ALGORITHMS FOR PERFORMANCE, DEPENDABILITY, AND PERFORMABILITY EVALUATION USING STOCHASTIC ACTIVITY NETWORKS

SUMMARY OF RESEARCH

Principal Investigator: William H. Sanders

Period Covered by the Report: 1/16/1996- 1/15/1997

Daniel D. Deavours, M. Akber Qureshi, and William H. Sanders
Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois
1308 West Main Street
Urbana, IL 61801-2307
(217) 333-0345
{deavours,qureshi,whs}@crhc.uiuc.edu

Grant Number: NAG 1 1782

Chapter 1

INTRODUCTION

Modeling tools and technologies are important for aerospace development. At the University of Illinois, we have worked on advancing the state of the art in modeling by Markov reward models in two important areas: reducing the memory necessary to numerically solve systems represented as stochastic activity networks and other stochastic Petri net extensions while still obtaining solutions in a reasonable amount of time, and finding numerically stable and memory-efficient methods to solve for the reward accumulated during a finite mission time.

A long standing problem when modeling with high level formalisms such as stochastic activity networks is the so-called state space explosion, where the number of states increases exponentially with size of the high level model. Thus, the corresponding Markov model becomes prohibitively large and solution is constrained by the the size of primary memory. To reduce the memory necessary to numerically solve complex systems, we propose new methods that can tolerate such large state spaces that do not require any special structure in the model (as many other techniques do). First, we develop methods that generate row and columns of the state transition-rate-matrix on-the-fly, eliminating the need to explicitly store the matrix at all. Next, we introduce a new iterative solution method, called modified adaptive Gauss-Seidel, that exhibits locality in its use of data from the state transition-rate-matrix, permitting us to cache portions of the matrix and hence reduce the solution time. Finally, we develop a new memory and computationally efficient technique for Gauss-Seidel based solvers that avoids the need for generating rows of A in order to solve $Ax = b$. This is a significant performance improvement for on-the-fly methods as well as other recent solution techniques based on Kronecker operators. Taken together, these new results show that one can solve very large models without any special structure.

The second approach, we developed a tool that makes no assumptions about the underlying structure of the Markov process, and it requires only slightly more memory than what

is necessary to hold the solution vector itself. It uses a disk to hold the state-transition-rate matrix, a variant of block Gauss-Seidel as the iterative solution method, and an innovative implementation that involves two parallel processes: the first process retrieves portions of the iteration patrix from disk, and the second process dŏes repeated computation on small portions of the matrix. Thus, only a part of the matrix need be in memory at any one time. To illustrate its use, we consider two realistic models – a Kanban manufacturing system and the Courier protocol stack. Depending on model parameter values, these models have up to 10 million states and about 100 million transitions, but we can still efficiently solve the models on a workstation with 128 Mbytes of memory and 4 Gbytes of disk. This is a significant improvement over the present state of the art with respect to the size of the model we can solve, the solution time, and the class of high level models we may solve.

The above techniques address exact numerical results for steady state behavoir. A much more difficult problem is finding the probability distribution the reward accumulated during a finite interval of time. The interval may correspond to the mission period in a mission-critical system, the time between scheduled maintenances, or a warranty period. In such models, changes in state correspond to changes in system structure (due to faults and repairs), and the reward structure depends on the measure of interest. For example, the reward rates may represent a productivity rate while in that state, if performability is considered, or the binary values zero and one, if interval availability is of interest. We present a new methodology to calculate the distribution of reward accumulated over a finite interval. In particular, we derive recursive expressions for the distribution of reward accumulated given that a particular sequence of state changes occurs during the interval, and we explore paths one at a time. The expressions for conditional accumulated reward are new and are numerically stable. In addition, by exploring paths individually, we avoid the memory growth problems experienced when applying previous approaches to large models. The utility of the methodology is illustrated via application to a realistic fault-tolerant multiprocessor model with over half a million states.

The rest of this report is organized in the following way. Chapter 2 discusses on-the-fly solution techniques, the new iterative solution technique called modified adaptive Gauss-Seidel, and a technique for performing Gauss-Seidel based iterations by accessing only columns of A. Chapter 3 discuesses the disk-based tool we developed for solving very large Markov systems. Chapter 4 explains the new path-based technique for solving for distributions of reward variables that are both computationally and memory efficient, and numerically stable.

Chapter 2

"ON-THE-FLY" SOLUTION TECHNIQUES FOR STOCHASTIC PETRI NETS AND EXTENSIONS

Abstract

Use of a high-level modeling representation, such as stochastic Petri nets, frequently results in a very large state space. In this paper, we propose new methods that can tolerate such large state spaces and that do not require any special structure in the model. First, we develop methods that generate rows and columns of the state transition-rate-matrix on-the-fly, eliminating the need to explicitly store the matrix at all. Next, we introduce a new iterative solution method, called modified adaptive Gauss-Seidel, that exhibits locality in its use of data from the state transition-rate-matrix. This permits the caching of portions of the matrix, hence reducing the solution time. Finally, we develop a new memory- and computationally-efficient technique for Gauss-Seidel-based solvers that avoids the need for generating rows of A in order to solve $Ax = b$. Taken together, these new results show that one can solve very large SPN, GSPN, SRN, and SAN models without any special structure.

I Introduction

Problems of scalability in models and the resulting state-space explosion are daunting. The traditional approach of generating a state-level model from a high-level specification, such as stochastic Petri nets, typically results in very large state spaces for practical models. Such problems are further compounded with even higher-level formalisms, such as stochastic Petri nets with tokens that have attributes. This problem is often called the "largeness problem," and is a major impediment to accurately modeling large and complex systems.

There have been numerous attempts to address the largeness problem, resulting in techniques that produce either exact or approximate results. The exact approaches tend to fall into two general categories: those that attempt to reduce the state-space size (e.g., methods based on stochastic well-formed nets [3] or reduced base model construction [17]), and those that attempt to tolerate the large state space. Several techniques that tolerate large state spaces take advantage of the fact that some components of a model (called submodels) interact in a limited way with other submodels, so that the state-transition-rate matrix of the model is a function of Kronecker operators on the state-transition-rate matrix of the submodels. Solution methods for stochastic automata networks [18] are an example of this type of method.

More recently, there has been work on superposed generalized stochastic Petri nets (SGSPNs), which are essentially independent submodels that may be joined by synchro-

nizing on a timed transition. This class seems to be more promising as a less restrictive modeling technique. First introduced in [7], solutions for SGSPNs were restricted by the so-called product space (the product of the submodels' state spaces), which could be much larger than the set of tangible reachable states. Kemper, in [10, 11], devised a method to operate on the tangible space, rather than the product space, by providing a mapping from product space to the tangible reachable space. Ciardo and Tilgner [6] built on Kemper's work by removing some of the imposed restrictions, e.g., by allowing synchronizing transitions to be immediate.

We believe that there are three substantial restrictions with current SGSPN techniques. First, all known methods based on Kronecker operators require models to have a structure such that there are partially independent components with limited interaction between them. While Ciardo and Tilgner relax these requirements significantly, many models still do not exhibit the structure required to use these methods.

Second, the sum of the state spaces' sizes of the component models must be smaller than the size of state space of the combined model for Kronecker-based methods to be advantageous. This requires the submodels to be approximately the same size.

Third, Kronecker-based methods have generally been limited to the Power or Jacobi methods, both of which usually exhibit poor convergence behavior. This is particularly undesirable because large systems of equations tend to exhibit worse convergence characteristics than small systems. A notable exception is the work of Ciardo [4], who presents algorithms for doing a Gauss-Seidel iteration, although we are unaware of any tool that uses them.

In contrast, we develop methods in this paper that can be used with all variants of stochastic Petri nets, regardless of the structure of the model. We develop new techniques that permit the use of more general iterative methods, which often converge more quickly. We do this in three ways. First, we develop algorithms that can generate, on-the-fly, the required incoming and outgoing transition rates from a state. In particular, we give two algorithms: one for standard stochastic Petri nets (SPNs) and one for generalized stochastic Petri nets (GSPNs) [12]. We assume the reader has a basic understanding of Markov models, state space generation, and basic iterative solution techniques.

Second, since the generation of the state-transition-rate matrix on-the-fly takes significantly more time than doing an iteration with the matrix in memory, we develop a new iterative solution method that exhibits locality in its use of data from the state-transition-

rate matrix. This algorithm, which we call *modified adaptive Gauss-Seidel* (MAGS), reuses generated rows and columns in the state-transition-rate matrix within an iteration in order to reduce the performance penalty incurred by their generation.

Third, any solution algorithm based on Gauss-Seidel, such as SOR, requires access to rows of A in order to solve $Ax = b$, which corresponds to accessing the incoming rates of a state in the corresponding Markov model. We describe a new approach that only needs to compute outgoing (columns), not incoming (rows), rates, at the cost of having to keep two vectors of size equal to the number of tangible reachable states: one vector for the solution and another additional vector. This approach can be used with any solution method that is based on Gauss-Seidel, such as SOR or adaptive Gauss-Seidel.

These three contributions, namely on-the-fly rate generation, MAGS, and column Gauss-Seidel, are somewhat orthogonal in that they are independent contributions that can be applied in other contexts. For example, solutions based on Kronecker operators could benefit from both MAGS and column Gauss-Seidel. However, each contribution enhances the other, and taken as a whole, they present a new solution technique that addresses all restraining aspects of computing a solution to models that are otherwise intractable.

The remainder of the paper is organized as follows. Section II presents algorithms for computing incoming and outgoing transition rates for SPN and GSPN models. These algorithms are the core of our on-the-fly solution methods, since they compute the needed rows and columns of the state-transition-rate matrix directly from the net representation, without requiring explicit storage of the matrix in memory or on disk. Section III then presents a new iterative solution algorithm that exhibits locality in its access to rows and columns of the state-transition-rate matrix. Then, Section IV introduces a new approach that avoids the need to have incoming transition rates for Gauss-Seidel-based iterative methods, at the expense of keeping one additional vector of size equal to the number of states in the model. This technique can easily double the speed of a solution if sufficient memory is available. Finally, Section V presents some empirical results from a prototype implementation of the method.

II Forward/Backward Access Algorithms

The first class of algorithms we develop makes use of both incoming and outgoing state transition rates. In this section, we show two algorithms: one for SPN models, and one for

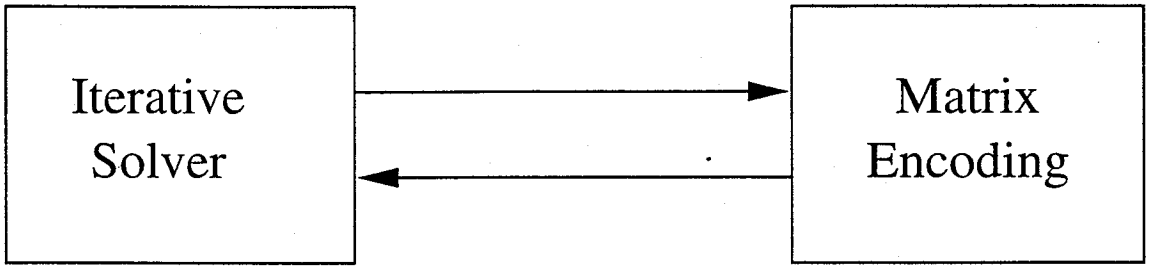

Figure 2.1: Solution Paradigm.

GSPN models.

Before proceeding, we will introduce some helpful notation. In particular, we will address the solution of a system of simultaneous linear equations written as $\pi Q = 0$, where π is a row vector and Q is the state-transition-rate matrix. Since we focus on numerical solution techniques, we adopt the notation $Ax = b$, or more precisely $Ax = 0$, where $A = Q^T$ and $x = \pi^T$. Here, the off-diagonal i-th row elements of Q represent outgoing rates of state i in the corresponding Markov chain, and the off-diagonal column elements of A represent the same. Similarly, the off-diagonal i-th column elements of Q and the off-diagonal i-th row elements of A represent the incoming rates to state i in the corresponding Markov chain.

To facilitate understanding our new approach, we present in Figure 2.1 a simple paradigm for viewing the solution process. Instead of viewing the matrix as data, we view it as a function returning the requested portion of the matrix. Hence, when the matrix is stored explicitly in memory, the function may be quite trivial and efficient in terms of computation, but costly in terms of memory consumption. In this paradigm, the superposed GSPN methods use Kronecker operators on smaller matrices and a mapping function to generate an element of A. Thus, accessing an element of A requires more computation, but (usually) less memory. Kronecker-based methods have the disadvantage of requiring a special structure in the model in order to work efficiently. In contrast, our methods act directly on the net representation to generate a row or column of A. This requires significant computation, but it will work with any model and will always take memory proportional to the size of the model.

More specifically, let s_i represent an encoding of the i-th state of the model. The encoding may be a simple concatenation of the bit encoding of the number of tokens in places, or a more sophisticated encoding suggested by Kemper [10, 11], called a *mix*. The encodings of all the states in the model form the set $S = \{s_1, s_2, \ldots, s_n\}$ (computed initially

by a state space search). To compute the i-th column of A, we take the state encoding s_i with the model and compute the successor states and the rates to those states. The significant computational requirements to compute a column in A are to

1. Decode s_i
2. Determine all enabled timed transitions in s_i
3. Fire all enabled transitions and possibly search a network of immediate transitions to determine the rate to each successor state j
4. For each successor state j:
   (a) Encode s_j
   (b) Search for s_j in S to determine index j

If we must do a binary search to look for an element in S (for the most efficient use of space), the most expensive operation is probably 4 (b), which takes time $O(\log n)$, where n is the number of states in the model. If we are willing to use more memory, we may use a hash table to do the lookup in $O(1)$ time. Since the problems we are addressing are limited by the memory of the machine, we must be careful how we use the memory.

Generating a column of A is therefore straightforward, but accessing A only by columns limits our choice of iterative methods to Jacobi or the Power method (unless the new approach from Section IV is used). In order to use the more powerful Gauss-Seidel method, or variants of it, we need to have access to rows of A. To illustrate the need for access to rows of A, consider the basic action in the Gauss-Seidel method that we call a Gauss-Seidel *step*:

$$x_i^{(k+1)} = \frac{-1}{a_{ii}} \left(\sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} + \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} - b_i \right) \tag{2.1}$$

where $x^{(k)}$ is the solution vector after k iterations. Doing a Gauss-Seidel step for i from 1 to n is called an *iteration*. In order to explicitly do the summation as shown above, one must have access to a row of A. Entries in the i-th row correspond to the incoming rates from predecessor states, so the task is to find the predecessor states and the corresponding incoming rates. Finding the diagonal element, a_{ii} in Equation 2.1, is also non-trivial, since it is defined as the negative sum of the outgoing rates. In general, to compute a_{ii} we must

```
/* Return the vector of off-diagonal row i */
a = 0
for each t ∈ T_i^{-1} do
    s_j <-r(t)- s_i
    if s_j ∈ S
        a_j = a_j + r(t)
return a
```

Figure 2.2: Algorithm to get i-th column for SPN model.

compute the outgoing rates and sum them. In the following, we describe how to compute the rows of A for SPN and GSPN models.

To find the off-diagonal elements of the i-th row of A, we must know the predecessor states and incoming rates. In a model, this corresponds to finding the set of states that lead to the state s_i. The approach we take to finding these predecessor states is basically to execute the model one step "backwards" in time.

A SPNs

For SPNs, by which we mean Petri nets (with no inhibitor arcs) and exponentially timed transitions, the algorithm is simple. To understand it, we first introduce the notion of a reverse model. A *reverse model* is the corresponding model where the directions of all the arcs have been reversed. The firing rules are the same except that any marking-dependent rates are determined *after* a transition fires. We let T_i be the set of (timed) transitions enabled in a marking or state i, and T_i^{-1} be the set of transitions enabled in the reverse model. The notation $s_i \xrightarrow{r(t)} s_j$ means state i goes to state j with rate $r(t)$ by firing transition t. Similarly, $s_j \xleftarrow{r(t)} s_i$ means state i goes to state j with rate $r(t)$ in the reverse model, or, equivalently, state j goes to state i with rate $r(t)$ in the forward model. The symbol S denotes the reachable set of states. The algorithm for computing the non-diagonal row entries is shown in Figure 2.2.

To perform a Gauss-Seidel step on x_i, we need to access the i-th row of A, including the diagonal element a_{ii}. To compute the diagonal, we must also compute the i-th column vector. The necessity of computing both the i-th row and the i-th column presents an additional significant cost to computing the row vector. In Section III we show how we can

make use of the already computed column vector in a more powerful iterative technique. In Section IV we show how to perform Gauss-Seidel by *only* accessing A by columns.

The SPN modeling paradigm is simple, but modeling complex systems with simple SPNs is difficult. We present this algorithm because SPNs are simple and fast. This also gives a framework on which we can build more complex algorithms.

B GSPNs

The procedure for computing the outgoing states and rates for a GSPN model is a straightforward extension of SPNs and is generally well known. However, it is less trivial to compute the incoming states and rates, or correspondingly, the i-th off-diagonal row elements of A. Figure 2.3 shows the algorithm we propose to do this. This algorithm allows for general marking-dependent rates and weights, so we can replace inhibitor arcs with transitions with marking-dependent rates or weights. The new notation is as follows: T_i is the set of transitions enabled in state s_i, T_i^{-1} is the set of transitions enabled in the reverse model in state s_i, T^{-1} is a set containing transitions enabled in the reverse model that have become enabled exclusively by the firing of some immediate transition, and $s_j \xleftarrow{w(m)} s_i$ means s_i goes to s_j by firing a single immediate transition m with weight $w(m)$ in the reverse model. I_i is the set of immediate transitions enabled in state s_i in the forward model, and I_i^{-1} is the set of immediate transitions enabled in state s_i in the reverse model.

The algorithm consists of two basic procedures that correspond to searching timed and immediate transitions. At a high level, we simply reverse the directions of the arcs and search all paths involving the firing of any number of immediate transitions followed by the firing of a timed transition. The algorithm we present does this in an organized way.

