the analysis of Section 6.2, the CDF-based policy

- does not idle wait if the idle times have low CV, and
- estimates the idle wait based on the empirical distribution if the idle times have high CV.

CDF/w-estimates: This policy estimates the idle wait the same way as the CDF-based policy but is more conservative because it limits the number of background jobs to be served in an idle interval according to the following equation:

$$T \cdot \frac{90^{th} \text{ percentile of idle intervals - idle wait}}{\text{Average background service time}}, \tag{6.1}$$

where $0 < T \leq 1$ is a parameter that adjusts the estimated number of background jobs assuming that the interval is large (i.e., equal to the 90th percentile of idle times). This parameter controls the performance degradation of foreground jobs. As T increases, foreground performance degrades. T is adjusted to reflect variability in the distribution of idle intervals, i.e., T is close to 1 under idle intervals of high variability and less than 1 for low variable intervals.

In all of the above policies, if a new foreground request finds a background job in service, it waits until that background job completes.

Simulation Environment

The three policies are evaluated via simulation of a single server queue. We assume that there is no limit on the waiting queue capacity and the service process is FCFS. Consistently with [26], we also assume that there are *always* background jobs waiting for service. This is the case in storage systems where background media scans happen continuously to ensure that any existing disk latent errors are detected and recovered before the user accesses the data [8].

In the scenarios evaluated here, we aim to maintain background service transparent from the user. It is common practice, to consider an additional 5%-10% slowdown in performance as small enough to not be noticed by the system user. Consequently, we set the degradation target D to 7%, i.e., the middle point in the 5%-10% range. Slowdown of foreground jobs caused by background activity is computed as the ratio of foreground response time when background jobs are served to foreground response time when no background job is served.

The service times of background jobs are exponentially distributed. We expect this to be a realistic assumption, because disk-level service times have variability (i.e., measured via the CV) close to that of the exponential distribution [77]. The background service times are adjusted so that two different systems are simulated: (a) one system where both foreground and background jobs have the same mean service time (dubbed also as "short foreground" –"short background" system) and (b) one system where the average

background service time is 7 times longer than the average foreground service time (dubbed also as "short foreground" –"long background"). We remark that the results with other background service time ranges are qualitatively the same.

We first use synthetic workloads to quantitatively evaluate the policies under controlled systems with different levels of variability or burstiness. In Section 6.3.3, two real disk-level traces are used for evaluation. In these synthetic workloads, foreground interarrival times are drawn from an Erlang distribution, resulting in idle intervals of low variability. Drawing foreground interarrival times from an Lognormal distribution results in a system with high variability in its idle intervals. For both systems, the mean interarrival times are adjusted such that we evaluate system utilizations due to foreground jobs only, equal to 10%, 30%, and 70%, representing a system under low, medium, and high foreground load, respectively. All simulations are done with a 1 million sample space of foreground jobs and results are reported with 98% confidence intervals.

Idle Intervals with Low Variability

Results of the experiments with low variability in idle intervals are given in Figure 6.2. The first row of graphs corresponds to the system with "short foreground – short background" jobs, and the second row corresponds to the system with "short foreground – long background" jobs. Four performance metrics are presented: (a) the number of completed background jobs in millions (first column), (b) the overall system utilization (second column), (c) the background-caused slowdown of foreground jobs (third column), and (d) the foreground response times (fourth column). The last two metrics capture, respectively, the relative and the absolute background-caused degradation in foreground performance.

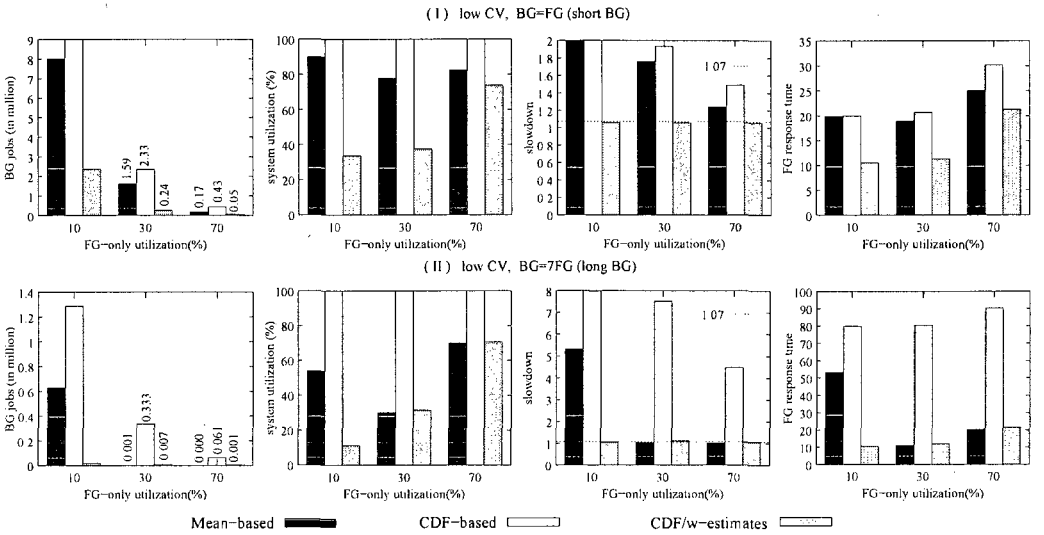


Figure 6.2: Overall system performance measured by number of completed background jobs in millions, overall system utilization, slowdown of the foreground jobs attributed to background activity (the horizontal line corresponds to 7% slowdown), and the absolute foreground response time. The idle intervals are independent and with low variability. Three foreground system utilizations are evaluated, i.e., 10%, 30%, and 70%. Foreground utilization is controlled by changing the foreground arrival rate and fixing its service time. The first row of graphs shows the case when the background jobs are "short", i.e., as long as foreground jobs, and the second row shows the case when the background jobs are "long", i.e., 7 times longer than foreground jobs.

Successful policies should increase the system utilization while the slowdown of foreground jobs is kept up to the pre-defined target of 7%. Results shown in Figure 6.2 can be summarized as follows:

- The mean-based and the CDF-based policies are very aggressive in the number of background jobs that they serve (first column), which results in high system utilization (second column). However, the penalty on foreground jobs is significant (third and fourth column). Note that because there is always an infinite supply of background jobs, if the system serves background jobs as much as possible till foreground jobs arrive, then the overall system utilization reaches 100%.

- The CDF/w-estimates policy consistently meets the performance target of fore-

ground jobs, across both experiments with short and long background jobs. This is due to the parameter T in Eq.(6.1), which determines how many background jobs to serve such that the effect on foreground performance is contained within the pre-determined limits.

- Under low foreground-only utilizations (i.e., 10%) there is more room to exploit idle times and serve large quantities of background jobs while limiting the effect on foreground slowdown. For example, with CDF/w-estimates, the overall system utilization for short background jobs increases to 35% from the initial 10% and 2.3 million background jobs are completed which is about 10 times and 46 times more than the completed background jobs when the foreground-only utilization is 30% and 70%, respectively.

- As foreground utilization in the system increases, the relative impact of background activity on foreground jobs reduces. The reason is that response times of foreground jobs are already dominated by waiting in the queue due to the high foreground load. As a result, waiting because of background work is not as noticeable. In low utilizations, foreground response time is dominated by the service time rather than the waiting time in the queue. Any background-caused delay is immediately observed because it may be the only wait that the foreground jobs experience. As a result, background work can be scheduled effectively even when the foreground system utilization is high.

- There is a significant difference in relative policy performance if background jobs are short or long. First, the mean-based policy performs poorly under long back-

ground jobs. This is because in the case of low variability in the idle times and high utilization, most idle times are short, i.e., shorter than service times of long background jobs. If the idle wait is equal to the average background service time, then the majority of idle intervals are not used for servicing any background activity. Note that the number of long background tasks completed under the mean-based policy is only 1000 for a foreground utilization of 30% and none for a foreground utilization of 70%. By self-adjusting T in Eq.(6.1), the CDF/w-estimates policy remains flexible, avoids background work starvation, and maintains the foreground performance targets. For example, even under the case of long background jobs and medium or high utilization, the respective numbers of background jobs completed are 7000 for a foreground utilization of 30% and 1000 for a foreground utilization of 70%.

- As expected, absolute foreground response time (shown in column four of Figure 6.2) increases with foreground utilization, even if the delays due to background jobs are limited. In this chapter, the focus is to achieve foreground performance targets measured by slowdown (a relative measure) rather than response time (an absolute measure). If the latter were the case, and foreground performance under 70% utilization would be the performance target, then all three policies meet that target if the foreground utilization is 10% or 30%.

For idle intervals with low variability, the three policies use idle intervals differently. The mean-based policy "consumes" the beginning of an idle interval via the idle wait and background jobs are served at the end of the interval. The CDF-based does not wait

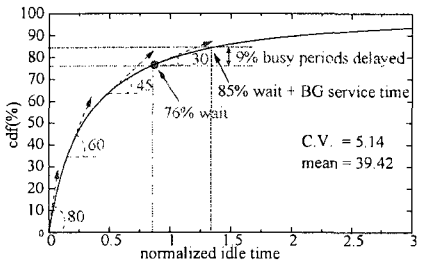


Figure 6.3: Relation between the slope of the CDF and the length of idle wait when the distribution has high CV

idle and serves background jobs as long as there is no waiting foreground job, utilizing the system 100%. The CDF/w-estimates policy schedules background activity at the beginning of the idle interval and not at the end (as the mean-based one) by estimating the number of background jobs to be served in any idle interval. This proves to be effective and strikes a good balance between the performances of foreground and background jobs.

Idle Intervals with High Variability

If the idle intervals have high variability, then policies that worked well under low variability conditions cease to be effective. The long tail of the distribution of idle times suggests that delaying background jobs is promising as now only long idle intervals are used for background jobs. No jobs are scheduled in idle intervals that are too short to fit a single background job.

Determining the length of idle wait is done dynamically by constructing on-line the cumulative histogram of idle times. The idle wait is defined by the CDF as the point in the histogram where the sharp increasing portion of the body ends and the slow increasing part of the tail starts (see Figure 6.3).

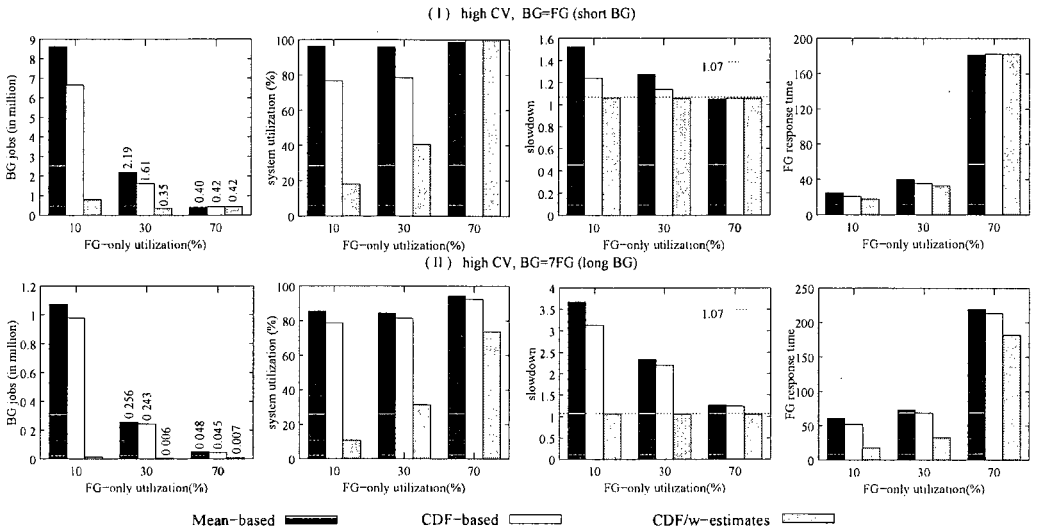


Figure 6.4: Overall system performance measured by number of completed background jobs in millions, overall system utilization, slowdown of the foreground jobs attributed to background activity (the horizontal line corresponds to 7% slowdown), and the absolute foreground response time. The idle intervals are independent and with high variability. Three foreground system utilizations are evaluated, i.e., 10%, 30%, and 70%. Foreground utilization is controlled by changing the foreground arrival rate and fixing its service time. The first row of graphs shows the case when the background jobs are "short", i.e., as long as foreground jobs, and the second row shows the case when the background jobs are "long", i.e., 7 times longer than foreground jobs.

Changes in the histogram shape are detected by inspecting the slope of both portions of the CDF curve. When the slope decreases to a predefined angle, e.g., 30 degrees in this case, then the desired point that separates the body from the tail of the histogram, is found. The number of jobs to be served in each interval is then computed using Eq.(6.1).

The predefined slope angle that determines the separation point between the body and the tail of the histogram, defines how aggressive the usage of idle intervals is, i.e., the higher the slope, the smaller the idle wait. In order to contain the slowdown of foreground jobs to a minimum, the angle should be set such that the CDF's slope is small and the usage of idle times is conservative.

Figure 6.4 shows the results for two experiments: short foreground–short background

(first row) and short foreground–long background (second row) under different utilization levels of systems with foreground jobs only. Similar to Figure 6.2, the following metrics are reported: the number of completed background jobs in millions, the overall system utilization, the relative slowdown in response time of the foreground jobs, and the absolute foreground response time. The performance target of foreground job slowdown remains 7% in these experiments also.

The figure illustrates that the high variability of the idle time distributions offers better opportunities to take advantage of idle intervals. Especially for the first experiment with short background jobs, system utilization significantly increases. Results from these experiments are summarized as follows:

- The mean-based policy, because of short idle waits, utilizes the system best but at the expense of higher delays for foreground jobs. For long background jobs, foreground jobs experience slowdown as much as 3.75 for low utilization and 2.5 for medium utilization (see third column in Figure 6.4-(II)).

- The CDF/w-estimates policy is always below or right at the 7% slowdown target (see dotted line in the third column) at the expense of scheduling less background jobs and lower utilization levels (see first and second columns of Figure 6.4).

- The system can be utilized as much as 100% with minimal performance degradation of foreground jobs when background jobs are short and the foreground utilization is high (e.g., 70% foreground utilization bars in Figure 6.4-(I)). The CDF/w-estimates policy becomes aggressive here by selecting a higher slope and by estimating the number of background jobs to be served using $T = 1$ in Eq.(6.1) (see rightmost set

of bars in the plots of Figure 6.4-(I)).

- Recall, that for idle intervals with low variability, for medium and high utilization and long background jobs (see first plot in Figure 6.2-(II)), the mean-based policy is not able to serve any background jobs at all. Under highly variable idle times, the result is different (see first plot in Figure 6.4-(II)). This is due to the existence of some very long idle times that enable completion of long background jobs even for long idle wait.

If idle intervals have high CV, then the mean-based, the CDF-based, and the CDF/w-estimates policies (different from the case of idle times with low CV) operate similarly, i.e., they all idle wait in the beginning of an interval and utilize its end. For medium to high utilization and short background jobs, the CDF-based and the CDF/w-estimates policies estimate an idle wait that is similar to the static one used by the mean-based policy. For long background job, the number of estimated background jobs to be served in an idle interval for the CDF/w-estimates policy is high, similar to the number of other two policies that are oblivious of such estimation.

The results presented here show that there are cases where the usage of idle times can be aggressive without affecting the performance of foreground jobs. If the background jobs are equal to or even shorter than foreground ones, then a foreground slowdown is not large, especially under high foreground utilization. Thus, under these cases, estimating the number of background jobs to serve is not necessary – it is enough to just wait for only a short period of time and serve background jobs till a foreground job arrives.

It may first appear counter-intuitive that high foreground utilization provides a good opportunity to serve short background jobs and reach nearly 100% system utilization with

minimal impact on foreground performance. Although the same observation exists for low variable idle intervals, the results are more pronounced if idle times have high variability. It is the tail of the idle time distribution that can be exploited to serve large quantities of short background jobs, with insignificant additional delay on the already slow response time of foreground jobs.

The following summarizes a comparison of the results in Figures 6.2 and 6.4.

