Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 80 (2007) 493–509
The Journal of Systems and Software
www.elsevier.com/locate/jss

Quantifying software performance, reliability and security: An architecture-based approach

Vibhu Saujanya Sharma [a,*], Kishor S. Trivedi [b]

[a] Department of Computer Science and Engineering, Indian Institute of Technology Kanpur, Kanpur, UP 208016, India
[b] Department of Electrical and Computer Engineering, Duke University, Durham, NC 27708, USA

Available online 30 August 2006

Abstract

With component-based systems becoming popular and handling diverse and critical applications, the need for their thorough evaluation has become very important. In this paper we propose an architecture-based unified hierarchical model for software performance, reliability, security and cache behavior prediction. We employ discrete time Markov chains (DTMCs) to model software systems and provide expressions for predicting the overall behavior of the system based on its architecture as well as the characteristics of individual components. This approach also facilitates the identification of various bottlenecks. We illustrate its use through some case studies and also provide expressions to perform sensitivity analysis.

Keywords: Software architecture; Performance; Reliability; Security; Markov models

1. Introduction

Complex software systems are working behind the scenes in almost every aspect of life today, and much more depends on the reliability, performance and security of these systems than ever before. Unreliable software systems could cause anything from inconvenience (like an online banking system failure) to even fatal accidents (like the failure of a critical software system during a space launch). Software applications that do not perform well could cause inefficiency and major delays in day-to-day operations causing wastage of time and money. On the other hand, insecure software systems could allow an intruder to take control over them, which depending upon the kind of application, could range from bank frauds to terrorist acts.

Adequate analysis of a software system for reliability, performance and security could help ascertain its shortcomings. Apart from preventing big mishaps, it may also be helpful to establish the behavior of the software system with respect to the hardware it is deployed on, and predictions related to specific aspects, such as cache-miss behavior of the same could be made.

With the component-based design becoming popular and many software systems being built using off the shelf components, systems can no longer be treated as monolithic entities for the purpose of the aforementioned analysis and evaluation. The architecture of the software system determines how the different components interact and is a major factor contributing to the way the system behaves and performs. The prevalent black box models clearly are inept to model such component-based software systems.

In recent past there has been some work on architecture-based analysis of systems. Some of these approaches analyze the system for reliability (Gokhale and Trivedi, 2002; Goseva-Popstojanova and Trivedi, 2001; Cheung, 1980) while some for performance (Smith et al., 2005; Sharma et al., 2005; Petriu et al., 2000). These approaches employ a number of techniques for representing the software system ranging from Markov chains, execution graphs, to UML, etc. Cache-miss behavior of a system is also an

* Corresponding author. Tel.: +91 512 2597579; fax: +91 512 2597586.
E-mail addresses: vsharma@cse.iitk.ac.in, vibhusharma@gmail.com (V.S. Sharma), kst@ee.duke.edu (K.S. Trivedi).

0164-1212/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2006.07.021

important factor affecting the performance of the system and some approaches for analyzing cache-miss behavior of software systems have been proposed (Clark and Emer, 1985; Stone et al., 1992; Rao, 1978). However these approaches are not specifically concerned with architecture-based analysis of the system. Some approaches have also been proposed for security evaluation of a system, but these approaches again do not take into account the architecture of the system in general and are not reproducible many times. Only a few approaches like AVA (Voas et al., 1996) yield reproducible results, but they again tend to treat the software monolithically.

In general, known quantitative methods usually concentrate on a particular attribute only, and a lot of effort has to be put in to employ different modeling and analysis approaches for a multi-attribute analysis of the same system. Moreover, most of the existing approaches for security and cache-miss analysis are not suitable for an architecture level analysis.

In this paper, we provide a hierarchical approach that unifies the analysis of component-based systems for performance, reliability, cache-miss behavior and security. The initial step in the hierarchical approach is the modeling of the software architecture using *Discrete Time Markov Chains* or DTMCs (Trivedi, 2001). The DTMC model is then analyzed to get the important information for the subsequent attribute specific analysis. The unique ability of the approach to allow for quantitative analysis of different attributes, make it very suitable for comparing various software architectures as well as component types, to achieve the desired overall behavior along each of the attributes.

Our approach is useful both for analysis at the time of system design as well as for the evaluation of existing systems. The approach could be well accommodated in the design phase of software development, as changes in the software architecture of the system do not cause the whole model to collapse. Moreover changes in individual component behavior can also be accommodated without any change to other model parameters, making it suitable for use in the later phases of development when the testing and tuning of the components is going on. We also provide ways in which the architecture of the system could be extracted to construct the model.

This approach incorporates the software architecture of the system as an important input to the prediction of its security and cache performance behavior, which has been overlooked in most of the related studies. We try to take into account the second-order architectural effects in our approach, wherever possible, for increasing the accuracy. Results show that the predictions are pretty close to the observed behavior. We also provide the approach to measure the impact of variation in the behavior of individual components as well as workload on the overall behavior of the system by performing sensitivity analysis.

The rest of the paper is divided as follows: Section 2 provides a brief overview of related work and a background on DTMCs, Section 3 introduces the hierarchical model and provides expressions to predict the reliability, cache-miss behavior, and time spent in the system, apart from defining vulnerability index and providing the expressions for the same as well. Section 4 provides insights into ways to extract the architecture of the system. Section 5 provides the illustrations and case studies wherein the approach has been applied and presents the obtained results. Section 6 addresses the issue of performance evaluation in presence of multiple users. Section 7 discusses techniques to perform sensitivity analysis of the system under consideration with respect to the model parameters. Section 8 is concerned with the modifications required to incorporate dependence among the components and Section 9 concludes the paper.

2. Related work and background

We present the related work in quantifying various software attributes and present a brief overview of DTMCs, which we use to model the control flow in a component-based system.

2.1. Related work

Architecture-based analysis aims to take into account the behavior of the components and the architecture that constitute the application. In recent past there have been some studies (Gokhale and Trivedi, 2002; Goseva-Popstojanova and Trivedi, 2001; Gokhale et al., 1998; Goseva-Popstojanova et al., 2001) that focus both on reliability and performance of software applications taking into account their software architecture. Different approaches to software reliability assessment have been classified into three categories: state-based, path-based and additive in Goseva-Popstojanova and Trivedi (2001). State-based models use the control flow graph of the software to represent the architecture of the system, which could be modeled as *Discrete Time Markov Chains* or DTMCs (Gokhale and Trivedi, 2002; Goseva-Popstojanova and Trivedi, 2001; Cheung, 1980; Gokhale et al., 1998; Goseva-Popstojanova et al., 2001; Reussner et al., 2003), *Continuous Time Markov Chains* or CTMCs (Laprie, 1984; Ledoux, 1999) or *Semi-Markov Processes* or SMPs (Kubat, 1989; Littlewood, 1975). In path-based models (Yacoub et al., 1999; Shooman, 1976) the system reliability is computed considering the possible execution paths of the program. Additive models (Everett, 1999; Xie and Wohlin, 1995) do not consider the architecture of the software explicitly and estimate the system failure intensity as the sum of component failure intensities under the assumption that individual component reliabilities can be modeled by a non-homogenous Poisson process (NHPP).

State-based models could be further classified into two categories: composite and hierarchical. Composite models combine the software architecture and the failure behavior of the software in the same model, while hierarchical models incorporate separate modeling of the software

architecture, which is then solved and the solution is superimposed with the failure behavior of the components in order to predict reliability. Changes in software architecture or individual component behavior are comparatively difficult to accommodate in composite models. Moreover composite models are prone to the problem of stiffness arising out of the huge relative difference in the magnitudes of the failure and transition probabilities of components (the former being very small usually compared to the latter).

Hierarchical models provide much more flexibility than composite models as software architecture is modeled separately. Different architectural alternatives could be evaluated by changing the architectural model only, thus involving minimal overhead. The problem of stiffness (Reibman and Trivedi, 1988; Bobbio and Trivedi, 1986) is also reduced to a large extent, as the software architectural model is solved separately first and the failure behavior is superimposed on this solution for finding the overall reliability. However the hierarchical model only provides an approximation to the composite models and hence accurate hierarchical modeling is an important topic. The first step towards the same is including the second-order architectural effects in modeling and has been presented in Gokhale and Trivedi (2002).

C.U. Smith first coined the term Software Performance Engineering (SPE) in her pioneering work (Smith, 1990) as a methodological approach for performance evaluation of software systems. The tool that implements this approach is called SPE·ED. The tool can be used to specify the software execution model using Execution Graphs (EGs), which are like annotated flowcharts. The system execution model, which models contention, is realized by using simulation. Recently an effort for importing UML models to SPE·ED has been proposed (Smith et al., 2005). Petriu and Woodside use a tool called UCM2LQN, to convert Use case Map based representation of software systems into LQN model (Petriu et al., 2000) which can then be solved using their LQN solver for performance evaluation (Petriu and Woodside, 2002).

An important aspect of system performance is its cache behavior and it is dependent both on the software as well as hardware architecture of the system. Three methods have been commonly used for cache performance evaluation: real-time execution, trace driven simulation and modeling. The real-time execution approaches (Clark and Emer, 1985) have the drawback of being restricted to existing hardware only. Trace driven simulation approaches (Wang and Baer, 1990) tend to be very space and time consuming (Li, 2000). Moreover both these approaches lack insights on how software architecture impacts the cache performance. Empirical models such as (Stone et al., 1992; Chow, 1976; Thiebaut, 1987) try to parameterize cache performance in terms of capacity, working set size, spatial and temporal locality and the interaction between the two. These models have the nice feature of yielding fast and accurate results for certain cache configurations for a given program. But they apply only to certain cache configurations and workloads. Analytical and hybrid models such as (Rao, 1978; Horowitz et al., 1989; Martonosi et al., 1997) do provide insights into the nature of programs and the factors affecting cache performance. But they have their own disadvantages such as large number of parameters (Rao, 1978; Horowitz et al., 1989) and application to only designated program structures (Martonosi et al., 1997). Reader is referred to (Li, 2000) for detailed study of these models.