In particular, the algorithm starts by searching predecessor states reached by firing timed transitions in the reverse model. Those are the states that lead to the current state by firing only a single timed transition. After those are searched, T^{-1} is set to $\{\}$. Transitions are added to T^{-1} only as they become enabled by firing an immediate transition in the reverse model. An intuitive explanation for this is that in the forward model, a stable marking goes to a stable marking by firing a timed transition followed by a number of immediate transitions. Therefore, if we trace the same path backwards in the reverse model, the path can not end with the firing of a timed transition that does not become enabled by the firing of immediate transitions along the path. We can avoid examining many vanishing states

/* Return the vector of off-diagonal column i */
$a = 0$
for each $t \in T_i^{-1}$ do
    $s_j \xleftarrow{r(t)} s_i$
    if $s_j \in S$
        $a_j = a_j + r(t)$
set $T^{-1} = \{\}$
call `search_back_im`(s_i, 1)

procedure `search_back_im`(s_i, r)
for each $m \in I_i^{-1}$
    $s_j \xleftarrow{w(m)} s_i$ (update T^{-1})
    $\hat{r} = r \times w(m) / \sum_{\forall k | s_j \xrightarrow{\hat{w}(k)} s_k} \hat{w}(k)$
    for each $t \in T^{-1}$ do
        $s_k \xleftarrow{\bar{r}(t)} s_j$
        if $I_k = \{\}$ and $s_k \in S$
            $a_k = a_k + \hat{r}\bar{r}(t)$
    call `search_back_im`(s_j, $\hat{r}$)

Figure 2.3: Algorithm to get i-th column for GSPN model.

this way and therefore prevent unnecessary computation.

The `search_back_im` procedure recursively searches through the network of immediate transitions. After an immediate transition is fired in the reverse model, we determine the probability $\hat{r}$ and try firing each $t \in T^{-1}$ to see if it results in a stable marking. We have found that maintaining I_i can be done efficiently and can prevent unnecessary searching in S for a vanishing marking (which is usually computationally more expensive).

Figure 2.3 shows the basic algorithm, but there are some possible improvements. We noted above that inhibitor arcs are a special case of marking-dependent values, which is the simplest way to deal with them. We could also build static data structures that can tell us if a transition in the reverse model is "inhibited," that is, that there is no need to fire a transition in the reverse model because it would result in a state where that transition is inhibited in the forward model.

III Numerical Solution Methods That Exhibit Locality

Given algorithms to generate desired row/columns of A on-the-fly, we need iterative methods to solve $Ax = 0$ for the non-trivial solution of x. Typically, A is very sparse, so a vector multiplied by a row (or column) of A requires few operations. For superposed GSPN methods and the methods we present here, the time to compute a row or column of A is much greater than the time to do a vector-vector multiply, so we would like to have iterative techniques that can re-use the row (or column) of A as much as possible within a single iteration. In this section, we will present a method that has this property. Informally, the strategy is to generate a sequence of rows and columns, store them in a software cache, re-use that part of the matrix as long as it is useful, and then discard the sequence, generate a new sequence, and continue. We are willing to do more work in the solution process in return for fewer accesses to the matrix in order to speed up the overall time to compute the solution.

Adaptive Gauss-Seidel Modified adaptive Gauss-Seidel (MAGS) is an extension to adaptive Gauss-Seidel [8, 9] that exhibits locality. To motivate its formulation, we first review adaptive Gauss-Seidel. Adaptive Gauss-Seidel (AGS) is based intuitively on the observation that some elements sometimes converge or change more quickly than others, that is, $|x_i^{(k+1)} - x_i^{(k)}| > |x_j^{(k+1)} - x_j^{(k)}|$. If this is true, then a Gauss-Seidel step on x_i is considered

more *effective* than a Gauss-Seidel step on x_k, and therefore more work should be done on x_i. The intuition is that x_i is getting to the solution faster, so we should do steps on it more frequently. AGS is thus a variant of Gauss-Seidel where Gauss-Seidel steps are not necessarily performed in sequential order. Adaptive Gauss-Seidel is based on the methods of Rüde [16], for which he shows rigorously the effectiveness of the algorithm for the case where A is symmetric, positive definite. Since A is not symmetric or positive definite for Markov models, we use AGS as a heuristic. Our belief in its effectiveness is based on the fact that Horton [9] shows empirically that AGS needs significantly fewer floating point operations than standard point Gauss-Seidel to solve certain Markov models to the same accuracy.

Heuristically, if we do a Gauss-Seidel step on element i and we find that $|x_i^{(k+1)} - x_i^{(k)}|$ is large, then we have done effective work on element i. Because the change in x_i is large, we should also do work on states whose occupancy probability directly depends on x_i, since they too could change significantly. These states are the successor states of state i in the corresponding Markov chain and are also the non-zero off-diagonal elements of the i-th column of A. For simplicity, we quantify effectiveness by a single number ϵ, and if $|x_i^{(k+1)} - x_i^{(k)}| > \epsilon$, then we should also do work on the successors of state i. In Section II, we noticed that in order to compute a_{ii}, we need to compute the outgoing rates of state i. This heuristic can take advantage of this by noticing the successor states of s_i (i.e., the non-zero entries of the i-th column). Now we may begin to formulate the basis of an algorithm based on these observations.

In particular, let M be the set of states on which we need to perform work, which is initially set to S. Figure 2.4 shows the algorithm in detail for a given ϵ. The algorithm continues until M is empty. We call this one AGS iteration. The strategy to get a solution efficiently is to pick an initial large ϵ_0, call `AGS`, and then repeat the process with a successively smaller ϵ.

The way to decrease ϵ at each iteration is a difficult problem. Horton [8, 9] proposes decreasing it by a multiplicative constant $\Delta\epsilon$, shown here.

$\epsilon = \epsilon_0$
while not converged
    `AGS(`ϵ`)`
    $\epsilon = \epsilon \times \Delta\epsilon$

```
procedure AGS(ε)
M = s_1, ..., s_n
while M ≠ {}
    choose state s_i ∈ M
    M = M \ {s_i}
    t = x_i
    Gauss-Seidel_Step(i)
    if |t − x_i| > ε
        for all j ≠ i, a_ji ≠ 0
            M = M ∪ {s_j}
```

Figure 2.4: Adaptive Gauss-Seidel iteration.

Choosing a good $\Delta\epsilon$ is also difficult. If we choose a value near one, it makes MAGS work like normal Gauss-Seidel. If $\Delta\epsilon$ is too small, fast-changing elements may start to converge to the wrong values, resulting in unnecessary work. Horton suggests values between 0.5 and 0.1, and our experimentation shows that these values are good for our modification to AGS as well. The convergence criteria could be any of the known criteria, or it could be a sufficiently small ϵ. This seems to be at least as good as the commonly used $||x^{(k+1)} - x^{(k)}||$ method.

Modified Adaptive Gauss-Seidel Although AGS may speed convergence, since it works on states according to an "effectiveness" criterion, it does not ensure any kind of locality for data re-use. In particular, we note that AGS does not specify which state should be removed from M. We have modified the algorithm to narrow the choices in order to create locality. Specifically, we modify AGS by adding another set C, which is used to represent a software cache of M. The set C has two types associated with each element: *activated* and *deactivated.* We modify AGS by first limiting our working set to C, and when we would add s_i to M, we instead first check whether $s_i \in C$, and if it is, activate s_i; otherwise we add s_i to M. The algorithm for modified AGS (MAGS) is given in Figure 2.5.

In practice, the order in which we choose elements from M or C plays a very significant role in the convergence characteristics. Experience has shown that the best convergence occurs when elements are chosen from C or M in a breadth first order. Experience has also shown that MAGS, while it is a valid implementation of AGS, does not usually perform as well as Horton's implementation of AGS. This tells us that the convergence characteristics

```
procedure MAGS(ε)
M = s_1, ..., s_n
while M ≠ {}
    C ⊂ M
    M = M \ C
    while there exists an active element in C
        choose an active s_i ∈ C
        deactivate s_i in C
        t = x_i
        call Gauss-Seidel_Step(i)
        if |t − x_i| > ε
            for all j ≠ i, a_ji ≠ 0
                if s_j ∈ C then activate s_j in C
                else M = M ∪ {s_j}
```

Figure 2.5: Modified adaptive Gauss-Seidel iteration.

of AGS are very dependent on the order in which elements are removed from M or C. In the worst performance, MAGS performs roughly as well as Gauss-Seidel.

IV Forward Solution Methods

The complexity in applying the above iterative solution techniques comes because they are based on Gauss-Seidel iteration steps, and hence require row access to A. One can avoid accessing rows, but this restricts solution techniques to the Jacobi or Power methods. In the two previous sections, we showed how to solve models using Gauss-Seidel-like methods by computing the predecessor states. Finding predecessor states requires more computation per iteration than finding successor states, but allows the use of iterative methods that typically converge with fewer iterations.

In this section, we show how, with a little additional work and memory requirements identical to those for the Jacobi method (one additional vector the size of a solution vector), we can also perform Gauss-Seidel-based methods, and yet require only the computation of successor states. This result is very important, since it shows that if one can use the Jacobi method, then with little additional work and no additional memory, one can use Gauss-Seidel-based iterative methods. If we have the memory to hold a second vector of size equal to the number of states, we can perform all iterative solution techniques that are based on

Gauss-Seidel iteration steps without the cost of computing predecessor states. When this is done, we can compute solutions on-the-fly to more expressive modeling paradigms, such as stochastic activity networks (SANs) [13, 14] and stochastic reward networks (SRNs) [5]. Although the method works for all iterative solution techniques based on Gauss-Seidel steps, we develop it in terms of standard Gauss-Seidel first, and then show how it can be used in more sophisticated variants, such as SOR and modified adaptive Gauss-Seidel.

A Column Gauss-Seidel

To understand how we can eliminate the need for row access, we recall that the basic operation in many Gauss-Seidel-based iteration schemes is the Gauss-Seidel step, given in (2.1). By using this step as the basic unit of computation, we can seamlessly replace Gauss-Seidel steps with the new variant, which requires only column (successor) access in other iterative methods.

We introduce our strategy with a vector δ, which we define as

$$\delta_i = x_i^{(k+1)} - x_i^{(k)},$$

so that a Gauss-Seidel step on element i is equivalent to setting $x_i^{(k+1)} = x_k^{(k)} + \delta_i$. We show how to initialize δ, and then given δ, we show how doing a Gauss-Seidel step on element i affects δ_j for all $j \neq i$.

In particular, let $x^{(1)}$ be some initial guess. We initialize δ by the following:

$$\begin{array}{l}
\delta = 0 \\
\text{for } i = 1 \text{ to } n \\
\quad \text{for } j = 1 \text{ to } n | j \neq i \\
\quad\quad \delta_j = \delta_j + a_{ji} x_i^{(1)} \\
\text{for } i = 1 \text{ to } n \\
\quad \delta_i = (b_i - \delta_i)/a_{ii} - x_i^{(1)}
\end{array}$$

This essentially does a Jacobi iteration and places $x^{(2)} - x^{(1)}$ in δ. This is what we want because if we choose to start Gauss-Seidel at x_i, then $x_i^{(2)} = x_i^{(1)} - \delta_i$. (The first Gauss-Seidel step is identical to the first Jacobi step.)

Now we may do a Gauss-Seidel step on any element by simply doing the computation $x_i^{(2)} = x_i^{(1)} + \delta_i$. Once we do the computation, however, δ_j is in general obsolete. We now

show how to update δ_j after each Gauss-Seidel step. Say we do a Gauss-Seidel step on x_i, in the most general form

$$x_i^{(k)} = \frac{1}{a_{ii}} \left(-\sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^* + b_i \right) ,$$

where x_j^* is the most recently computed value of x_j. After this step, $\delta_i = 0$. Now say we do step p on x_c, and then observe the effects that this computation has on δ_i.

$$x_c^{(p+1)} = x_c^{(p)} + \delta_c$$

$$x_i^{(k+1)} - x_i^{(k)} = \frac{-1}{a_{ii}} \left(a_{ic} x_c^{(k+1)} - a_{ic} x_c^{(k)} \right)$$

Finally,

$$\delta_i = \frac{-a_{ic}\delta_c}{a_{ii}} .$$

Now let us not assume $\delta_i = 0$. We denote δ_i^0 as the value of δ_i before performing a Gauss-Seidel step on x_c. Inductively, we can show that after we do a Gauss-Seidel step on x_c, we can compute the new δ_i from the value of δ_i^0 and δ_c.

$$x_i^{(k+1)} - x_i^{(k)} = \delta_i^0 + \frac{-1}{a_{ii}} \left(a_{ic} x_c^{(k+1)} - a_{ic} x_c^{(k)} \right)$$

$$\delta_i = \delta_i^0 - \frac{a_{ic}\delta_c}{a_{ii}} \tag{2.2}$$

Now we can see that updating δ_i after performing a Gauss-Seidel step on x_c requires access to the c-th column of A. In addition, computing δ_i also needs a_{ii}, but this dependency is easy to eliminate. If we let $d_i = \delta_i a_{ii}$, and d_i^0 is the value of d_i before performing a Gauss-Seidel step on x_c, then

$$d_i = d_i^0 - a_{ic}\delta_c .$$

Then, when doing the Gauss-Seidel step on x_i, simply divide d_i by a_{ii} and update all $d_j | a_{ji} \neq 0$. We now have everything necessary to perform a Gauss-Seidel step on x_i by accessing only the i-th column of A.

Successive Over-Relaxation We now show how to easily extend this method to Successive Over-Relaxation (SOR). Recall the basic step for SOR:

$$x_c^{(k+1)} = \omega \bar{x}_c^{(k+1)} + (1-\omega) x_c^{(k)} \ ,$$

where $\bar{x}_i$ is the Gauss-Seidel iterate. We computed above

$$\bar{x}_c^{(k+1)} = x_c^{(k)} + \delta_c \ ,$$

and by substitution,

$$x_c^{(k+1)} = x_c^{(k)} + \omega \delta_c \ .$$

The updating of δ_i, $\forall i \neq c$ is done by (2.2). From this, we can see that the column-only SOR step involves only a minor extension to the column-only Gauss-Seidel.

Algorithm Figure 2.6 shows the algorithm for doing a Gauss-Seidel step using only column access. We show the algorithm with the feature of over-relaxation parameter ω, which is set to 1 for standard Gauss-Seidel, but in general can take on values $\omega \in (0, 2)$. Notice that the column Gauss-Seidel step procedure accesses only the i-th column of A. Consequently, we can perform any Gauss-Seidel-based step on element x_i using on-the-fly matrix generation solely by computing the successor states and corresponding rates of state s_i.

As an example of the use of this implementation of a Gauss-Seidel step, we show an implementation of standard Gauss-Seidel.

```
x = initial guess
call Gauss-Seidel_Step_Init()
while x not converged
    for i = 1 to n
        call cGauss-Seidel_Step(i)
```

We call this algorithm *column Gauss-Seidel.* Notice that we can do column Gauss-Seidel with the same memory requirements as Jacobi, and after an initialization cost, the same number of operations per iteration as Jacobi and Gauss-Seidel.

Recall that the diagonal element a_{ii} is the negative sum of the off-diagonal elements in the i-th column. Performing a Jacobi iteration while accessing A by columns requires two basic steps. In the first step, we do the matrix-vector multiply $x^{(k+1)} = (A - D)x^{(k)}$, where D is the diagonal matrix of A. This step accesses the off-diagonal elements of A.

```
/* Matrix A ∈ R^{n×n} */
/* arrays x, b, and d ∈ R^n */
/* Solve Ax = b using d. */
procedure cGauss-Seidel_Step_Init()
d = 0
for i = 1 to n
    for j = 1 to n| j ≠ i
        d_j = d_j + a_ji x_i
for i = 1 to n
    d_i = (b_i − d_i/a_ii − x_i)a_ii

procedure cGauss-Seidel_Step(int i)
δ = ωd_i/a_ii
x_i = x_i + δ
for j = 1 to n| j ≠ i
    d_j = d_j − a_ji × δ
```

Figure 2.6: Gauss-Seidel step requiring only column access to A.

The next step does $x^{(k+1)} = D^{-1}x^{(k+1)} + D^{-1}b$, which accesses the diagonal elements of A. If A is encoded, the two steps require two sweeps of A, one for the off-diagonal elements and one for the diagonal elements. The alternative is one sweep of A and explicit storage of the diagonal elements. Two sweeps of A would substantially decrease performance, and explicit storage of D requires additional memory of the same size as x. Therefore, the Jacobi method requires two sweeps of the matrix per iteration with $|A| + 2n$ memory, or one sweep per iteration with $|A|+3n$ memory. Column Gauss-Seidel, on the other hand, only needs $|A| + 2n$ memory and a single sweep of the matrix per iteration. Thus, for on-the-fly techniques, column Gauss-Seidel takes either less work or less memory than Jacobi.

Furthermore, column Gauss-Seidel has some improved numerical properties relative to Gauss-Seidel or Jacobi. As the iteration process approaches the solution, the algorithm keeps d to full precision, even while variations in x are small. Column Gauss-Seidel may thus proceed as if x were kept to greater precision, because all the important information about x (namely $x^{(k+1)} - x^{(k)}$) is stored in d; this is useful when elements in x vary in size by many orders of magnitude and the user requires a high degree of accuracy. The algorithm is not self-correcting, however. If somehow (due to rounding errors, for example) x is perturbed, the algorithm will converge to the wrong answer. An easy solution to this is

to reinitialize d when the iteration process is near the solution, or after every several digits of accuracy acquired.

B Column Modified Adaptive Gauss-Seidel and Block Gauss-Seidel

As mentioned earlier, the approach used to obtain column-only Gauss-Seidel can be extended to all Gauss-Seidel-based algorithms. In this case of modified adaptive Gauss-Seidel, this is very straightforward; the routine `cGauss-Seidel_Step` is a direct replacement for the routine `Gauss_Seidel_Step`. This substitution results in an algorithm that can solve any model class, especially SAN or SRN models, with much greater speed than the variant that requires row access. The cost of using column-only Gauss-Seidel is some extra time spent in initialization (negligible) and the extra memory to hold d. If this memory is available, the column-only variant should be used.

V Prototype Implementation Performance Comparison

We have made a prototype implementation to compare the speed of our technique to that of existing solvers based on Kronecker methods. As is typically the case with such comparisons, the prototypical nature of each implementation makes it very difficult to fairly compare performance of different methods. The use of different computing platforms and differences in the algorithms themselves make a comparison difficult. For example, Kemper has reported the time for a single Jacobi iteration for a particular model, where an iteration is defined as a sweep through the entire state space. Our new method (MAGS) uses a different notion of iteration. The algorithm does a dynamic number of basic Gauss-Seidel steps, that is determined by certain parameter values and an depending on an "effectiveness" criterion, rather than by a simple sweep through the state space. Furthermore, recently used rates are cached using our technique, and we expect that each operation in our solution technique will be more effective in leading the solver to convergence.