- The number of completed background job decreases faster as foreground utilization increases for low variability than high variability idle times (first column in each figure).
- Overall system utilization is better under high variable than low variable idle times (second column in each figure).
- Foreground slowdowns are higher under low variable than high variable idle times (third column in each figure), because foreground response times (fourth column in each figure) are smaller (and more sensitive to additional delays) under low variable than high variable idle times.

Tail of the Response Time Distribution for Foreground Jobs

We have shown that the CDF/w-estimates policy consistently maintains foreground slowdown less than 7% while serving as many background jobs as possible. The figures of the two previous subsections present average slowdowns of foreground jobs. Here, the distribution of foreground response times is also presented and allows to study the tails of response times of foreground jobs.

In order to focus on the tail in the distribution, Figure 6.5 depicts the complementary cumulative distribution function (CCDF) of the response time distribution for foreground jobs only, and for foreground utilization levels equal to 30%. Here we only compare the tails of foreground response time distributions with and without background jobs under the CDF/w-estimates policy because their two average foreground response times are similar (with at most 7% difference). Comparisons with other policies that yield significantly higher response times are irrelevant.

Figure 6.5 shows that the impact on the tail of the response time distribution is caused by the length of the background job. For short background jobs, irrespectively of the variability of idle intervals, the foreground response time distribution with background jobs follows the distribution of the foreground response time without background jobs. The slight difference in average foreground response times exists throughout the distribution. Short background jobs, in general, delay the foreground jobs for a short period of time only. In this case, there are many background jobs that are scheduled, so there is a large portion of foreground jobs that are slightly delayed.

In the case of long background jobs, the behavior is different. Although, the CDF/w-estimates policy schedules only a few large background jobs to contain delays to a few foreground jobs only (less than 2% of all foreground jobs), the tail of the foreground response time distribution is much longer than when no background jobs are served. The long tail is a result of the significant delays caused to foreground jobs by long background ones.

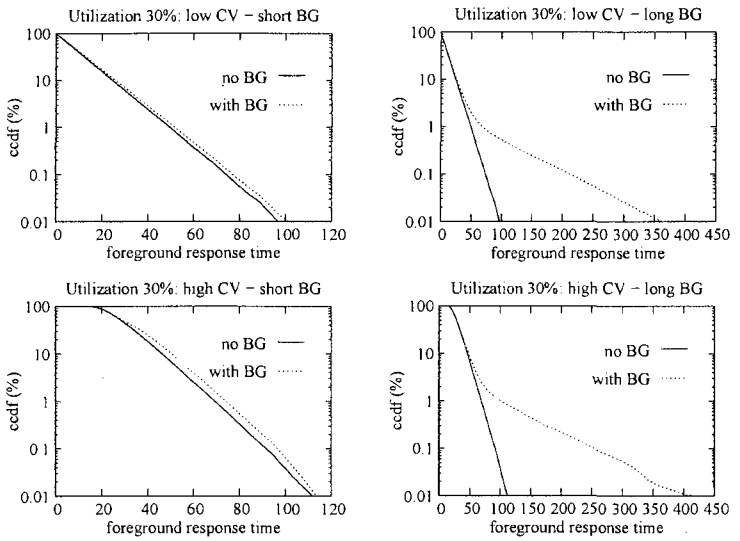


Figure 6.5: Tail of the foreground response time in the presence of or not of background activity when the idle intervals are independent. The dashed lines in the graph correspond to results for the CDF-w/estimates policy. Utilization of the system under only foreground activity is 30%.

6.3.2 Background Activity in Bursty Idle Intervals

The policies presented in the previous subsection are based on the cumulative histogram of the empirical distribution of idle times. Here, the focus is on idle intervals that, in addition to being highly variable, are also bursty. Section 6.2.2 presents analysis of bursty processes and shows that if burstiness exists then it is possible to predict the near future based on the recent past. If a sequence of observations is positively bursty, then this implies that long observations (i.e., several times larger than the mean) are clustered together in the sequence and that short ones (i.e., several time smaller than the mean) are also clustered together. This property can be used to manage system idleness more efficiently by exploiting long intervals aggressively.

To detect burstiness in idle times, a similar structure as the one depicted in Figure 2.1 is constructed on-line. First, observations of idle times are classified as small or large.

The value to partition the range of idle times in "small" and "large" is $(CV + 1) \cdot mean$. Then, every pairs of idle times is classified in the appropriate category, i.e., (small,small), (large,small), (small,large), or (large,large), and the corresponding probability is calculated online. Pairs do not include consecutive observations only, but also those that are separated by up to 9 observations (lags) apart, which is a measure of the burst length. Once these conditional probabilities are constructed, they are used to predict more accurately whether the next idle interval is short or long. The CDF/w-estimates policy is augmented into the Bursty+CDF/w-estimates policy as follows:

- if the current interval belongs to the "large" category, then the next interval is predicted to be "large" with probability ρ. Note that the probability for occurrence of (large,large) pair is not equal to 1, which implies the existence of mis-predicted "large" intervals. The probability ρ used in this policy is just to control the number of miss-predicted "large" intervals.

- if the next idle interval is predicted to be "large", instead of using Eq.(6.1) to estimate the number of background jobs to be served in that interval, the following equation is used:

$$T \cdot \frac{(CV + 1) \cdot mean - \text{idle wait}}{\text{Average background service time}}, \tag{6.2}$$

  which implies that the length of the incoming idle interval is at least the "large" value $(CV + 1) \cdot mean$.

Exploiting the long intervals in a bursty sequence allows to increase overall system utilization, because during those intervals the policy has the opportunity to be aggressive

without affecting foreground jobs. Benefits are different for short and long background jobs:

for short background jobs: stringent foreground slowdowns are achieved without radically reducing the number of completed background jobs,

for long background jobs: a given amount of background work can now be completed with less degradation on foreground performance, resulting in shorter tails in their response times.

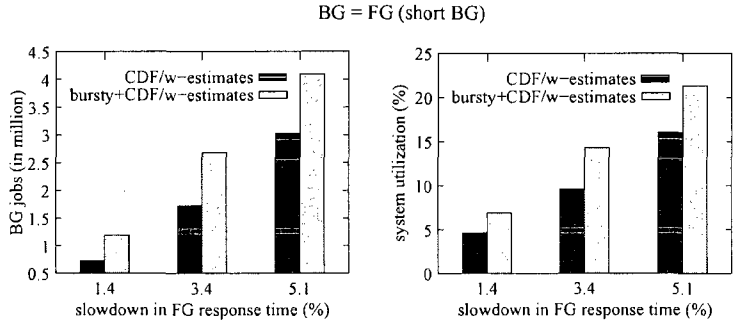


Figure 6.6: Number of completed background jobs and overall system utilization when under bursty idle intervals. Three different foreground slowdowns are considered, i.e., 1.4%, 3.4% and 5.1%. The more stringent requirements on foreground slowdown the higher the relative improvement between the CDF/w-estimates and Bursty+CDF/w-estimates.

Experiments in this subsection assume that the interarrival and service processes of foreground jobs are drawn from an MMPP distribution. The service times of background jobs are exponentially distributed. The results in idle times with probability of (small,small), (small,large), (large,small), and (large,large) pairs are shown in Figure 2.1. Evaluation of improvements due to the use of conditional probabilities in the CDF/w-estimates policy, is done via two sets of experiments: one with short and the other with long background jobs.

Figure 6.6 presents the number of completed background jobs and the overall system utilization as a function of foreground slowdown, when short background jobs are served. There are more background jobs completed with Bursty+CDF/w-estimates than with CDF/w-estimates. The relative performance gap between the two policies increases as foreground slowdown decreases. If the requirements on foreground slowdown are relaxed, then the difference between the two policies diminishes. In general, overall system utilization improves with Bursty+CDF/w-estimates.

In a system that serves long background jobs, background jobs are chosen to be 300 times longer than the foreground service times, on the average. An example of such scenario is when disks spin-down to conserve power. Spinning them back-up and ready for work is orders of magnitude larger than serving a single request. This extreme case is difficult to address – scheduling a background job in the wrong interval may have a tremendous impact on the tails of foreground jobs.

Figure 6.7 presents the foreground job slowdowns for the CDF/w-estimates and the Bursty+CDF/w-estimates policies when background jobs are large. The dotted horizontal line represents the performance target of 7% average slowdown for foreground jobs. The Bursty+CDF/w-estimates policy attempts to detect pairs of long idle intervals and utilize them by serving the background jobs because then the probability to affect foreground is low. We select three levels of completed background work, which are (1) High: with 7455 completed background jobs, the background work is more than two times the foreground work; (2) Medium: with 1816 completed background jobs where the background work is more than half of the foreground work; and (3) Low: with 1252 completed background jobs, here the background work is only a third of foreground work. These three levels are

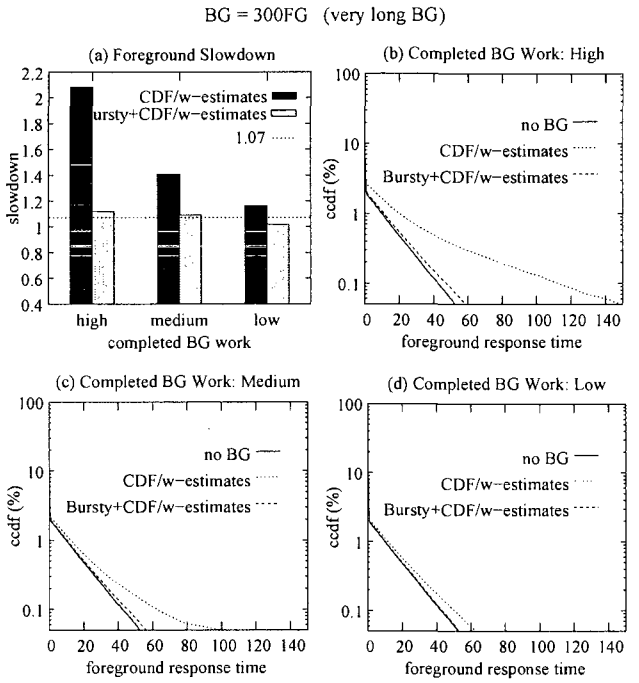


Figure 6.7: System performance under background jobs when the idle intervals are bursty. Three different settings are considered that yield three different numbers of completed background jobs and foreground slowdowns. Plot (a) gives the foreground slowdown for each level of background completions and plots (b)–(d) give the respective CCDFs in foreground response time distribution.

a result of different values of the parameter T used in Eq.(6.1).

Figure 6.7 shows the foreground job slowdown for each of three level of completed background work. For high, medium, and small amounts of completed background work, the foreground slowdown under CDF/w-estimates is, respectively 110%, 40% and 17%. The Bursty+CDF/w-estimates policy reaches foreground slowdowns of 12%, 9% and 2%, respectively.

Experiments in Section 6.3.1 showed that the long background jobs change the tail of the foreground response time distribution. The tail of the foreground response time distribution under the large background jobs is also given in Figure 6.7 with the plots of the CCDFs of the foreground response times. The figure also plots the CCDFs of

foreground jobs with no background activity (labeled "no BG") as a baseline comparison. When there is no background activity, the tail of the foreground response time is not long, e.g., only 0.05% of foreground jobs have response time larger than 52. Only 1% of jobs have response time larger than 10 when service time is 1 on the average.

The CDF/w-estimates policy, being oblivious to burstiness, affects significantly both average foreground response time as well as its distribution tail. For example, when the completed background work is high, 0.05% of foreground jobs have response time larger than 150, making the tail almost 3 times longer from the cases when there is no background activity. As the amount of completed background work decreases, the tail of the foreground response time distribution shortens and approaches the baseline tail.

Figure 6.7 also plots the tail of the foreground response time distribution under the Bursty+CDF/w-estimates policy. This policy utilizes mostly large idle intervals while the CDF/w-estimates policy is oblivious to them. Therefore, the number of delayed foreground jobs by long background jobs under Bursty+CDF/w-estimates policy reduces. As a result, not only the slowdown of foreground work is substantially smaller than that under CDF/w-estimates, but also the tail of the foreground response time distribution is close to the baseline tail.

The results presented in this subsection show that if idle intervals are bursty (i.e., the series of idle intervals contains information on the order of observations) then one can predict the occurrences of long idle intervals, which in turn can be used to efficiently schedule large quantities of background work without affecting foreground jobs.

Trace	Mean Arrival	Mean Service	Util (%)	Mean Idle	CV Idle	Burty
T1	62.64	5.50	8.3	190.08	6.41	No
T2	252.29	5.50	2.2	731.34	3.90	Yes

Table 6.1: Overall characteristics of traces used in this evaluation. The measurement unit is ms.

6.3.3 Case Study: Disk Drives

We validate the results of the high variance case in Section 6.3.1 and of the burstiness case in Section 6.3.2 using traces that are measured in actual storage systems. Traces used in this subsection are measured at different disks of a 40-disk storage system of an in-the-field e-mail server. The traces record, in micro seconds, both arrival and departure times for each foreground request in the system and allow for *exact* computation of idle and busy times at the disk level of the system where the traces were measured. The main statistical characteristics of the traces relevant for background work scheduling are given in Table 6.1.

The main observation from statistics in Table 6.1 is the substantial difference between the mean of foreground interarrival times and the mean length of idle intervals. Results in Table 6.1 further confirm that neither the foreground arrival or service process can provide enough information for background scheduling, instead one should focus on idle times. Therefore, instead of using foreground arrivals to guide idle time management (as in [26]), we suggest to monitor and estimate idle time characteristics and use them to guide background scheduling in idle times.

Both traces, that we have selected, have highly variable idle times and can be used to validate the results of Section 6.3.1, which deals with scheduling background work in

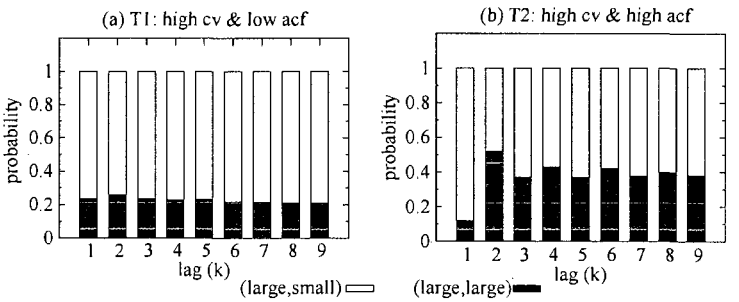


Figure 6.8: Probabilities of (large,small) and (large,large) pairs for traces T1 and T2.

highly variable idle times. We also identify if any of the two traces has idle interval lengths that are bursty. Figure 6.8 gives the probability of pairs of "large" idle intervals of up to 9 observations (lags) apart. The maximum probability of pairs of long idle intervals is only 0.2 for trace T1 and about 0.5 for trace T2. Consequently, T1 is viewed as a trace with very weak dependency structure (i.e., the sequence of observations are nearly independent). Trace T2 has much stronger dependence among the observations. Although the probability of (long, long) pairs is not as high as for the synthetic trace in Section 6.3.2. Trace T2 is clearly a trace with bursty idle intervals. Trace T1 is used to validate the results of Section 6.3.1 and trace T2 is used to validate results of Section 6.3.2.

We do not deal with identifying why idle intervals are bursty or not, although the fact that burstiness exists in many processes associated with disk drives [77] implies also idle time burstiness. As shown by the results for trace T2 in Figure 6.8, burstiness in idle times exist and taking it into consideration for managing idle time utilization as in Section 6.3.2 yields realistic benefits.

In the experiments with T1, all three policies are evaluated, i.e., mean-based, CDF-based, and CDF/w-estimates. The mean background service time is chosen to be 5 ms, 50 ms and 300 ms, representing the cases of short, medium, and long background jobs,

respectively, when compared to the mean foreground service time of 5.5 ms. These service times of background jobs are all exponentially distributed. Similar demands of disk background activities are write verification (short), moving large chunks of data (medium), and flushing the write cache (long). More details on such storage system background tasks can be found in [33].