Software security is another field where considerable research is taking place. There have been some attempts to quantify the security of software system by means of Tiger Team Penetration practices, where a group of experts, sit together and try to break in, by exploiting any weaknesses it might possess. However this practice is very subjective to the kind of people consisting the Tiger Team, and thus is non-reproducible in nature. There have been some approaches, which focus on the process, which is adopted while the software is being developed, to assess the security of the final product. One example of this is the SSECMM or the Systems Security Engineering Capability Maturity Model. However, branding the software to be secure by evaluating its development process has not found much popularity. This is because even after following the best practices, there is scope of some weakness present in the final product, which would not be uncovered, until it is rigorously tested for its vulnerabilities.

Cigital labs (formerly Reliable Software Technologies) have proposed a product evaluation approach called the AVA or the Adaptive Vulnerability Analysis (Voas et al., 1996). The basic emphasis is on observing the impact of incoming simulated infections (which model threats in AVA) upon an executing system. Hazardous output conditions are specified through assertions upon the program variable states. However this approach treats the software monolithically. The Software Engineering Institute (SEI) at CMU also has been very active in this field and they have proposed V-RATE or Vendor Risk Assessment and Threat Evaluation (Lipson et al., 2001) which could be used while deciding which vendor to choose for buying a certain COTS component.

In a gist, in much of the existing work, different attributes of software systems such as reliability, performance and security have been treated separately and many a times without incorporating the software architecture in the approach.

2.2. DTMCs – a brief background

In this section, we discuss Discrete Time Markov Chains (DTMCs), which we use to model the software architecture of a system. A Markov process is a stochastic process whose dynamic behavior is such that probability distributions for its future development depend only on the present state and not on how the process arrived in that state. If we assume that the state space, I, is discrete (finite or countably infinite), and the parameter space, T, is also discrete,

then we have a DTMC. A DTMC is characterized by its states and transition probabilities among the states. The one-step transition probability matrix $\mathbf{P} = [p_{i,j}]$ is a stochastic matrix as all the elements in a row of $\mathbf{P}$ add up to 1 and each of the $p_{i,j}$'s lie in the range $[0,1]$.

For our purposes, we classify DTMCs in the following two categories:

- Irreducible: If every state can be reached from every other state in a finite number of steps.
- Absorbing: If at least one state has no outgoing transition.

We can partition the transition probability matrix of an absorbing DTMC as

$$\mathbf{P} = \begin{bmatrix} \mathbf{Q} & \mathbf{C} \\ \mathbf{0} & \mathbf{1} \end{bmatrix}$$

If the DTMC has n states with m absorbing states, $\mathbf{Q}$ would be a $(n-m)$ by $(n-m)$ sub-stochastic matrix (with at least one row sum <1) describing the probabilities of transition only between transient states, $\mathbf{1}$ being an m by m identity matrix, $\mathbf{0}$ would be an m by $(n-m)$ matrix of zeros, and $\mathbf{C}$ would be an $(n-m)$ by m matrix. The k-step transition probability matrix, given by $\mathbf{P^k}$ has the form

$$\mathbf{P^k} = \begin{bmatrix} \mathbf{Q^k} & \mathbf{C'} \\ \mathbf{0} & \mathbf{1} \end{bmatrix}$$

The (i,j)th entry of $\mathbf{Q^k}$ denotes the probability of arriving in state s_j after exactly k steps, starting from state s_i. It can be shown that $\sum_{k=0}^{t} Q^k$ converges as t approaches infinity. Hence the inverse matrix $(\mathbf{I}-\mathbf{Q})^{-1}$ exists. This is called the fundamental matrix $\mathbf{M}$

$$\mathbf{M} = (\mathbf{I}-\mathbf{Q})^{-1} = \mathbf{I} + \mathbf{Q} + \mathbf{Q^2} + \cdots = \sum_{\mathbf{k=0}}^{\infty} \mathbf{Q^k}$$

Let $X_{i,j}$ represent the number of visits to state j starting from state i before the process is absorbed. It can be shown that the expected number of visits to a state j with the system starting from state i, before entering an absorbing state is given by the (i,j)th entry of the fundamental matrix $\mathbf{M}$ (Bhat, 1984; Trivedi, 2001). So

$$E[X_{i,j}] = m_{i,j}$$

The variance of the expected number of visits could also be computed using the fundamental matrix. Let $\sigma_{i,j}$ denote the variance of the number of visits to the state j starting from state i. Define $\mathbf{M_D} = [md_{i,j}]$ such that

$$md_{i,j} = \begin{cases} m_{i,j} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

In other words, $\mathbf{M_D}$ represents a diagonal matrix with the diagonal entries same as that of $\mathbf{M}$. If we define $\mathbf{M_2} = [m^2_{i,j}]$, we have

$$\boldsymbol{\sigma^2} = \mathbf{M}(\mathbf{2M_D} - \mathbf{I}) - \mathbf{M_2}$$

Hence

$$\mathrm{Var}[X_{i,j}] = \sigma^2_{i,j}$$

For an application consisting of a number of software components, we can represent its software architecture using a DTMC. The state of the DTMC at an execution step is given by the component in execution in that step. Moreover, transitions between states represent transfer of control from one component to another.

3. The hierarchical model and its application

The software architecture of a terminating application could be represented as an absorbing DTMC as mentioned in Section 2. In this section, we present the approach for utilizing the DTMC representation in a hierarchical model for predicting the reliability, performance, security and cache behavior of the software. We assume that the time spent by the application in each component per visit is a random variable with known mean and variance. We also assume that the reliability, the vulnerability index and the cache-miss ratios of each component are deterministic and could be ascertained.

We assume that the application has n components with the initial component indexed 1 which receives the control flow first, and a final component n, after which the program terminates. These two components are denoted respectively by the initial and the absorbing or the completion state of the DTMC. In general, the transition from a state i to j in this DTMC represents the transfer of control from component or module i to j in the software or program. The control flow through the components of the application is given by the one-step transition probability matrix $\mathbf{P}$. We can therefore find the expected visit counts to various components as well as the variance of the visit counts as explained in Section 2.2. The architecture of an existing software system can be extracted by using various profiling tools like *gprof* (GNU gprof, 1998), ATOM (Srivastava and Eustace, 1994), ATAC (Horgan and London, 1992), etc. We elaborate upon this issue in Section 4. During the development of the system, in the architectural phase, the individual properties of the components such as reliability, performance attributes etc., need to be estimated from experience with similar previous systems. The parameters can be however be computed for existing systems by testing and measurement of component attributes (Gokhale and Trivedi, 2002).

Fig. 1 shows the architecture-based hierarchical analysis approach followed in the subsequent subsections. One needs to model the software architecture of the application as a DTMC only once and then by assigning suitable rewards, the performance, reliability, security, and cache performance analysis of the application can be carried out.

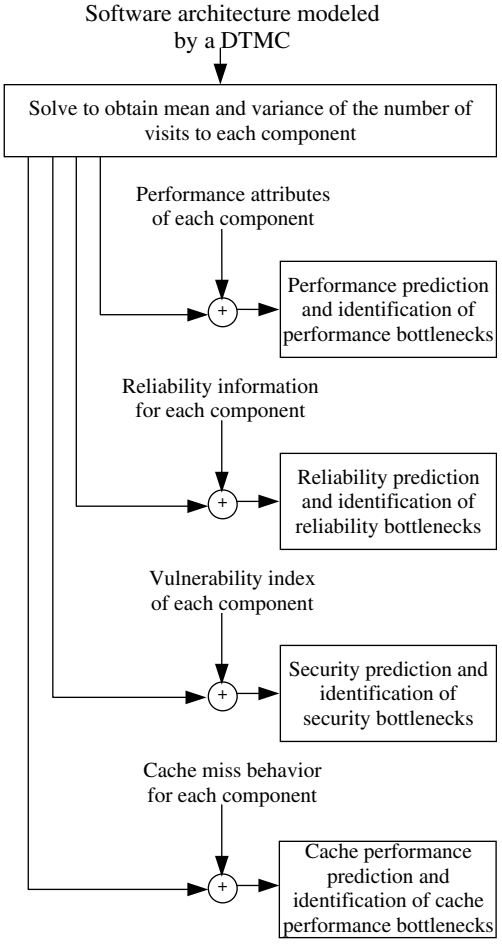

Fig. 1. The hierarchical modeling approach.

3.1. Reliability prediction

The reliability of a software application composed of a number of software components could be calculated by utilizing the mean and variance of the number of visits obtained from DTMC analysis and combining these with the reliabilities of individual components (Gokhale and Trivedi, 2002). Consider that the state i represents the ith component in execution. Assuming that the components fail independently of each other as well as in successive executions, if R_i denotes the reliability of the component i, we can denote the reliability of the application as

$$R = \prod_{i}^{n} R_i^{X_{1,i}}$$

Note that as the number of visits to each component is a random variable (except for component n), R itself is a random variable. The expected reliability of the application is given by

$$E[R] = E\left[\prod_{i}^{n} R_i^{X_{1,i}}\right] = \prod_{i}^{n} E[R_i^{X_{1,i}}]$$

The term $E[R_i^{X_{1,i}}]$ in the product on the right is the expected reliability of component i for a single execution. Using the Taylor series expansion we have

$$E[R_i^{X_{1,i}}] = R_i^{E[X_{1,i}]} + \frac{1}{2}(R_i^{E[X_{1,i}]})(\log R_i)^2 \mathrm{Var}[X_{1,i}]$$

which could be written as

$$E[R_i^{X_{1,i}}] = R_i^{m_{1,i}} + \frac{1}{2}(R_i^{m_{1,i}})(\log R_i)^2 \sigma_{1,i}^2$$

Note that the number of visits to the nth component (the absorbing state in the DTMC) is always 1, so $E[X_{1,n}] = 1$, and $\mathrm{Var}[X_{1,n}] = 0$. So

$$E[R_n^{X_{1,n}}] = R_n$$

Hence the overall expected reliability of the application taking into account the impact of second-order architectural effects is given by (Gokhale and Trivedi, 2002)

$$E[R] = \left[\prod_{i}^{n-1}\left(R_i^{m_{1,i}} + \frac{1}{2}(R_i^{m_{1,i}})(\log R_i)^2 \sigma_{1,i}^2\right)\right] R_n$$

The second-order architectural effects are captured by the variance of the number of visits to a component and provide for more accurate predictions, the only source of approximation in the model being the Taylor series cut-off. However if the second-order architectural effects are ignored, we get the expected reliability of the application as

$$E[R] \approx \left[\prod_{i}^{n-1} R_i^{m_{1,i}}\right] R_n$$

We mark the component with the lowest value of $E[R_n^{X_{1,n}}]$ as the reliability bottleneck of the software application.