In spite of these difficulties, we will try to make a comparison between the Kronecker-based implementation by Kemper and our prototype on-the-fly method implementation. In doing so, we make the assumption that the time taken by the iterative solver in actually performing vector-vector multiplications is negligible relative to the cost of generating the required data for both the Kronecker and on-the-fly methods. This is reasonable, since both methods perform many more operations in obtaining the required rates than in using them.

Our computer (a 120 MHz Hewlett-Packard Model C110) can perform Gauss-Seidel with over-relaxation by accessing A at a rate of 50 MB/s, for example, if the state-transition-rate matrix is stored explicitly in memory.

Since the methods have a different notion of iteration and use the obtained rates very differently (our algorithm should typically converge with fewer iterations, using a consistent notion of iteration), it is not possible to compare iteration costs. Instead, we compare data generation rates of the two implementations. To calculate the data generation rate for Kemper's implementation, we divide the time to do an iteration by the number of non-zero entries in the matrix, and we obtain a data generation rate of approximately 700 Kbytes/second on an 85 MHz Sparc 4 machine. Measurements of an SPN model on our machine show that we can generate data at a rate of about 440 KByte/second.

We guess that our machine is roughly three times faster than the one Kemper used, resulting in a data generation rate for the Kronecker-based implementation that we guess is five times faster than the on-the-fly methods. Thus (as might be expected), it takes longer to generate data for a completely general model representation than for one that exhibits the special structure needed for Kronecker-based solution methods. However, we reuse data that is generated (in a small, fixed-size cache) and use a solution algorithm that should be faster than Jacobi for most models. Thus, the solution times for the prototype on-the-fly implementation are roughly similar to those for Kronecker-based methods, and since our methods are applicable to a much wider class of models and can use more effective iterative solvers, it is reasonable to use them for models that do not exhibit the special structure necessary for the Kronecker approach.

VI Conclusion

We make three important contributions in this paper. First, we propose a technique that can solve very general models of approximately the same size as Kemper and Ciardo [6, 10, 11], using approximately the same amount of memory. Instead of requiring the special model structure needed for a Kronecker-based solution, we generate the required row and/or columns of the state-transition-rate matrix on-the-fly, allowing for a completely general structure. In particular, we develop algorithms for doing this for SPN and GSPN models.

Second, we recognized that for all such methods (both ours and Kronecker-based ones),

generation of rates from the state-transition-rate matrix is much slower than the iterative solution process. To account for this, we developed a new iterative solution process, called modified adaptive Gauss-Seidel, that exhibits locality in its use of rates from the state-transition-rate matrix, and hence does not require as high a data generation rate to be competitive. Because of this, we can cache recently generated rates in memory, minimizing the bottleneck of rate generation.

Third, we showed how to solve $Ax = b$ using Gauss-Seidel and variants by accessing A only by columns. This method is no more computationally expensive than Gauss-Seidel (except for initialization, which is a very small part of the cost) and takes no more (or even less) memory than Jacobi. This result is very important, since it shows that if we have the memory to hold a second vector of size equal to the number of states in the model, (the same requirement that the Jacobi method has,) we can perform all iterative solution techniques that are based on Gauss-Seidel iteration steps without the cost of backward execution.

Finally, we compare the performance of a prototype implementation of our method to a prototype implementation using the Kronecker approach. We saw that our implementation generates data (transition rates) at a speed about five times slower than the Kronecker-based prototype did. However, we believe that the faster convergence rate of MAGS and the locality of its use of rate information, combined with column-only Gauss-Seidel, makes the on-the-fly approach viable for the solution of large models that can not be solved by the Kronecker approach.

REFERENCES

[1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, PA, 1994.

[2] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Hadaad, "On well-formed coloured nets and their symbolic reachability graph," in *Proc. Eleventh Conference on Application and Theory of Petri Nets*, Paris, France, June 1990. Reprinted in K. Jensen and G. Rozenberg, ed., *High-Level Petri Nets. Theory and Application*, Springer Verlang, 1991.

[3] G. Chiola and G. Franceschinis, "Colored GSPN models and automatic symmetry detection," in *Proc. Third Int. Workshop on Petri Nets and Performance Models (PNPM'89)*, pp. 50–60, Kyoto, Japan, Dec. 1989.

[4] G. Ciardo, "Advances in compositional approaches based on Kronecker algebra: Application to the study of manufacturing systems," in *Third International Workshop on Performability Modeling of Computer and Communication Systems*, pp. 61–65, Bloomingdale, IL, Sept. 7–8, 1996.

[5] G. Ciardo, A. Blakenmore, P. F. J. Chimento, J. K. Muppala, and K. S. Trivedi, "Automatic generation and analysis of Markov reward models using Stochastic Reward Nets," in C. Meyer and R. J. Plemmons, ed., *Linear Algebra, Markov Chains, and Queueing Models*, vol. 48 of *IMA Volumes in Mathematics and its Applications*, pp. 141–191, Springer-Verlag, 1993.

[6] G. Ciardo and M. Tilgner, "On the use of Kronecker operators for the solution of generalized stochastic Petri nets," ICASE Report #96-35 CR-198336, NASA Langley Research Center, May 1996.

[7] S. Donatelli, "Superposed generalized stochastic Petri nets: Definition and efficient solution," in R. Valette, ed, *Application and Theory of Petri Nets 1994, Lecture Notes in Computer Science 815 (Proc. 15th Int. Conf. on Application and Theory of Petri Nets, Zaragoza, Spain)*, pp. 258–277, Springer-Verlag, June 1994.

[8] G. Horton, "Adaptive relaxation for the steady-state analysis of Markov chains," in William J. Stewart, ed., *Computations with Markov Chains*, pp. 585–586, Kluwer Academic Publishers, Boston, 1995.

[9] G. Horton, "Adaptive Relaxation for the Steady-State Analysis of Markov Chains," ICASE Report #94-55 NASA CR-194944, NASA Langley Research Center, June 1994.

[10] P. Kemper, "Numerical analysis of superposed GSPNs," in *Proc. Int. Workshop on Petri Nets and Performance Models (PNPM'95)*, pp. 52–61, Durham, NC, Oct. 1995.

[11] P. Kemper, "Numerical Analysis of Superposed GSPNs," in *IEEE Transactions on Software Engineering*, vol. 22, no. 9, Sept. 1996.

[12] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franseschinis, *Modeling with generalized stochastic Petri nets*, John Wiley & Sons, 1995.

[13] J. F. Meyer, A. Movaghar, and W. H. Sanders, "Stochastic activity networks: Structure, behavior, and application," In *Proc. International Workshop on Timed Petri Nets*, pp. 106–115, Torino, Italy, July 1985.

[14] A. Movaghar and J. F. Meyer, "Performability modeling with stochastic activity networks," In *Proc. 1984 Real-Time Systems Symp.*, pp. 215–224, Austin, TX, Dec. 1984.

[15] M. A. Qureshi, W. H. Sanders, A. P. A. van Moorsel, and R. German, "Algorithms for the Generation of State-Level Representations of Stochastic Activity Networks with General Reward Structures," In *Proc. Int. Workshop on Petri Nets and Performance Models (PNPM'95)*, pp. 180–190, Durnham, NC, Oct., 1995.

[16] U. Rüde, "On the multilevel adaptive iterative method," in *Preliminary Proceedings of the Second Copper Mountain Conference on Iterative Methods*, April 9–14, 1992. Also in T. Manteuffel, ed., *SIAM J. Sci. Comput.*, 15, 1994.

[17] W. H. Sanders and J. F. Meyer, "Reduced Base Model Construction Methods for Stochastic Activity Networks," in *IEEE Journal on Selected Areas in Communications*, vol. 9, no. 1, pp. 25–36, Jan. 1991.

[18] W. J. Stewart, *Introduction to the Numerical Solution of Markov Chains*, Princeton University Press, 1994.

Chapter 3

AN EFFICIENT DISK-BASED TOOL FOR SOLVING VERY LARGE MARKOV MODELS

Abstract

Very large Markov models often result when modeling realistic computer systems and networks. We describe a new tool for solving large Markov models on a typical engineering workstation. This tool does not require any special properties or a particular structure in the model, and it requires only slightly more memory than what is necessary to hold the solution vector itself. It uses a disk to hold the state-transition-rate matrix, a variant of block Gauss-Seidel as the iterative solution method, and an innovative implementation that involves two parallel processes: the first process retrieves portions of the iteration matrix from disk, and the second process does repeated computation on small portions of the matrix. We demonstrate its use on two realistic models: a Kanban manufacturing system and the Courier protocol stack, which have up to 10 million states and about 100 million nonzero entries. The tool can solve the models efficiently on a workstation with 128 Mbytes of memory and 4 Gbytes of disk.

I Introduction

A wide variety of high-level specification techniques now exist for Markov models. These include, among others, stochastic Petri nets, stochastic process algebras, various types of block diagrams, and non-product form queuing networks. In most cases, very large Markov models result when one tries to model realistic systems using these specification techniques. The Markov models are typically quite sparse (adjacent to few nodes), but contain a large number of states. This problem is known as the "largeness problem." Techniques that researchers have developed to deal with the largeness problem fall into two general categories: those that avoid the large state space (for example, by lumping,) and those that tolerate the large state space (for example, by recognizing that the model has a special structure and storing it in a compact form). While many largeness avoidance and tolerance techniques exist, few are applicable to models without special structure. Methods are sorely needed that permit the solution of very large Markov models without requiring them to have special properties or a particular structure.

In this paper, we describe a new tool for solving Markov models with very large state spaces on a typical engineering workstation. The tool makes no assumptions about the underlying structure of the Markov process, and requires little more memory than that necessary to hold the solution vector itself. It uses a disk to hold the state-transition-rate

matrix, a variant of block Gauss-Seidel as the iterative solution method, and an innovative two-process implementation that effectively overlaps retrieval of blocks of the state-transition-rate matrix from disk and computation on the retrieved blocks. The tool can solve models with ten million states and about 100 million transitions on a machine with 128 Mbytes of main memory. The state-transition-rate matrix is stored on disk in a clever manner, minimizing overhead in retrieving it from disk. In addition, the tool employs a dynamic scheme for determining the number of iterations to perform on a block before beginning on the next, which we show empirically to provide a near optimum time to convergence. Solution time is typically quick even for very large models, with only about 20% of the CPU time spent retrieving blocks from disk and 80% of the CPU resources available to perform the required computation.

In addition to describing the architecture and implementation of the tool itself, we illustrate its use on two realistic models: one of a Kanban manufacturing system [2], and another of the Courier protocol stack executing on a VME bus-based multiprocessor [14]. Both models have appeared before in the literature, and are excellent examples of models that have very large state spaces for realistic system parameter values. In particular, both models have been used to illustrate the use of recently developed Kronecker-based methods [2, 7], and the Courier protocol has been used to illustrate an approximate method based on lumping [14]. Both numerical results and solution times are presented for each model and, when possible, compared to previously obtained values and solution times. In each case we can obtain an exact solution (to the desired precision) in significantly less time than previously reported using Kronecker-based methods. It is thus our belief that if sufficient disk space is available to hold the state-transition-rate matrix, our approach is the method of choice for exact solutions.

The remainder of the paper is organized as follows. First, in Section II, we address issues in the choice of a solution method for very large Markov models, comparing three alternatives: Kronecker-based methods (e.g., [7, 2]), "on-the-fly" methods [3], and the disk-based method that we ultimately choose. This section presents clear arguments for the desirability of disk-based methods if sufficient disk space is available. Section III then describes the architecture and implementation of the tool, describing solutions to issues we faced in building a practical implementation. Finally, Section IV presents the results of the use of the tool on the two models described earlier.

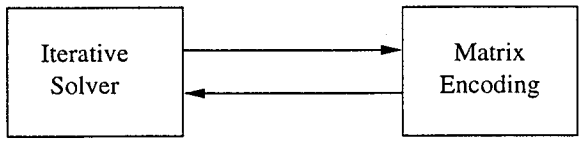

Figure 3.1: Solution paradigm.

II The Case for Disk-Based Methods

For our tool implementation, we considered three numerical solution techniques for tolerating large state spaces: Kronecker-based techniques, "on-the-fly" techniques, and disk-based techniques. To evaluate each method, we introduce a paradigm based on Figure 3.1. Here, we divide the numerical solution process into the iterative solver and the matrix encoding. The key to solving large matrices is to encode the matrix so that it takes little main memory (RAM), but still allows quick access to matrix elements. The iterative solver is thus a data consumer, and the matrix encoder is a data producer. We would like for both to be as fast as possible to obtain a solution quickly. An additional important factor is how effectively a particular iterative method uses the data it consumes. For example, certain iterative methods, such as Gauss-Seidel [12] and adaptive Gauss-Seidel [5], typically do more effective work with the same number of accesses to the matrix as Jacobi or the Power method, and hence do not require as high a data production rate to efficiently obtain a solution. We want to find a fast but general matrix encoding scheme and an effective iterative method with a low data consumption rate.

The first class of encoding schemes we consider are those of Kronecker-based methods. These methods require and make use of the fact that in certain models, particular parts of the model (called submodels) interact with one another in a limited way. One way to insure a model has this structure is to construct it according to a prescribed set of rules from smaller models, as is done, for example, in the case of stochastic automata networks [12]. If one follows these rules, one may easily express the transition rate matrix for the entire model as a function of Kronecker operators on the transition rate matrices of the submodels.

More recently there has been work on a type of model decomposition called superposed generalized stochastic Petri nets (SGSPNS) [1, 2, 7, 10, 11]. SGSPNs are essentially independent models that may be joined by synchronization of a transition. We believe [2] to be the state of the art in Kronecker operator methods, and although the more recent

techniques can solve a much larger class of models than originally proposed in [4], they are still restrictive in the models that they can effectively solve.

To evaluate the speed of the Kronecker operator methods, we observe rates in which the iterative solver and matrix encoding operate. We have observed that on our computer (120 MHz Hewlett-Packard Model C110), the SOR iterative solver can consume data at a rate of about 50 Mbyte/second. From numbers published by Kemper [7], we estimate that his implementation of the Kronecker-based method can produce data at a rate of 700 Kbyte/second on an 85 MHz Sparc 4, and we extrapolate that the rate would be about 2 Mbyte/second on our HP C110. Since both the data production and consumption require the CPU, the whole process will proceed at a rate of about 1.9 Mbyte/second. Kemper's method is also restricted to Jacobi or the Power method, which usually exhibit poor convergence characteristics, so the effectiveness of its use of generated data is low. Ciardo and Tilgner [2] present their own tool, but they do not present data in such a way that we can analyze the data generation rate. We can compare actual times to solution for their benchmark model, however, and do so in Section IV. Ciardo gives algorithms to perform Gauss-Seidel on a Kronecker representation in [1], but has not yet built a tool with which we can compare our approach.

The second class of encoding schemes we considered for implementation in this tool are "on-the-fly" methods introduced in [3]. On-the-fly methods have none of the structural restrictions of Kronecker-based methods, and they can operate on nets with general enabling predicate and state change functions, such as are present in stochastic activity networks [9, 10]. In addition, they can obtain a solution with little additional memory, or perhaps even less memory than needed by SGSPN solvers, while at the same time using Gauss-Seidel or variants. However, the prototype implementation described in [3] generates data at about 440 Kbyte/second on a HP C110. Although [3] introduces iterative methods that are usually more effective than Jacobi or the Power method in their use of data, the overall solution speed for these methods will be somewhat slower than for Kronecker-based methods, but still reasonable, given that they can be used without restrictions on the structure of a model.

The final class we considered was that of disk-based methods, where the workstation disk holds an encoding of the state-transition matrix. If we can find an iterative method that accesses data from the state-transition-rate matrix in a regular way and use a clever encoding, disks can deliver data to an iterative algorithm at a high rate (5 Mbyte/second or higher) with low CPU overhead. Furthermore, high performance disks are inexpensive

relative to the cost of RAM, so we would like to find a way to utilize disks effectively. Experimental results show that if we can do both disk I/O and computation in parallel, we can perform Gauss-Seidel at a rate of 5 Mbyte/second while using the CPU only 30% of the time. Thus disk-based methods have the potential to greatly outperform Kronecker and on-the-fly methods, at the cost of providing a disk that is large enough to hold the state-transition-rate matrix of the Markov model being solved. The challenge is to find a more effective solution method that has a data consumption rate of about 5 Mbytes/second at 80% CPU utilization.

Clearly, the method of choice depends on the nature of the model being solved, and the hardware available for the solution. If the state-transition-rate matrix is too large to fit on available disk space and the model meets the requirements of Kronecker-based methods, then they should be used. If the model does not fit on disk and does not meet the requirements of Kronecker-based methods, on-the-fly methods should be used. However, SCSI disks are inexpensive relative to RAM (in September 1996, approximately $1400 for 4 Gbyte fast wide SCSI), so space may inexpensively be made available to store the state-transition-rate matrix. Since a single disk can provide the high data production rate only for sequential disk access, the efficiency of disk-based methods will depend on whether we can find a solution algorithm that can make effective use of the data in the sequential manner. We discuss how to do this in the following sections.

III Tool Architecture and Implementation

In this section, we discuss the architecture of our tool and its implementation on an HP workstation. In particular, we discuss the basic block Gauss-Seidel (BGS) algorithm and how it maps onto a program or set of programs that run on a workstation. An important issue we solve is how to effectively do computation and disk I/O in parallel. We develop a flexible implementation with many tunable parameters that can vary widely on different hardware platforms and models.

The mathematics of BGS is generally well known (see [12], for example). We wish to solve for the steady state probability vector π given by $\pi Q = 0$. To review BGS briefly, partition the state-transition-rate matrix Q into $N \times N$ submatrices of (roughly) the same

size, labeled Q_{ij}. BGS then solves

$$\Pi_i^{(k+1)} Q_{ii} = -\left(\sum_{j=1}^{i-1} \Pi_j^{(k+1)} Q_{ji} + \sum_{j=i+1}^{N} \Pi_j^{(k)} Q_{ji} \right) \tag{3.1}$$

for Π_i for i ranging from 1 to N, where Π_i is the corresponding subvector of π. This is called the k-th BGS *iteration.* Solving for Π_i can be done by any method; our tool uses (point) Gauss-Seidel. One Gauss-Seidel iteration to solve (3.1) is called an *inner* iteration, and solving (3.1) for $1 \leq i \leq n$ is an *outer* iteration.