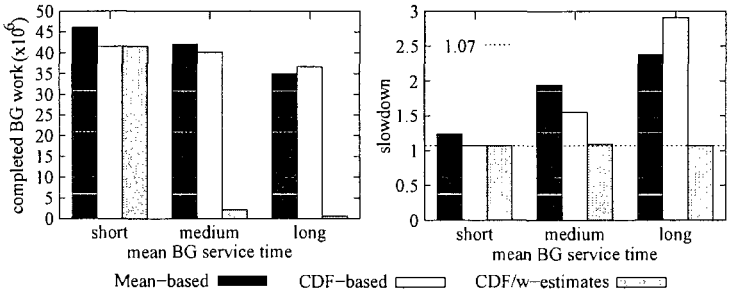


Figure 6.9: Number of completed BG jobs and foreground slowdown for trace T1. Three background service demands are chosen that correspond to short, medium, and long background jobs.

Figure 6.9 plots experimental results for trace T1. The CDF/w-estimates policy, consistently with results in Section 6.3.1, outperforms the other two policies when it meets foreground performance requirements. In the experiments with T2, background service time is set to be 300 ms. Experiments are conducted only with long background jobs and not short ones, because with this set of experiments, the emphasis is on exploiting burstiness of idle times to schedule the most challenging long background jobs. Consistently with the results in Section 6.3.2, benefits are higher for long rather than short background jobs. Here only the CDF/w-estimates and the Bursty-CDF/w-estimates policies are evaluated (as in Section 6.3.2). Figure 6.10 presents average slowdown of foreground jobs for three levels of completed background work, i.e., high (83,998 jobs), medium (47,702 jobs), and low (23,322 jobs). By exploiting burstiness, both average foreground slowdown and the tail of the foreground response time distribution are improved. These trace-based

experiments confirm our analysis in previous subsections using synthetic workloads.

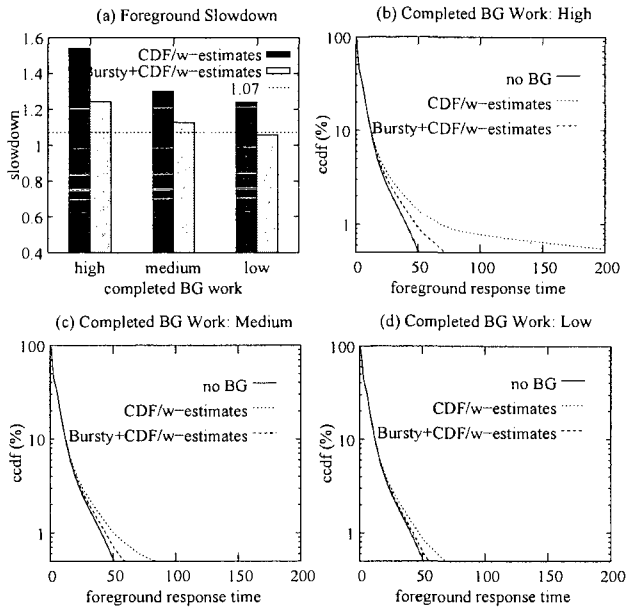


Figure 6.10: Average slowdown of foreground jobs and the tail of response time distribution for three levels of completed background work, for trace T2.

6.4 Application: Enhancing Data Availability

In this section, we evaluate the impact that the idle time management policies have on performance of background activities that are scheduled to improve data reliability when they are constraint by the degradation on foreground performance. Such idle time management is in the same spirit as the techniques proposed in the previous section, where idle times stochastic characteristics (i.e., variability and burstiness) guide background work scheduling.

The trend of disk failure in storage systems with a large disk drive population has been

observed and analyzed in recent works [72]. Upon a single disk failure, storage systems are designed to restore the lost data using redundant information. During data restoration, the storage system operates with reduced redundancy and any additional failure causes data loss. Although a second entire disk failure during this period is less likely, data loss may occur even if a few disk sectors are not accessible. Disk sector errors, often related to localized media failures, are more frequent than entire disk failures and are known as "latent sector errors" because they are detected only when the affected area on the disk is accessed and not when they truly occur [86, 9, 27].

In general, there are two prevalent strategies for protecting data from latent sector errors: *disk scrubbing* is an error detection technique that aims at detecting latent sector errors via background media scan and *before* the affected data is accessed by the user or before any other disk failure [84], and *Intra-disk data redundancy* is used as an error recovery technique by adding parity for sets (segments) of sectors within the same disk [23, 45] which is effective in multiple- and single-disk storage systems. However, scrubbing could cause delays to the foreground work because disk operations such as seeks are not preemptive. Multiple redundancy levels and intra-disk parity do impose additional work in the storage system when data is modified (i.e., during WRITE operations) because the parity need to be updated. Consecutively, both scrubbing and intra-disk parity updates can operate as system background processes, because if the execution of this additional work competes with regular user traffic, it may cause additional undesired delays. Therefore, in this section, we use the "body-based", the "tail-based", and the "tail+bursty" policies (see Section 6.3 for the detailed description) to schedule scrubbing and intra-disk parity updates as background activities.

6.4.1 Background: MTTDL estimation

Latent sector errors rather than total disk failures cause loss of data but not necessary result in storage system failure. Consequently, an important reliability metric for storage systems is the Mean-Time-To-Data-Loss (MTTDL). Approximate models for the MTTDL as a function of various system parameters are given in [9]. Here, we calculate MTTDL of systems with scrubbing and intra-disk data redundancy using the same models as in [9]. For details on the models, we direct the interested reader in [9]. Here, we only provide a quick overview as follows. The model defines MTTDL in terms of the following parameters:

MV, ML: mean interarrival time of visible and latent disk errors, respectively,

MRV, MRL: mean recovery time from visible and latent errors, respectively,

MDL: mean detection time of latent sector errors,

α: errors temporal locality parameter,

β_{XY}, errors spatial locality parameters, where consecutive errors X and Y are either visible (i.e., type V) or latent (i.e., type L).

If no scrubbing is initiated, then MTTDL is given by the following equation:

$$\frac{1}{MTTDL} \approx \frac{\beta_{VV}}{\alpha}\frac{MRV}{MV^2} + \frac{\beta_{LV}}{\alpha}\frac{MRV}{MV \cdot ML} + \frac{1}{ML} \tag{6.3}$$

If scrubbing is performed then the above equation accounts for the average time it takes to detect the error via scrubbing (i.e., MDL) and recover from it (i.e., MRL) as follows:

$$\frac{1}{MTTDL} \approx \frac{\beta_{VV}}{\alpha k^2}\frac{MRV}{ML^2} + \frac{\beta_{LV}}{\alpha k}\frac{MRV}{ML^2} + \frac{\beta_{VL} + k\beta_{LL}}{\alpha k} \cdot \frac{MDL + MRL}{ML^2} \tag{6.4}$$

where k is defined in [9] as $k = ML/MV$. The parameter values for Eqs.(6.3) and (6.4) used in [9] and in the following subsections are given in Table 6.2.

MV	ML	MRV
120,000 hrs	84,972 hrs	1.4 hrs
k	$\alpha, \beta_{VV}, \beta_{LV}, \beta_{VL}$	β_{LL}
1.41	1	0

Table 6.2: Parameters used for MTTDL estimation.

6.4.2 Trace Characteristics and Simulations

All policies presented here are evaluated via trace driven simulations. All simulations are driven by disk drive traces, see [77] for a detailed description of the statistical characteristics of the selected ones. We selected three disk traces that were measured in a personal video recording device (PVR), a software development server, and an e-mail server, which we refer throughout this section by T3, T4, T5, respectively. Table 6.3 gives a summary of the overall characteristics such as request mean interarrival time, request mean service time, utilization, as well as the mean and the coefficient of variation (CV) of idle intervals in the trace.

Trace	Mean Arrival	Mean Service	Util (%)	Mean Idle	CV Idle	Burty
T3	62.85	10.68	17.4	91.98	0.98	No
T4	96.72	4.20	4.2	236.08	6.41	No
T5	252.29	5.59	2.2	760.84	3.79	Yes

Table 6.3: Overall characteristics of traces used in our evaluation. The measurement unit is ms.

The focus of this section is the evaluation of two background activities. Scrubbing is an *infinite* background process because upon completion of one entire disk scan, commonly a new one starts. The parity updates depend on the WRITE user traffic and are considered a *finite* background process. Table 6.4 gives the specific parameters of scrubbing and intra-disk parity update used in our simulations.

	Scrubbing		Intra-disk Parity Update			
Tra -ce	Short BG Number	Short BG Mean Service	Short BG Number	Read BG Mean Service	Write BG Mean Service	Write Portion (%)
T3	100,000	6.0	2	10.0	5.0	40
T4	100,000	6.0	2	10.0	5.0	1; 10; 50; 90
T5	100,000	6.0	2	10.0	5.0	50

Table 6.4: Background activities characteristics. The unit of measurement is ms.

Scrubbing is abstracted as a long background job that is preemptive at the level of a single disk request. Hence, it is assumed that an entire scan of a 40GB disk, i.e., one completed scrubbing, requires 100,000 disk IOs each scanning approximately 1000 sectors. Assuming disk capacities of 40GB might be conservative given that modern disk drives reach capacities of up to 500GB. Nonetheless, the analysis presented in this section still

holds for larger disks as well. One single disk scan request as part of the scrubbing job is assumed to take in average as much time as a READ disk request. In our simulation, this is drawn from an exponential distribution with mean 6 ms (similarly to the mean service time of traces in Table 6.3). The time to serve 100,000 disk IOs as part of a single scrubbing corresponds the average scrubbing time.

Parity updates are abstracted as short background jobs. To update the parity of a segment of sectors, the following steps are taken. First the entire set of sectors should be read, then the parity must be calculated, and finally the new parity is written on the disk. Therefore, each parity update consists of one READ (assumed to take in average 10 ms) and one WRITE (assumed to take in average 5 ms), both exponentially distributed. The preemption level of parity updates is at the disk request level. If a parity update is preempted after the READ, then the system maintains no memory of the work done and the update has to restart again during another idle period. Parity updates are served in a first-come-first-served (FCFS) fashion.

Scrubbing and intra-disk parity update processes are scheduled using the three policies outlined in the beginning of this section. All three policies degrade the performance of user traffic up to 7% (this is a pre-set system parameter) by restricting the amount of background jobs served. Their efficiency regarding the performance of timely completion of background tasks (i.e., scrubbing or parity updates) depends on the variability of idle times in traces T3, T4, and T5. The following sections further elaborate on policy sensitivity with respect to idle time variability.

6.4.3 Infinite Background Activities: Scrubbing

Background media scans can be abstracted as an *infinite* background process that detects any possible media errors on disk drives and thus prevents any data loss caused by the latent sector errors. As a preventive feature, scrubbing is completed in background and can be conducted by the storage system or the disk drive itself. Based on the system specifications described in Section 6.4.2, we evaluate the effectiveness of scrubbing aiming at degrading performance of user traffic by at most 7%.

Table 6.5 presents the number of completed media scans, their average scrubbing time, and the overall system utilization for the three traces of Table 6.3, when the body and the tail of idle times are utilized. Consistently with results reported in Section 6.3, for lowly variable idle times (i.e., trace T3) utilizing the body rather than the tail of idle times results in faster scrubbing and better overall system utilization. In particular, scrubbing under the body-based policy is twice faster than under the tail-based policy (see first row of Table 6.5). For T4 that has highly variable idle times, the tail-based rather than the body-based policy yields faster scrubbing and better system utilization (i.e., at least an order of magnitude difference, see second row of Table 6.5). Finally, if idle times are in addition bursty (i.e., trace T5), then utilizing the tail of idle times and predicting long idle periods performs better than utilizing only the tail of idle times. Utilizing burstiness to benefit scrubbing scheduling results in a five-fold improvement in mean scrubbing time. The body-based policy is not evaluated for T5 because the results of T4 establish that tail rather than body of idle times should be utilized if idle times have high CV.

In addition to the average performance presented in Table 6.5, we also evaluate the distribution of scrubbing time. The distribution is built with a sample space of completed

Tra -ce	Policy	Completed Scrubbing	Mean Scrubbing Time (s)	System Util (%)
T3	body	6	3,617.8	33.1
	tail	4	6,484.7	26.8
T4	body	4	11,519.6	9.7
	tail	63	726.4	83.1
T5	tail	20	4,476.3	14.3
	tail+bursty	94	972.9	62.6

Table 6.5: Scrubbing performance for traces T3, T4, and T5 under body-based, tail-based, and tail+bursty-based idle time managing policies.

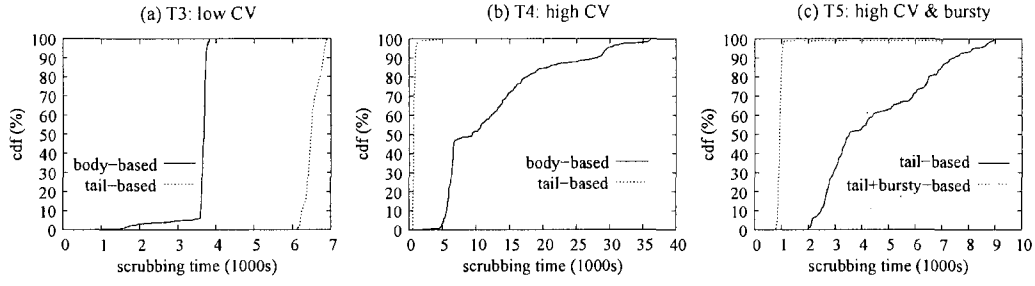


Figure 6.11: CDF of scrubbing time distribution for traces (a) T3, (b) T4, and (c) T5.

scrubbing as large as 500 by replaying the traces several times. Figure 6.11 shows the cumulative distribution function (CDF) of scrubbing time for traces T3, T4, and T5. For all three traces, the best performing scheduling policy for scrubbing identified in Table 6.5 achieves the shortest scrubbing distribution tail. However, the differences between the scrubbing scheduling policies are more drastic when it comes to the distributions than the average values. For example, for trace T3 (see Figure 6.11(a)), almost 100% of scrubbings have scrubbing times less than 3831.9 seconds under the body-based policy while a twice larger scrubbing time is achieved only for 1.4% of scrubbings under the tail-based policy.

Similarly for trace T4 (see Figure 6.11(b)), the tail of scrubbing time under the tail-based policy is about 7.5 times shorter than under the body-based policy. Exploiting burstiness with the tail+bursty-based policy, as shown in Figure 6.11(c), further reduces the tail of scrubbing time distribution.

The goal of scrubbing as a preventive background feature is to improve the MTTDL. The average time of scrubbing, given in Table 6.5, allows for MTTDL calculation when scrubbing is not running and when it is running, using Eqs.(6.3) and (6.4), respectively. The mean detection time of sector errors (MDL) in Eq.(6.4) is set to be equal to $0.5 \times$ average scrubbing time. Moreover, compared to detection times, the recovery times of latent sector errors are insignificant (i.e., MRL $\ll$ MDL). We thus assume MRL ≈ 0 in Eq.(6.4). Table 6.6 gives the improvements in MTTDL when scrubbing is running over the case when it is not running. The overall improvement of MTTDL because of scrubbing is 4 orders of magnitude. The differences in the MTTDL improvement between the scheduling policies that are used to manage the idle times are between 20% and 40%.

T3		T4		T5	
body	tail	body	tail	tail	tail+bursty
4 $\times 10^4$	3 $\times 10^4$	3 $\times 10^4$	5 $\times 10^4$	4 $\times 10^4$	5 $\times 10^4$

Table 6.6: MTTDL improvement via scrubbing.

6.4.4 Finite Background Activities: Intra-disk Parity Update

Intra-disk data redundancy requires maintaining updated parity that becomes dirty if the corresponding data is modified [23, 45]. This extra amount of work required to

maintain updated parity consists of an extra READ and an extra WRITE for each user-issued WRITE. Completing this work instantaneously upon completion of each user-issued WRITE is called *instantaneous parity* (IP) update. Naturally, IP causes degradation in user performance but provides the highest level of data reliability.