There are many studies in recent past that assume independent component failures for overall reliability estimation (Goseva-Popstojanova et al., 2001; Yacoub et al., 2004). However, this assumption of independent component behavior can be relaxed. We discuss this in Section 8.

3.2. Performance prediction

Consider that we represent the time spent executing the ith component for an application by a random variable T_i. If the DTMC state i, represents the control flow residing in component i, the mean time spent in state i is $E[T_i] = \tau_i$ and its variance is θ_i^2. We thus use the number of visits to state i starting from state 1 (represented by $X_{1,i}$) to find the mean time to completion of the application (Gokhale and Trivedi, 2002). The time to completion of the application for a single run denoted by random variable T, is given by

$$T = \sum_{i=1}^{n} T_i X_{1,i}$$

Note that here we are assigning T_i as a *reward* (Trivedi, 2001) to every state i, and we are interested in computing the expected accumulated reward till reaching the absorbing state, signifying the completion of the execution of

the application. This expected reward denotes the expected time to completion of the application and can be expressed as

$$E[T] = E\left[\sum_{i=1}^{n} T_i X_{1,i}\right] = \sum_{i=1}^{n} E[T_i X_{1,i}]$$

As $T_i X_{1,i}$ is a function of random variables T_i and $X_{1,i}$, so $T_i X_{1,i}$ is a random sum (Gokhale and Trivedi, 2002; Trivedi, 2001). Hence using the expressions for random sums

$$E[T_i X_{1,i}] = E[T_i]E[X_{1,i}] = \tau_i m_{1,i}$$

where, $E[T_n X_{1,n}] = \tau_n$. Hence we get the expected time to completion of the application as

$$E[T] = \sum_{i=1}^{n-1} \tau_i m_{1,i} + \tau_n$$

We can denote the variance of the random variable $T_i X_{1,i}$ as (Gokhale and Trivedi, 2002)

$$\begin{aligned}\mathrm{Var}[T_i X_{1,i}] &= \mathrm{Var}[T_i]E[X_{1,i}] + (E[T_i])^2 \mathrm{Var}[X_{1,i}] \\ &= \theta_i^2 m_{1,i} + \tau_i^2 \sigma_{1,i}^2\end{aligned}$$

Hence the variance of the time to completion of the application, assuming independence among the time spent in each of the modules is given by

$$\mathrm{Var}[T] = \sum_{i}^{n} (\theta_i^2 m_{1,i} + \tau_i^2 \sigma_{1,i}^2)$$

We mark the component with the largest value of $\tau_i m_{1,i}$ as the performance bottleneck of the software application.

The concept of expected execution time for a component has been well-known (Trivedi, 2001; Smith, 1990; Knuth, 1997) in literature. One should note that rather than a combined measure like execution time, many performance attributes such as CPU time, Disk I/O time, etc., can be specified for a component. Also note that the above discussion of performance prediction assumes that the application is handling only one request, i.e. there is no contention at any machine hosting the system. In such cases, a detailed underlying performance model (Trivedi, 2001; Smith, 1990; Petriu et al., 2000) will be required to find out the expected time to service completion of a request. We present a queueing modeling based approach for performance evaluation in Section 6.

3.3. Security prediction

The need for quantifying security of software systems is gaining importance with many mission critical applications solely relying on them. Moreover the use of COTS components as building blocks in software architecture mandates thorough understanding of the security implications of their use.

Vulnerability is a potential defect or weakness in a system, together with the knowledge required to exploit the defect. A *security attack* is an active attempt to exploit the vulnerabilities in the system and it could pose a *threat* to the system.

We define *Vulnerability Index* (VI) as the probability of a component's vulnerability being exposed in a single execution. The VI for a component may be measured by subjecting the component to attacks for various possible vulnerabilities in the system and finding out the ratio of successful attacks to the total number of attempts. Ascertaining whether an attack was successful or not could be done by examining the state of the variables of the software component using appropriate interfaces in case of COTS components. Reaching one of the states that signifies a security breach implies a successful attack. Existing techniques such as the AVA (Voas et al., 1996) could also be used to determine the vulnerability index of a software component, wherein the basic emphasis is on observing the impact of incoming simulated infections (which model threats) upon an executing program and giving a vulnerability assessment of the same. Note however that the source code of the component needs to be known for applying AVA.

Consider a system consisting of n components some of which are more vulnerable to attacks (have higher VI), while others are less vulnerable. It is fairly non-intuitive to comment on the security of this system, as all the components do not contribute equally to the security (or insecurity) of the system. Moreover same component might influence the security of different systems, differently depending on their software architecture. We define Effective Vulnerability Index (EVI) as a relative measure of the impact of a component on the system's insecurity. We propose to assign the component's vulnerability index as the *reward* for the corresponding state in the DTMC. Let VI_i denote the vulnerability index of the component i. Thus for component i we have

$$\mathrm{EVI}_i = m_{1,i} * \mathrm{VI}_i$$

Note that this means that the impact of a component's vulnerability on the software's insecurity is directly proportional to the number of visits to it during the actual execution, which is fairly intuitive. Note that EVI is no longer a probability and is a relative measure. We are assuming that the vulnerability of one component does not affect that of any other. Reducing the EVIs of the components in the system would strengthen the system in general. If any effort or investment is to be made to secure the system, then the component having the highest EVI value should be targeted first. Thus this technique maximizes the security return for an effort aimed at making the system more secure. Moreover the EVI for individual components could also be used in case of dynamic monitoring of the executing system. An alert could be issued, whenever an execution path traverses through components having high EVI values, so that the system could then be closely monitored for this highly vulnerable execution.

We can use the individual VIs and the software architecture to find out the vulnerability index, VI, for the whole

system. The vulnerabilities present in the system, would be considered exposed if at least one component is broken into (has its vulnerability exposed), hence we have

$$\mathrm{VI} = 1 - \prod_i^n (1 - \mathrm{VI}_i)^{X_{1,i}}$$

where, VI is the random variable denoting the vulnerability index of the whole software system and $X_{1,i}$ denotes the random variable corresponding to the number of visits to a transient state i starting from state 1. The expected vulnerability of the system is given by

$$E[\mathrm{VI}] = E\left[1 - \prod_i^n (1 - \mathrm{VI}_i)^{X_{1,i}}\right] = 1 - E\left[\prod_i^n (1 - \mathrm{VI}_i)^{X_{1,i}}\right]$$
$$= 1 - \prod_i^n E[(1 - \mathrm{VI}_i)^{X_{1,i}}]$$

Using similar technique as in case of reliability prediction we have

$$E[(1 - \mathrm{VI}_i)^{X_{1,i}}] = (1 - \mathrm{VI}_i)^{m_{1,i}} + \frac{1}{2}((1 - \mathrm{VI}_i)^{m_{1,i}}) \times (\log(1 - \mathrm{VI}_i))^2 \sigma_{1,i}^2$$

and as the total number of visits to the last component n is always 1, so $E[X_{1,n}] = 1$, and $\mathrm{Var}[X_{1,n}] = 0$. So

$$E[(1 - \mathrm{VI}_n)^{X_{1,n}}] = 1 - \mathrm{VI}_n$$

Hence the expected vulnerability index of the software application, taking into account the second-order architectural effects is given by

$$E[\mathrm{VI}] = 1 - \left[\prod_i^{n-1} \left[(1 - \mathrm{VI}_i)^{m_{1,i}} + \frac{1}{2}((1 - \mathrm{VI}_i)^{m_{1,i}}) \times (\log(1 - \mathrm{VI}_i))^2 \sigma_{1,i}^2\right]\right](1 - \mathrm{VI}_n)$$

Neglecting the second-order architectural effects we have

$$E[\mathrm{VI}] \approx 1 - \left[\prod_i^{n-1} (1 - \mathrm{VI}_i)^{m_{1,i}}\right](1 - \mathrm{VI}_n)$$

Note that the probability that the component operates without a security failure during one execution of the software system is given by $E[(1 - \mathrm{VI}_i)^{X_{1,i}}]$. Hence we can also find out the probability of component i's vulnerabilities being exposed during the execution of the software system as

$$1 - E[(1 - \mathrm{VI}_i)^{X_{1,i}}] = 1 - (1 - \mathrm{VI}_i)^{m_{1,i}} + \frac{1}{2}((1 - \mathrm{VI}_i)^{m_{1,i}}) \times (\log(1 - \mathrm{VI}_i))^2 \sigma_{1,i}^2$$

An interesting observation is that if we make an approximation and ignore the variance of the number of visits, on expanding the term $(1 - \mathrm{VI}_i)^{m_{1,i}}$ using Taylor series we get

$$1 - E[(1 - \mathrm{VI}_i)^{X_{1,i}}] \approx 1 - \left[1 + \sum_k^\infty \binom{m_{1,i}}{k} (-1)^k \mathrm{VI}_i^k\right]$$

If we neglect the higher order terms, we get

$$1 - E[(1 - \mathrm{VI}_i)^{X_{1,i}}] \approx 1 - [1 - m_{1,i}\mathrm{VI}_i] = m_{1,i}\mathrm{VI}_i$$

This is the same expression that we defined to be the Effective Vulnerability Index of a component in the software. This shows that the intuitive concept of Effective Vulnerability Index also has an analytical basis. The security bottleneck of the software system, is the component having the highest value of $1 - E[(1 - \mathrm{VI}_i)^{X_{1,i}}]$ or simply the highest EVI if variance in the visit counts is ignored and higher order terms are not included (which might typically be the case if VIs are very small).

3.4. Cache performance analysis

Cache behavior and its performance largely depends on the data layout and the program structure apart from the hardware architecture of the system. The study of the execution behavior of the program is very important for cache performance analysis and such studies often divide an application into several phases to signify the changes in execution. Indeed, the locality rule of program references also suggests that the reference behavior of a program displays a clustered nature around different regions (Shedler and Tung, 1976). In our model we utilize the software architecture of the application to define program execution phase differentiation. We use the hierarchical model to provide an accurate means to predict cache performance of an application with respect to its execution behavior. Each phase of program execution corresponds to the control flow residing in a component or a module in the software architecture of the application, and we provide a formula to find overall cache-miss ratio in terms of the cache-miss ratios of the individual modules. This work was done as part of the Masters thesis by Li (2000) at Duke University.