The sequential algorithm for a single BGS iteration follows directly. In particular, let $r \in \mathcal{R}^n$ be an auxiliary variable.

```
for i = 1 to N
    r = 0
    for j = 1 to N | j ≠ i
        r = r − Π_j Q_ji
    Solve Π_i Q_ii = r for Π_i
```

One can easily see that the access to Q is very predictable, so we may have blocks of Q ordered on disk in the same way that the program accesses them. This way the program accesses the file representing Q sequentially entirely throughout an iteration. One could then easily write an implementation of BGS and a utility to write Q appropriately to a file. What is not trivial is to build a tool that overlaps computation (the solution of $\Pi_i Q_{ii} = r$) and reading from disk in a flexible, efficient way.

Tool Architecture Our solution to this is to have two cooperating processes, one of which schedules disk I/O, and the other of which does computation. Obviously, they must communicate and synchronize activity. We use System V interprocess communication mechanisms since they are widely available and simple to use. For synchronization, we use semaphores, and for passing messages, such as a block of Q, we use shared memory. We call the process that schedules I/O the *I/O process*, and we call the process that solves $\Pi_i Q_{ii} = r$ the *compute process.* To minimize memory usage, we want to have as few blocks of Q in memory at one time as possible, so we must be careful how we compute the step $r = r - \Pi_j Q_{ji}$, $\forall j \neq i$. For simplicity, we assign the task of computing r to the I/O process.

We first looked at several large matrices that were generated by GSPN models. (We looked at GSPNs because they can easily create large transition rate matrices, not because

our solution technique is limited to them.) We noticed that the matrix is usually very banded; that is, for reasonable choices for N, the number of non-zero elements in the blocks $Q_{i,j} : |i-j| > 1$ is small, if not zero. By lumping all the blocks in a column into a smaller number (three) of larger blocks, we can eliminate the overhead of reading small or empty blocks. For the i-th column, we call $Q_{i,i}$ the *diagonal* block; $Q_{i-1,i}$ is the *conflict* block; all other blocks are lumped into a single block that we call the *off* block. We use the term *off block* because it includes all the off diagonal blocks except the conflict block. Let D represent the diagonal block, C the conflict block, and O the off block. The following represents a matrix where $N = 4$.

$$\left(\begin{array}{c|cc|c} D & \multicolumn{2}{c|}{O} & C \\ \hline C & D & \multicolumn{1}{c|}{O} & \\ \hline O & C & D & O \\ \hline \multicolumn{2}{c|}{O} & C & D \end{array}\right)^T$$

The reason we have a conflict block will be apparent soon.

Lumping several blocks into the off block complicates 3.1), but does not require any extra computation. The actual mechanics of the computation of $\Pi Q_{\text{off},i}$ are no different than for the computation of $\Pi_j Q_{ji}$. For the formula $r = \Pi Q_{\text{off},i}$, we compute $r = \sum_{k \neq i,i-1} \Pi_k Q_{ki}$. We may now compute r the following way:

$$\begin{aligned} r &= -\Pi Q_{\text{off},i} \\ r &= r - \Pi_{i-1} Q_{\text{conflict},i} \end{aligned}$$

Let us denote $r_i = \Pi_i Q_{ii}$ to distinguish between different r vectors. In order to make the computation and disk I/O in parallel, the program must solve $\Pi_i Q_{ii} = r_i$ while at the same time compute r_{i+1}. Therefore, while the compute process is solving $\Pi_i Q_{ii} = r_i$, the I/O process is prefetching $Q_{i+1,i+1}$, and reading $Q_{\text{off},i+1}$ and $Q_{\text{conflict},i+1}$ to compute r_{i+1}. Notice that when computing r_{i+1}, we need the most recent value of Π_i to multiply by $Q_{\text{conflict},i}$, which introduces a data dependency. Thus, we can not completely compute r_{i+1} while in parallel computing Π_i. (We could also use a less recent value of Π_i, but that would reduce the effectiveness of BGS.)

Finally, we add synchronization to ensure that the I/O process has the most recent version of Π_i to compute r_{i+1}. The full algorithm we use is presented in Figure 3.2. We used a large, shared memory array to represent the steady state probability vector Π, two shared diagonal block buffers Q_{diag0} and Q_{diag1}, and two r vectors r_0 and r_1. The processes

Shared variables: Π, Q_{diag0}, Q_{diag1}, r_0, r_1

Semaphores: S_1 locked, S_2 unlocked

Compute Process

Local variable (unshared): t
$t = 0$
while not converged
  for $i = 1$ to N
    Lock(S_1)
    for $j = 1$ to *MinIter*
      Do GS iteration: $\Pi_i Q_{\text{diag}t} = r_t$
    $j = MinIter + 1$
    while $j \leq MaxIter$ and
        I/O process not blocked on S_2
      Do GS iteration: $\Pi_i Q_{\text{diag}t} = r_t$
      j = j + 1
    Unlock(S_2)
    $t = \bar{t}$

I/O Process

Local variable (unshared): t, Q_{tmp}
$t = 0$
do forever
  for $i = 1$ to N
    $Q_{\text{diag}t} = \text{disk read}(Q_{ii})$
    $Q_{\text{tmp}} = \text{disk read}(Q_{\text{off},i})$
    $r_t = -\Pi Q_{\text{tmp}}$
    $Q_{\text{tmp}} = \text{disk read}(Q_{\text{conflict},i})$
    Lock(S_2)
    $r_t = r_t - \Pi_{i-1} Q_{\text{tmp}}$
    Unlock(S_1)
    $t = \bar{t}$

Figure 3.2: Compute and I/O processes for BGS algorithm.

share two diagonal block and r variables so that one can be used to compute (3.1) while the other one is being prepared for the next computation. The processes also share two locking variables, S_1 and S_2, which they use to communicate and control the relative progress of the other process.

Compute Process We first explain the compute process. A local variable t alternates between 0 and 1, which indicates which of the two shared block and r variables the process should use. After each step, t is alternated between 0 and 1, which we denote $t = \bar{t}$. The function Lock(S_1) will lock S_1 if S_1 is unlocked. If S_1 is already locked, it will block until S_1 is unlocked (by the I/O process); then it will lock S_1 and proceed. While the compute process is blocked on S_1, it uses no CPU resources.

The compute process has two parameters, *MinIter* and *MaxIter*. The compute process is guaranteed to do at least *MinIter* Gauss-Seidel inner iterations to approximately solve (3.1). Then the compute process will proceed to do up to *MaxIter* iterations or until the I/O process is complete with the current file I/O and is waiting for the compute process to unlock S_2, whichever comes first. This allows the compute process to do a dynamic number

of Gauss-Seidel iterations, depending on how long the I/O process takes to do file I/O. We ignore the boundary conditions in the figures for simplicity. If $i-1=0$, for example, then we use N for $i-1$ instead.

The convergence criterion we use in this tool is a modification to the $||\pi^{(k+1)}-\pi^{(k)}||_\infty$ criterion. In particular, we compute $||\Pi_i^{(k+1)}-\Pi_i^{(k)}||_\infty$ for the *first* inner iteration and take the $\max_i ||\Pi_i^{(k+1)}-\Pi_i^{(k)}||_\infty$ to be the number we use to test for convergence. We use this for two reasons: the first inner iteration usually results in the greatest change of Π_i, so computing the norm for all inner iterations is usually wasteful, and the computation of the norm takes a significant amount of time. We have observed experimentally that this measured norm is at least as good as the the $||\pi^{(k+1)}-\pi^{(k)}||_\infty$ criterion.

The dynamic nature of the iteration count is an interesting feature of this tool. If the system on which the program is running is doing other file I/O and slowing the I/O process down, the compute process may continue to proceed to do useful work. At some point, however, additional Gauss-Seidel iterations may not be useful at all, presumably after *MaxIter* inner iterations, so the process will stop doing work and block waiting for S_1 to become unlocked. Choosing a good *MinIter* and *MaxIter* is difficult and requires some knowledge about the characteristics of the transition rate matrix. If we allow the compute process to be completely dynamic, some blocks may consistently get fewer inner iterations and converge more slowly than other blocks, causing the whole system to converge slowly. In Section IV, we show some experimental results of varying these parameters.

Input/Output Process The I/O process is straightforward. The greatest complexity comes in managing the semaphores properly. This is a case of the classical producer-consumer or bounded buffer problem, and we defer the reader to [13] or a similar text on operating systems to show the motivation and correctness of this technique. The primary purpose of the I/O process is to schedule disk reads and compute r_t. It does this by issuing a C function to read portions of the file directly into the shared block variable or the temporary block variable. Because the I/O process may execute in parallel with the compute process, the I/O process may issue read requests concurrently with the computation, and since file I/O uses little CPU (under 20%), we can effectively parallelize computation and file I/O on a modern, single-processor workstation.

This implementation of BGS uses relatively little memory. The size of the steady state

probability vector Π is proportional to the number of states, which is unavoidable using BGS or any other exact method. Other iteration methods, such as Jacobi, require additional vectors of the same size as Π, which our program avoids. Two diagonal blocks, Q_{diag0} and Q_{diag1}, are necessary; the compute process requires one block to do the inner iteration, and the I/O process reads the next diagonal block at the same time. Two r variables are also required for the same reason. Finally, the I/O process requires a temporary variable Q_{tmp} to hold Q_{off} and Q_{conflict}. We could eliminate Q_{tmp} by instead using $Q_{\text{diag}t}$, but doing so would require us to reverse the order in which we read the blocks, causing us to read $Q_{\text{diag}t}$ last. This would reduce the amount of time we could overlap computation and file I/O. We chose to maximize parallelization of computation at the expense of a modest amount of memory.

IV Results

To better understand the algorithms presented in the previous section, we implemented them and tested the resulting tool on several large models presented in the literature. We present the models here and discuss the performance measures we took in order to better understand the issues in building and using a tool to solve large matrices, so we are not so interested here in the results of the models as much as using the models to understand the characteristics of our solver. All the solutions we present here, with the exception of one, can be solved on our HP workstation with 128 Mbyte of RAM (without using virtual memory) and 4 Gbyte of fast disk memory.

Kanban Model The Kanban model we present was previously used by Ciardo and Tilgner [1, 2] to illustrate a Kronecker-based approach. They chose this model because it has many of the characteristics that are ideal for superposed GSPN solution techniques, and it also does not require any mapping from the product space to the tangible reachable states. We refer to [2] for a description of the model and specification of the rates in the model. Briefly, the model is composed of four subnets. At each subnet, a token enters, spends some time, and exits or restarts with certain probabilities. Once the token leaves the first subnet, it may enter the second or third subnet, and to leave the system, the token must go through the fourth subnet. We chose to solve the model where the synchronizing transitions are timed transitions.

N	States	NZ Entries	Size (MB)	e_1	e_2	e_3	e_4	τ
1	160	616	0.008	0.90742	0.67136	0.67136	0.35538	0.09258
2	4,600	28,128	0.34	1.81006	1.32851	1.32851	0.76426	0.17387
3	58,400	446,400	5.3	2.72211	1.94348	1.94348	1.52460	0.23307
4	454,475	3,979,850	47	3.64641	2.51298	2.51298	1.50325	0.27589
5	2,546,432	24,460,416	290	4.58301	3.03523	3.03523	1.81096	0.30712
6	11,261,376	115,708,992	1,367	5.53098	3.50975	3.50975	2.07460	0.33010

Table 3.1: Characteristics and reward variables for the Kanban model.

Table 3.1 shows some information about the model and the corresponding transition rate matrix. Here, N represents the maximum number of tokens that may be in a subnet at one time. There are two important variables that we may vary, the number of blocks and the number of inner iterations, that greatly affect performance. We present two experiments. First, we vary the number of blocks while keeping the number of inner iterations fixed, and second, we vary the number of inner iterations while keeping the number of blocks fixed.

For the first experiment, we use the Kanban model where $N = 5$. We divide the transition rate matrix into 32×3 blocks, and perform a constant number of inner iterations. We vary the number of inner iterations from 1 to 20. The results of the solution execution time and the number of BGS iterations are shown in the top two graphs in Figure 3.3. All the timing measurements that we present in this paper are "wall clock" times. The plots show the time to achieve three levels of accuracy based on the modified $||\Pi^{(k+1)} - \Pi^{(k)}||_\infty < \{10^{-6}, 10^{-9}, 10^{-12}\}$ convergence criterion explained in Section III.

Figure 3.3 shows how doing an increased number of inner iterations yields diminishing returns, so that doing more than about 7 inner iterations does not significantly help reduce the number of BGS iterations. For this model, setting *MaxIter* to 6 or 7 makes sense. It also shows that the optimal number of inner iterations with respect to execution time is 4. For fewer than four inner iterations, the compute process spends time idle and waiting for the I/O process. This leads us to choose *MinIter* to be 3 or 4.

It is interesting to note that solving this model with a dynamic number of inner iterations takes 10,436 seconds, which is more time than is required if we fix the number of inner iterations to be 3, 4, or 5 (10269, 10044, and 10252 seconds respectively). We observed that some blocks always receive 4 or fewer inner iterations, while others always receive 7 or more. This shows us several important things. First, some blocks always receive more iterations than others, and we know that the solution vector will converge only as fast as its slowest

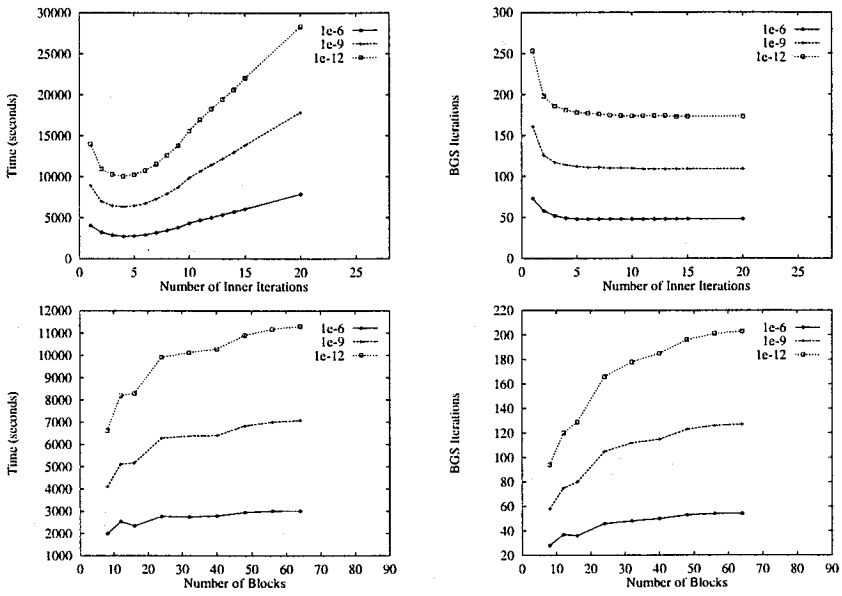

Figure 3.3: Performance graphs of Kanban model ($N = 5$).

converging component. Second, we argued above that doing more than 7 inner iterations is wasteful, so allowing the number of inner iterations to be fully dynamic is wasteful since the I/O process does not read data quickly enough to keep the compute node doing useful work. Finally, if the compute process is always doing inner iterations, it checks to see if the I/O process is blocked on S_2 only after completing an inner iteration. This requires the I/O process to always block on S_2 and wait for the compute process to complete its inner iteration, which is wasteful since the I/O process is the slower of the two processes.

For the next experiment, we set the number of inner iterations to be 5, vary the number of blocks, and observe convergence rate, execution time, and memory usage. The bottom two plots of Figure 3.3 range the number of blocks from 8 to 64 and plot execution time and number of iterations respectively for the convergence criteria $||\Pi^{(k+1)} - \Pi^{(k)}||_\infty < \{10^{-6}, 10^{-9}, 10^{-12}\}$. Table 3.2 shows how memory usage varies with the number of blocks. Notice that between 8 and 64 blocks, the execution time is nearly double while the memory usage is about one third. We see that there is clearly a memory/speed tradeoff. Note that the solution vector for a 2.5 million state model alone takes about 20 Megabytes of memory.

Finally, as a basis for comparison, we present results given in [2] in Table 3.3 and compare the solution times to those of our tool. The column titled 'Case 1' represents the tool in [2] with mapping from the product space to tangible reachable states enabled, while 'Case 2'

N	Memory (MB)
8	95
12	71
16	59
24	48
32	40
40	37
48	35
56	33
64	32

Table 3.2: Number of blocks versus memory.

N	Case 1	Case 2	BGS time	No. Blocks	Memory
1	1 s	1s	-	-	-
2	13 s	2 s	-	-	-
3	310 s	2 s	-	-	-
4	4,721 s	856 s	225 s	4	28 MB
5	22,215 s	6,055 s	2,342 s	16	59 MB
6	-	-	18,563 s	128	111 MB

Table 3.3: Comparison of performance.

is with no mapping (an idealized case). Cases 1 and 2 are performed on a Sony NWS-5000 workstation with 90 MB of memory. We present no results for $N = 1, 2, 3$ because the matrix was so small that the operating systems buffered the entire file in memory. In addition to computing the reward variables (see Table 3.1) for the Kanban model to greater accuracy than [2], we were also able to solve for the case where $N = 6$.

Courier Protocol Model The second model we examine is a model of the Courier protocol given in [8, 14]. This is a model of an existing software network protocol stack that has been implemented on a Sun workstation and on a VME bus-based multiprocessor. The model is well specified in [8, 14]. The GSPN models only a one-way data flow through the network. For our experiment, we are only interested in varying the window size N. The transport space and fragmentation ratio is kept at one. Varying N corresponds to initially placing N tokens in a place, and it has a substantial impact on the state space size!

Table 3.4 shows the characteristics of the model. The column 'Matrix Size' contains the size of the matrix in megabytes if the matrix were to be kept entirely in memory. One can see that this transition-rate matrix is less dense than the one for the Kanban model. For this model, we wish to show how the solution process varies as the size of the problem gets larger. We set *MaxIter* to be 6 and let N range. Table 3.4 summarizes these results.

There are several interesting results from this study. First, we note that for $N < 3$, the file system buffers significantly decrease the conversion and solution times, so they should not be considered as part of a trend. More traditional techniques would probably do as

	$N=1$	$N=2$	$N=3$	$N=4$	$N=5$	$N=6$
States	11,700	84,600	419,400	1,632,600	5,358,600	15,410,250
Nonzero	48,330	410,160	2,281,620	9,732,330	34,424,280	105,345,900
Matrix	0.6	5	28	118	414	1,264
Blocks	4	4	4	32	64	128
Generation Time (s)	5	38	218	938	3,716	11,600
Conversion Time (s)	3	24	136	581	2,076	*14,482
Solution Time (s)	2	16	143	1,138	6,040	20,742
Iterations	18	19	23	49	69	85
Memory (MB)	0.4	3.4	18	21	57	144

(*) Time abnormally high because the computer was not dedicated.