Here, we show that it is possible to complete the parity updates as a background job scheduled in idle intervals in a timely fashion while keeping user performance slowdown less than a predefined target. In the experiments presented here acceptable user slowdown is set to 7% only. Delays in parity updates reduce the effect of intra-disk parity on data reliability. We quantify how delayed intra-disk parity affects data reliability for the three idle scheduling techniques.

We present results for traces T3 and T4. Traces T4 and T5 yield similar results because both have high variability in idle times and because for the finite work generated by parity updates exploiting burstiness does not yield any further improvement. The following three metrics are monitored: (a) the ratio of completed parity updates to the total trace WRITE traffic, (b) the average time of parity updates which is the time interval between the completion of a user-issued WRITE operation and the update of the parity, and (c) the overall (foreground + background) system utilization.

Parity Updates under Trace T3

Table 6.7 gives the parity update results under the body-based and tail-based idle time scheduling policies. Trace T3 has nearly 40% user WRITEs. Different from the behavior under infinite background activities, the tail-based rather than the body-based idle time scheduling performs best overall. Most importantly, the tail-based policy updates parities

Policy	Completed Ratio (%)	Mean Update Time (ms)	System Util (%)
body	38.6	180,629.0	24.7
tail	41.6	3,321.0	22.9

Table 6.7: Parity update performance for trace T3 (low variability).

almost by two orders of magnitude faster than the body-based policy. Quick parity update times are particularly desirable because the average parity update time is the metric that affects data reliability. Note that system utilization is higher under the body-based than under the tail-based policy. Under the body-based policy, there are more cases where a user request preempts a parity update, which unfortunately results in wasted work. Under the tail-based policy, only long idle intervals are used to update the finite parities which results in only few of them being preempted by user traffic.

Figure 6.12 shows the distribution of the parity update times. While about 68% of parity updates under the body-based policy are faster than under the tail-based policy, the tail of parity update times is longer than under the tail-based policy, which dominates the average parity update time and causes a two orders of magnitude advantage for the average tail-based performance.

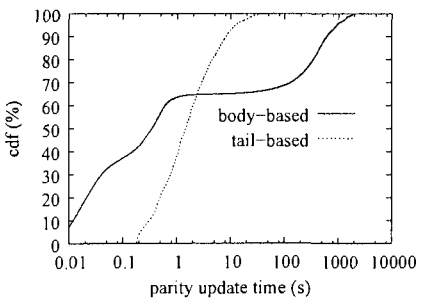


Figure 6.12: CDF of parity updates time for trace T3 (low variability).

Because parity updates are postponed in idle periods, some user WRITEs may find dirty parity in their corresponding parity segment. Updating parity when multiple WRITEs have occurred in the parity segment is more prone to errors than when only one WRITE has been completed. Table 6.8 gives the probabilities that by the time a parity is updated, the corresponding parity segment has been overwritten up to five times by the user. Although the metric depends on parity update times, it also depends on the spatial locality of the user WRITE workload. Trace T3 does have this characteristic. Results in the table show that the majority of parity updates (approximately 75%) for both policies occur when the segment has been written at most twice.

Trace	Policy	User Issued WRITEs				
		1	2	3	4	5
T3	body	0.65	0.16	0.04	0.10	0.01
	tail	0.44	0.29	0.09	0.12	0.02

Table 6.8: Probabilities of user WRITES in trace T3 (low variability) that find dirty parity.

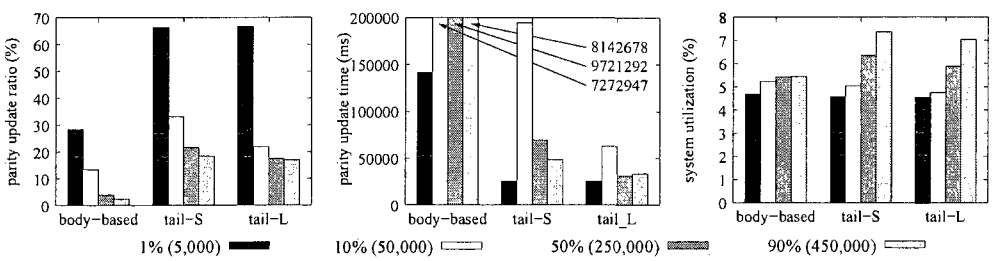


Figure 6.13: Performance of parity updates for trace T4 (high variability) and four different user WRITE traffic, i.e., 1%, 10%, 50% and 90% (numbers in parenthesis indicate the absolute number of user WRITEs).

Parity Updates under Trace T4

User issued WRITE traffic in T4 represents only 1% of the total requests. To experiment with traces with more WRITE traffic, we generate three additional traces that have 10%, 50%, and 90% WRITEs, respectively. These traces are generated based on T4, by probabilistically selecting an entry in the trace to be a READ or a WRITE.

Figure 6.13 presents parity update performance for trace T4 (and its variants) using the body-based and tail-based policies to schedule work in idle times. Figure 6.13 shows two different performances for the tail-based policy (marked in the plots as "tail-S" and "tail-L"). Although both tail-based policies utilize the tail of the idle times, under "tail-S" the idle wait is (approximately 40%) shorter than under "tail-L".

Because T4 has highly variable idle times, the tail-based policy outperforms the body-based one. For example, the body-based policy performs at least two to three times worse than the tail-based policy with respect to the total number of completed parity updates and the average parity update time. The differences in performance between the body-based and the tail-based policies increase as the amount of parity updates increases. Among the tail-based policies, "tail-L" achieves better update time while "tail-S" achieves better number of completed updates. Note, two tail-based policies perform exactly same when the amount of parity updates is small (cases with 1% WRITES). Timely updates are critical for MTTDL, we elaborate more on this later in this subsection.

The overall system utilization in Figure 6.13 is not as high as the 80% utilization level under scrubbing in Table 6.5 because parity updates represent a finite amount of work. Similarly to the results of trace T3, if the amount of parity updates is small (cases with 1% and 10% WRITEs), then the body-based policy utilizes the system more than the

tail-based policy because of the preempted updates. As the amount of parity updates increases, the effect of this phenomenon diminishes.

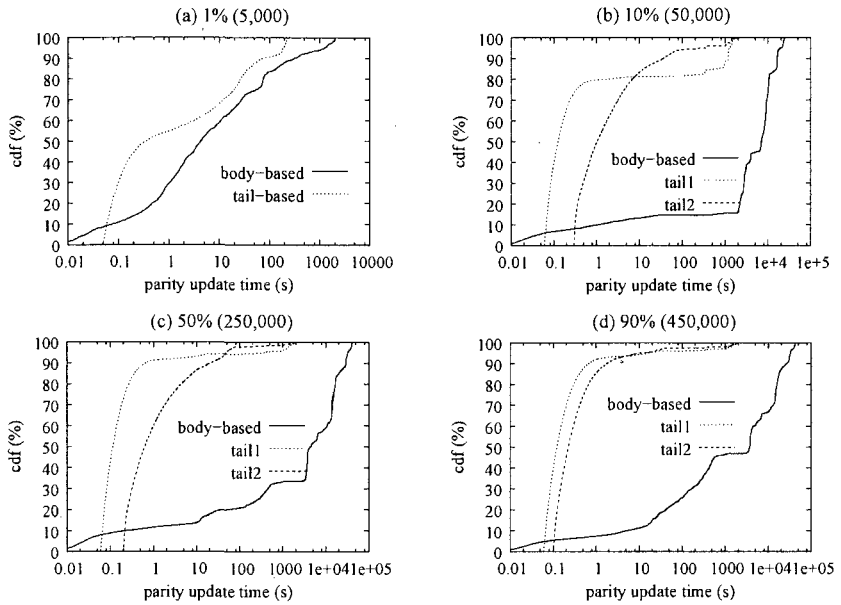


Figure 6.14: CDF of parity update time for trace T4 (high variability) and four different user WRITE traffic, i.e., 1%, 10%, 50% and 90% (numbers in parenthesis indicate the absolute number of user WRITEs).

Figure 6.14 plots the CDFs of parity update times for all four variants of trace T4. Consistently with results in Figure 6.13, under the body-based policy, the distribution has longer tail than under the tail-based policy. The "tail-L" variant has the shortest tail indicating that the best average performance comes from the policy that results in a shorter tail of update times. The "tail-L" variant has also the longest idle waiting, which indicates that it uses the smallest number of idle intervals among all policies evaluated and has to wait for the very long intervals to arrive. Nevertheless, it results in the shortest average and distribution tail for update times. As parity updates increase in number, the differences in the distribution of update times between "tail-S" and "tail-L" decrease.

Table 6.9 presents the probabilities that by the time a parity update occurs, up to five user WRITEs have modified the parity segment for all four variants of trace T4. As the portion of user WRITEs increases in the trace, the probability of one user WRITE updates decreases. The body-based policy results are consistently worse than the results under the tail-based policies. In the best case (i.e., 1% WRITEs) 100% of parity updates happen when the parity segment has been modified at most twice for both policies.

Trace	Policy	User Issued WRITEs				
		1	2	3	4	5
T4	body	0.98	0.02	N/A	N/A	N/A
(1%)	tail	0.99	0.01	N/A	N/A	N/A
T4	body	0.75	0.17	0.05	0.01	0.01
(10%)	tail	0.85	0.12	0.02	0.006	0.001
T4	body	0.53	0.22	0.10	0.05	0.03
(50%)	tail	0.65	0.22	0.07	0.03	0.01
T4	body	0.46	0.21	0.11	0.06	0.04
(90%)	tail	0.59	0.24	0.08	0.03	0.02

Table 6.9: Probabilities of user WRITES in trace T4 (high variability) that find dirty parity.

MTTDL in Data Redundant Drives

The estimation of MTTDL for disks with intra disk redundancy is based on Eq.(6.3). Assuming that latent sector errors are spatially and temporally correlated, the improvement in the mean interarrival time of latent sector errors is 0.48×10^2 [23], or equivalently,

$ML^{(2)} = 0.48 \times 10^2 \cdot ML^{(1)}$, where $ML^{(1)}$ represents the mean interarrival time of latent errors if there is no intra-disk data redundancy, and $ML^{(2)}$ represents the mean interarrival time of latent errors if there is intra-disk data redundancy.

If instantaneous parity (IP) is supported (i.e., parity updates occur without delay), then MTTDL is calculated using Eq.(6.3) and $ML^{(2)}$ is used in place of ML, i.e.,

$$MTTDL = MTTDL_{ML^{(2)}}.$$

If parity updates are delayed, then Eq.(6.3) is modified as follows:

$$\begin{aligned} MTTDL &\approx p \cdot MTTDL_{ML^{(1)}} \\ &\quad +(1-p) \cdot MTTDL_{ML^{(2)}}, \end{aligned} \tag{6.5}$$

where p represents the probability that the parity is dirty and $MTTDL_{ML^{(1)}}$ is computed using Eq.(6.3) and value $ML^{(1)}$ is used for ML. We assume that if the parity is dirty then latent errors arrive in intervals of $ML^{(1)}$ and that if parity is updated, then errors arrive in intervals of $ML^{(2)}$. We approximate p as the portion of the disk with dirty parity as follows:

$$\begin{aligned} p &\approx \frac{QL_{Update} \cdot Length_{Parity\ segment}}{Capacity_{Disk}} \\ &= \frac{RT_{Update} \cdot \lambda_{Update} \cdot Length_{Parity\ segment}}{Capacity_{Disk}}, \end{aligned} \tag{6.6}$$

where RT_{Update} is the average parity update time, λ_{Update} is the arrival rate of parity updates and $Length_{Parity\ segment}$ is the number of sectors in each parity segment. The

performance of the policy to schedule background processes during idle intervals determines RT_{Update} and consequently affects the MTTDL.

Assuming that the disk capacity is 40GB, the relative MTTDL improvement is estimated for parity updates for trace T3 and the four variants of trace T4. Results are given in Table 6.10. The improvement attributed to intra-disk parity are only one order of magnitude – recall that those attributed to scrubbing are as high as four orders of magnitude. The important result of Table 6.10 is that there is almost no difference between the MTTDL improvement achieved via instantaneous parity (IP) updates and the delayed parity updates evaluated in this subsection, which strongly argues in favor of delayed intra-disk parity.

Policy	T3	T4			
		1%	10%	50%	90%
body	48.1	48.4	46.6	38.6	35.1
tail-S	48.4	48.4	48.3	48.2	48.2
tail-L	N/A	48.4	48.4	48.3	48.3
IP	48.4	48.4	48.4	48.4	48.4

Table 6.10: MTTDL improvement via intra-disk data redundancy.

6.4.5 Multi-feature Case: Scrubbing and Intra-disk Parity

Scrubbing and intra-disk parity can be used simultaneously to improve MTTDL. In this subsection, we evaluate performance of these two features when running concurrently in idle times, dubbed as "scrubbing+parity". Because both features run in background

without any buffer requirement, their queue capacity is assumed to be infinite. Recall that scrubbing generates *infinite* work while parity updates require *finite* work. Here, we evaluate a scenario when parity updates have higher priority than scrubbing. This means that scrubbing is scheduled only if there is no parity update waiting. As in previous sections, the performance degradation of user traffic is kept below the preset 7% threshold.

Results for Trace T4

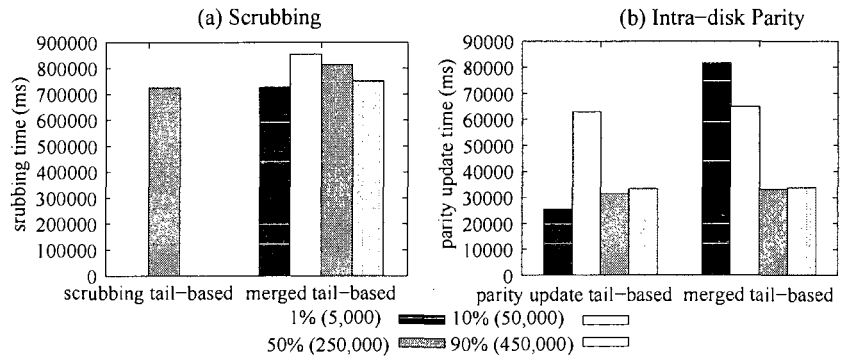


Figure 6.15: Average (a) scrubbing and (b) parity update times when running individually and together.

Initially, we present results for T4. As for this trace, both scrubbing and parity updates individually perform better using the tail-based policy, Figures 6.15(a) and (b) give the average scrubbing and parity update times under this policy. For comparison, in each plot the results of only disk scrubbing and only intra-disk parity are also included. For the case of scrubbing, all variants of trace T4 perform the same because scrubbing is workload independent.

Although scrubbing has lower priority than intra-disk parity update, enabling it concurrently with parity updates does not affect its performance considerably (i.e., only 10% in the worst case). Similarly, parity updates see minimal change in their performance

because they are processes of higher priority than scrubbing. The only exception is the case with the smallest amount of parity updates (i.e., 1% user WRITEs). As discussed in Section 6.4.4, the effect of parity updates in user traffic performance is almost zero for this case and parity update times are the smallest. However, adding the infinite scrubbing work degrades parity update performance by as much as 3 times.

Figure 6.16 shows overall system utilization, which is dominated by the work done for scrubbing. Because the work related to parity updates is small, its completion barely adds to the system utilization. It is scrubbing with its infinite amount of work that keeps the system continuously utilized.

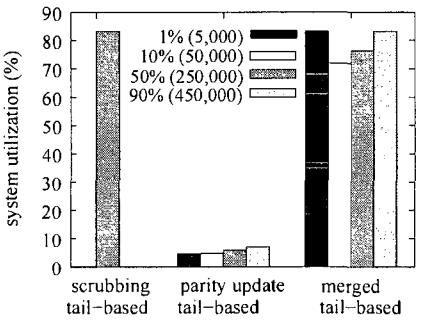


Figure 6.16: Overall system utilization under scrubbing and parity updates when they run individually and together.