Let there be n components or modules in the application, then we can model the software architecture of the application as a DTMC with n states and associated transition probabilities. From this we can also find $m_{1,i}$, the expected number of visits to component i starting from component 1, in an execution, as explained in Section 2. Let n_i be the number of memory references during execution, per visit to module i. One can compute this as

$$n_i = \frac{\text{Number of total memory references in a module}}{\text{Number of actual visits to that module } (V_i')}$$

Note that V_i' is different from $m_{1,i}$, the former being the number of visits to a module directly measured from one sample execution, while the latter is calculated from the DTMC as mentioned in Section 2. Let M_i be the miss ratio for individual module i, measured from a sample execution of the application and is defined as the total cache misses that occur in a module divided by the total number of

memory references in that module. The overall miss ratio is then computed as

$$M = \frac{\sum_i^n M_i m_{1,i} n_i}{\sum_i^n m_{1,i} n_i}$$

This miss ratio function gives a prediction for cache misses which incorporates both factors from software architecture and hardware configuration. The visit counts mainly involve the software architecture while different hardware configuration issues are addressed by the miss ratio of individual modules. The component i, having the highest value of $M_i m_{1,i} n_i$ is the component causing the most cache misses, and thus is the cache performance bottleneck of the system. We have not as yet computed the expression for the variance of the overall miss ratio in terms of the mean and variance of the visit counts, and leave it as future work. Also note that this approach is only an approximation as we are assuming here that the different modules or components are totally independent and there is no data sharing between them. However in real world systems programs exhibit data locality and many times modules do share the some data. We tackle this issue in the case study in Section 5.

4. Extracting the software architecture

Knowledge about the software architecture of the system is essential for applying the proposed hierarchical analysis approach. The individual components need to be identified and their interaction with other components needs to be taken into account for constructing the DTMC model corresponding to the system.

While applying the approach to software systems under construction, the software architecture is mostly available in some standard form (like UML diagrams) and individual components are identifiable. The information regarding interaction among components can be estimated from previous experience with similar software components. A method to estimate the control flow transition probabilities among components from the occurrence probabilities of various execution scenarios based on the operational profile of the system has been shown in Yacoub et al. (2004). Quantitative information regarding the individual characteristics of the components, i.e., reliability, performance demands, etc., also is not available in entirety and thus estimates of these values are used. However as the software development continues, these estimates become better and the analysis thus improves with time in terms of accuracy.

For existing systems however, the main concern is to identify the software architecture and different components that constitute the system. Tools like gprof, ATOM, SWAG Kit etc. can be used for extracting the software architecture of the system. The GNU profiler or gprof (GNU gprof, 1998) is very useful to derive the flat profile and the call graph of C and C++ programs. The flat profile shows how much time a program spent in each function, and how many times that function was called. The call graph provides the information regarding the other functions calling a particular function and the functions called by it, along with the number of calls. This information is helpful to construct the DTMC model and also get the relevant transition probabilities.

ATOM (Srivastava and Eustace, 1994) is a toolkit that provides a set of instrumentation APIs that can be used to extract the call graph information of a program including execution times of procedures and the number of times they are called. Unlike gprof, it does not need the source code of the application, and uses object modules, thus making its use independent of the programming languages and compilers. However ATOM is available only for Tru64 UNIX operating system. Recently, a toolkit called the SWAGkit has been developed (SWAGkit, 2006) which can be used to extract, abstract and present software architectures. Currently SWAGkit supports the extraction of C/C++ code, the abstraction to the architectural level and the presentation in a graphical form. It has been used to analyze and visualize some complex software systems, including the Linux operating system kernel (Bowman et al., 1999).

5. Illustrations and case studies

We illustrate the approach described in Section 3, with the help of case studies. The program chosen for the purpose of experimental illustration of the concepts of reliability, performance and security, was developed for the European Space Agency in C language and consists of about 10,000 lines of code. Its purpose is to provide a language-oriented user interface to the user for describing the configuration of an array of antennas. The program is divided into three main modules: the Parser module, the Computational module and the Formatting module. This program was used for a case study for investigating sensitivity of software reliability growth models to operational profile errors in Krishnamurthy and Mathur (1997).

Upon analysis of the code, the software architecture of the system was found to be as in Fig. 2 (Goseva-Popstojanova et al., 2001). The states 1, 2 and 3 correspond to the Parser, Computational and Formatting modules respectively. We shall refer to these as components 1, 2, and 3 respectively. The state C represents the completion of execution. Two faulty versions of the same program were

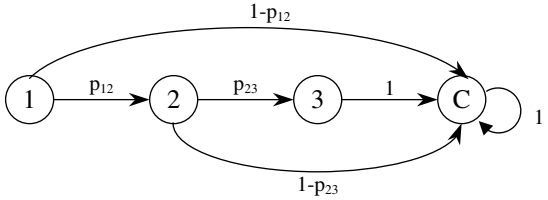

Fig. 2. ESA program architecture.

obtained by reinserting the faults discovered during integration testing and operational usage. Faulty version A consisted of faulty components 1 and 2 with fault free component 3, while faulty version B consisted of faulty component 2 and fault free components 1 and 3. The reliabilities of each component were estimated (Goseva-Popstojanova et al., 2001) using

$$R_i = 1 - \lim_{n_i \to \infty} \frac{f_i}{n_i}$$

where f_i is the number of failures of component i and n_i is the number of executions of component i in N randomly generated test cases. The mean execution times were measured by measuring the total time for a fixed number of executions of the program using the Unix utility *clock* and then dividing it by the number of executions. For the purpose of illustrating the approach we assume values for the vulnerability index of each component. The different parameters are given in Tables 1 and 2. The transition probabilities, the expected visit counts and the variance in the visit counts for both the faulty versions are presented in Table 3.

We show the results for both versions of the program, in Table 3. One can see that the bottlenecks for reliability, performance and security in case of Faulty version A are components 1, 3 and 2 respectively. Note that when considered as individual components, component 2 does not have the highest value of vulnerability index, but it is the security bottleneck in the context of the software system as a whole. Using the results in Goseva-Popstojanova et al. (2001) for comparison we can see from Table 4 that the overall reliability prediction is very close to the composite model solution when the second-order architectural effects are taken into consideration.

For illustrating the cache performance analysis approach we chose SHARPE (Sahner et al., 1996), which is a versatile software package for analyzing performance, reliability and performability models. The architecture of SHARPE is extracted by using ATOM wherein a module or component represents a file in the SHARPE package. The transition probabilities for the 30 components, which constitute the transient states of the absorbing DTMC are represented as a 3D mesh in Fig. 3. The visit counts ($m_{1,i}$'s) to the various components are computed as explained in Section 2.

For the computation of the miss ratios of individual files/modules, the experimental setup consisted of simulation of level-one data cache on DEC ALPHA machines. Typical cache configuration of a directly mapped cache of 32 K bytes was modeled with a cache block size of 16 bytes. Simulations were performed by incrementing the application object code with ATOM routines. The addition of these routines does not affect the execution flow or the caching behavior significantly (Srivastava and Eustace, 1994). These routines were used to record the execution

Table 1
Component reliabilities and transition probabilities

Faulty Version	Reliability of components			Transition probabilities	
	1	2	3	p_{12}	p_{23}
A	0.8428	0.8346	1	0.5933	0.7704
B	1	0.8346	1	0.7364	0.6866

Table 2
Component vulnerability indices and expected time spent

Component #	Time (ms)	VI
1	20	0.015
2	6.5	0.05
3	76	0.06

Table 3
Expected visit counts and variances

Faulty version	Component #	Expected visit counts $m_{1,i}$	Variance in visit counts $\sigma^2_{1,i}$	$E[R_i^{X_{1,i}}]$	$E[T_i X_{1,i}]$	$1 - E[(1 - \mathrm{VI}_i)^{X_{1,i}}]$
A	1	1	0	0.8428	20	0.0150
	2	0.5933	0.2413	0.9018	3.8565	0.0297
	3	0.4571	0.2482	1	34.7396	0.0274
B	1	1	0	1	20	0.0150
	2	0.7364	0.1941	0.8781	4.7866	0.0368
	3	0.5056	0.2500	1	38.4256	0.0303

Table 4
Reliability prediction comparison

Faulty version	Measured reliability	Composite model		Hierarchical model		Hierarchical model (using variance)	
		Reliability	Error (%)	Reliability	Error (%)	Reliability	Error (%)
A	0.7393	0.7601	2.81	0.7571	2.41	0.7601	2.81
B	0.8782	0.8782	0	0.8753	0.33	0.8781	0.01

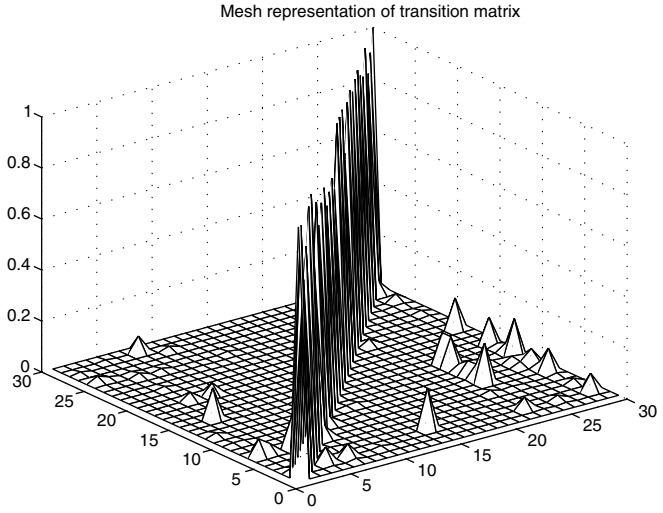

Fig. 3. Transition probabilities for SHARPE modules.