Table 3.4: Characteristics of Courier protocol model.

well or better for such small models. For $N \geq 3$, the model becomes interesting. We wrote our own GSPN state generator for these models, and it was optimized for memory (so we could generate large models), not for speed. It was also designed to be compatible with the *UltraSAN* solvers and reward variable specification. The conversion time is the time it took to convert the Q-matrix in *UltraSAN*'s format to one used by the tool, which involves taking the transpose of Q and converting from an ASCII to binary floating point representation. The conversion time shown for $N = 6$ is the wall clock time, but it is abnormally large since it was not run in a dedicated environment. We estimate from the user time that the conversion process would take about 7,000 seconds on a dedicated system.

The data gives us a rough idea about the relative performance of each step in the solution process. The conversion process takes between half and two thirds the generation time. We believe that much of the conversion time is spent translating an ASCII representation of a real number into the computer's internal representation. The solution times are the times to reach the convergence criterion $||\Pi_i^{(k+1)} - \Pi_i^{(k)}||_\infty < 10^{-12}$ described above, and are roughly twice the generation times. This shows that the solution process does not take a disproportionate amount of time more than the state generation or conversion process.

Another interesting observation of this model is that in the case where $N = 6$, the transition-rate matrix generated by the GSPN state generator (a sparse textual representation of the matrix) would be larger than 2 Gbytes, which is larger than the maximum allowable file size on our workstation. To solve this system, we rounded the rates to 6 decimal places. This obviously affects the accuracy of the solutions. There are obvious and simple ways to use multiple files to avoid this problem; we simply state this observation to

	$N=1$	$N=2$	$N=3$	$N=4$	$N=5$	$N=6$
λ	74.3467	120.372	150.794	172.011	187.413	198.919
P_{send}	0.01011	0.01637	0.02051	0.02334	0.02549	0.02705
P_{recv}	0.98141	0.96991	0.96230	0.95700	0.95315	0.95027
P_{sess1}	0.00848	0.01372	0.01719	0.01961	0.02137	0.02268
P_{sess2}	0.92610	0.88029	0.84998	0.82883	0.81345	0.80197
$P_{transp1}$	0.78558	0.65285	0.56511	0.50392	0.45950	0.42632
$P_{transp2}$	0.78871	0.65790	0.57138	0.51084	0.46673	0.43365

Table 3.5: Reward variables for Courier protocol model.

give the reader a feel for the size of the data that the tool is manipulating. Also, of the 144 Mbytes necessary to compute the solution, 118 Mbytes of it are needed just to hold the solution vector.

In Table 3.5 we show several of several of the reward variables in the model as N varies from 1 to 6. The λ we compute here corresponds to measuring λ_{lsp} in the model, which corresponds to the user's message throughput rate. The measures λ_{frg} can easily be computed as $\lambda_{frg} = \lambda q_1/q_2$. Similarly, $\lambda_{ack} = \lambda_{lsp} + \lambda_{frg}$. From this, we can see how the packet throughput rate (λ) increases as the window size increases. Other reward variables are explained in [8], and they correspond to the fraction of time different parts of the system are busy. We note that the values we computed here differ from those Li found by approximate techniques [8, 14]. We suspect that Li used a fragmentation ratio in his approximation techniques that is different (and unpublished) from the ratio for which he gives "exact" solutions because we were able to reproduce the exact solutions.

V Conclusion

We have described a new tool for solving Markov models with very large state spaces. By devising a method to efficiently store the state-transition-rate matrix on disk, overlap computation and data transfer on a standard workstation, and utilize an iterative solver that exhibits locality in its use of data, we are able to build a tool that requires little more memory than the solution vector itself to obtain a solution. This method is completely general to any model for which one can derive a state-transition-rate matrix. As illustrated in the paper, the tool can solve models with 10 million states and 100 million non-zero entries on a machine with only 128 Mbytes of main memory. Because we make use of an innovative implementation using two processes that communicate via shared memory, we

are able to keep the compute process utilizing the CPU approximately 80% of the time.

In addition to describing the tool, we have illustrated its use on two large-scale models: a Kanban manufacturing system and the Courier protocol stack executing on a VME bus-based multiprocessor. For each model, we present detailed results concerning the time and space requirements for solutions so that our tool may be compared with existing and future tools. The results show that the speed of solution is much faster than those reported for implementations based on Kronecker operators. These results show that our approach is the current method of choice for solving large Markov models if sufficient disk space is available to hold the state-transition rate matrix.

REFERENCES

[1] G. Ciardo, "Advances in compositional approaches based on Kronecker algebra: Application to the study of manufacturing systems," in *Third International Workshop on Performability Modeling of Computer and Communication Systems*, Bloomingdale, IL, Sept. 7–8, 1996.

[2] G. Ciardo and M. Tilgner, "On the use of Kronecker operators for the solution of generalized stochastic Petri nets," ICASE Report #96-35 CR-198336, NASA Langley Research Center, May 1996.

[3] D. D. Deavours and W. H. Sanders, " 'On-the-fly' solution techniques for stochastic Petri nets and extensions," to appear in *Petri Nets and Performance Models*, 1997.

[4] S. Donatelli, "Superposed generalized stochastic Petri nets: Definition and efficient solution," in R. Valette, editor, *Application and Theory of Petri Nets 1994, Lecture Notes in computer science 815 (Proc. 15th Int. Conf. on Application and Theory of Petri Nets, Zaragoza, Spain)*, pp. 258–277, Springer-Verlag, June 1994.

[5] G. Horton, "Adaptive Relaxation for the Steady-State Analysis of Markov Chains," ICASE Report #94-55 NASA CR-194944, NASA Langley Research Center, June 1994.

[6] P. Kemper, "Numerical analysis of superposed GSPNs," in *Proc. Int. Workshop on Petri Nets and Performance Models (PNPM'95)*, pp. 52–61, Durham, NC, Oct. 1995. IEEE Comp. Soc. Press.

[7] P. Kemper, "Numerical Analysis of Superposed GSPNs," in *IEEE Transactions on Software Engineering*, 1996, to appear.

[8] Y. Li, "Solution Techniques for Stochastic Petri Nets," Ph.D. Dissertation, Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, May 1992.

[9] J. F. Meyer, A. Movaghar, and W. H. Sanders, "Stochastic activity networks: Structure, behavior, and application," In *Proc. International Workshop on Timed Petri Nets*, pp. 106–115, Torino, Italy, July 1985.

[10] A. Movaghar and J. F. Meyer, "Performability modeling with stochastic activity networks," In *Proc. 1984 Real-Time Systems Symp.*, Austin, TX, December 1984.

[11] W. H. Sanders, W. D. Obal II, M. A. Qureshi, F. K. Widjanarko, "The *UltraSAN* modeling environment," in *Performance Evaluation*, pp. 89–115, Vol. 24, 1995.

[12] W. J. Stewart, "Introduction to the Numerical Solution of Markov Chains," Princeton University Press, 1994.

[13] A. S. Tanenbaum, *Modern Operating Systems*, Prentice Hall, 1992.

[14] C. M. Woodside and Y. Li, "Performance Petri Net Analysis of Communications Protocol Software by Delay-Equivalent Aggregation," in *Proc. Fourth Int. Workshop on Petri Nets and Performance Models*, pp. 64–73, Melbourne, Australia, Dec. 2–5, 1991.

Chapter 4

A NEW METHODOLOGY FOR CALCULATING DISTRIBUTIONS OF REWARD ACCUMULATED DURING A FINITE INTERVAL

Abstract

Markov reward models are an important formalism by which to obtain dependability and performability measures of computer systems and networks. In this context, it is particularly important to determine the probability distribution function of the reward accumulated during a finite interval. The interval may correspond to the mission period in a mission-critical system, the time between scheduled maintenances, or a warranty period. In such models, changes in state correspond to changes in system structure (due to faults and repairs), and the reward structure depends on the measure of interest. For example, the reward rates may represent a productivity rate while in that state, if performability is considered, or the binary values zero and one, if interval availability is of interest. This paper presents a new methodology to calculate the distribution of reward accumulated over a finite interval. In particular, we derive recursive expressions for the distribution of reward accumulated given that a particular sequence of state changes occurs during the interval, and we explore paths one at a time. The expressions for conditional accumulated reward are new and are numerically stable. In addition, by exploring paths individually, we avoid the memory growth problems experienced when applying previous approaches to large models. The utility of the methodology is illustrated via application to a realistic fault-tolerant multiprocessor model with over half a million states.

Keywords: Markov Reward Models, Performability, Interval Availability.

I Introduction

Performability evaluation is an important approach for calculating the performance of a dependable computing system or network, taking into account changes in performance due to faults. Many methods have been proposed to evaluate system performability, but one of the most popular has been through the use of reward models [1, 2]. In this context, it is important to determine the probability distribution function of reward accumulated during a finite interval. The interval may correspond to the mission period in a mission-critical system, the time between scheduled maintenances, or a warranty period.

Most early work in this regard has been limited to acyclic Markov reward models (see, for example, [2, 3, 4, 5, 6]). Determining the distribution of reward accumulated over a

finite interval for general (possibly cyclic) Markov reward models is more difficult due to the possible presence of an infinite number of paths. The first attempt to solve such models was made by Kulkarni et al. [7]. They numerically inverted an expression obtained in the transform domain to obtain a solution. Later Smith et al. [8] presented an improved version of the algorithm in [7], with a computational complexity of $O(n^3)$, where n is the number of states. In [9], Pattipati et al. applied partial differential equation techniques to compute system performability.

Also notable is the work of de Souza e Silva and Gail, who were the first to present a performability solution based on uniformization [10]. They formulated the probability distribution of reward accumulated over a finite interval by first conditioning on the number of transitions n of a uniformized process and then further conditioning on vectors corresponding to the number of visits to states with different rewards, given n. This algorithm was limited in applicability since the storage and computational complexity was combinatorial with the number of distinct rewards. Two attempts have been made to deal with this complexity. In particular, Donatiello and Grassi [11] presented an algorithm with a polynomial complexity in number of distinct rewards by combining uniformization and Laplace transform methods. In addition, in [12], de Souza e Silva and Gail presented a significant improvement on their previous algorithm, albeit limited to rate-based reward models. The storage complexity of their new algorithm is independent of the number of distinct rewards assigned to the states of the Markov process.

In spite of these significant advances, the developed algorithms have been limited to solving small models, and for short time intervals. The reasons are two-fold: 1) the storage complexity is still polynomial with number of states and transitions of the uniformized process, and 2) in addition to the stated storage complexity, the algorithm requires the storage of the state transition matrix of the entire subordinated Markov process. When state spaces are large (as is often the case) or the interval is reasonably long (again, as is typical), these storage requirements make the computation of distribution of accumulated reward over a finite interval very difficult.

In this paper, we present a path-based approach (used previously to solve dependability models [13, 14]) to compute the distribution of time-averaged reward accumulated over a finite interval for general (possibly cyclic) reward models. In doing so, we trade storage complexity for time complexity. We show that the trade is advantageous if the probability mass is concentrated on a reasonable number of paths (up to tens of millions). Moreover, to use

the path-based approach effectively, we present a numerically stable and computationally efficient algorithm to compute the conditional distribution of accumulated reward given a path. This formulation is new and not based on the often used Weisberg result [15]. Finally, we illustrate the usefulness of the path-based approach by computing the distribution of time-averaged accumulated reward over a finite interval of a highly redundant fault-tolerant multiprocessor system. The new approach presented in this paper is significant in that it can be applied to much larger systems than previously possible, and it is numerically stable.

II Background

Uniformization (also known as Jenson's method and randomization) is a well-known method for computing the state-occupancy probabilities of a Markov process at specific time t. Its formulation involves construction of a discrete-time Markov process and Poisson process from an original continuous-time process. In particular, consider a continuous-time time-homogeneous Markov process $\{X_t : t \geq 0\}$ with generator matrix A and initial state distribution vector π_0. Let λ be greater than or equal to the maximum departure rate from any state in the process, and $\{Z_n : n \in N\}$ be a discrete-time time-homogeneous Markov chain defined on the same state space as $\{X_t : t \geq 0\}$ with single step transition matrix $P = A/\lambda + I$. Furthermore, let $\{N_t : t \geq 0\}$ be a Poisson process with rate λ. Then π_t, the row vector of state occupancy probabilities at time t, can be expressed as

$$\pi_t = \sum_{n=0}^{\infty} \frac{e^{-\lambda t}(\lambda t)^n}{n!} \pi_0 P^n.$$

de Souza e Silva and Gail [10, 16] formulated the probability distribution function (PDF) of the accumulated reward averaged over a finite interval of time using uniformization and several additional properties of a Markov chain subordinated to a Poisson process. Specifically, consider that the Poisson process has arrivals at instances $T_1 < T_2 < \ldots < T_n$ in an interval $(0, t)$. These are the instances when the subordinated Markov chain makes transitions. Furthermore, consider n independent random variables $U_1, U_2, \ldots U_n$ uniformly distributed on the interval $(0, 1)$. Let $U_{(1)}, U_{(2)}, \ldots, U_{(n)}$ be their order statistics. It is well known (e.g., [17]) that the joint distribution of the transition times of subordinated Markov chains, given n transitions of the Poisson process in an interval $(0, t)$, is identical to the joint distribution of the order statistics of n uniformly distributed random variables over

the interval $(0, t)$. Moreover, it can easily be shown that tU_i, the product of the interval t and the random variable U_i, is uniformly distributed over an interval $(0, t)$. Therefore

$$P\{T_1 \leq t_1, T_2 \leq t_2, \ldots, T_n \leq t_n\} \;=\; P\{tU_{(1)} \leq t_1, tU_{(2)} \leq t_2, \ldots, tU_{(n)} \leq t_n\}, \qquad (4.1)$$

for $t_1 < t_2 < \ldots < t_n$.

Given n transitions of the Poisson process, each path of the subordinated Markov chain consists of $n + 1$ sojourn times. Using the result from Equation 4.1, these $n + 1$ sojourn times Y_i can be represented as $Y_1 = tU_{(1)}$, $Y_2 = t(U_{(2)} - U_{(1)})$, ..., $Y_{n+1} = t(1 - U_{(n)})$. It is also known [18] that the random variables Y_i, $i = 1, 2, \ldots, n + 1$, are exchangeable. In particular, exchangeability says that

$$P\{Y_1 \leq t_1, Y_2 \leq t_2, \ldots, Y_{n+1} \leq t\} \;=\; P\{Y_{j_1} \leq t_1, Y_{j_2} \leq t_2, \ldots, Y_{j_{n+1}} \leq t\},$$

for all permutations of j_i's from $1, 2, \ldots, n + 1$. Therefore, by using order statistics, the sequence of sojourn times in a state trajectory can be altered. In particular, suppose all paths with n transitions of the uniformized process are divided into sets such that each path in a set has an equal number of visits to states with identical rate rewards. Then exchangeability enables us to describe all paths within each set by a vector $\mathbf{k} = (k_1, k_2, \ldots, k_{K+1})$, where $K + 1$ is the number of distinct rate rewards and k_i, $i = 1, 2, \ldots, K + 1$, is equal to the number of visits to states with rate reward i. (Note $|\mathbf{k}| = n + 1$.)

Using these ideas, de Souza e Silva and Gail formulated the PDF of reward accumulated over a finite interval by further conditioning on the vectors $\mathbf{k}$ possible for each n. Accordingly, $P\{AR(t) \leq r\}$, the distribution of the time-averaged reward accumulated over a finite interval, was expressed as

$$P\{AR(t) \leq r\} \;=\; \sum_{n=0}^{\infty} \frac{e^{-\lambda t}(\lambda t)^n}{n!} \sum_{\forall \mathbf{k}} P\{\mathbf{k} \mid n\} P\{AR(t) \leq r \mid n, \mathbf{k}\}, \qquad (4.2)$$

where $P\{\mathbf{k} \mid n\}$ is the probability of vector $\mathbf{k}$ given n transitions.

The exact solution for $P\{AR(t) \leq r\}$ is not possible in this formulation, since the first summation in (4.2) is over an infinite set. However, as shown in [10, 16], a solution with error bound can be computed by truncating the first summation to some finite number N. In particular, to a certain error bound, the probability distribution of time-averaged reward

accumulated over a finite interval can be formulated as

$$P\{AR(T) \leq r\} = \sum_{n=0}^{N} \frac{e^{-\lambda t}(\lambda t)^n}{n!} \sum_{\forall \mathbf{k}} P\{\mathbf{k} \mid n\} P\{AR(T) \leq r \mid n, \mathbf{k}\}. \tag{4.3}$$

For Markov processes with large state spaces, second summation is challenging to compute due, primarily, to the computation of $P\{\mathbf{k} \mid n\}$. This computation requires knowledge of all $\mathbf{k}$ vectors that are possible given $n-1$ transitions of the uniformized process and the probability of occurrence of paths within these sets of vectors. Since there can be $\begin{pmatrix} K+n \\ n \end{pmatrix}$ $\mathbf{k}$ vectors for $n-1$ transitions and the number of paths which can generate a vector $\mathbf{k}$ can be equal to the size of the state space, the task of computing PDF for large state spaces becomes immensely difficult. Moreover, this approach also requires the complete generation of the state space prior to starting the PDF computation. Generation of large state spaces in itself is an intensively memory complex problem.

In [12], de Souza e Silva and Gail improved on their earlier algorithm by finding a recursion which, for a given number of transitions, depends on sets of vectors $\mathbf{k}$. Instead of computing path probabilities for each vector $\mathbf{k}$, they directly computed the conditional distribution given n for sets of $\mathbf{k}$ vectors. Their new algorithm significantly improved the memory complexity since individual $\mathbf{k}$ vectors no longer need to be computed. However, due to the recursion on sets of $\mathbf{k}$ vectors, their algorithm needs a separate computation for every point on the probability distribution curve. When both rate and impulse rewards are used, their algorithm has $O(dMN^2\kappa(N,r))$ storage requirement [19], where d is the average number of non-zero entries in the transition matrix, M is the state-space size, N is the truncation point of the infinite series, and $\kappa(N,r)$ is the number of distinct values obtained by adding any combination of N impulse rewards that are less than r. Furthermore, their algorithm operates on the state transition matrix of the subordinated Markov process, which also must be stored. While the new algorithm takes significantly less memory than the original algorithm, it still requires prohibitively large amounts of memory for realistically sized models, which may have hundreds of thousands of states. The next section presents a formulation that avoids this problem, albeit at the cost of additional time, and avoids numerical difficulties in the formulation presented in [10].