Results for Trace T3

Here we present results for trace T3 which is characterized by idle periods with low variability. For this trace, scrubbing performs better using the body-based policy while parity updates are done more efficiently using the tail-based policy. Thus, in addition to the body-based and the tail-based policies for the combined background work, we also evaluate another scheduling policy which schedules scrubbing work via the body-based policy

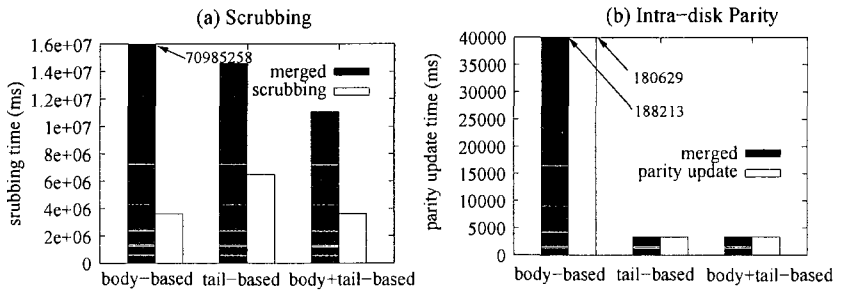


Figure 6.17: Average time for (a) an entire scrubbing, (b) parity updates for trace T3 (low variability). The body+tail policy schedules scrubbing via the body-based policy and parity-updates via the tail-based policy

and parity-updates via the tail-based policy. This policy is dubbed "body+tail-based" policy.

Figure 6.17(a) presents the average time for a complete scrubbing when run individually (and together) with parity updates. If the body-based policy is used to schedule both types of background jobs, performance degradation on scrubbing is significant. With the body+tail-based variation, each background activity (i.e., scrubbing or parity update) is scheduled using the policy under which it performs best when running individually. Parity updates, because they have higher priority than scrubbing, are not penalized as much as scrubbing (see Figure 6.17(b)). Furthermore, parity updates perform significantly better if they are scheduled using the tail-based policy, independently of how scrubbing is scheduled.

Figure 6.18 presents system utilization for trace T3. Results are in agreement with those shown in Figure 6.17, the body+tail-based policy utilizes best the entire system providing room for both scrubbing and parity updates to perform similar to their best individual performance.

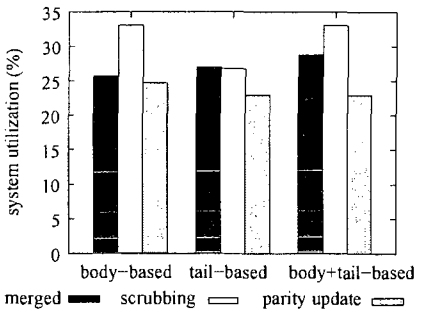


Figure 6.18: Overall system utilization

MTTDL in Data Redundant Drives

We use Eq.(6.5) to estimate the MTTDL improvement when both scrubbing and intra-disk parity are enabled. Differently from the MTTDL estimation in Section 6.4.4, the $MTTDL_{ML^{(1)}}$ and $MTTDL_{ML^{(2)}}$ in Eq.(6.5) are computed using Eq.(6.4). The average time for a complete disk scrubbing when it runs concurrently with parity updates is used in Eq.(6.4) to estimate both $MTTDL_{ML^{(1)}}$ and $MTTDL_{ML^{(2)}}$, i.e., $ML^{(1)}$ is set to half of average scrubbing time and $ML^{(2)} = 0.48 \times 10^2 \cdot ML^{(1)}$. Also assuming MRL ≈ 0. The parameter p in Eq.(6.5) is estimated using Eq.(6.6) and the average parity update time when it runs concurrently with scrubbing. Results are presented in Table 6.11. For trace T3 and four variants of trace T4, the MTTDL improvement attributed to scrubbing and intra-disk parity are as high as 7 and 8 orders of magnitude, respectively. Consistently with the results shown in Figure 6.17, the body+tail-based policy achieves better improvement in the MTTDL than both the body-based and the tail-based policies.

Policy	T3	T4			
		1%	10%	50%	90%
body	1.8 $\times 10^7$	N/A	N/A	N/A	N/A
tail	6.3 $\times 10^7$	1.12 $\times 10^8$	1.09 $\times 10^8$	1.07 $\times 10^8$	1.04 $\times 10^8$
body+ tail	7.1 $\times 10^7$	N/A	N/A	N/A	N/A

Table 6.11: MTTDL improvement via scrubbing and intra-disk parity.

6.5 Background Schedulability Algorithm

In this section, we define a general *schedulability* framework to determine *when* and for *how long* the system can serve background jobs during idle times. This framework is generic yet adaptable to system dynamics and it works consistently well under a broad variety of system conditions and background demands, dealing effectively with the challenging problem of prescribed solutions that cannot possibly apply in every environment.

6.5.1 Algorithmic Framework

The target of this framework is to determine the length of the idle wait period I *and* the length of a background busy period T within an idle interval. As a result, the schedulability of background work is determined from the pair of parameters I and T, as depicted in Figure 6.19.

Depending on the length of the idle intervals and the amount of background work

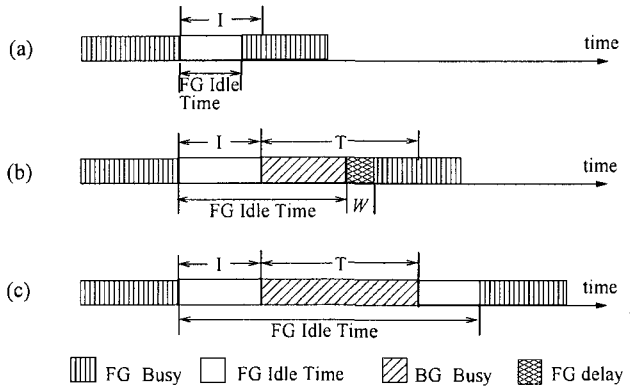


Figure 6.19: Three cases of idleness utilization: (a) no BG job are served in an idle interval shorter than I; (b) BG jobs are served and FG jobs are delayed in an idle interval longer than I and shorter than $(I+T)$; (c) BG jobs are served without delaying FG ones for idle intervals longer than $(I+T)$.

served, foreground jobs may get delayed. We therefore classify the idle intervals into three categories:

(a) Idle intervals that do not serve any background jobs because they are shorter than I (see Figure 6.19(a));

(b) Idle intervals that serve background jobs, but experience a foreground arrival during the execution of background work, because their length is between I and $(I+T)$. In this situation, the background job that is in service continues its service to completion, but the system stops serving additional background jobs even if T has not elapsed yet (see Figure 6.19(b));

(c) Idle intervals that serve background jobs for T units of time and do not experience any foreground arrival in the meantime, because they are longer than $(I+T)$. After serving background work for T units of time, the system remains idle without serving additional background work until a foreground job arrives (see Figure 6.19(c)).

Among all idle intervals, those of the (b) category are of imminent importance, because they can cause foreground performance degradation. As our goal is to contain foreground delays within targets, we especially focus on this case.

Data Structure and Parameters

The framework that we propose here determines the (I, T) pair using the length of idle intervals that are obtained via system monitoring. The empirical distribution of idle intervals is maintained in the form of a cumulative data histogram (CDH) which consists of a compact list of (t_j, C_j) pairs. The finite list of the CDH (t_j, C_j) pairs indexed by the histogram bin j, where t_j is the smallest length of idle intervals falling on the jth histogram bin, and C_j is the corresponding empirical cumulative probability of occurrence $C_j = Pr(idle_{interval} <= t_j)$. The empirical distribution of idle times incorporates foreground workload demands into the decision making without including complex processes, such as the foreground arrival and service processes.

Additional metrics necessary to determine the (I, T) pair are also obtained via system monitoring and include:

- S^{BG}, the average service demands of background jobs,

- RT^{FG}, the average foreground response time without background jobs, which is estimated by monitoring the response times of foreground jobs that are in the busy periods without any background-caused delay, i.e., the proceeding idle interval falls in the (a) and (c) categories of idle intervals (see Figure 6.19), and

- W, the average wait time that the foreground requests experience due to the execution of background jobs, which is estimated by recording the time a foreground job arrives in

a system idle of foreground jobs and the time it actually gets service (see Figure 6.19(b)).

The only user level input in our framework is the degradation target D in foreground performance. Yet, we stress that D may *not* be explicitly provided as a user input. For example, the user input may be in the form of the required amount of background work to be completed. In that case, we find the (I, T) pair that satisfies the user input, i.e., completes the required background work, with the smallest possible foreground degradation target D.

The algorithmic framework first estimates the portion of idle intervals that delay foreground requests, i.e., the idle intervals that fall into the (b) category (see Figure 6.19). This portion of idle intervals is denoted by E and its estimation is central to our algorithmic framework. Once E is estimated, the (I, T) pair is derived based on the histogram of idle interval lengths.

Estimation of E

We define E to be the portion of idle intervals that are utilized by background work which delays foreground jobs. Once a foreground job is delayed with the amount of time W, the entire set of foreground jobs belonging in the same foreground busy period will be delayed by the same amount W. If we assume that all foreground busy periods have the same number of foreground jobs, then E approximates the probability that a foreground job experiences a background-caused delay. Hence, the average response time of foreground jobs RT would be the expected foreground-only response time RT^{FG}, plus the average additional delay W attributed to the background work, which occurs only E percent of

the time,

$$RT = RT^{FG} + E \cdot W. \tag{6.7}$$

Our goal here is to express E via the monitored system metrics RT^{FG}, W, and the degradation target D. We relate D with the expected foreground response time RT and the average foreground-only response time RT^{FG} as follows:

$$D = \frac{RT - RT^{FG}}{RT^{FG}}. \tag{6.8}$$

Combining Eq.(6.7) with Eq.(6.8), we get

$$D = \frac{RT^{FG} + E \cdot W - RT^{FG}}{RT^{FG}} = \frac{E \cdot W}{RT^{FG}}, \tag{6.9}$$

which can be re-written to express E as

$$E = \frac{D \cdot RT^{FG}}{W}. \tag{6.10}$$

Because we use the degradation target D in foreground performance in Eq.(6.10), the estimated E ensures that the background-caused delay does not exceed D and does not violate foreground performance targets. The estimation of E is critical, because it represents the mapping of the user input D on to our main data structure, i.e., the CDH of idle interval lengths, and facilitates the estimation of the (I, T) pair. The accuracy of E depends on the accuracy of the monitored values for RT^{FG} and W. In our evaluation, we show that even if we use average monitored estimates, the final output is consistently satisfactory.

Estimation of (I, T)

We use the parameter E estimated via Eq.(6.10) and the histogram (CDH) of idle times to derive the (I, T) pair. For this, we scan the sorted list of the CDH (t_j, C_j) pairs, for intervals of length E. In practice, there may not exist an interval with exact length E. Instead, for each (t_j, C_j), we find a $(t_{j'}, C_{j'})$ such that $|C_{j'} - C_j - E| = \varepsilon$, where $C_{j'} > C_j$ and ε is a small number (e.g., 0.05). Each such (t_j, t'_j) pair represents one choice for (I, T), which we index by i and denote as $(I_i = t_j, T_i = t'_j - t_j)$. See Figure 6.20 for a high level depiction. The result of the entire scanning process is a set of (I_i, T_i) pairs.

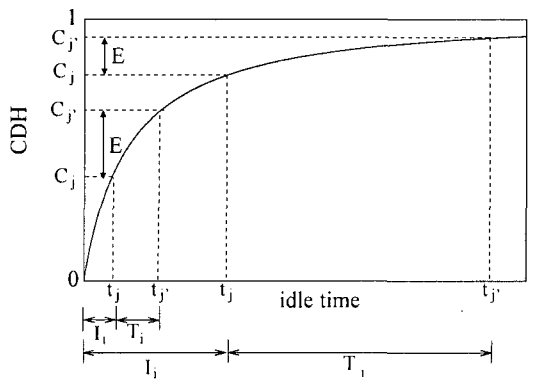


Figure 6.20: Transition from E to (I_i, T_i) in a cumulative data histogram. Any interval of length E in the y-axis is mapped uniquely onto an interval in the x-axis described by the pair (I_i, T_i). Because E defines multiple intervals in the y-axis (between 0 and 1), multiple (I_i, T_i) pairs exist.

Avoiding background starvation If in the set of all (I_i, T_i) pairs there is no interval T_i which is at least S^{BG} long, then no background job can be served and background jobs may experience starvation. To avoid starvation, we substitute E with a larger E' value and estimate a new set of (I_i, T_i) pairs for E' such that at least one of the new $T_i \geq S^{BG}$. To prevent additional delays in foreground performance after substituting E with E', the background jobs are served with probability E/E' in any eligible idle interval (i.e.,

interval longer than the idle wait I_i). The transition from E to E' is conservative with small increments (e.g., 0.05) in order to delimit foreground degradation and maintain it as close to its degradation target D as possible.

Selecting among the (I_i, T_i) pairs Because every (I_i, T_i) pair is chosen such that only E percent of idle intervals delay foreground jobs, the foreground performance target is met by any of the (I_i, T_i) pairs. The final (I, T) pair is selected such that as much as possible outstanding background work is served as soon as possible.

Every (I_i, T_i) can serve in average B_i amount of background work measured in units of time. Note that measuring work in units of time or number of jobs is qualitatively equivalent, because one is derived from the other using only the average background service time S^{BG}. We estimate B_i as follows:

$$B_i = T_i \cdot Pr(idle > (I_i + T_i)) + \sum_{r=1}^{\lceil T_i/S^{BG} \rceil} r \cdot S^{BG} \cdot (C_r - C_{r-1}), \tag{6.11}$$

where $Pr(idle > (I_i + T_i))$ is the probability that an idle interval is greater than $(I_i + T_i)$ and C_0 is the probability that an idle interval is less than I_i. Idle intervals longer than $(I_i + T_i)$ can serve T_i background work. Thus, the first term in Eq.(6.11) gives the amount of background work completed in these idle intervals. The second summation term gives the amount of background work completed in idle intervals longer than I_i but shorter than $(I_i + T_i)$. In such idle intervals, less than T_i background work will be served. Figure 6.21 demonstrates the estimation of the background work to be completed in these idle intervals. The rth subinterval of length S^{BG} has probability $C_r - C_{r-1}$ and serves $r \cdot S^{BG}$ background work, see Figure 6.21. An idle interval shorter than I_i does not serve any background work, thus it is not represented in Eq.(6.11).

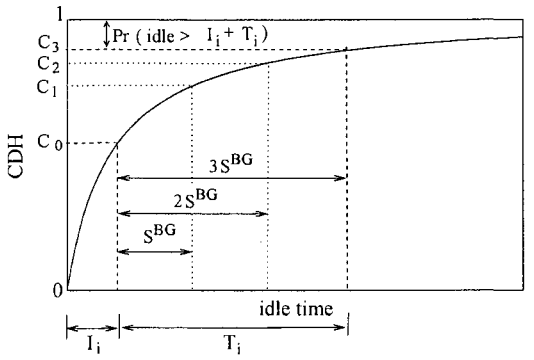


Figure 6.21: Estimation of the BG work B_i that completes during idle intervals if (I_i, T_i) is the schedulability pair.

Each (I_i, T_i) is augmented by the corresponding B_i and the selection of the final (I, T) is done according to the type and amount of background work available in the system. Estimating the amount of available background work B is system/feature dependent. Media scans may run continuously [8] and the amount of work associated to them is infinite, i.e., $B = \infty$.

Unlike background media scans, the work associated with the majority of background features in storage systems depends on the foreground workload. For example, WRITE verification [79] and parity updates [23], generate background work that depends linearly on the amount of WRITE foreground traffic. In these cases, the monitored foreground traffic is used to estimate the amount of background work. For example, if an average of M WRITEs arrive in every foreground busy period, then the amount of background work associated with WRITE verification, where for each foreground WRITE, a background READ is generated, is $B = M \cdot S^{BG}$ in average. Also, the amount of background work associated with parity updates, where for each foreground WRITE, a background READ and a background WRITE are generated, is $B = 2M \cdot S^{BG}$.

Once B is estimated, the final pair (I, T) is selected such that I is the smallest among

all possible (I_i, T_i) pairs, where $B_i > B$. The condition to select the shortest idle wait I_i enables the fastest possible background response time. If $B = \infty$, then the final (I, T) is the one with the maximum estimated B_i.