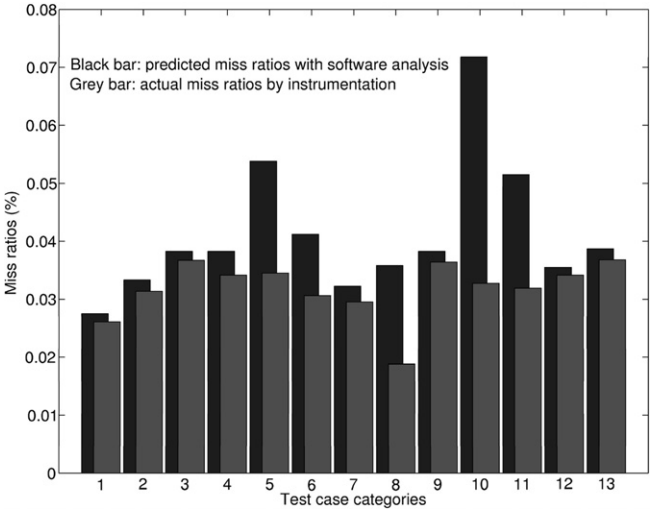

Fig. 4. Comparison between predicted and actual miss ratios.

Table 5
Miss ratios for each module

Module	Miss (%)	Module	Miss (%)	Module	Miss (%)
analyze.c	2.67	bind.c	2.80	bitlib.c	3.54
cexpo.c	2.24	cg.c	0.79	debug.c	4.52
expo.c	4.63	ftree.c	0.75	in_qn_pn.c	24.8
inchain.c	3.64	indist.c	9.95	inshare.c	31.9
inspade.c	3.90	maketree.c	17.4	mpfqn.c	4.35
factor.c	4.72	multpath.c	5.23	newcg.c	12.1
inlinear.c	7.50	newphase.c	4.54	pfqn.c	3.53
phase.c	0.72	reachgraph.c	5.85	read1.c	1.83
result.c	17.3	share.c	22.5	sor.c	3.13
symbol.c	2.73	uniform.c	37.4	util.c	7.95

details such as number of memory references and misses and the number of visits to each module. The average miss ratios (M_i's) for individual components as obtained from the simulation study are given in Table 5.

We use equations from Section 3 to get an estimate of the overall cache-miss ratio. Since the example application, SHARPE, deals with different types of probability models, it is expected that different model inputs would involve quite distinct control flow. Hence experiments for the 13 distinct categories in the suite were conducted. The miss ratios obtained from thus constructed models are compared with the actual miss ratios obtained directly from cache simulations in Fig. 4.

Fig. 4 shows that the model predictions are quiet close to the actual cache-miss behavior except for a few cases. The four cases with the largest discrepancies solve models in Markov chain, fault tree, multi-chain product form queuing network and single-chain product form queuing network models, respectively, and the differences are largest for the latter two categories. Careful inspection of related application source code finds that for these two cases, a large data file is accessed by multiple program files consecutively. As we have set our module to be a file, such data locality cannot be captured by this model, whereas the

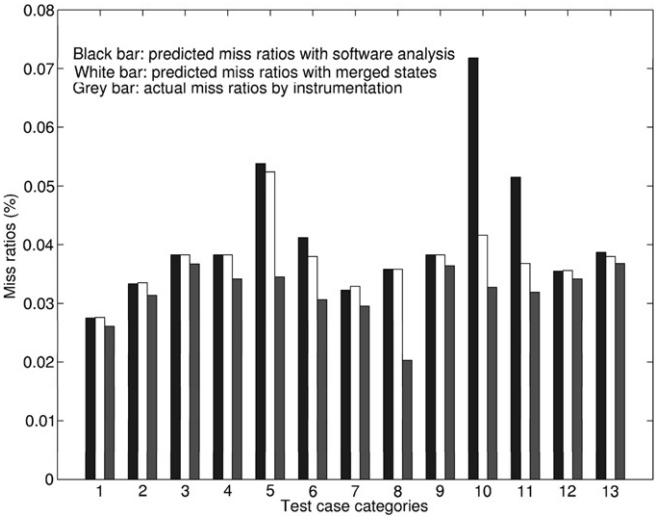

Fig. 5. Comparison between predicted and actual miss ratios with and without the merged states.

cache warm-up (occurrence of initial cache misses) is double counted in these modules. To capture this kind of data locality, we merge these modules that deal with the same data file into a large phase and run the experiments again. As only the software architecture is altered, cache simulation is only needed for the merged phase. Results are shown in Fig. 5. The figure shows a closer match on predictions on these cases without compromising the accuracy of other cases. This merging process also demonstrates a way of achieving better modularization.

6. Performance evaluation under resource contention

For real world systems, which have to cater to a large number of requests at a time, performance becomes one of the most critical concerns. For such systems, the response times that the users encounter largely depend on the load on the system and are always more than the total

average execution times (without any contention as computed in Section 3.2). The cause for this increase in response time with load is the increase in contention at the various resources, which causes jobs to queue up waiting for the resources to be free. These resources include hardware resources such as the CPU, the I/O subsystem, the physical connectors, etc. Moreover limitations on the number of concurrent jobs that can be processed by a machine at a time, also causes jobs to wait, for free processes. There are several pertinent questions that gain importance in such situations, like:

- How many clients will a system be able to handle?
- How does varying the number of clients affect the throughput and the average response time?
- What changes need to be done at the hardware or the software level to improve the system performance?

We extend our approach as mentioned in Section 3.2 to incorporate contention of resources and the queueing of jobs. Our approach consists of first generating the equivalent DTMC model, given the information about the architecture of the software system. This is then annotated with the individual resource requirements to find the requirements of various components on the different resources. Then for a given deployment of the various components of the software system on the available hardware, a queueing network model of the system can be generated. This is then solved for the likely workload for the system.

To illustrate this approach, we take the example of a typical tiered system. In such systems, the building blocks are components (sometimes a group of components) called tiers, which are allowed to interact only with the adjacent tiers. A typical computation in such systems involves a tier doing some processing and forwarding the request to the next tier till the request is fulfilled and the reply is sent back across the various tiers back to the client. We have addressed the issue of performance evaluation for a tiered software architecture-based system in the presence of multiple clients in another related work (Sharma et al., 2005) and we present it briefly here. A tiered system with nl tiers can be visualized as in Fig. 6a. We present the DTMC model for the system in Fig. 6b.

In this DTMC model, one starting (S_0) and one ending state (S_{2nl+1}) represent the start and end of the computation. Any pair of states S_i and S_{2i+1}, represent the control flow residing in the tier i. The probability of transition between different tiers can be used to label the DTMC model. This DTMC can be used to find the expected number of visits to different tiers as explained in previous sections.

The visit counts for different tiers can be used to find out the total resource requirements for each tier $tcpu(i)$, once the average resource demands for a single execution for each tier are known. These resource demands can be measured for individual tiers by using system tools such as *iostat* and *top*. As the different tiers have to be deployed on the available machines, corresponding to each machine

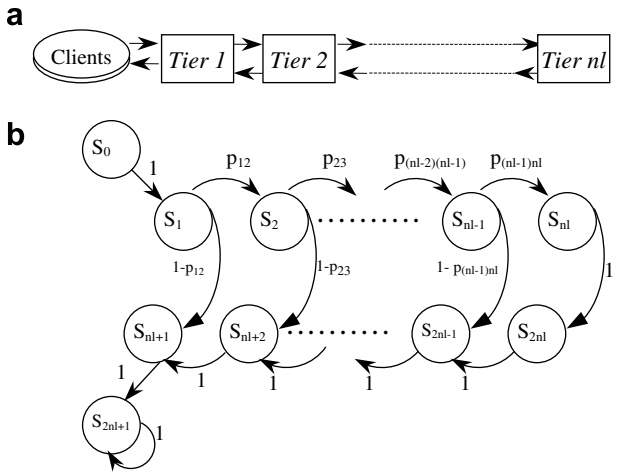

Fig. 6. A typical tiered system (a) and the equivalent DTMC representation (b).

j, we have a set $L(j)$ containing indices of the tiers residing on machine j. Using this information one can find the total CPU and I/O requirements at each machine as

$$tmcpu(j) = \sum_{i \in L(j)} tcpu(i)$$

$$tmdisk(j) = \sum_{i \in L(j)} tdisk(i)$$

Similarly, with information about the bandwidth of connectors joining the machines and the message sizes communicated between the machines, the total requirements on the various connectors could also be found.

For constructing the queueing network, each hardware device is modeled as a queueing node, with the corresponding service demands mapped to each of them. Systems with multiple processors and disks are modeled accordingly. We model systems with limited number of possible concurrent processes by using a hierarchical combination of models. A lower level closed form queueing network (QN) model is created whose expected throughput is used to find the rate of the upper level flow equivalent server.

We have developed a web-based tool that automates our approach (Sharma et al., in preparation). This tool is simple to use and does the job of the system performance modeler and analyst. It generates a set of web-based forms to allow the architect/administrator to enter the software and hardware specifications. The tool creates and solves the DTMC model and then generates the QN model that is fed to SHARPE, which is a versatile package for analyzing and solving reliability, performance, and performability models. As part of the output, average throughput and mean response time graphs as shown in Fig. 7, for the range of number of clients in the given workload are provided to the user. The tool also locates the performance bottlenecks in the system and information regarding such components and the suggested scaleups is provided along with the predicted performance improvements.

This approach has been recently extended further to help the user, with a step-by-step guidance for improving

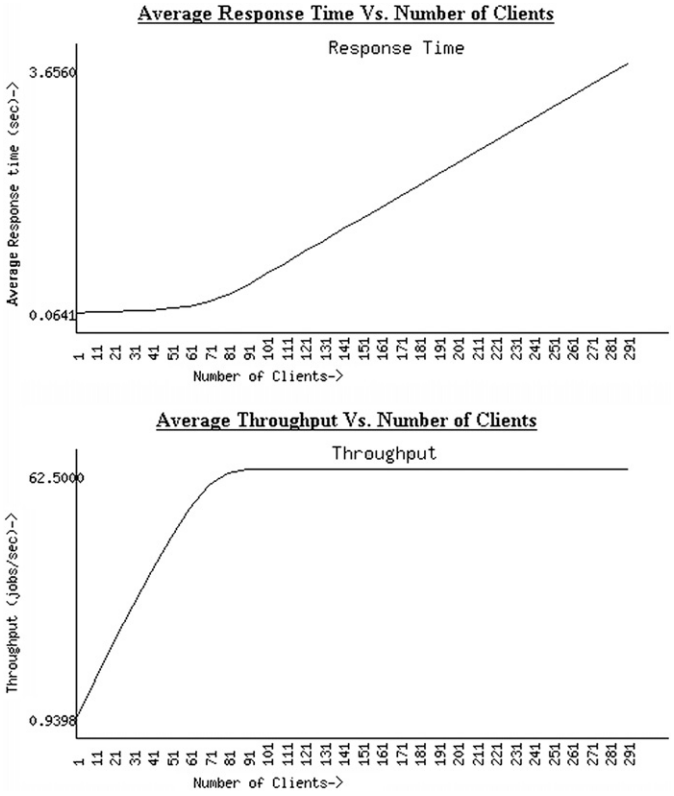

Fig. 7. Variation in average response time and throughput with the number of clients for an example system.

the performance to a desired level (Sharma et al., in preparation). The user is asked to specify the target performance for a particular workload and then an iterative bottleneck analysis and removal is done to reach the target and determine the choices available for the same. These are then presented to the user.