III Path-Based Algorithm

In this new formulation, we compute the probability distribution function of the time-averaged reward accumulated over a finite interval of time by conditioning on possible paths, rather than **k** vectors, that can occur. These paths are generated in a depth-first search manner, from a higher level description (such as a SAN [20]), thus avoiding the memory complexity problems associated with previous approaches. However, this gain is not free, since the approach results in an increased time complexity. In the worst case, the number of paths of the uniformized process that must be generated grows exponentially with an increase in the truncation point. However, if the uniformized process has a sparse state-transition probability matrix or is such that the probability mass is concentrated on a reasonable number of the many paths, then the depth-first search approach becomes a reasonable choice.

In this case, we can compute the solution with a reasonable error bound by only considering the set of probable paths. Moreover, if we know the highest rate of the underlying Markov process (as would be the case if it were generated from a higher-level formalism), then the depth-first search approach does not even require generation of the state space prior to the solution. In this case, the paths of the uniformized process can be explored without generating the complete state space, and the solution for the time-averaged accumulated reward can be obtained on the fly.

The depth-first search approach for computing the PDF of the time-averaged reward accumulated over a finite interval is based on exploring paths of the uniformized process and computing conditional distribution given the path. To understand this approach, we first look at a path in the uniformized process. A path in the uniformized process corresponds to a sequence of states through the state space. Let $< s_0, s_1, \ldots, s_n >$ be a sequence in the uniformized process that occurs in time $(0, t)$. The probability of its corresponding path in time $(0, t)$ can be obtained by computing $P\{s_0, s_1, \ldots, s_n \mid n\} = p(s_0, s_1) \times p(s_1, s_2) \times \ldots p(s_{n-1}, s_n)$, and $P\{n\}$,

$$P\{path\} \;=\; P\{s_0, s_1, \ldots, s_n \;\wedge\; n\} \;=\; P\{n\}P\{s_0, s_1, \ldots, s_n \mid n\}$$

where $p(s_i, s_j)$ is the transition probability from state s_i to s_j in the discrete time embedded process.

For the path-based approach, the distribution of the time-averaged reward accumulated

over a finite interval can be expressed as

$$P\{AR(t) \leq r\} = \sum_{path \in P} P\{path\} P\{AR(t) \leq r | path\}, \tag{4.4}$$

where P is the (possibly infinite) set of possible paths in the process.

As with the formulation in (4.2), an exact solution of (4.4) is not possible because there are an infinite number of paths in a uniformized process. Since our intent is to only consider paths that are significant relative to the solution, we limit the number of paths considered by discarding those whose path probabilities are smaller than a specified value, defined as w. Let P_w denote the set of paths that have path probabilities greater than or equal to w. Accordingly, PDF can be expressed as

$$P\{AR(t) \leq r\} = \sum_{path \in P_w} P\{path\} P\{AR(t) \leq r | path\}. \tag{4.5}$$

As with the previous approach, considering a finite number of paths results in an error in the solution which can be bounded. In particular, let $E(w)$ be the error induced by discarding paths for which $P\{path\} < w$ and $0 \leq w \leq 1$. In order to bound $E(w)$, we first compute $E(path)$, the error produced by discarding a particular path. Let this path consist of the sequence $< s_0, s_1, \ldots, s_n >$. Note that by discarding a path, we are also discarding all longer paths which have states $s_0, s_1, \ldots, s_n$ as their first n states. We first look at the error induced by discarding the path of length n. In particular, the error will be

$$E(path) \;=\; \frac{e^{-\lambda t}(\lambda t)^n}{n!} P\{s_0, s_1, \ldots, s_n \mid n\} P\{AR(t) \leq r | path\}.$$

Since the conditional distribution $0 \leq P\{AR(t) \leq y | path\} \leq 1 \; \forall y$, we can bound the error by assuming it is equal to 1. This implies that

$$E(path) \;\leq\; \frac{e^{-\lambda t}(\lambda t)^n}{n!} P\{s_0, s_1, \ldots, s_n \mid n\}.$$

Now, since the entries in a row of a transition matrix sum to 1, the sum of the probabilities of all paths of length $n+1$ discarded due to the discarding of the path with state sequence $< s_0, s_1, \ldots, s_n >$ can be bounded by

$$\sum_{\forall s_{n+1}} P\{s_0, s_1, \ldots, s_{n+1} \mid n+1\} \times \frac{e^{-\lambda t}(\lambda t)^{n+1}}{(n+1)!} \;=\; P\{s_0, s_1, \ldots, s_n \mid n\} \frac{e^{-\lambda t}(\lambda t)^{n+1}}{(n+1)!}.$$

Using the above argument repeatedly, the total error (denoted E^*) induced by discarding all paths of length n or longer with starting states $s_0, s_1, \ldots s_n$ can be bounded by

$$E^*(path) \leq P\{s_0, s_1, \ldots, s_n \mid n\} \times \sum_{i=n}^{\infty} \frac{e^{-\lambda t}(\lambda t)^i}{i!} = P\{s_0, s_1, \ldots, s_n \mid n\} \times \left(1 - \sum_{i=0}^{n-1} \frac{e^{-\lambda t}(\lambda t)^i}{i!}\right).$$

Hence, a bound on the total error induced by discarding paths can be computed as

$$E(w) \leq \sum_{path \in P^D(w)} E^*(path),$$

where $P^D(w)$ is the set of paths discarded, during exploration, since they do not meet the criteria described.

IV Calculation of Conditional Distribution

Given the approach of the previous section, we need a way to compute the distribution of time-averaged reward accumulated over an interval of time conditioned on a particular path. Finding an efficient and numerically stable way to compute this distribution is the key to the development of a methodology that can be applied to realistically sized systems. Since our process is uniformized with paths that are exchangeable (see Section II), the conditional accumulated reward will be the same for all paths that have the same number of visits to states with particular rate rewards. In other words, for all paths with $\mathbf{k}$ vector $\mathbf{k} = k_1, k_2, \ldots, k_{K+1}$ such that $|\mathbf{k}| = n + 1$,

$$P\{AR(t) \leq r \mid path\} = P\{AR(t) \leq r \mid n, \mathbf{k}\}.$$

This suggests that, in principle, we could use the result from Weisberg [15], which gives the probability distribution function of a linear combination of selected order statistics of i.i.d. uniform random variables. Unfortunately, while this is correct, it is numerically unstable for practical problems, since for large n, it requires the subtraction of extremely large numbers (generated from factorials of large numbers) with nearly identical magnitude at multiple points in the algorithm. Multiple precision arithmetic could potentially be used to avoid this problem, but it is extremely difficult to know what precision to use, because of the multiple subtractions.

To avoid these problems, we have developed a new algorithm that does not make use of the Weisberg result. It is based on an alternative formulation given in [21] and makes use of

three new lemmas (presented in the following) that reduce both the amount of computation and the magnitude of numbers that must be stored as intermediate results. Furthermore, we are able to formulate the expression in a form that requires very few subtractions of numbers with large but nearly identical magnitude and, hence, are able to determine exactly the number of digits required to achieve a desired precision in the result.

In particular, the new formulation is based on expressing the problem as the linear combination of order statistics, rewriting these as a linear combination of Drichlet random variables, and then computing the distribution of the combination. (The Drichlet distribution is widely used in statistical mathematics, see [22] for example.) Specifically, consider, as was done previously, that $K+1$ different rate rewards are assigned to the states of the uniformized process. Furthermore, suppose that rate rewards are ordered such that

$$r_1 > r_2 > \ldots > r_{K+1} \geq 0.$$

Then define l_i to be the sum of the lengths of sojourn times with rate reward r_i. After doing this, the conditional averaged accumulated reward, $AR(t)$, given n and $\mathbf{k}$, can be expressed as

$$AR(t \mid n, \mathbf{k}) \;=\; \frac{1}{t} \sum_{i=1}^{K+1} r_i \times l_i.$$

Now recall that using the result from (4.1), the $n+1$ sojourn times Y_i can be represented as $Y_1 = tU_{(1)}$, $Y_2 = t(U_{(2)} - U_{(1)})$, ..., $Y_{n+1} = t(1 - U_{(n)})$. Resultingly, $P\{Y_1 + Y_2 \leq r\} = P\{tU_2 \leq r\}$. According to the exchangeability property, given n transitions, $P\{Y_{j_1} + Y_{j_2} + \ldots + Y_{j_m} \leq r\} = P\{tU_m \leq r\}$ for all permutations $j_1, j_2, \ldots, j_m$, $m \leq n+1$, of $1, 2, \ldots, n+1$. Therefore, we can rearrange the sojourn times in a state trajectory such that first k_1 intervals are of rate r_1, the next k_2 intervals are of rate r_2, and so on. Then, sum of the sojourn times for rates rewards r_i, $i = 1, 2, \ldots, K+1$, can be expressed as

$$\begin{aligned}
l_1 &= Y_1 + Y_2 + \ldots + Y_{k_1}, \\
l_2 &= Y_{k_1+1} + Y_{k_1+2} + \ldots + Y_{k_1+k_2}, \\
&\vdots \\
l_{K+1} &= Y_{k_1+\cdots+k_K+1} + Y_{k_1+\cdots+k_K+2} + \ldots + Y_{k_1+\cdots+k_{K+1}}.
\end{aligned}$$

Given this, the conditional distribution function can be formulated as

$$P\{AR(t) \leq r \mid n, \mathbf{k}\} = P\left\{\frac{1}{t} \times \begin{bmatrix} r_1(Y_1 + \ldots + Y_{k_1}) + \\ r_2(Y_{k_1+1} + \ldots + Y_{k_1+k_2}) + \\ \vdots \\ r_{K+1}(Y_{k_1+\cdots+k_K+1} + \ldots + Y_{k_1+\cdots+k_{K+1}}) \end{bmatrix} \leq r\right\}.$$

Furthermore, by scaling the rate rewards such that $b_i = r_i - r_{K+1}$, for $i = 1, 2, \ldots, K+1$, the conditional distribution can be expressed as

$$P\{AR(t) \leq r \mid n, \mathbf{k}\} = P\left\{\frac{1}{t} \times \begin{bmatrix} b_1(Y_1 + \ldots + Y_{k_1}) + \\ b_2(Y_{k_1+1} + \ldots + Y_{k_1+k_2}) + \\ \vdots \\ b_K(Y_{k_1+\cdots+k_{K-1}+1} + \ldots + Y_{k_1+\cdots+k_K}) \end{bmatrix} \leq r - r_{K+1}\right\}.$$

Now define the random variables V_i such that

$$\begin{aligned} V_1 &= \tfrac{Y_1+\ldots+Y_{k_1}}{t}, \\ V_2 &= \tfrac{Y_{k_1+1}+\ldots+Y_{k_1+k_2}}{t}, \\ &\vdots \\ V_K &= \tfrac{Y_{k_1+\cdots+k_{K-1}+1}+\ldots+Y_{k_1+\cdots+k_K}}{t}. \end{aligned}$$

As given in [21], these random variables $(V_1, V_2, \ldots, V_K)$ have a K-variate Drichlet probability density function (pdf)

$$f(v_1, \ldots, v_K) = \frac{\Gamma(n+1)}{\prod_{i=1}^{K+1}\Gamma(k_i)} \prod_{i=1}^{K} (v_i)^{(k_i-1)} \left\{1 - \sum_{i=1}^{K} v_i\right\}^{(k_{K+1}-1)},$$

at any point in the simplex:

$$\left\{(v_1, \ldots, v_K) : v_i \geq 0, i = 1, \ldots, K, \sum_{i=1}^{K} v_i \leq 1\right\},$$

in the K-dimensional real space and zero outside.

Using these random variables, we can then express the conditional time-averaged accumulated reward as

$$P\{AR(t) \leq r \mid n, \mathbf{k}\} = P\left\{\sum_{i=1}^{K} b_i V_i \leq b\right\}, \text{ where } b = r - r_{K+1}. \tag{4.6}$$

Given (4.6), the problem of computing conditional time-averaged accumulated reward is reduced to the computation of the distribution of a linear combination of random variables with K-variate Drichlet distribution. To compute this, we use a result given in [21] for the density function of this distribution. In particular, the density function of $AR(t)$, given $\mathbf{k}$ and n is

$$f_{AR(t)}(b \mid n, \mathbf{k}) = \prod_{a=1}^{K} b_a^{-k_a} \sum_{l=1}^{K} \sum_{m=1}^{k_l} C_{l,m} \frac{\chi(b/b_l)\chi(1-(b/b_l))b^{m-1}(1-(b/b_l))^{n-m}}{B(m, n-m+1)}, \quad (4.7)$$

where

$$\chi(z) = \begin{cases} 1, & \text{if } z > 0, \\ 0, & \text{else,} \end{cases}$$
$$B(i,j) = \frac{\Gamma(i)\Gamma(j)}{\Gamma(i+j)},$$

and $C_{l,m}$ are constant coefficients for the partial fraction of

$$G(s) = \frac{1}{(s+\beta_1)^{k_1}(s+\beta_2)^{k_2}\ldots(s+\beta_k)^{k_K}},$$

where $\beta_i = (1/b_i)$, $i = 1, 2, \ldots, K$, are zeros of $1/G(s)$ and k_i, $i = 1, 2, \ldots, K$, denote their order.

The PDF of $AR(t)$, given n and $\mathbf{k}$, is thus (by integration)

$$F_{AR(t)}(b \mid n, \mathbf{k}) = \prod_{a=1}^{K} b_a^{-k_a} \sum_{l=1}^{K} \sum_{m=1}^{k_l} \frac{C_{l,m}}{B(m, n-m+1)} \int_0^b \chi(s/b_l)\chi(1-(s/b_l))s^{m-1}(1-(s/b_l))^{n-m} ds. \quad (4.8)$$

To evaluate the integral in (4.8), we must know the regions in which the unit step functions have a non-zero value. Note that

$$\chi(s/b_l) = \begin{cases} 1 & if\ s > 0, \\ 0 & else. \end{cases}, \text{ and } \quad \chi(1-(s/b_l)) = \begin{cases} 1 & if\ s < b_l, \\ 0 & else. \end{cases}.$$

Thus

$$F_{AR(t)}(b \mid n, \mathbf{k}) = \prod_{a=1}^{K} b_a^{-k_a} \sum_{l=1}^{K} \sum_{m=1}^{k_l} \frac{C_{l,m}}{B(m, n-m+1)} \begin{cases} \int_0^{b_l} s^{m-1}(1-(s/b_l))^{n-m} ds & if\ b > b_l \\ \int_0^{b} s^{m-1}(1-(s/b_l))^{n-m} ds & if\ 0 \le b \le b_l \end{cases}.$$

After solving the integrals

$$F_{AR(t)}(b \mid n, \mathbf{k}) = \prod_{a=1}^{K} b_a^{-k_a} \sum_{l=1}^{K} \sum_{m=1}^{k_l} C_{l,m} \begin{cases} b_l^m & if\ b \ge b_l \\ \frac{b^m\ H[m,m-n,m+1,(b/b_l)]}{m\ B(m,n-m+1)} & if\ 0 \le b < b_l \end{cases}, \quad (4.9)$$

where $H[m, m-n, m+1, (b/b_l)]$ is a hyper-geometric function defined by the following series,

$$1 + \sum_{u=1}^{\infty} \frac{m(m+1)\ldots(m+u-1)\ (m-n)(m-n+1)\ldots(m-n+u-1)}{u!\ (m+1)(m+2)\ldots(m+u)} \left(\frac{b}{b_l}\right)^n. \quad (4.10)$$

Similarly, the complementary probability distribution function of $AR(t)$, given $\mathbf{k}$ and n, denoted $(\overline{F}_{AR(t)}(b \mid n, \mathbf{k}))$, can be computed as

$$\overline{F}_{AR(t)}(b \mid n, \mathbf{k}) = \begin{cases} 0 & if\ b \geq b_l \\ \prod_{a=1}^{K} b_a^{-k_a} \sum_{l=1}^{K} \sum_{m=1}^{k_l} C_{l,m} \left(b_l^m - \frac{b^m\ H[m,m-n,m+1,(b/b_l)]}{m\ B(m,n-m+1)}\right) & if\ 0 \leq b < b_l \end{cases}. \quad (4.11)$$

In Equation 4.11, we assume that $0 \leq k_l \leq n$ such that $\sum_{l=1}^{K} k_l = n+1$. When $k_l = n+1$, the Markov chain only visits states with rate reward r_l. The conditional probability distribution is then 1, if the rate reward b_l is greater than b, or 0, if it is less. More generally, suppose x out of K rate rewards are greater than b, i.e., $b_1 > b_2 > \cdots > b_x > b$, then

$$\overline{F}_{AR(t)}(b \mid n, \mathbf{k}) = \prod_{a=1}^{K} b_a^{-k_a} \sum_{l=1}^{x} \sum_{m=1}^{k_l} C_{l,m} \times b_l^m \times \left(1 - \left(\frac{b}{b_l}\right)^m \frac{H[m, m-n, m+1, (b/b_l)]}{m\ B(m, n-m+1)}\right). \quad (4.12)$$

By looking at (4.12), one can easily realize that its computation by straightforward means, like the Weisberg result, is susceptible to numerical errors. The main reason for these errors is the computation of factorials in computing $H[m, m-n, m+1, (b/b_l)]$, $B(m, n-m+1)$, and $C_{l,m}$. These expressions result in extremely large numbers which when subtracted using normal floating point arithmetic may introduce significant loss of precision. However, with appropriate manipulation, and selective use of extended precision math, these problems can be avoided. In the remainder of this section, we derive three lemmas, which show how to derive a computationally efficient and numerically stable means to compute $\overline{F}_{AR(t)}(b \mid n, \mathbf{k})$.