6.5.2 Analysis and Evaluation

We develop a trace-driven discrete event simulation model for the evaluation. Because the focus of the methodology is to determine when to start and stop serving background jobs, our simulation aims at correctly modeling the interaction between foreground and background busy periods rather than the specifics of scheduling each job inside a busy period.

In our evaluation, we use a set of disk-level traces measured in a number of personal and enterprise-level systems. We experimented with the entire set of traces, but to keep the presentation concise, we present here detailed results on traces $T3$ and $T6$, which we considered challenging and representative. Table 6.12 summarizes the main characteristics of these two traces. Trace $T3$ is selected because it is the trace with the highest utilization and idle intervals with low variability. Trace $T6$ is selected because it has the highest utilization among traces with high variability in idle intervals. We also note that there is significant burstiness in the idle times of $T6$.

Trace	Mean Arrival	Mean Service	Util (%)	Mean Idle	CV Idle	Bursty
T3	62.85	10.68	17.4	91.98 ms	0.98	No
T6	69.20	5.74	8.3	30.68 ms	6.16	Yes

Table 6.12: Overall characteristics of traces used in our evaluation. The measurement unit is ms.

Figure 6.22 gives the empirical cumulative data histograms (CDHs) of idle interval lengths for traces $T3$ and $T6$. The tail of the distribution of the idle interval lengths for $T6$ is longer than for $T3$, which implies that trace $T6$ has many short idle intervals and some very long ones, while in trace $T3$ most idle intervals are of similar lengths.

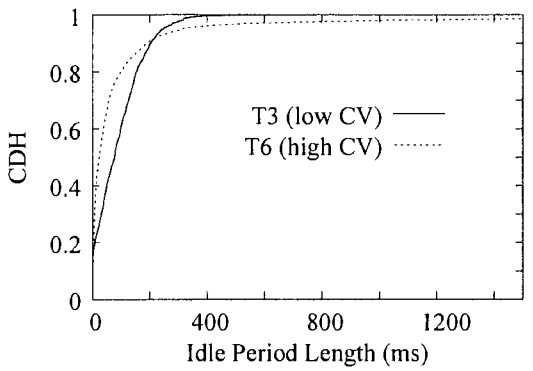


Figure 6.22: CDH of idle times for traces $T3$ and $T6$.

We evaluate the performance of our methodology under different amount of background work. Specifically, we experiment with background work that is 10%, 40%, and 90% of the foreground work, as well as the extreme case of having "infinite" background work in the system. While foreground busy and idle periods are determined by the traces, in our model we set the service time of background jobs to be exponentially distributed with a mean of 6.0 ms, which is similar to the mean service time of foreground jobs in trace $T3$.

Same as in the previous sections, the acceptable slowdown of foreground jobs due to background jobs is set to 7%. We have conducted experiments with various values of D and results are qualitatively similar to those reported here. The metrics of interests are: (a) the average relative delay of foreground jobs due to background work, defined as $(RT - RT^{FG})/RT^{FG}$, and (b) the number of completed background jobs.

System Performance

Table 6.13 shows the results for the four levels of background work. We observe that in most scenarios the relative foreground delay is well below the degradation target D. Under trace $T6$, the system serves significantly more background jobs than under trace $T3$, because the utilization of trace $T3$ is twice as high as the utilization of trace $T6$. If the background work is infinite, then the results in Table 6.13 show that trace $T3$ can accommodate background work that is twice as much as the foreground one and that trace $T6$ accommodates as much as six times more background work than foreground one.

Trace	BG Target Work	FG Delay (Target D=7%)	Completed BG	
			Reqs.	Work
T3	10%	1.4%	3,861	10%
	40%	2.0%	15,514	40%
	90%	3.9%	34,953	90%
	infinite	7.0%	74,234	190%
T6	10%	3.2%	132,362	10%
	40%	6.8%	528,287	40%
	90%	4.7%	1,190,208	90%
	infinite	3.9%	7,862,813	610%

Table 6.13: FG delay, completed BG requests, and completed BG work relative to the incoming FG work.

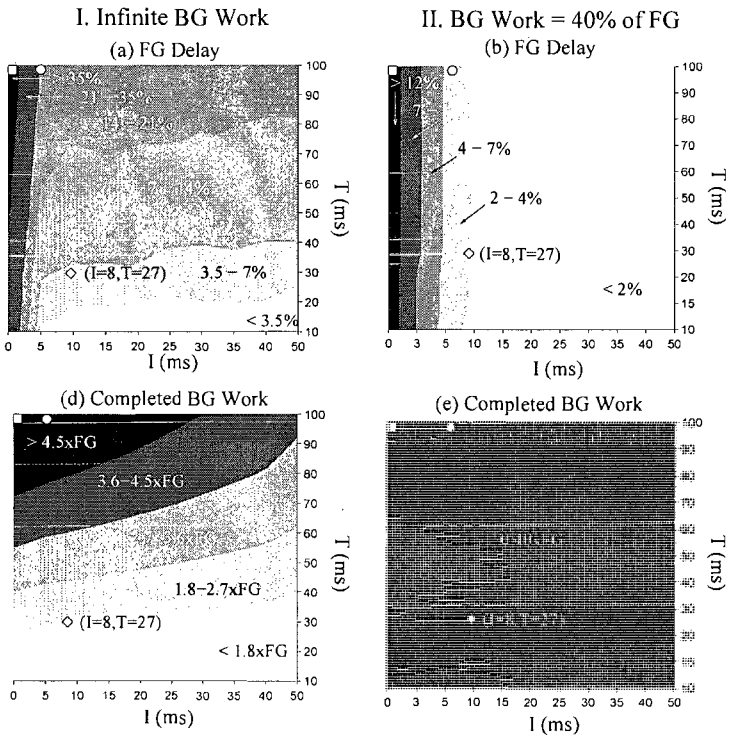


Figure 6.23: Trace $T3$. FG delay and completed BG work for *any* (I, T) pair. Diamond shapes mark our solution. Square and circle shapes mark common practices.

Optimality of the (I, T) Pair

To evaluate the effectiveness of our methodology in utilizing idleness, we perform a state space exploration, i.e., estimating the foreground and background performance for *any* (I, T) pairs. Figures 6.23 and 6.24 give the results of the state space exploration analysis for traces $T3$ and $T6$, respectively. We evaluate the cases of infinite background work in the first column, and background work that is 40% of the arriving foreground work in the second column. The first row in Figures 6.23 and 6.24 shows the background-caused delay on foreground performance and the second row presents the completed background work. In each plot, we mark the pair generated by our approach with a diamond. For

comparison with common practices [26], we also mark with a square the results for the pair $(I = 0, T = \infty)$, i.e., no idle wait, and with a circle the results for the pair $(I = 6, T = \infty)$, i.e., fixed idle wait equal to the average background service demand.

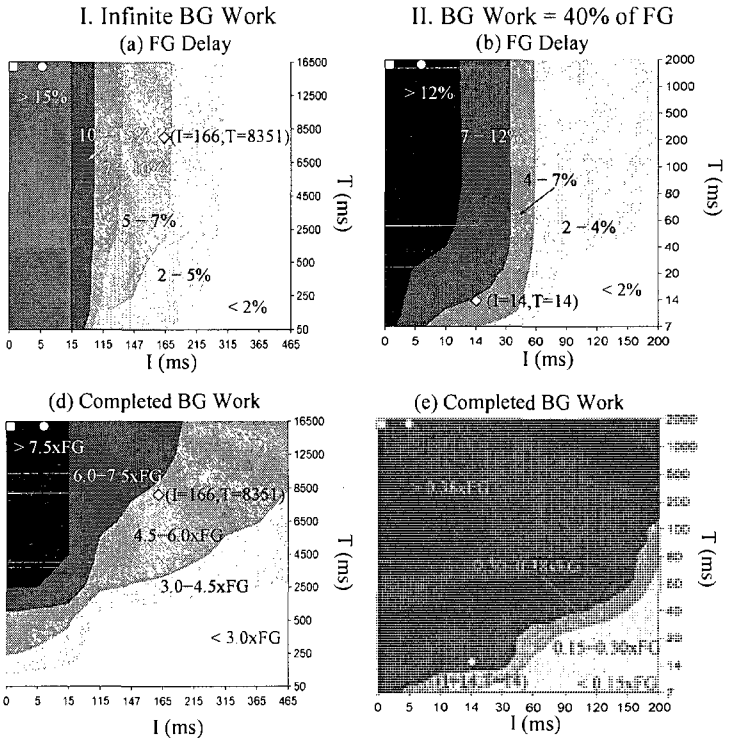


Figure 6.24: Trace $T6$. FG delay and completed BG work for *any* (I, T) pair. Diamond shapes mark our solution. Square and circle shapes mark common practices.

Figures 6.23 and 6.24 clearly indicate that the pairs representing common practices provide a fixed solution independent of the effect they have on foreground or background performance (see the fixed position of the circle and square shapes in all plots). The pair $(I = 0, T = \infty)$ significantly degrades foreground performance, by more than 10% for both traces, and confirms that idle wait is necessary in scheduling background work. With an idle wait equal to the average background service demand, the pair $(I = 6, T = \infty)$ keeps the background-caused delays low for several scenarios, in particular for trace $T3$ with low

variability in idle periods. However, it fails to meet performance targets if the amount of background work is large (e.g., the infinite case) or if idle intervals are variable (e.g., trace $T6$).

Figures 6.23 and Figures 6.24 indicate that there is a set of pairs (I, T) that would satisfy the degradation target $D = 7\%$. For example, plots (a) and (d) in Figure 6.23, indicate that the idle wait I should be at least 5 ms and the length of background busy period T should be at most 40 ms. However, having T shorter than 25-30 ms or I larger than 20 ms would result in reduced levels of completed background work.

The pair (I, T) estimated using our methodology is *consistently* among the ideal choices that strike a good balance between the completed background work and foreground performance. Our results confirm that it is necessary not only to idle wait but also to limit the amount of background work completed in every idle interval (i.e., have $T < \infty$) to sustain foreground performance at desired levels. Furthermore, controlling foreground performance by only changing the idle wait length I (as in common practices) would result in background work starvation.

6.6 Chapter Summary

In this chapter, we proposed some new schemes to efficiently schedule non-preemptive background jobs during idle intervals, such that two conflicting goals are met: (1) degrade foreground performance by no more than a predefined target, and (2) avoid background work starvation. We show that monitoring the stochastic characteristics of idle times is important, which allows us to incorporate accurately the complex interaction between the arrival and service processes of foreground traffic. We also identify burstiness as a

source of additional information to improve idle time utilization. The analysis shows that if burstiness exists in the observed idle interval lengths, then it can be used to predict the near future. Predicting that the next idle interval is long given that the current interval is also long is of particular interest, because scheduling of background jobs can become more aggressive. As a result, more background work completes with less delays in foreground jobs and tremendously shorter tails in the foreground response time distribution.

The proposed background scheduling schemes are exploited to evaluate the performance of two data loss prevention techniques, i.e., disk scrubbing and intra-disk data redundancy. Scrubbing (representing infinite amount of background work) and parity-updates related to intra-disk redundancy (representing finite amount of background work) can even be scheduled simultaneously and still meet user performance targets. Each of the evaluated background features improves data reliability by orders of magnitude. The enhancement on data reliability when both background activities are scheduled in the system is higher than the linear combination of their individual benefits.

Chapter 7

Conclusions and Future Work

The main contribution of this dissertation is on the design of effective systems using the knowledge of burstiness in workloads. The new techniques and tools that are developed in this dissertation are summarized as follows.

- For performance prediction, we have developed new effective and robust capacity planning methods that model burstiness in the arrival and/or service process of multi-tier enterprise systems.
- For general scheduling, we have designed two new scheduling policies for systems with bursty workloads, which can achieve good estimates of service times of upcoming requests and improve system performance by selectively delaying requests.
- For idleness management, we have proposed a model to evaluate the performance trade-offs of foreground and background work under bursty arrivals and develop new background scheduling algorithms to determine the schedulability of background work during idle times in storage systems.

We have shown that burstiness is extremely important in performance models and system design, as it has a dramatic impact on system performance. The focus of this dissertation focus is on the development and the design of new techniques and tools for performance prediction, scheduling, and resource allocation that leverage on knowledge about the future workload that can be derived if burstiness exists.

We develop a new capacity planning model to capture burstiness in the *service process* of multi-tier enterprise systems. Using the index of dispersion together with other measurements that reflect the estimated mean and the 95th percentile of service times, a Markov-modulated process is derived that captures well both burstiness and variability of the service process. The model parameterization is done by inferring essential process information from inexact and limited measurements in a real system. Experimental results demonstrate that this parameterized model can accurately predict performance in systems even in the very difficult case where there is persistent bottleneck switch among various servers.

This dissertation also proposes a simple and robust approach that injects burstiness into the *arrival process* of the TPC-W benchmark. The revised TPC-W benchmark can thus be practically used for assessing the effectiveness of mechanisms that counteract burstiness. This new approach incorporates different intensities of burstiness into the arrival flows via the index of dispersion. Detailed experimentation in a real testbed proves the effectiveness and robustness of the proposed approach.

This dissertation also exploits the knowledge of burstiness in scheduling, another important component in system design. We illustrate that the information of the future workload, e.g., service demands of upcoming requests, can be derived from its burstiness

profile. By taking advantage of this derived information, two new measurement-based scheduling policies, called SWAP and ALoC, are designed to maintain high availability by delaying those requests that contribute to burstiness. Using simulations, we show that SWAP consistently improves performance and availability compared to the first-come first-served (FCFS) scheduling and is able to effectively approximate the shortest job first (SJF) scheduling. We also show that ALoC, as an extension of SWAP, adaptively controls system load to meet pre-defined quality-of-service levels and significantly improves system performance by infinitely delaying (i.e., denying service) only a small fraction of requests.

For storage systems, the completion of background work is critical for system operation. Yet, scheduling non-preemptive background jobs should not degrade foreground performance more than predefined targets. This dissertation presents a new background scheduling scheme which can determine when and for how long idle times can be used for serving background jobs, without violating predefined performance targets of foreground jobs. We also show that burstiness in idle times provides additional information to improve idle time utilization with less degradation on foreground performance. An extensive set of trace-driven simulation results proves that our approach is effective and robust in a wide range of system conditions. Furthermore, we demonstrate that this new approach can successfully schedule two maintenance features, namely disk scrubbing and intra-disk data redundancy, without affecting foreground task performance, while improving system reliability.

7.1 Future Work

There are several extensions to the results presented in this dissertation that are subject of future work.

- **Resource Allocation Mechanisms:** Supporting service level agreement (SLA) guarantees for bursty arrivals is a challenging task for resource allocation mechanisms, as they should understand and tune system parameters under bursty traffic scenarios. Burstiness may impact in an unexpected way the performance of different resource allocation mechanisms, e.g., the Session-based admission control (SBAC) mechanism [18]. We will improve SBAC via considering burstiness in both arrival streams and service demands and consequently counteracting its performance effect. We expect that the new version of SBAC can prevent the overload conditions and support the SLAs guarantees under bursty workload conditions. We will also design a new resource allocation algorithm for autonomic system management when there is burstiness in the arrival stream. By online monitoring the arrival flows and measuring the corresponding index of dispersion, the algorithm autonomically detects the burst of client requests and then appropriately allocates system resources based on this information. For instance, if a burst of arrivals is detected, then the system may add more servers at the front tier for providing adequate service for all requests. Also, the system may shut off some servers for energy conservation when less client requests are coming during the next monitoring period. We expect that this new algorithm can not only support the SLAs guarantees but also save power, especially in the case of bursty arrivals.