7. Sensitivity analysis

Usually in the architectural phase it might be difficult to ascertain some model parameter values accurately. Moreover a significant amount of effort might be needed depending upon the required precision. Sensitivity analysis is very important in a study of this nature because it can predict the impact of changes in different parameters on the final result. This would help determine (change in) which parameters is the system most sensitive to. The hierarchical modeling approach, which we present in this paper, aids in sensitivity analysis along the following two dimensions:

- Study the effect of the change in individual component parameters (reliability, vulnerability index, etc.) on the overall system behavior.
- Study the effect of the change in workload on the overall system behavior.

We present the sensitivity analysis based on the two dimensions in the following subsections.

7.1. Effect of change in reliability of a component

We consider the effect of change in the reliability of individual component k by finding the derivative of expected reliability of the overall system as given in Section 3.1 with respect to R_k. We get the following (Gokhale and Trivedi, 2002):

$$\frac{\mathrm{d}E[R]}{\mathrm{d}R_k} = \left[m_{1,k} R_k^{m_{1,k}-1} + \frac{1}{2}\sigma_{1,k}^2 \left(m_{1,k} R_k^{m_{1,k}-1} (\log R_k)^2 + \frac{2\log R_k}{R_k} R_k^{m_{1,k}} \right) \right] \times \left[\prod_{i=1, i\neq k}^{n-1} \left(R_i^{m_{1,i}} + \frac{1}{2}(R_i^{m_{1,i}})(\log R_i)^2 \sigma_{1,i}^2 \right) \right] R_n$$

where, $\mathrm{d}E[R]/\mathrm{d}R_k$ is the rate of change in overall reliability of the system. This shows that there exists a complex dependence of the change in overall reliability of the software on the change in individual component's reliability. As presented in Gokhale and Trivedi (2002), the revised expected reliability of the software system is given by

$$E[R_{\mathrm{rev}}] = E[R_{\mathrm{orig}}] + (\mathrm{d}E[R]/\mathrm{d}R_k)\Delta R_k$$

where, R_{rev} denoted the revised reliability of the system, ΔR_k is the change in the reliability of component k, and R_{orig} denotes the original reliability of the same. We show the effects of variations in the reliability of components 1 and 2 in the Faulty version A of the ESA program introduced in Section 4. Fig. 8 shows the variation in overall expected reliability as the reliabilities of components 1 and 2 are varied between 0.75 and 0.99.

As is evident from Fig. 8, the variation in the estimated reliability of component 1 has a greater effect on the overall reliability than that component 2. This is due to the fact that the mean number of visits to component 1 is significantly higher than those to component 2.

7.2. Effect of change in time spent in a component

We consider the effect of the change in the time spent in a component k, given by τ_k, on the overall time spent in the

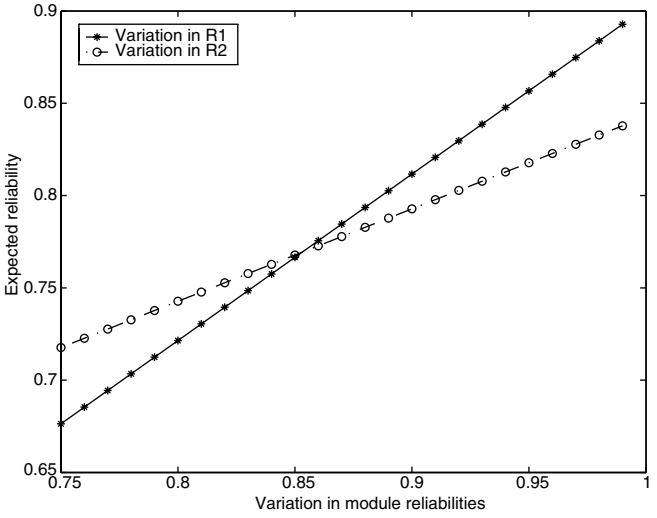

Fig. 8. Variation in expected overall reliability with individual reliabilities.

software system by differentiating the expected overall time spent in the software system as given in Section 3.2 with respect to τ_k. We get

$$dE[T]/d\tau_k = m_{1,k}$$

where $dE[T]/d\tau_k$ is the rate of change in overall time spent in the software system. Let T_{orig} denote the original overall time spent in the software system, and $\Delta\tau_k$ is the change in the time spent in component k, then the expected revised overall time spent in the system, $E[T_{rev}]$ is given by

$$E[T_{rev}] = E[T_{orig}] + (dE[T]/d\tau_k)\Delta\tau_k$$

We show the variation in the overall time spent in the software system when the time spent in modules 1, 2, and 3 in the ESA program's faulty version A is varied from 5 to 50 ms in Fig. 9.

It can be noticed that the software system is more sensitive to variation in the time spent in component 1 and is not affected much by variation in component 3.

7.3. Effect of change in vulnerability index of a component

We differentiate the expected VI expression as given in Section 3.3 with respect to VI_k, the vulnerability index of component k, to get the rate of change in the overall expected vulnerability index of the system for a change in VI_k. We have

$$\frac{dE[VI]}{dVI_k} = \left[m_{1,k}(1-VI_k)^{m_{1,k}-1} + \frac{1}{2}\sigma_{1,k}^2 \left(m_{1,k}(1-VI_k)^{m_{1,k}-1}(\log(1-VI_k))^2 + \frac{2\log(1-VI_k)}{(1-VI_k)}(1-VI_k)^{m_{1,k}} \right) \right] \left[\prod_{i=1,i\neq k}^{n-1} ((1-VI_i)^{m_{1,i}} + \frac{1}{2}((1-VI_i)^{m_{1,i}})(\log(1-VI_i))^2\sigma_{1,i}^2 \right) \right] (1-VI_n)$$

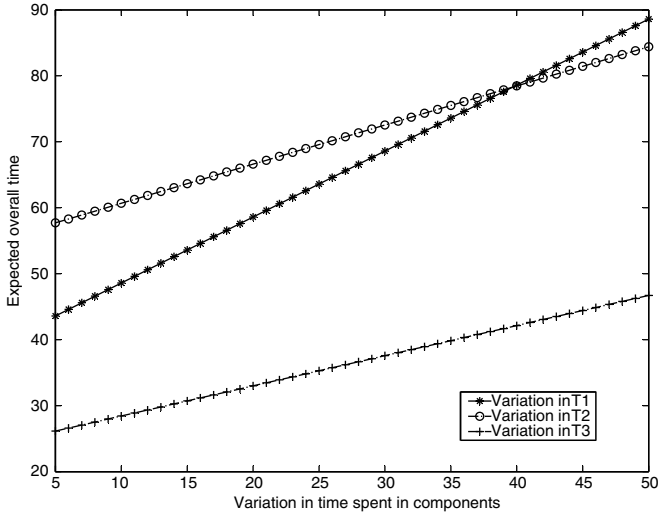

Fig. 9. Variation in expected overall time spent with individual spent times.

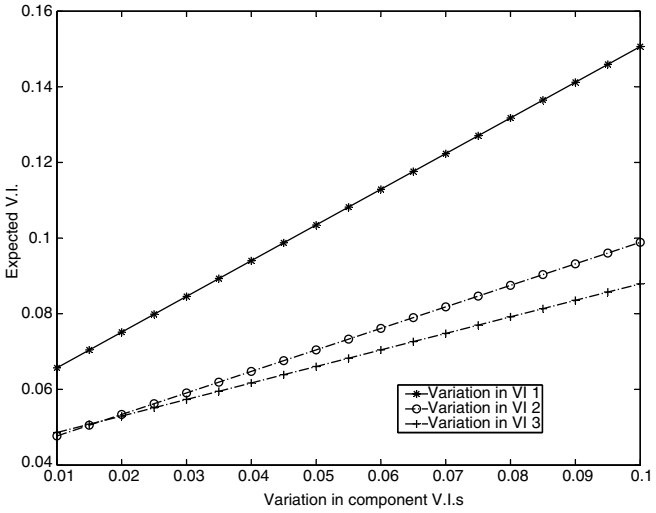

Fig. 10. Variation in expected overall VI with individual VIs.

Taking revised vulnerability index of the whole system as VI_{rev}, ΔVI_k as the change in the vulnerability index of component k, and the original to be VI_{orig}, we have

$$E[VI_{rev}] = E[VI_{orig}] + (dE[VI]/dVI_k)\Delta VI_k$$

The variation in overall VI of the system when the vulnerability indices of components 1, 2 and 3 in the ESA program's faulty version A are varied from 0.01 to 0.1 is shown in Fig. 10.

As in the case of reliability and the expected time spent in the software system, the vulnerability index of the system is affected more by the variation in the vulnerability index of component 1, than that of component 2 or 3. So the estimation of the vulnerability index for component 1 should be done much more precisely for an appropriate prediction.

7.4. Effect of change in cache-miss behavior of a component

We consider the effect of the change in the cache-miss ratio in a component k, given by M_k, on the overall cache-miss ratio by differentiating the expected overall time spent in the software system as given in Section 3.4 with respect to M_k. We get

$$\frac{dE[M]}{dM_k} = \frac{m_{1,k}n_k}{\sum_i^n m_{1,i}n_i}$$

where, $dE[M]/dM_k$ is the rate of change in overall cache-miss ratio in the software system. Let M_{orig} denote the original overall cache-miss ratio of the software system, then for ΔM_k change in the miss ratio of module k, the expected revised overall cache-miss ratio of the system, $E[M_{rev}]$ is given by

$$E[M_{rev}] = E[M_{orig}] + (dE[M]/dM_k)\Delta M_k$$

We show the variation in the overall miss ratio of the SHARPE software package for one of the test scenarios, when the miss ratio of modules *result.c* and *share.c* is varied from 0.01 to 0.45 in Fig. 11.