For simplicity, in the remaining discussion, let

$$F_l(n, m) = \left(1 - \left(\frac{b}{b_l}\right)^m \frac{H[m, m-n, m+1, (b/b_l)]}{m\ B(m, n-m+1)}\right). \quad (4.13)$$

Accordingly,

$$\overline{F}_{AR(t)}(b \mid n, \mathbf{k}) = \prod_{a=1}^{K} b_a^{-k_a} \sum_{l=1}^{x} \sum_{m=1}^{k_l} C_{l,m} \times b_l^m \times F_l(n, m),$$

which after algebraic manipulation can be written as

$$\overline{F}_{AR(t)}(b \mid n, \mathrm{k}) \;=\; \sum_{l=1}^{x} \prod_{\substack{a=1 \\ a \neq l}}^{K} b_a^{-k_a} \sum_{m=1}^{k_l} \frac{1}{b_l^{(k_l-m)}} \times C_{l,m} \times F_l(n,m).$$

A Computation of $C_{l,m}$

To compute $C_{l,m}$, the constant coefficients of the partial fraction expansion of $G(s)$, we need to compute higher derivatives of the function $G_l(s)$ (the ith derivative of G_l is denoted $G_l^{(i)}$), where

$$G_l(s) \;\equiv\; (s+\beta_l)^{k_l} G(s) \;=\; \prod_{\substack{a=1 \\ a \neq l}}^{K} \frac{1}{(s+\beta_a)^{k_a}},$$

since

$$C_{l,m} \;=\; \lim_{s \to (-\beta_l)} \frac{1}{(k_l-m)!}\, G_l^{(k_l-m)}(s) \;\equiv \frac{1}{(k_l-m)!}\, G_l^{(k_l-m)}(-\beta_l). \tag{4.14}$$

To compute $C_{l,m}$, we use Bell polynomials [23, 24], the higher derivatives of a composition of two functions. In particular, let $h = f \circ g$ be the composition of f with g, that is, $h(t) = f(g(t))$. Denote n-fold application of an operator d/dt by $d^{(n)}/dt^{(n)}$. Furthermore, denote $h_n = d^{(n)}h/dt^{(n)}$, $f_n = d^{(n)}f(g)/dg^{(n)}$, and $g_n = d^{(n)}g/dt^{(n)}$. Assume that the functions f, g, and h have derivatives of all the orders. Then the Bell polynomials can be expressed as follows:

$$\begin{aligned} h_1 &= f_1 g_1 \\ h_2 &= f_2 g_1^2 + f_1 g_2 \\ h_3 &= f_3 g_1^3 + f_2(3 g_1 g_2) + f_1 g_3 \\ &\vdots \end{aligned}$$

In general, $h_q = \sum_{p=1}^{q} f_p \alpha_{p,q}$, where α_{pq}'s are polynomials in g_i's that do not depend upon the choice of f. According to the Faa DiBurno's formula, [23], the α_{pq} can be explicitly

represented as

$$\alpha_{p,q} = \sum_{\substack{n_i \geq 0 \\ \sum_{i=1}^{q} n_i = p \\ \sum_{i=1}^{q} i n_i = q}} \frac{q!}{(1!)^{n_1}(2!)^{n_2}\ldots(q!)^{n_q}(n_1!)(n_2!)\ldots(n_q!)} g_1^{n_1} g_2^{n_2} \ldots g_q^{n_q}. \tag{4.15}$$

Note that the summation is over all n_i such that $n_i \geq 0$, $\sum_{i=1}^{q} n_i = p$, and $\sum_{i=1}^{q} i n_i = q$. The computation of $\alpha_{p,q}$ for large values of q thus becomes prohibitive. This, in turn, makes the computation of higher derivatives of a composition of two functions impractical. However, if the function $h = f(g)$ is defined such that all its derivatives with respect to g are identical, then an efficient recursive expression can be written to compute the higher derivatives. This fact is expressed formally in the following lemma, which due to lack of space is stated without proof.

Lemma 1 *Let $h = f(g)$ be a function such that $h = f_1 = f_2 = \ldots$. Then h_q, for some $q \geq 1$, can be computed as*

$$h_q = \sum_{i=1}^{q-1} \binom{q-1}{q-i} g_i h_{q-i} + g_q h \ .$$

Proof: See *Appendix.*

We can use Lemma 1 to efficiently compute the higher derivatives of $G_l(-\beta_l)$. To do this, we define $f(s) = \exp(s)$, $h(s) = f(g(s))$, and

$$g(s) = - \sum_{\substack{a=1 \\ a \neq l}}^{K} k_a \ln(s + \beta_a).$$

Note that $h(s) = \exp(g(s)) = G_l(s)$. Furthermore, observe that

$$\frac{d^{(p)}}{dg} G_l(g(s)) = \exp(g(s)), \text{ and}$$

$$G_l(-\beta_l) \;=\; \prod_{\substack{a=1 \\ a \neq l}}^{K} (\beta_a - \beta_l)^{-k_a}, \tag{4.16}$$

for $p \geq 1$. Using Lemma 1, $G^{(k_l-m)}(-\beta_l) = h_{k_l-m}$ can then be expressed as

$$G_l^{(k_l-m)}(-\beta_l) \;=\; \sum_{i=1}^{k_l-m-1} \left(\begin{array}{c} k_l-m-1 \\ k_l-m-i \end{array} \right) g_i G_l^{(k_l-m-i)}(-\beta_l) + g_{k_l-m} G_l(-\beta_l), \tag{4.17}$$

where

$$g_p \;=\; (-1)^p (p-1)! \sum_{\substack{a=1 \\ a \neq l}}^{K} \frac{k_a}{(\beta_a - \beta_l)^p} \quad \text{for } p \geq 1.$$

By looking at Equation 4.17 and using an inductive argument, it can be realized that

$$G_l^{(k_l-m)}(-\beta_l) \;=\; G_l(-\beta_l) \times \tilde{G}_l^{(k_l-m)}(-\beta_l), \text{ where} \tag{4.18}$$

$$\tilde{G}_l^{(k_l-m)}(-\beta_l) \;=\; \sum_{i=1}^{k_l-m-1} \left(\begin{array}{c} k_l-m-1 \\ k_l-m-i \end{array} \right) g_i \tilde{G}_l^{(k_l-m-i)}(-\beta_l) + g_{k_l-m}. \tag{4.19}$$

Now, using Equations 4.14 and 4.18, $C_{l,m}$ can be expressed as

$$C_{l,m} \;=\; \frac{G_l(-\beta_l) \times \tilde{G}_l^{(k_l-m)}(-\beta_l)}{(k_l-m)!}$$

Accordingly,

$$\overline{F}_{AR(t)}(b \mid n, \mathbf{k}) \;=\; \sum_{l=1}^{x} \prod_{\substack{a=1 \\ a \neq l}}^{K} b_a^{-k_a} \times G_l(-\beta_l) \sum_{m=1}^{k_l} \frac{1}{b_l^{(k_l-m)}} \times \frac{\tilde{G}_l^{(k_l-m)}(-\beta_l)}{(k_l-m)!} \times F_l(n,m).$$

Since $\beta_i = (1/b_i)$, using Equation 4.16, we can write

$$G_l(-\beta_l) \;=\; \prod_{\substack{a=1 \\ a \neq l}}^{K} \left(\frac{b_a b_l}{b_l - b_a} \right)^{k_a}.$$

Finally,

$$\overline{F}_{AR(t)}(b \mid n, \mathbf{k}) = \sum_{l=1}^{x} \prod_{\substack{a=1 \\ a \neq l}}^{K} \left(\frac{b_l}{b_l - b_a}\right)^{k_a} \sum_{m=1}^{k_l} \frac{1}{b_l^{(k_l-m)}} \times \frac{\tilde{G}_l^{(k_l-m)}(-\beta_l)}{(k_l - m)!} \times F_l(n, m).$$

In the remainder of this section, we present two more lemmas, one which gives a recursive expression to compute

$$H_l(m) = \frac{1}{b_l^{(k_l-m)}} \times \frac{\tilde{G}_l^{(k_l-m)}(-\beta_l)}{(k_l - m)!}$$

and another which gives a recursive expression for computing $F_l(n, m)$. Based on these results, we can then present an algorithm to compute $\overline{F}_{AR(t)}(b \mid n, \mathbf{k})$. Proofs of the lemmas are omitted due to space limitations.

B Computation of $H_l(m)$

Lemma 2 *For* $1 \leq m \leq k_l - 1$,

$$H_l(m) = \frac{1}{k_l - m} \left[\sum_{i=1}^{k_l-m-1} g_i'' H_l(m+i) + g_{k_l-m}'' \right],$$

where

$$g_i'' = (-1)^i \sum_{\substack{a=1 \\ a \neq i}}^{K} k_a \left(\frac{b_a}{b_l - b_a}\right)^i,$$

and for $m = k_l$, $H_l(m) = 1$.

Proof: See *Appendix.*

C Computation of $F_l(n, m)$

Lemma 3 *For* $m = 1$,

$$F_l(n, m) = \left(1 - \frac{b}{b_l}\right)^n,$$

and for $2 \leq m \leq n$,

$$F_l(n,m) = \binom{n}{m-1} \left(\frac{b}{b_l}\right)^{m-1} \left(1 - \frac{b}{b_l}\right)^{n-(m-1)} + F_l(n,m-1).$$

Proof: See *Appendix.*

V Algorithm to Compute $\overline{F}_{AR(t)}(b \mid n, \mathbf{k})$

Recall that we can express the complementary conditional distribution in terms of $H_l(m)$ and $F_l(n,m)$ as

$$\overline{F}_{AR(t)}(b \mid n, \mathbf{k}) = \sum_{l=1}^{x} \prod_{\substack{a=1 \\ a \neq l}}^{K} \left(\frac{b_l}{b_l - b_a}\right)^{k_a} \sum_{m=1}^{k_l} H_l(m) \times F_l(n,m).$$

While the recursions developed in the previous section give us an efficient method to calculate $H_l(m)$ and $F_l(n,m)$, $H_l(m)$ will take on extremely large positive and negative values for certain $\mathbf{k}$ and large n. These values can cause loss of precision in a practical implementation of the algorithm. This loss of precision can be avoided, to some extent, by writing a recursion for $H_l(m) \times F_l(n,m)$ directly, which we denote by $X_l(n,m)$. In this notation,

$$\overline{F}_{AR(t)}(b \mid n, \mathbf{k}) = \sum_{l=1}^{x} \prod_{\substack{a=1 \\ a \neq l}}^{K} \left(\frac{b_l}{b_l - b_a}\right)^{k_a} \sum_{m=1}^{k_l} X_l(n,m). \tag{4.20}$$

Then, using Lemma 2, we can write a recursive expression to compute $X_l(n,m)$. Since $\frac{X_l(n,m)}{F_l(n,m)} = H_l(m)$, Lemma 2 says that

$$\frac{X_l(n,m)}{F_l(n,m)} = \frac{1}{(k_l - m)} \sum_{i=1}^{k_l - m - 1} g_i'' \times \frac{X_l(n,m+i)}{F_l(n,m+i)} + g_{k_l - m}'',$$

for $1 \leq m \leq k_l - 1$. This implies that

$$X_l(n,m) = \frac{1}{(k_l - m)} \sum_{i=1}^{k_l - m - 1} g_i'' \times X_l(n,m+i) \times \frac{F_l(n,m)}{F_l(n,m+i)} + g_{k_l - m}'' \times F_l(n,m). \tag{4.21}$$

Note for $m = k_l$, $X_l(n,m) = F_l(n,m)$, since $H_l(k_l) = 1$. By writing the recursion this way, we avoid the problem with large $H_l(m)$, since the $F_l(n,m)$ are probabilities.

The formulation presented in (4.20) and (4.21) diminishes the numerical difficulty in computing $\overline{F}_{AR(t)}(b \mid n, \mathbf{k})$, but still requires careful implementation to avoid loss of precision in the computation. In particular, the formulation calls for the addition of a large number of positive and negative numbers, which will be large in magnitude for certain $\mathbf{k}$ and large n. These numbers may be very close to one another, since when added together they should equal a potentially very small probability. The main loss of precision in such cases is caused by the subtraction of two numbers which are close in value. In this case, a large shift to the left will be made after the operation to normalize the result, and many digits of precision will be lost.

Keeping this in mind, to minimize the number of subtractions, we split $X_l(n,m)$ into positive $X_l^{pos}(n,m) > 0$ and negative $X_l^{neg}(n,m) < 0$ parts such that

$$X_l(n,m) \;=\; X_l^{pos}(n,m) + X_l^{neg}(n,m).$$

Similar to the recursion in Equation 4.21, for $1 \le m \le k_b - 1$, recursions for $X_l^{pos}(n,m)$ and $X_l^{neg}(n,m)$ can be expressed as

$$X_l^{pos}(n,m) \;=\; \frac{1}{(k_l - m)} \left[\sum_{i=1}^{k_l-m-1} I(g_i'' > 0) g_i'' X_l^{pos}(n, m+i) \frac{F_l(n,m)}{F_l(n,m+i)} + \sum_{i=1}^{k_l-m-1} I(g_i'' < 0) g_i'' X_l^{neg}(n, m+i) \frac{F_l(n,m)}{F_l(n,m+i)} \;+\; I(g_{k_l-m}'' > 0) g_{k_l-m}'' \right] \tag{4.22}$$

and

$$X_l^{neg}(n,m) \;=\; \frac{1}{(k_l - m)} \left[\sum_{i=1}^{k_l-m-1} I(g_i'' > 0) g_i'' X_l^{neg}(n, m+i) \frac{F_l(n,m)}{F_l(n,m+i)} + \sum_{i=1}^{k_l-m-1} I(g_i'' < 0) g_i'' X_l^{pos}(n, m+i) \frac{F_l(n,m)}{F_l(n,m+i)} \;+\; I(g_{k_l-m}'' < 0) g_{k_l-m}'' \right], \tag{4.23}$$

where $I(x)$ is an indicator function (returns 1 if x is true and else 0) and for $m = k_l$, $X_l^{pos}(n,m) = F_l(n,m)$ and $X_l^{neg}(n,m) = 0$.

Given that all the additions are performed in ascending order of absolute magnitudes, then, for each shifted reward $b_l > b$, Equations 4.22 and 4.23 yield an accurate (to the

precision of the computation) positive value and negative value. Both these values are then multiplied by

$$\prod_{\substack{a=1 \\ a \neq l}}^{K} \left(\frac{b_l}{b_l - b_a}\right)^{k_a},$$

which is denoted z in the algorithm below. Then, these two resulting numbers with powers greater than 0 are subtracted to yield a probability. Since the subtraction will result in a probability, the order of magnitude of the two numbers will always be the same (called *magnitude* in the following algorithm). This implies that to obtain an answer accurate to some specified precision, we need a decimal-digit precision in the computation equal to the sum of *magnitude* and the desired precision. If normal floating point math (as per the IEEE standard) does not provide the required precision, the algorithm obtains the required precision by invoking multi-precision routines, rolling back, and recomputing the required terms and sums. Thus, by the use of selective multi-precision, the algorithm always achieves the desired precision and, hence, is numerically stable. Since there is only a single subtraction needed for each shifted rate reward, each summation will at most roll back once, leading to an efficient algorithm.

More precisely, the algorithm to compute $\overline{F}_{AR(t)}(b \mid n, \mathbf{k})$ is as follows. (Note that the given algorithm does not consider impulse rewards. However, an extension of the algorithm to include impulse rewards is straightforward and can be done in a manner similar to one presented by Qureshi and Sanders [25].)

Algorithm 1 *(Compute the complementary conditional distribution, $\overline{F}_{AR(t)}(b \mid n, \mathbf{k})$, using (4.20) and (4.21).)*

$\overline{F}_{AR(t)}(b \mid n, \mathbf{k}) = 0.$

For $l = 1$ *to* x

    For $m = 1$ *to* k_l
        Compute $F_l(n, m)$ *by using recursion given in Lemma 3.*
    End(For).

    For $m = k_l$ *to* 1
        If $(m = k_l)$
            $X_l^{pos}(n, k_l) = F_l(n, k_l)$, *and* $X_l^{neg}(n, k_l) = 0.$
        Else

Use Equation 4.22 to compute $X_l^{pos}(n,m)$.
Use Equation 4.23 to compute $X_l^{neg}(n,m)$.
End(For).

Let $X_{negative}$ *be the sum of all* $X_l^{neg}(n,m)$, *added in the ascending order of their absolute magnitudes.*

Let $X_{positive}$ *be the sum of all* $X_l^{pos}(n,m)$, *added in the ascending order of their magnitudes.*

$X_{positive} = z \times X_{positive}$.
$X_{negative} = z \times X_{negative}$.

Let magnitude be the logarithm (base 10) of $X_{positive}$.

Let needed digits = magnitude + desired precision.

If (IEEE floating point precision > needed digits)
$X_{total} = X_{positive} + X_{negative}$
Else
Recompute X_{total} *by recomputing* l_{th} *iteration of for loop with needed digits precision.*

$\overline{F}_{AR(t)}(b \mid n, \mathbf{k}) = \overline{F}_{AR(t)}(b \mid n, \mathbf{k}) + X_{total}$.

End(For)

This algorithm, together with the path-based approach described in Section III, provide the basis for stable and efficient calculation of the distribution of time-averaged reward accumulated over a finite interval. The overall algorithm to compute the distribution of accumulated reward starts by first generating possible paths and computing their respective probabilities. The probabilities of paths which correspond to identical $\mathbf{k}$ vectors are added together, while the probabilities of paths corresponding to distinct $\mathbf{k}$ vectors are stored separately. This exploration of paths and computation of path probabilities continues until either all paths with probabilities greater than or equal to the discarding weight are explored or the memory used reaches a preset machine-dependent limit. After reaching one of these conditions, the conditional distribution of accumulated reward is computed for each $\mathbf{k}$ vector generated, and after being multiplied by the probability of the explored paths corresponding to the $\mathbf{k}$ vector, is added together with the results from other paths to form the unconditional distribution. If all paths with probability greater than the threshold have not yet been

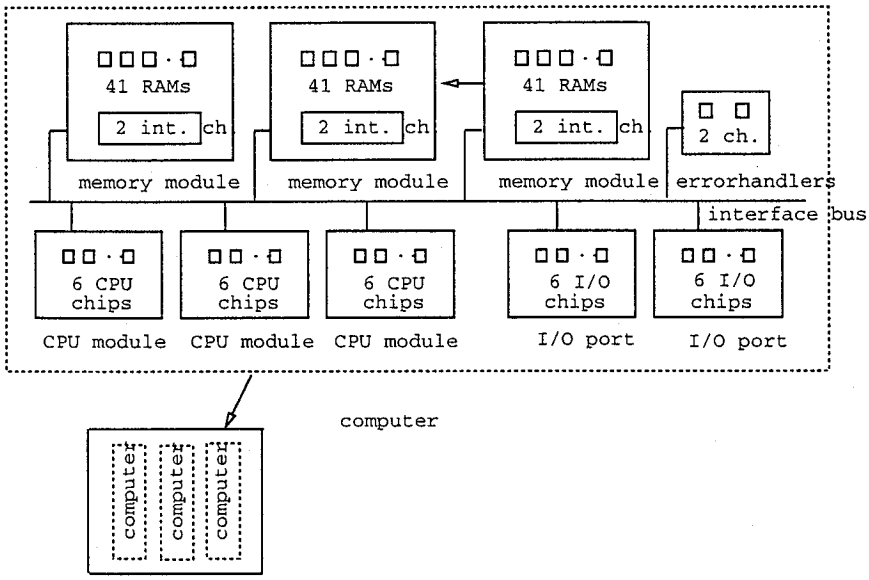

Figure 4.1: Fault-tolerant parallel computing system

explored, the process is repeated, exploring more paths and then computing the required conditional distributions.