- **Refined Background Job Scheduling Schemes:** The scheduling schemes for background jobs bases its scheduling decisions on the empirical distribution of idle times, as well as on the mean of service times of background jobs. However, we have observed in the preliminary simulations that the effectiveness of the scheme diminishes quickly when the service times of background jobs are not exponentially distributed. This indicates that monitoring the mean of background jobs is not enough. One future direction on background scheduling is to improve the algorithm accuracy by exploring how to incorporate more statistical characteristics (i.e., not only the mean) of the service process of background jobs. Another possible direction for future work on background scheduling scheme is to extend the current one to support a wider range of background work. In this dissertation, the background tasks are treated strictly with lower priority than foreground ones. For example, background media scans [84] are *always* of lower priority. However, background activities in storage systems [33] may not always have lower priority than foreground ones. Examples of such background activities include disk cache flushing and RAID rebuild. Such activities can be deferred in background, but not indefinitely, i.e., there is a deadline associated with their completion. For example, flushing the disk cache is commonly a background activity, but it puts foreground jobs on wait if the cache is full and must be flushed right away. We will work to refine the background scheduling schemes to account for the above conditions.

- **Intelligent Power/Energy Management:** Efficient and intelligent power control is one of the most crucial but challenging research issues in computer systems. For example, reducing energy consumption is an important issue for a data center.

However, the storage subsystem, among the various components of a data center, consumes significant amounts of energy. Even worse, the fraction of energy consumption tends to increase as storage requirements rise. We will deploy the background job scheduling scheme to evaluate in-depth specific system architectures and features, e.g., effectively power-off disks in a storage system with the goal of reducing power/energy consumption. In storage systems, a tradeoff exists between disk energy conservation and performance penalty because it is not instantaneous to bring a disk up to the active mode when a new request arrives and thus causes some amount of delay on the upcoming requests. Multi-level controls in the idle mode, such as not engaging disk heads or spinning down disks, may conserve more disk energy at the cost of more performance penalties on disk requests. Consequently, we will develop intelligent power control techniques to select an optimal idle mode level, aiming at well balancing the tradeoff between disk energy conservation and performance penalty. In addition, this dissertation has found that idle times in enterprise storage systems are sometimes bursty. Therefore, we will also use the knowledge of burstiness, as well as the distribution of idle times, to predict the length of the future idle intervals. If the upcoming idle interval is predicted as long, then the disks in the data center can be sent to low-power mode immediately, aiming at saving more power/energy.

Appendix A

MAP(2) Generation

In order to better understand how the 2-state Markovian Arrival Process (MAP(2)) works, we provide the following pseudo code to generate a sample of n_t values $\{X_1, X_2, \ldots, X_n, \ldots, X_{n_t}\}$ from a MAP(2) parameterized by the tuple $(l_{12}, l_{21}, l_{11}, l_{22}, l_{12}, v_{12}, v_{21})$, as shown in Figure 2.4 of Chapter 2.

function: MAP_sample(l_{12}, l_{21}, l_{11}, l_{22}, l_{12}, v_{12}, v_{21}, n_t)

1. initialize active state S_a and inactive state S_i

a. $S\_a =$ "1";

b. $S\_i =$ "2";

2. generate a sample for $n = 1, 2, \ldots, n_t$

a. $X_n = 0$;

b. calculate the transition probabilities

I. $p_1 = l_{S\_aS\_a}/(l_{S\_aS\_a} + l_{S\_aS\_i} + v_{S\_aS\_i})$;

II. $p_2 = l_{S\_aS\_i}/(l_{S\_aS\_a} + l_{S\_aS\_i} + v_{S\_aS\_i})$;

III. $p_3 = v_{S\_aS\_i}/(l_{S\_aS\_a} + l_{S\_aS\_i} + v_{S\_aS\_i})$;

c. r = random number in $[0, 1]$;

d. if transitions only signify real events, i.e., $r < p_1$

I. X_n += sample from exponential distribution with rate $l_{S\_aS\_a}$;

II. go to **Step 2.**; //does signify a real event

e. else if transitions signify real events and change states, i.e., $r < p_2$

I. set S_a to the previous S_i;

II. set S_i to the previous S_a;

III. X_n += sample from exponential distribution with rate $l_{S\_aS\_i}$;

IV. go to **Step 2.**; //does signify a real event

f. else if transitions only change states, i.e., $r > p_2$

I. set S_a to the previous S_i;

II. set S_i to the previous S_a;

III. X_n += sample from exponential distribution with rate $v_{S\_aS\_i}$;

IV. go to **Step 2.c.**; //does not signify any real event

Appendix B

Revised TPC-W Benchmark

We exemplify the effectiveness of the new methodology (see Chapter 3) by introducing a new module into the TPC-W, a benchmark that is routinely used for capacity planning of e-commerce systems. This new module uses a shared Markovian Arrival Process (MAP) to draw think times for all users emulated on the same client machine, and hence injects burstiness into the arrival flows of the system. The modified code of TPC-W and related scripts are as follows.

B.1 www_map.m

A MATLAB script that generates a 2-state MAP for user think times. Note that in order to run this script, we refer the reader to download the MAP Queueing Network Toolbox at `http://www.cs.wm.edu/~ningfang/tpcw_codes/`.

```
 function MAPZ=www_map(Z,I,N)
% MAPZ=www_map(Z,I,N) - generate a 2-state MAP for user think time
%
```

```
% Input:
% Z: the mean user think time, e.g., 7 seconds
% I: the index of dispersion
% N: the maximum number of emulated browers
%
% Output:
% MAPZ: a 2-state MAP for user think time
%
% Examples:
% - www_map(7,4000,1200)


%% MAPs for the service process at the front server
FS_D0=[-127.9035   27.0132
        78.2006 -633.9898];
FS_D1=[ 100.8902   0
        0  555.7892];
MAPFS={FS_D0,FS_D1};


%% MAPs for the service process at the dababase server
DB_D0=[-74.0    50.7
       328.7   -6550.2];
DB_D1=[23.3      0
       0       6221.5];
```

```
MAPDB={DB_D0,DB_D1};

%%for two-tier system
meanFS=map_mean(MAPFS)+map_mean(MAPDB);

F=1.0001;
ACF=0;
threshold = 0.4;

while ACF<threshold
    %%mean long think time
    Tflushout=max([F*Z,F*N*meanFS]);
    %%mean short think time
    Tflushin=meanFS/F;
    D0=diag([-1/Tflushout,-1/Tflushin]);
    %%probability of jump from short state to long state
    p2=0.01;
    %%probability of jump from long state to short state
    p1=-p2*(Tflushout-Z)/(Tflushin-Z);
    P=[1-p1,p1;p2,1-p2]; D1=-D0*P;
    MAPZ={D0,D1};

    MAXITER=1000;
```

```
    while (MAXITER>0 && abs(map_idc(MAPZ)-I)>0.01*I)
        MAXITER = MAXITER - 1;
        if map_idc(MAPZ)>I
        %%real I is larger than the target I,
        %%increase p2 to be in long state more frequently
            p2=p2/rand;
            p1=-p2*(Tflushout-Z)/(Tflushin-Z);
            P=[1-p1,p1;p2,1-p2]; D1=-D0*P;
            MAPZ={D0,D1};
        elseif map_idc(MAPZ)<I
        %%real I is smaller than the target I
        %%derease p2 to be in long state less frequently
            p2=p2*rand;
            p1=-p2*(Tflushout-Z)/(Tflushin-Z);
            P=[1-p1,p1;p2,1-p2]; D1=-D0*P;
            MAPZ={D0,D1};
        end
    end
    I=map_idc(MAPZ);
    ACF=map_acf(MAPZ,1);
    F=F*2;
end
end
```

B.2 rbe.MMPP.java

A new java file that generate random numbers following the Markov-Modulated Poisson Process (MMPP(2)) distribution.

```
package rbe;

import java.net.*;

import java.io.*;

import java.util.Random;

import java.util.Date;

import java.util.Vector;


public class MMPP {

/********************
 * Each state in MMPP
 ********************/
class STATE {

    double   mean; // mean service time

    double[]    p; // transmission probabilities

                      p[i*#states+j] is the probability of

                      transmission from this state to state j

                      in D_i */

    Random rand_ind; // rand stream index of service time

    Random rand_trans; // rand stream index of state transmission
```

```
    double  during; // during time in this state

    public STATE() {
        mean = 0.0;
        rand_ind = null;
        rand_trans = null;
        during = 0.0;
        p = null;
    }
} //class States

STATE[] states;     // states in BMAP
int     numState;   // number of states
int     numBulk;    // number of bulk arrivals
int     curr_ind;   // index of current state
double  mean;       // mean
double  svar;       // \sum (x_i - mean)
int     number;     // number of intervals generated
final double INF = 999999999;

// Constructor of MMPP
// rand: rand seed for random to be used in this object
public MMPP(Random rand){
```

```
    initialization(rand);
}

public MMPP(){
    Random rand = new Random();
    initialization(rand);
}

// Initialize the MMPP
// Get D0 and D1
void initialization(Random rand) {
    int i, j, k;
    try{
        numState = 2;
        numBulk = 1;
        //2-state MAP with Z=7sec,I=4000
        double [][][] D =
        {{{-0.03443249765465439,  0},
          {0,    -192.8605531916577}},
         {{0.03439111976070266,     0.00004137789395173437},
          {0.07354649281846881,     192.7870066988392}}
        };
        states = new STATE[numState];
```

```
for (i = 0; i < numState; i++) {
    states[i]=new STATE();
    states[i].mean = 0.0;
    states[i].rand_ind = new Random(rand.nextLong());
    states[i].rand_trans = new Random(rand.nextLong());
    states[i].during = 0.0;
    states[i].p = new double[(numBulk+1) * numState];
}
for (j = 0; j < numBulk + 1; j++) {
    for (i = 0; i < numState; i++) {
        for (k = 0; k < numState; k++) {
            states[i].p[j*numState+k] = D[j][i][k];
            if (states[i].p[j*numState+k] < 0.0)
                states[i].p[j*numState+k] = 0.0; //diagonal in D0
            states[i].mean += states[i].p[j*numState+k];
        } // k
    } // i
} // j
for (j = 0; j < numBulk + 1; j++) {
    for (i = 0; i < numState; i++) {
        for (k = 0; k < numState; k++) {
             if (states[i].mean == 0)
                states[i].p[j*numState+k] = 0;
```

```
                    else
                       states[i].p[j*numState+k] =
                           states[i].p[j*numState+k] / states[i].mean;
                    if (j*numState+k > 0)
                       states[i].p[j*numState+k] +=
                           states[i].p[j*numState+k-1];
                } // k
            } // i
        } // j
    } catch (java.lang.Exception ex) {
        System.out.println("Error in initialize MMPP");
        ex.printStackTrace();
    }
} // initialization()

/*******************************************************
 * Generate interarrival time for BMAP
 * Retruns the number which follows BMAP distribution
 *******************************************************/
public synchronized double gen_interval()
{
     double  interval    = 0.0;
     double  theo_mean   = 0.0;
```

```
        double  prob;
        int     i, bulk;
        theo_mean = states[curr_ind].mean;
           if (theo_mean < 0.000001)
               states[curr_ind].during = INF;
           else
               states[curr_ind].during =
                   Expo(1/theo_mean, states[curr_ind].rand_ind);
        interval += states[curr_ind].during;
        if (interval == INF)
            return interval;
           //find the next state based on prob
        prob = states[curr_ind].rand_trans.nextDouble();
        for (i = 0; i < numState*(numBulk+1); i++)
            if (prob <= states[curr_ind].p[i]) break;
        bulk= i / numState;
        i   = i % numState;
        curr_ind = i;
        if (bulk == 0) //instate transition,
            interval += gen_interval();
           return interval;
    } // get_interval()
```

```
/*******************************************************
* Returns a random number from an exponential distribution
*******************************************************/
private double Expo(double m, Random rand)
{
        return (-m * Math.log(1.0 - rand.nextDouble()));
}
```

B.3 rbe.RBE.java

TPC-W Remote Browser Emulator. In this java file, a new function is added to draw a new user think time from an MMPP(2) distribution.

```
...
/****************************************************
 * Returns a MMPP(2) distributed positive real number
 ****************************************************/
public final long MMPP2(MMPP mmpp, long min, double lMin,
                        long max, double lMax)
{
        return((long) mmpp.gen_interval());
}
...
```

B.4 rbe.EB.java

TPC-W Emulated Browsers. In this java file, the user think times are generated by an MMPP(2) distribution instead of an exponential distribution.

```
...
//define a MMPP(2) used to generate think times
public static MMPP mmpp_tt = new MMPP();
...
//comment the original function negExp
//generate user think times from an exponential distrribution
//long r = rbe.negExp(rand, 7000L, 0.36788, 70000L, 4.54e-5, 7000.0);
//generate user think times from an MMPP(2) distrribution
long r = 1000*rbe.MMPP2(mmpp_tt, 7000L, 0.36788, 70000L, 4.54e-5);
...
```

Bibliography

[1] *ISO/IEC 13818: Generic coding of moving pictures and associated audio (MPEG-2).*

[2] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Lazy verification in fault-tolerant distributed storage systems. In *Proc. of IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 179-190, 2005.

[3] V. Almeida, M. Arlitt, and J. Rolia. Analyzing a web-based system's performance measures at multiple time scaless. *ACM Perf. Eval. Rev.*, 30(2):3-9, 2002.

[4] A. T. Andersen and B. F. Nielsen. On the statistical implications of certain random permutations in Markovian Arrival Processes (MAPs) and second-order self-similar processes. *Perf. Eval.*, 41(2-3):67-82, 2000.

[5] M. F. Arlitt and T. Jin. Workload characterization of the 1998 world cup web site. Technical Report HPL-1999-35R1, HP Labs, 1999.

[6] M. F. Arlitt and C. L. Williamson. Web server workload characterization: The search for invariants. In *Proc. of ACM Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, pages 126-137, 1996.

[7] E. Bachmat and J. Schindler. Analysis of methods for scheduling low priority disk drive tasks. In *Proc. of ACM Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, pages 55-65, 2002.

[8] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An analysis of latent sector errors in disk drives. In *Proc. of ACM Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, pages 289-300, 2007.

[9] M. Baker, M. Shah, D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, T. J. Giuli, and P. Bungale. A fresh look at the reliability of long-term digital storage. In *Proc. of European Systems Conference (EuroSys)*, pages 221-234, 2006.

[10] G. Balbo and G. Serazzi. Asymptotic analysis of multiclass closed queueing networks: Common bottlenecks. *Perf. Eval.*, 26(1):51-72, 1996.

[11] G. Banga and P. Druschel. Measuring the capacity of a web server under realistic loads. *World Wide Web*, 2(1-2):69-83, 1999.

[12] P. BARFORD AND M. CROVELLA. Generating representative web workloads for network and server performance evaluation. *ACM Perf. Eval. Rev.*, 26(1):151-160, 1998.

[13] J. BERAN. *Statistics for long-memory processes.* Chapman & Hall, New York, 1994.

[14] G. CASALE, P. CREMONESI, AND R. TURRIN. Robust workload estimation in queueing network performance models. In *Proc. of Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 183-187, 2008.

[15] G. CASALE, N. MI, AND E.SMIRNI. Bound analysis of closed queueing networks with workload burstiness. In *Proc. of ACM Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, pages 13-24, 2008.

[16] G. CASALE, E.Z. ZHANG, AND E. SMIRNI. Characterization of moments and autocorrelation in MAPs. *ACM Perf. Eval. Rev.*, 35(1):27-29, 2007.

[17] G. CASALE, E.Z. ZHANG, AND E. SMIRNI. Interarrival times characterization and fitting for markovian traffic analysis. Technical Report WM-CS-2008-02, College of William and Mary, 2008. Available at http://www.wm.edu/computerscience/techreport/2008/WM-CS-2008-02.pdf.

[18] L. CHERKASOVA AND P. PHAAL. Session based admission control: a mechanism for peak load management of commercial web sites. *IEEE Trans. on Computers*, 51(6):669–685, 2002.

[19] H. W. CHU, D. H. K. TSANG, AND T. YANG. Bandwidth allocation for VBR video traffic in ATM networks. In *Proc. of IEEE International Conference on Computer Communications and Networks (ICC)*, pages 612–615, 1995.

[20] D. R. COX AND P. A. W. LEWIS. *The Statistical Analysis of Series of Events.* John Wiley and Sons, New York, 1966.

[21] M. CROVELLA AND A. BESTRAVOS. Self-similarity in word wide web traffic: evidence and possible causes. *IEEE/ACM Trans. on Networking*, 5(6):835–846, 1997.