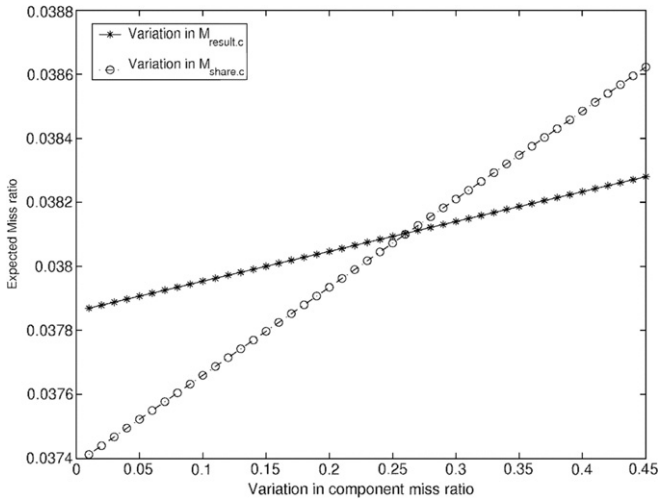

Fig. 11. Variation in expected overall miss ratio with individual miss ratios.

It is evident that the sensitivity of the software system is greater for a variation in the miss ratio of *share.c* rather *result.c*. Among others, the primary reason for that is the difference in the value of n_i of the two (which is about 2.93 for *share.c* and 0.65 for *result.c*).

7.5. Effect of change in workload

Any change in the software architecture of the system constitutes the change in workload of the system. The change reflects in the changed transition probabilities of the control flow between components and hence changed expected visit counts of various components. We show an illustration of the effect of changed workload through empirical analysis, as the expressions to deal with the same are very cumbersome. Consider the faulty version A of the ESA program. We vary the probability of transfer of control flow from component 1 to 2 from 0.05 to 0.95 and show the variation in the expected reliability and the expected vulnerability index of the system in Fig. 12. Notice that the reliability of the system is not at all affected by changes in p_{23} while it falls monotonically with increase in p_{12}. The reason is that though the change in p_{23} causes change in the software architectural model, which translates into change in the visit count of component 3, the reliability of component 3 in the Faulty version A of the program is 1. So the value of $E[R_3^{X_{1,3}}]$ will always be 1 irrespective of the value the expected visit counts to this component. The expected vulnerability index of the system shows a higher sensitivity to changes in p_{12} as can be seen in Fig. 12.

8. Incorporating dependence between software components

In the hierarchical modeling approach, we have assumed independence between behavior of components. Specifically, we have assumed that the control flow transition from a particular component does not depend on the path taken to reach this component. For the same, till now we have utilized first-order DTMCs – wherein the past is not important for prediction the future of the chain.

From the examples we have presented, one can see that modeling component-based software systems under such assumption does yield satisfactory results. However for some systems, this assumption may be too strong. In such systems, the transition probabilities of control flow from a certain component might no longer be independent of the path taken to reach it. In the simplest case, the probability of transition from a component might depend on the current component as well as the component from which the control flow arrived to this component. In general the dependence may exist for any length of the path taken to reach the present. In other words rather than using first-order Markov chains, we will have to resort to higher-order Markov chains (Haring, 1983).

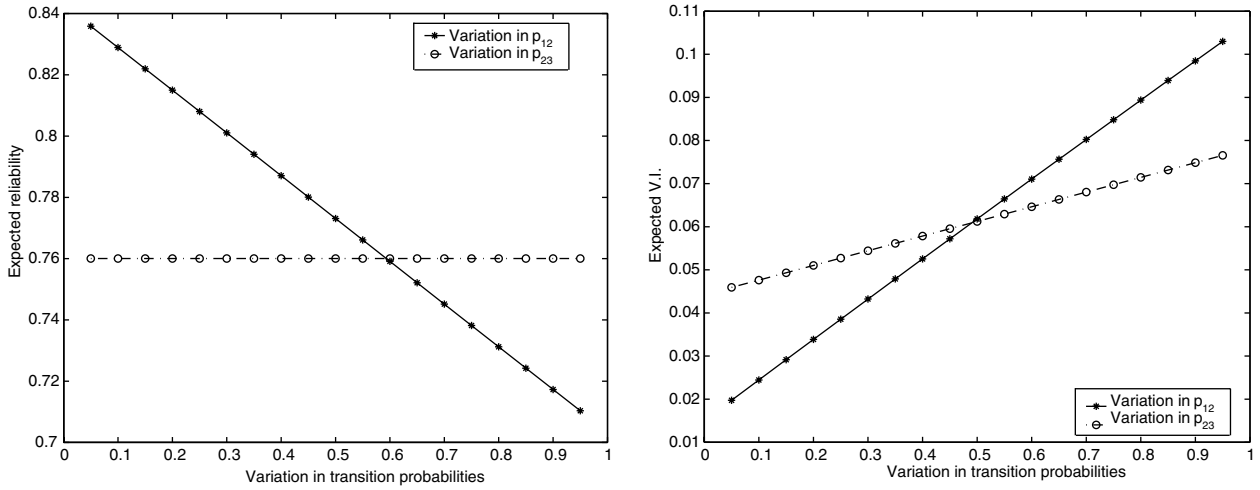

Fig. 12. Effect of workload variation on the expected reliability and vulnerability index of the system.

It can be shown that a nth-order Markov chain with state space S can always be specified as a first-order Markov chain with the state space S^n. For example, consider a software system having s components within which, the dependence of control flow transition is on the present component as well as the component before this in the execution path. We can thus represent this using a second-order Markov chain with state space S, with $|S| = s$. If $X_0, X_1, \ldots, X_n$, represent the observed state at discrete times, $t_0, t_1, \ldots, t_n$, then

$$P(X_{n+1} = i_{n+1} | X_0 = i_0,\ X_1 = i_1, \ldots, X_n = i_n) = P(X_{n+1} = i_{n+1} | X_n = i_n,\ X_{n-1} = i_{n-1})$$

This can also be represented using a first-order DTMC. This Markov chain will now have the state space as S^2, with states marked as a 2-tuples (x,k) where x represents the previous component (from where the control flow arrived to this state) and k denotes the current component (having the control flow). Each component in the software system can be represented using $|S|$ states of the DTMC, each of which will be representing the control having arrived from a different predecessor. Thus this first-order DTMC will incorporate the necessary dependence, and needs no special methods for analysis.

In general, for a system having s components, with the control flow depending on the last n components traversed, the corresponding first-order DTMC will have utmost s^n states, with each state represented by a n-tuple, showing the n components traversed in that path. The transition probability matrix accordingly will be a s^n by s^n matrix representing the different transitions.

The above representation of a system having dependence between the components being modeled by a first-order DTMC expands the applicability of the hierarchical approach. For calculating the visit counts, for such DTMC models describing a system with a higher (say nth) order dependence of control flow, we define $m_{1,k}$, as the expected number of visits to component k from the first component, as

$$m_{1,k} = \sum m_{1,(x2,x3,\ldots,xn,k)} \quad \text{for all tuples } i = (x_1, x_2, \ldots, x_n) \in S_n \text{ s.t. } p_{i,k} > 0$$

Once one gets this expected number of visits, the hierarchical method can then be followed. One can use the expected visit counts along with the component attributes to find the overall reliability, or calculate the expected execution time of the software with and without contention of hardware resources, calculate the vulnerability of the components or do cache-miss analysis of the system. We have not as yet explored the estimation of variance in visit counts for such models.

The dependence between components in terms of attributes like reliability can also be incorporated in the model. Specifically, failure dependence across different visits to the same component can be modeled using a non-homogenous DTMC. This has also been dealt with by using a time dependent failure rates, such as the NHPP type failure rate successfully used in Gokhale et al. (1998). Correlated failures of components can also be incorporated in a DTMC model (Goseva-Popstojanova and Trivedi, 2000). In any case since we assume that a failure of a component brings the application down, a correlated failure simply adds to this probability of individual component failure.

9. Conclusion

In this paper we presented a hierarchical approach for predicting various attributes of a software system based upon its software architecture and the attributes of its constituent components. Our approach is unique in that it brings together reliability, performance, security and cache-miss behavior prediction in the same model, which is hierarchical in nature. The paper extends the previous work in the field of architecture-based analysis of reliability and performance of software systems, and makes an original contribution in the field of security prediction and cache-miss analysis by taking software architecture into consideration, which other studies have lacked till now. This model is fairly accurate as it takes into account the second-order architectural effects also. We also suggest the use of higher-order Markov chains for representing systems where the control flow is not independent of the path through the software components.

Due to the hierarchical nature of the model, changes in the software architecture of the system could be accommodated very easily by changing the architectural model only. Our approach is thus, well suited for use in the design phase of the software development life cycle. Moreover the change in the individual behavior of a component does not cause the model to collapse, as the change is localized in nature, again due to the hierarchical nature. Hence this approach is also very effective in the later phases of the software development life cycle, when components are being tested and simultaneously corrected and/or updated. One limitation of this approach is that it is difficult to model concurrency of control flow using a DTMC and concurrently executing components have to be modeled as a single state in the DTMC model.

We present the analytical treatment of our approach in this paper and provide the expressions for the reliability, performance, cache-miss analysis, and security. We also show the application of this approach to real systems by presenting two case studies. We also illustrated the application of the approach for performance analysis of systems catering to multiple clients and thus having resource contention. We have presented both the formal sensitivity analysis (with derivatives), as well as the brute force approach. This is very important for systems where the individual parameters are not accurately measured or estimated.

This approach can be extended to incorporate the modeling of various levels of fault tolerance that may exist in a software system and can be used to study their effect on

various system attributes. This is one of the areas that we are focusing on currently (Sharma and Trivedi, in preparation). Another extension could be to automatically suggest alternatives to optimize various system attributes such as performance and reliability. Tradeoffs may exist between various attributes and it is important to choose a system configuration that optimizes these.