[22] P. J. DENNING AND J. P. BUZEN. The operational analysis of queueing network models. *ACM Computing Surveys*, 10(3):225–261, 1978.

[23] A. DHOLAKIA, E. ELEFTHERIOU, X. Y. HU, I. ILIADIS, J. MENON, AND K. K. RAO. Analysis of a new intra-disk redundancy scheme for high-reliability RAID storage systems in the presence of unrecoverable errors. *ACM Perf. Eval. Rev.*, 34(1):373–374, 2006.

[24] J. R. DOUCEUR AND W. J. BOLOSKY. Progress-based regulation of low-importance processes. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, pages 247-260, 1999.

[25] F. DOUGLIS, P. KRISHNAN, AND B. N. BERSHAD. Adaptive disk spin-down policies for mobile computers. In *Proc. of USENIX Symposium on Mobile and Location-Independent Computing*, pages 121–137, 1995.

[26] L. EGGERT AND J. D. TOUCH. Idletime scheduling with preemption intervals. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, pages 249-262, 2005.

[27] J. G. ELERATH AND M. PECHT. Enhanced reliability modeling of raid storage systems. In *Proc. of the International Conference on Dependable Systems and Networks (DSN)*, pages 175–184, 2007.

[28] H. FENG, Z. LIU, C. H. XIA, AND L. ZHANG. Load shedding and distributed resource control of stream processing networks. *Perf. Eval.*, 64(9-12):1102-1120, 2007.

[29] H. FENG, V. MISRA, AND D. RUBENSTEIN. PBS: a unified priority-based scheduler. In *Proc. of ACM Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, pages 203–214, 2007.

[30] H. W. FERNG AND J. F. CHANG. Connection-wise end-to-end performance analysis of queueing networks with MMPP inputs. *Perf. Eval.*, 43(1):39–62, 2001.

[31] E. J. FRIEDMAN AND S. G. HENDERSON. Fairness and efficiency in web server protocols. In *Proc. of ACM Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, pages 229–237, 2003.

[32] D. F. GARCIA AND J. GARCIA. TPC-W e-commerce benchmark evaluation. *IEEE Computer*, pages 42–48, 2003.

[33] R. GOLDING, P. BOSCH, C. STAELIN, T. SULLIVAN, AND J. WILKES. Idleness is not sloth. In *Proc. of USENIX Technical Conference*, pages 201–222, 1995.

[34] M. E. GOMEZ AND V. SANTONJA. On the impact of workload burstiness on disk performance. *Workload characterization of emerging computer applications*, pages 181-201, Kluwer Academic Publishers, 2001.

[35] D. GREEN. *Departure Processes from MAP/PH/1 Queues.* PhD thesis, Department of Applied Mathematics, University of Adelaide, 1999.

[36] S. D. GRIBBLE, G. S. MANKU, D. S. ROSELLI, E. A. BREWER, T. J. GIBSON, AND E. L. MILLER. Self-similarity in file systems. In *Proc. of ACM Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, pages 141–150, 1998.

[37] M. GROSSGLAUSER AND D. N. C. TSE. A framework for robust measurement-based admission control. *IEEE/ACM Trans. Networking*, 7(3):293–309, 1999.

[38] R. GUSELLA. Characterizing the variability of arrival processes with indexes of dispersion. *IEEE Journal on Selected Areas in Communications*, 19(2):203–211, 1991.

[39] M. HARCHOL-BALTER AND A. B. DOWNEY. Exploiting process lifetime distributions for dynamic load balancing. *ACM Trans. on Computer Systems*, 15(3):253–285, 1997.

[40] A. HEINDL. *Traffic-Based Decomposition of General Queueing Networks with Correlated Input Processes.* Ph.D. Thesis, Shaker Verlag, Aachen, 2001.

[41] D. P. HELMBOLD, D. D. E. LONG, T. L. SCONYERS, AND B. SHERROD. Adaptive disk spin-down for mobile computers. *Mobile Networks and Applications*, 5(4):285–297, 2000.

[42] D. HEYMAN AND D. LUCANTONI. Modeling multiple IP traffic streams with rate limits. *IEEE/ACM Trans. on Networking*, 11(6):948-958, 2001.

[43] A. HORVATH, G. ROZSA, AND M. TELEK. A MAP fitting method to approximate real traffic behaviour. In *Proc. of the IFIP Workshop on Performance Modelling and Evaluation of ATM & IP Networks*, pages 32/1–12, 2000.

[44] H. HUANG, W. HUNG, AND K. G. SHIN. Fs2: dynamic data replication in free disk space for improving disk performance and energy consumption. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 263–276, 2005.

[45] G. F. HUGHES AND J. F. MURRAY. Reliability and security of RAID storage systems and D2D archives using SATA disk drives. *ACM Trans. on Storage*, 1(1):95–107, 2005.

[46] D. M. JACOBSON AND J. WILKES. Disk scheduling algorithms based on rotational position. Technical Report HPL-CSP-91-7rev1, HP Laboratories, 1991.

[47] K. KANT, V. TEWARY, AND R. IYER. An internet traffic generator for server architecture evaluation. In *Proc. Workshop Computer Architecture Evaluation Using Commercial Workloads*, 2001.

[48] D. KRISHNAMURTHY, J. ROLIA, AND S. MAJUMDAR. A synthetic workload generation technique for stress testing session-based systems. *IEEE Trans. on Software Engineering*, 32(11):868–882, 2006.

[49] G. LATOUCHE AND V. RAMASWAMI. *Introduction to Matrix Analytic Methods in Stochastic Modeling.* SIAM, Philadelphia PA, 1999. ASA-SIAM Series on Statistics and Applied Probability.

[50] E. D. LAZOWSKA, J. ZAHORJAN, G. S. GRAHAM, AND K. C. SEVCIK. *Quantitative System Performance.* Prentice-Hall, 1984.

[51] H. LI AND M. MUSKULUS. Analysis and modeling of job arrivals in a production grid. *ACM Perf. Eval. Rev.*, 34(4):59–70, 2007.

[52] P. LIESHOUT, M. MANDJES, AND S. BORST. GPS scheduling: selection of optimal weights and comparison with strict priorities. *ACM Perf. Eval. Rev.*, 34(1):75–86, 2006.

[53] M. J. LITZKOW, M. LIVNY, AND M. W. MUTKA. Condor - a hunter of idle workstations. In *Proc. of International Conference of Distributed Computing Systems (ICDCS)*, pages 104–111, 1988.

[54] Z. Liu, N. Niclausse, and C.Jalpa-Villanueva. Traffic model and performance evaluation of web servers. *Perf. Eval.*, 46(2-3):77-100, 2001.

[55] Z. Liu, L. Wynter, C. H. Xia, and F. Zhang. Parameter inference of queueing models for it systems using end-to-end measurements. *Perf. Eval.*, 63(1):36–60, 2006.

[56] V. M. Lo, D. Zappala, D. Zhou, Y. Liu, and S. Zhao. Cluster computing on the fly: P2P scheduling of idle cycles in the internet. In *Proc. of the IEEE International Conference on Peer-to-Peer Systems (IPTPS)*, pages 227–236, 2004.

[57] D. M. Lucantoni. The BMAP/G/1 queue: A tutorial. *Models and Techniques for Performance Evaluation of Computer and Communication Systems*, pages 330–358. Springer-Verlag, 1993.

[58] D. Menasce, V. Almeida, R. Reidi, F. Pelegrinelli, R. Fonesca, and W. Meira Jr. In search of invariants in e-business workloads. In *Proc. of ACM Conference Electronic Commerce*, pages 56–65, 2000.

[59] D. A. Menascé and V. A. F. Almeida. *Capacity Planning for Web Performance: Metrics, Models, and Methods.* Prentice Hall, 1998.

[60] D. A. Menascé and V. A. F. Almeida. *Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning.* Prentice-Hall, Inc., 2000.

[61] D. A. Menascé, V. A. F. Almeida, and L. W. Dowdy. *Capacity planning and performance modeling: from mainframes to client-server systems.* Prentice-Hall, Inc., 1994.

[62] A. Merchant and P. S. Yu. An analytic model of reconstruction time in mirrored disks. *Perf. Eval.*, 20(1-3):115–129, 1994.

[63] N. Mi, G. Casale, L. Cherkasova, and E. Smirni. Burstiness in multi-tier applications: Symptoms, causes, and new models. In *Proc. of ACM/IFIP/USENIX International Middleware Conference (Middleware)*, 2008. The prelimilary paper appeared in the HotMetrics 2008 Workshop.

[64] N. Mi, Q. Zhang, A. Riska, E. Smirni, and E. Riedel. Performance impacts of autocorrelated flows in multi-tiered systems. *Perf. Eval.*, 64(9-12):1082–1101, 2007.

[65] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. In *Proc. of Workshop Internet Server Performance*, pages 59–67, 1998.

[66] R. R. Muntz and J. C. S. Lui. Performance analysis of disk arrays under failures. In *Proc. of International Conference on Very Large Databases (VLDB)*, pages 162–173, 1990.

[67] M. F. Neuts. *Structured Stochastic Matrices of M/G/1 Type and Their Applications.* Marcel Dekker, New York, 1989.

[68] M. F. NEUTS. *Algorithmic Probability: A Collection of Problems.* Chapman and Hall, 1995.

[69] B. F. NIELSEN. Modelling long-range dependent and heavy-tailed phenomena by matrix analytic methods. *Advances in Algorithmic Methods for Stochastic Models*, pages 265–278. Notable Publications, 2000.

[70] Z. NIU, T. SHU, AND Y. TAKAHASHI. A vacation queue with setup and close-down times and batch markovian arrival processes. *Perf. Eval.*, 54(3):225–248, 2003.

[71] T. OSOGAMI, M. HARCHOL-BALTER, AND A. SCHELLER-WOLF. Analysis of cycle stealing with switching times and thresholds. *Perf. Eval.*, 61(4):347–369, 2005.

[72] E. PINHEIRO, W.D. WEBER, AND L. A. BARROSO. Failure trends in a large disk drive population. In *Proc. of USENIX Conference on File and Storage Technologies (FAST)*, pages 17-28, 2007.

[73] S. RANJAN, J. ROLIA, H. FU, AND E. KNIGHTLY. Qos-driven server migration for internet data center. In *Proc. of International Workshop Quality of Service(IWQoS)*, pages 3–12, 2002.

[74] M. REISER. Mean-value analysis and convolution method for queue-dependent servers in closed queueing networks. *Perf. Eval.*, 1:7–18, 1981.

[75] M. REISER AND S. S. LAVENBERG. Mean-value analysis of closed multichain queueing networks. *Journal of the ACM*, 27(2):312–322, 1980.

[76] A. RISKA AND E. RIEDEL. Analysis of disk-level traces. Technical Report SEA-ARCH-2004-02, Seagate Research, 2004.

[77] A. RISKA AND E. RIEDEL. Disk drive level workload characterization. In *Proc. of the USENIX Annual Technical Conference*, pages 97–103, 2006.

[78] A. RISKA AND E. RIEDEL. Long-range dependence at the disk drive level. In *Proc. of Conference on Quantitative Evaluation of Systems (QEST)*, pages 41–50, 2006.

[79] A. RISKA AND E. RIEDEL. Idle Read After Write - IRAW. In *Proc. of the USENIX Annual Technical Conference*, pages 43–56, 2008.

[80] T. G. ROBERTAZZI. *Computer Networks and Systems.* Springer, 2000.

[81] J. ROLIA AND V. VETLAND. Correlating resource demand information with arm data for application services. In *Proc. of the International Workshop on Software and Performance (WOSP)*, pages 219–230, 1998.

[82] L. SCHRAGE. A proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 16:687–690, 1968.

[83] B. SCHROEDER AND G. A. GIBSON. Understanding disk failure rates: What does an mttf of 1,000,000 hours mean to you? *ACM Trans. Storage*, 3(3):8, 2007.

[84] T. J. E. SCHWARZ, Q. XIN, E. L. MILLER, D. D. E. LONG, A. HOSPODOR, AND S. NG. Disk scrubbing in large archival storage systems. In *Proc. of the International Symposium on Modeling and Simulation of Computer and Communications Systems (MASCOTS)*, pages 409-418, 2004.

[85] M. SELTZER, P. CHEN, AND J. OSTERHOUT. Disk scheduling revisited. In *Proc. of the USENIXTechnical Conference*, pages 313–323, 1990.

[86] S. SHAH AND J. G. ELERATH. Reliability analysis of disk drive failure mechanism. In *Proc. of Annual Reliability and Maintainability Symposium*, pages 226–231, 2005.

[87] M. SIVATHANU, V. PRABHAKARAN, A. C. ARPACI-DUSSEAU, AND R. H. ARPACI-DUSSEAU. Improving Storage System Availability with D-GRAID. In *Proc. of the USENIX Symposium on File and Storage Technologies (FAST)*, pages 15–30, 2004.

[88] H. TAKAGI. *Queuing Analysis Volume 1: Vacations and Priority Systems.* North-Holland, New York, 1991.

[89] M. M. THEIMER, K. A. LANTZ, AND D. R. CHERITON. Preemptable remote execution facilities for the v-system. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, pages 2–12, 1985.

[90] E. THERESKA, J. SCHINDLER, J. BUCY, B. SALMON, C. R. LUMB, AND G. R. GANGER. A framework for building unobtrusive disk maintenance applications. In *Proc. of the USENIX Symposium on File and Storage Technologies (FAST)*, pages 213–226, 2004.

[91] A. THOMASIAN AND V. F. NICOLA. Performance evaluation of a threshold policy for scheduling readers and writers. *IEEE Trans. on Computers*, 42(1):83–98, 1993.

[92] B. URGAONKAR, G. PACIFICI, P.J. SHENOY, M. SPREITZER, AND A. TANTAWI. An analytical model for multi-tier internet services and its applications. In *Proc. of ACM Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, pages 291–302, 2005.

[93] B. URGAONKAR, P. SHENOY, A. CHANDRA, AND P. GOYAL. Dynamic provisioning of multi-tier internet applications. In *Proc. of the International Conference on Automatic Computing(ICAC)*, pages 217–228, 2005.

[94] D. VILLELA, P. PRADHAN, AND D. RUBENSTEIN. Provisioning servers in the application tier for e-commerce systems. *ACM Trans. Interet Technology*, 7(1):7, 2007.

[95] J. L. WANG AND A. ERRAMILLI. A connection admission control algorithm for self-similar traffic. In *Proc. of IEEE Global Telecommunications Conference (GLOBECOM)*, pages 1623–1628, 1999.

[96] B. P. WELFORD. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4:419–420, 1962.

[97] M. Welsh, D. E. Culler, and E. A. Brewer. An architecture for well-conditioned, scalable internet services. In *Proc. of the ACM symposium on Operating systems principles (SOSP)*, pages 230–243, 2001.

[98] www.mercury.com/us/products/diagnostics. *Mercury Diagnostics.*

[99] E. Xu and A. S. Alfa. A vacation model for the non-saturated readers and writers system with a threshold policy. *Perf. Eval.*, 50(4):233–244, 2002.

[100] J. Zhang, M. Hu, and N. Shroff. Bursty data over cdma: Mai selfsimilarity, rate control, and admission control. In *Proc. of Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2002.

[101] Q. Zhang, L. Cherkasova, G. Mathews, W. Greene, and E. Smirni. R-capriccio: A capacity planning and anomaly detection tool for enterprise services with live workloads. In *Proc. of ACM/IFIP/USENIX International Middleware Conference (Middleware)*, pages 244–265, 2007.

[102] Q. Zhang, L. Cherkasova, and E. Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *Proc. of the International Conference on Automatic Computing(ICAC)*, pages 27, 2007.

VITA

Ningfang Mi

Ningfang Mi received her B.S. degree in Computer Science from Nanjing University, China, in 2000, and her M.S. degree in Computer Science from the University of Texas at Dallas, in 2004. She has been a Ph.D. candidate of Computer Science at the College of William and Mary since 2005. Her research interests include storage systems, multi-tiered systems, performance evaluation, energy/power management, web characterization, data analysis, system modeling, and scheduling/load balancing.