We have also not included costs in this model. Appropriate costs can be associated with various components, and configurations and the approach can be used to also output the costs of different system configurations. This will be helpful to software engineers and designers to decide the best system, while staying within the budgets. This is another direction for future research.

References

Bhat, U.N., 1984. Elements of Applied Stochastic Processes, second ed. John Wiley & Sons, Inc.

Bobbio, A., Trivedi, K.S., 1986. An aggregation technique for the transient analysis of stiff Markov chains. IEEE Transactions on Computers 35 (9), 803–814.

Bowman, I.T., Holt, R.C., Brewster, N.V., 1999. Linux as a case study: its extracted software architecture. In: Proceedings of the International Conference on Software Engineering (ICSE'99), Los Angeles, CA, USA, p. 555.

Cheung, R.C., 1980. A user-oriented software reliability model. IEEE Transactions on Software Engineering 6 (2), 118–125.

Chow, C.K., 1976. Determination of a cache's capacity and its matching storage capacity. IEEE Transactions on Computers C25 (2), 157–164.

Clark, D.W., Emer, J.S., 1985. Performance of the VAX-11/780 translation buffer: simulation and measurement. ACM Transactions on Computer Systems 3 (1).

Everett, W., 1999. Software component reliability analysis. In: Proceedings of the Symposium on Application-Specific Systems and Software Engineering Technology (ASSET'99), Dallas, TX, pp. 204–211.

GNU gprof, 1998. Available from: <www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html>.

Gokhale, S.S., Trivedi K.S., 2002. Reliability prediction and sensitivity analysis based on software architecture. In: Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02), Annapolis, MD, p. 64.

Gokhale, S.S., Trivedi, K.S., Wong, W.E., Horgan, J.R., 1998. An analytical approach to architecture-based software reliability prediction. In: Proceedings of the IEEE International Computer Performance and Dependability Symposium'98, Durham, NC, pp. 13–22.

Goseva-Popstojanova, K., Trivedi, K.S., 2000. Failure correlation in software reliability models. IEEE Transactions on Reliability 49 (1), 37–48.

Goseva-Popstojanova, K., Trivedi, K.S., 2001. Architecture-based approach to reliability assessment of software systems. Performance Evaluation 45 (2–3).

Goseva-Popstojanova, K., Mathur, A.P., Trivedi, K.S., 2001. Comparison of architecture-based software reliability models. In: Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE'01), Hong Kong, China, pp. 22–33.

Haring, G., 1983. On stochastic models of interactive workloads. In: Proceedings of the 9th International Symposium on Computer Performance Modelling, Measurement and Evaluation. North-Holland, pp. 133–152.

Horgan J.R., London, S., 1992. Atac: a data flow coverage testing tool for C. In: Proceedings of the Second Symposium on Assessment of Quality software Tools, New Orleans, LA, pp. 2–10.

Horowitz, M., Agarwal, A., Hennessy, J., 1989. An analytical cache model. ACM Transactions on Computer Systems 7 (2), 184–215.

Knuth, D.E., 1997. The Art of Computer Programming, third ed.Fundamental Algorithms, vol. 1 Addison Wesley Longman Publishing Co. Inc., Redwood City, CA, USA.

Krishnamurthy, S., Mathur, A.P., 1997. On the estimation of reliability of a software system using reliabilities of its components. In: Proceedings of the 8th International Symposium on Software Reliability Engineering (ISSRE'97), Albuquerque, NM, pp. 146–155.

Kubat, P., 1989. Assessing reliability of modular software. Operations Research Letters 8, 35–41.

Laprie, J.C., 1984. Dependability evaluation of software systems in operation. IEEE Transactions on Software Engineering 10 (6), 701–714.

Ledoux, J., 1999. Availability modeling of modular software. IEEE Transactions on Reliability 48 (2), 159–168.

Li, W., 2000. A Markovian analytical cache performance model, Masters Thesis, Duke University, Durham, NC.

Lipson, H.F., Mead, N.R., Moore, A.P., 2001. Can we ever build survivable systems from COTS components? Technical Note, CMU/SEI-2001.

Littlewood, B., 1975. A reliability model for systems with Markov structure. Applied Statistics 24 (2), 172–177.

Martonosi, M., Ghosh, S., Malik, S., 1997. Cache miss equations: an analytical representation of cache misses. In: Proceedings of the 11th ACM International Conference on Supercomputing, Vienna, Austria, pp. 317–324.

Petriu, D.C., Woodside, C.M., 2002. Software performance models from system scenarios in use case maps. In: Proceedings of the 12th international Conference on Computer Performance Evaluation, Modelling Techniques and Tools, London, pp. 141–158.

Petriu, D., Shousha, C., Jalnapurkar, A., 2000. Architecture-based performance analysis applied to a telecommunication system. IEEE Transactions on Software Engineering 26 (11), 1049–1065.

Rao, G.S., 1978. Performance analysis of cache memories. Journal of the ACM 25 (3), 378–395.

Reibman, A., Trivedi, K., 1988. Numerical transient analysis of Markov models. Computers & Operations Research 15 (1), 19–36.

Reussner, R., Schmidt, H.W., Poernomo, I., 2003. Reliability prediction for component-based software architectures. Journal of Systems and Software 66 (3), 241–252.

Sahner, R.A., Trivedi, K.S., Puliafito, A., 1996. Performance and Reliability Analysis of Computer Systems: An Example-based Approach Using the SHARPE Software Approach. Kluwer Academic Publishers., Boston, MA.

Sharma, V.S., Trivedi, K.S., in preparation. Reliability and performance of component based software systems with restarts, retries, reboots and repairs. In: Proceedings of the 17th IEEE International Symposium on Software Reliability Engineering (ISSRE'06), Raleigh, NC.

Sharma, V.S., Jalote, P., Trivedi, K.S., 2005. Evaluating performance attributes of tiered software architecture. In: Proceedings of the 8th International SIGSOFT Symposium on Component-based Software Engineering (CBSE'05), St. Louis, MO, pp. 66–81.

Sharma, V.S., Jalote, P., Trivedi, K.S., in preparation. A performance engineering tool for tiered software systems. In: Proceedings of the 30th IEEE Annual International Computer Software and Applications Conference (COMPSAC'06), Chicago, IL.

Shedler, G.S., Tung, C., 1976. Locality in page reference strings. SIAM Journal of Computing 1 (3), 218–241.

Shooman, M., 1976. Structural models for software reliability prediction. In: Proceedings of the 2nd International Conference on Software Engineering, San Francisco, CA, pp. 268–280.

Smith, C.U., 1990. Performance Engineering of Software Systems. Addison Wesley.

Smith, C.U., Llad, C.M., Cortellessa, V., Marco, A.D., Williams, L.G., 2005. From UML models to software performance results: an SPE process based on XML interchange formats. In: Proceedings of the 5th

International Workshop on Software and Performance (WOSP'05), Palma de Mallorca, Spain, pp. 87–98.
Srivastava, A., Eustace, A., 1994. Atom: a system for building customized program analysis tools. In: Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation, pp. 196–205.
Stone, H.S., Singh, J.P., Thiebaut, D.F., 1992. A model of workloads and its use in miss-rate prediction for fully associative caches. IEEE Transactions on Computers 41 (7), 811–825.
SWAGkit, 2006. Available from: <http://www.swag.uwaterloo.ca/swagkit/index.html>.
Thiebaut, D., 1987. Footprints in the cache. ACM Transactions on Computer Systems 1 (4), 281–293.
Trivedi, K.S., 2001. Probability and Statistics with Reliability, Queuing and Computer Science Applications. John Wiley and Sons.
Voas, J., Ghosh, A., McGraw, G., Chharon, F., Miller, K., 1996. Defining an adaptive software security metric from a dynamic software failure tolerance measure. In: Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS'96), pp. 250–263.
Wang, W.H., Baer, J.L., 1990. Efficient trace-driven simulation methods for cache performance analysis. In: Proceedings of the SIGMETRICS'90 Conference on Measurement and Modeling of Computer Systems, Boulder, CO, USA, pp. 27–36.
Xie, M., Wohlin, C., 1995. An additive reliability model for the analysis of modular software failure data. In: Proceedings of the 6th International Symposium on Software Reliability Engineering (ISSRE'95), Toulouse, France, pp. 188–194.
Yacoub S., Cukic, B., Ammar, H., 1999. Scenario-based reliability analysis of component-based software. In: Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE'99), Boca Raton, FL, pp. 22–31.
Yacoub, S., Cukic, B., Ammar, H.H., 2004. A scenario-based reliability analysis approach for component-based software. IEEE Transactions on Reliability 53 (4), 465–480.

Vibhu Saujanya Sharma is a Ph.D. candidate at the Department of Computer Science and Engineering at Indian Institute of Technology (IIT) Kanpur, India. He joined IIT Kanpur in 2002, after completing his B.Tech program in Computer Engineering from Aligarh Muslim University where he was a gold medalist. He visited the Department of ECE at Duke University during spring 2004 as an *Associate in Research* and has been selected as an *Infosys Fellow* at IIT Kanpur. He is a student member of IEEE and ACM. His research interests include architecture-based modeling, analysis and tuning of performance and reliability of software systems.

Kishor S. Trivedi holds the Hudson Chair in the Department of Electrical and Computer Engineering at Duke University, Durham, NC. He has been on the Duke faculty since 1975. He is the author of a well known text entitled, *Probability and Statistics with Reliability, Queuing and Computer Science Applications*, published by Prentice-Hall; this text was reprinted as an Indian edition; a thoroughly revised second edition (including its Indian edition) of this book has been published by John Wiley. He has also published two other books entitled, *Performance and Reliability Analysis of Computer Systems*, published by Kluwer Academic Publishers and *Queueing Networks and Markov Chains*, John Wiley. He is a Fellow of the Institute of Electrical and Electronics Engineers. He is a Golden Core Member of IEEE Computer Society. During a sabbatical year 2002–2003, he was the Poonam and Prabhu Goel Professor in the Department of Computer Science and Engineering at IIT Kanpur while at the same time he was a Fulbright Visiting Lecturer to India. He has published over 350 articles and has supervised 39 Ph.D. dissertations.