To test the algorithm on a non-trivial example, we have implemented it in C++, using the multi-precision library LiDIA [26]. The implementation has been used for several large examples and works well when the number of paths required to compute a desired accuracy is not more than tens of millions. One such example, with over one-half million states is given in the next section.

VI Example Study

The applicability of our approach is illustrated by considering the performability analysis of a highly redundant fault-tolerant multiprocessor system from Lee et al. [27]. The system, at the highest level, consists of three non-repairable computers. (Lee et al. considered a variable number of computers.) (Note that our algorithm applies to repairable, as well as non-repairable systems.) As shown in Figure 4.1, each computer is composed of three memory modules, of which one is a spare; three CPU units, of which one is spare; two I/O ports, of which one is spare; and two non-redundant error handling chips. A computer is classified as operational if at least two memory modules, two CPU units, one I/O port, and the two error-handling chips are working.

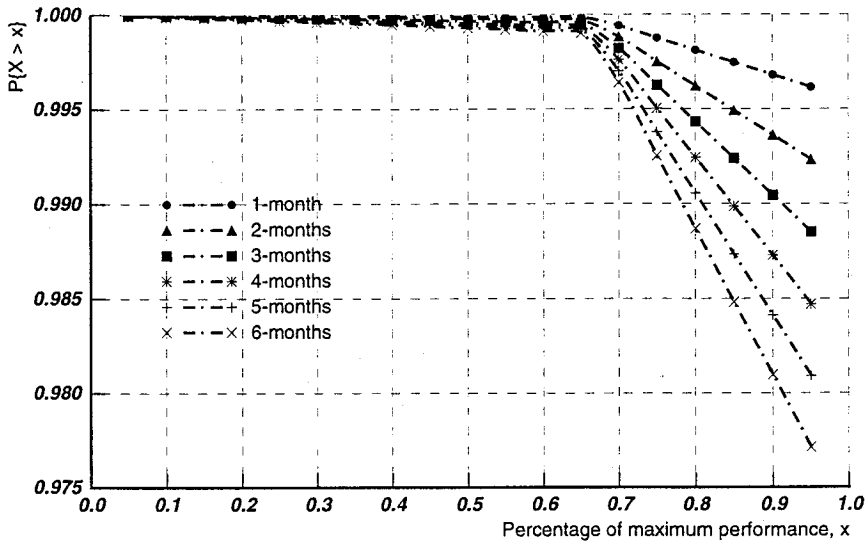

Figure 4.2: Performability Distribution

Internally, each memory module consists of 41 RAMs, out of which two are spare chips and two are interface chips. Each CPU unit and each I/O port consists of six non-redundant chips. A memory module is classified as operational if at least 39 of its 41 RAM chips and its two interface chips are working. Each component failure has coverage probabilities associated with it. The coverage probabilities are state dependent, and there are multiple levels of coverage (module, computer, multiprocessor). We used the same coverage probabilities and the failure rate for the chips as used in the original model by Lee et al. [27].

The performability measure considered is the complementary PDF of the fraction of jobs completed in the interval, considering faults, relative to the number of jobs that would complete in a fault-free system. More specifically, we consider a system where 300 jobs complete per second if 3 processors are working, 200 jobs complete per second if 2 processors are working, and 100 jobs complete per second if 1 processor is working. The measure, then, is the PDF of the number of jobs that complete divided by $300 * T$, where T is the length of the interval under consideration.

The model was represented as a stochastic activity network, as was done in [28]. Space does not permit presentation of the SAN model here. The Markov level representation of the model is very large, consisting of 463,268 states even if reduced base model construction [29] is used to detect symmetries in the process. To the best of our knowledge, this is by far the largest and most complicated model that has ever been solved for the distribution

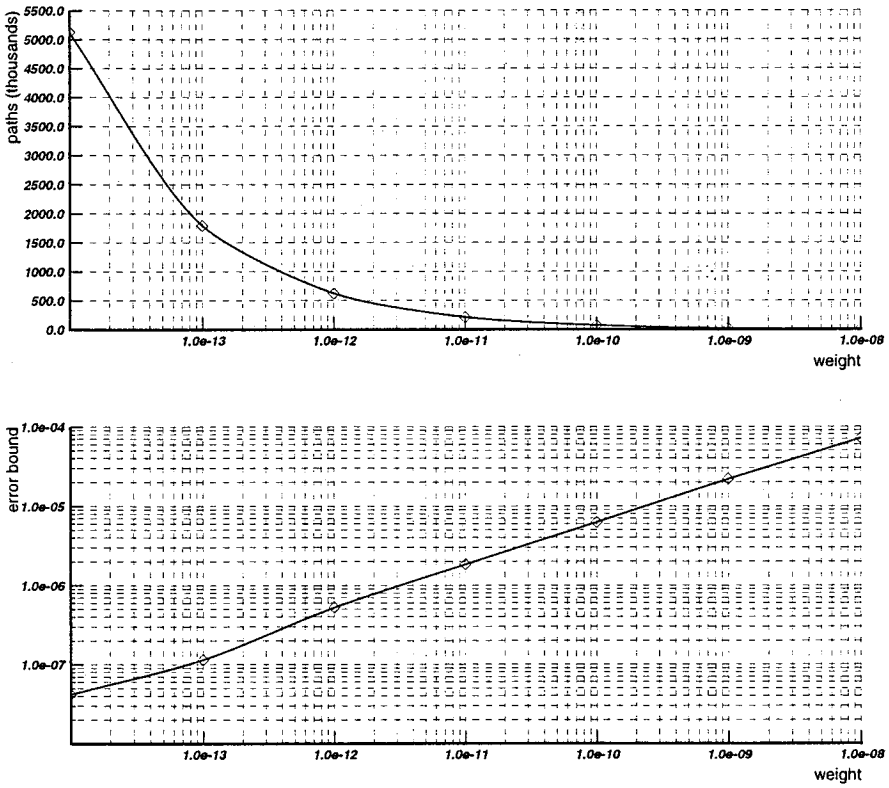

Figure 4.3: Effect of Discarding of Paths (t = one year)

of accumulated reward, and it illustrates the practicality of our approach.

We present results regarding both the performability of the considered system and the efficiency of the algorithm in the context of the example. In particular, Figure 4.2 depicts the complementary distribution of reward accumulated during time intervals of one to six months. Each line on the graph corresponds to a specific time interval, and it gives the probability that the system performs at a level higher than the value given on the x-axis. A discarding threshold value of $w = 10^{-11}$ was used for all time intervals, resulting in an accuracy of more than six digits for each result.

Figure 4.3 illustrates the effect of changing w on the number of paths considered and the bound on the error obtained due to discarding paths whose probabilities are less than the value w. For this example, we consider a utilization period of one year. Since the space required by previous approaches depends critically on the length of the interval,

this (reasonable) time point would result in an unreasonably large memory requirement for current non-path-based approaches. Note that the number of paths that need to be considered decreases dramatically with an increasing weight, while the error bound achieved remains reasonable. This illustrates the effectiveness of our approach on systems with up to millions of probable paths.

VII Conclusion

In this paper we have presented a new method for computing the distribution of time-averaged reward accumulated over a finite interval. The method is based on generating paths of the uniformized process which are probable, and computing the probability distribution of accumulated reward conditioned on each path. To make this practical, we have developed a new algorithm for computing the conditional distribution of accumulated reward which is computationally efficient and numerically stable. In addition, we have presented experimental results to show that the path-based approach is useful when solving Markov reward models with large state spaces and up to tens of millions of probable paths. We also have shown that, for the example considered, the number of paths required to obtain a result decreases dramatically as the path discarding threshold is increased while still obtaining reasonable accuracy.

These results bode well for the the practical use of reward models in performability modeling.

Acknowledgment

We would like to thank Luai Malhis, who built the SAN representation of the multiprocessor used as our example.

REFERENCES

[1] R. A. Howard, *Dynamic Probabilistic Systems (vol. II)*. John Wiley & Sons, 1971.

[2] J. F. Meyer, "Closed-form solutions of performability," *IEEE Transactions on Computers*, vol. 31, no. 7, pp. 648–657, 1982.

[3] D. G. Furchgott and J. F. Meyer, "A performability solution method for degradable non-repairable systems," *IEEE Transactions on Computers*, vol. 33, no. 6, pp. 550–554, 1984.

[4] L. Donatiello and B. R. Iyer, "Analysis of a composite performance reliability measure for fault-tolerant systems," *Journal of the ACM*, vol. 34, no. 1, pp. 179–199, 1987.

[5] A. Goyal and A. N. Tantawi, "Evaluation of performability for degradable computer systems," *IEEE Transactions on Computers*, vol. 36, no. 6, pp. 738–744, 1987.

[6] B. Ciciani and V. Grassi, "Performability evaluation of fault-tolerant satellite systems," *IEEE Transactions on Communications*, vol. 35, no. 4, pp. 403–409, 1987.

[7] V. G. Kulkarni, V. F. Nicola, R. M. Smith, and K. S. Trivedi, "Numerical evaluation of performability and job completion time in repairable fault-tolerant systems," in *Proceedings of 16th Int. Symposium on fault-tolerant computing*, (Vienna, Austria), pp. 252–257, July 1985.

[8] R. M. Smith, K. S. Trivedi, and A. V. Ramesh, "Performability analysis: measures, an algorithm, and a case study," *IEEE Transactions on Computers*, vol. 37, no. 2, pp. 406–417, 1987.

[9] K. R. Pattipati, Y. Li, and H. A. P. Blom, "A unified framework for the performability evaluation of fault-tolerant computer systems," *IEEE Transactions on Computers*, vol. 42, no. 3, pp. 312–326, 1993.

[10] E. de Souza e Silva and H. R. Gail, "Calculating availability and performability measures of repairable computer systems," *Journal of the ACM*, vol. 36, pp. 171–193, January 1989.

[11] L. Donatiello and V. Grassi, "On evaluating the cumulative performance distribution of fault-tolerant computer systems," *IEEE Transactions on Computers*, vol. 40, no. 11, pp. 1301–1307, 1991.

[12] E. de Souza e Silva and R. Gail, "Calculating transient distributions of cumulative reward," in *Proceedings of Sigmetrics/Performance-95*, (Ottawa, Canada), pp. 231–240, May 1995.

[13] R. Marie, A. L. Reibman, and K. S. Trivedi, "Transient analysis of acyclic Markov chains," *Performance Evaluation*, vol. 7, pp. 175–194, 1987.

[14] E. de Souza e Silva and P. M. Ochoa, "State space exploration in Markov models," *Performance Evaluation*, vol. 20, pp. 155–166, 1992.

[15] H. Weisberg, "The distribution of linear combinations of order statistics from the uniform distribution," *The Annals of Mathematical Statistics*, vol. 42, no. 2, pp. 704–709, 1995.

[16] E. de Souza e Silva and H. R. Gail, "Calculating cumulative operational time distributions of repairable computer systems," *IEEE Transactions on Computers*, vol. 35, pp. 322–332, April 1986.

[17] S. M. Ross, *Stochastic Processes.* John Wiley & Sons, 1982.

[18] A. M. Mood and F. A. Graybill, *Introduction to the Theory of Statistics.* McGraw-Hill Book Company, Inc., 1963.

[19] E. de Souza e Silva and R. H. Gail, "An algorithm to calculate transient distributions of cumulative reward," Tech. Rep. CSD-940021, University of California, Los Angeles, May 1994.

[20] J. F. Meyer, A. Movaghar, and W. H. Sanders, "Stochastic activity networks: Structure, behavior, and application," *Proceedings of International Workshop on Timed Petri Nets, Torino, Italy*, pp. 106–115, 1985.

[21] T. Matsunawa, "The exact and approximate distributions of linear combinations of selected order statistics from a uniform distribution," *The Annals of Institute of Statistical Mathematics*, vol. 37, pp. 1–16, 1985.

[22] S. M. Wilks, *Mathematical Statistics.* John Wiley & Sons, Inc., 1963.

[23] J. Riordan, *An Introduction to Combinatorial Analysis.* John Wiley & Sons, Inc., 1958.

[24] G. M. Constantine, *Combinatorial Theory and Statistical Design.* John Wiley & Sons, Inc., 1987.

[25] M. A. Qureshi and W. H. Sanders, "Reward model solution methods with impulse and rate rewards: An algorithm and numerical results," *Performance Evaluation*, vol. 20, pp. 413–436, 1994.

[26] V. . LiDIA Manual, "A library for computational number theory," tech. rep., Universitat des Saarlandes, August 1995.

[27] D. Lee, J. Abraham, D. Rennels, and G. Conte, "A numerical technique for the evaluation of large, closed fault-tolerant systems," in *Dependable Computing for Critical Applications*, pp. 95–114, Springer-Verlag, Wien, 1992.

[28] W. H. Sanders and L. M. Malhis, "Dependability evaluation using composed SAN-based reward models," *Journal of Parallel and Distributed Computing*, vol. 15, no. 3, pp. 238–254, 1992.

[29] W. H. Sanders and J. F. Meyer, "Reduced base model construction methods for stochastic activity networks," *IEEE Journal on Selected Areas in Communications*, vol. 9, pp. 25–36, 1991.

APPENDIX

Proof of Lemma 1
Basis Step. Let $q = 1$. Then the sum on the right is hg_1. From the definition of h_i, $h_1 = hg_1$.
Induction Hypothesis. Assume that for some $n \geq 1$,

$$h_n = \sum_{i=1}^{n} \left(\begin{array}{c} n-1 \\ n-i \end{array} \right) g_i h_{n-i} + g_n h \ .$$

Induction Step. Note $h_{n+1} = d^{(n+1)}h/dt^{(n+1)} = d(h_n)/dt$. Accordingly,

$$\begin{aligned}
h_{n+1} &= \frac{d}{dt}\left[h_n = \sum_{i=1}^{n} \left(\begin{array}{c} n-1 \\ n-i \end{array} \right) g_i h_{n-i} + g_n h\right] \\
&= \sum_{i=1}^{n} \left(\begin{array}{c} n-1 \\ n-i \end{array} \right) \frac{d}{dt}[g_i h_{n-1}] + \frac{d}{dt}[hg_n] \\
&= \left(\begin{array}{c} n-1 \\ n-1 \end{array} \right) \frac{d}{dt}[g_1 h_{n-1}] + \left(\begin{array}{c} n-1 \\ n-2 \end{array} \right) \frac{d}{dt}[g_2 h_{n-2}] + \cdots + \\
&\quad \left(\begin{array}{c} n-1 \\ 2 \end{array} \right) \frac{d}{dt}[g_{n-2} h_2] + \left(\begin{array}{c} n-1 \\ 1 \end{array} \right) \frac{d}{dt}[g_{n-1} h_1] + g_n \frac{d}{dt} h + g_{n+1} h \\
&= \frac{d}{dt}[g_1 h_{n-1}] + (n-1)\frac{d}{dt}[g_2 h_{n-2}] + \cdots + \frac{(n-1)(n-2)}{2}\frac{d}{dt}[g_{n-2} h_2] \\
&\quad + (n-1)\frac{d}{dt}[g_{n-1} h_1] + g_n h_1 + g_{n+1} h \\
&= g_1 h_n + g_2 h_{n-1} + (n-1) g_2 h_{n-1} + (n-1) g_3 h_{n-2} + \cdots + \\
&\quad \frac{(n-1)(n-2)}{2} g_{n-2} h_3 + \frac{(n-1)(n-2)}{2} g_{n-1} h_2 + \\
&\quad (n-1) g_{n-1} h_2 + (n) g_n h_1 + h g_{n+1} \\
&= g_1 h_n + (n) g_2 h_{n-1} + \cdots + \left[\frac{(n-1)(n-2)}{2} + (n-1)\right] g_{n-1} h_2 \\
&\quad (n) g_n h_1 + g_{n+1} h \\
&= \left(\begin{array}{c} n \\ n \end{array} \right) g_1 h_n + \left(\begin{array}{c} n \\ n-1 \end{array} \right) g_2 h_{n-1} + \cdots + \left(\begin{array}{c} n \\ 2 \end{array} \right) g_{n-1} h_2 + \\
&\quad \left(\begin{array}{c} n \\ 1 \end{array} \right) g_n h_1 + g_{n+1} h \\
&= \sum_{i=1}^{n+1} \left(\begin{array}{c} n \\ n+1-i \end{array} \right) g_i h_{n+1-i} + g_{n+1} h \ . \ \square
\end{aligned}$$

Proof of Lemma 2 For $m = k_l$, it is obvious that $H_l(m) = 1$. For $1 \leq m \leq k_l - 1$, consider

Equation (4.19, of the chapter), then we can write

$$H_l(m) = \frac{1}{b_l^{(k_l-m)}} \times \frac{1}{(k_l-m)!}\left[\sum_{i=1}^{k_l-m-1}\begin{pmatrix} k_l-m-1 \\ k_l-m-i \end{pmatrix} g_i \tilde{G}_l^{(k_l-m-i)}(-\beta_l) + g_{k_l-m}\right]$$

$$\begin{aligned} = \frac{1}{(k_l-m)!}\Bigg[&\begin{pmatrix} k_l-m-1 \\ k_l-m-1 \end{pmatrix} \times \frac{g_1}{b_l} \times \frac{\tilde{G}_l^{(k_l-m-1)}(-\beta_l)}{b_l^{k_l-m-1}} + \begin{pmatrix} k_l-m-1 \\ k_l-m-2 \end{pmatrix} \times \\ &\frac{g_2}{b_l^2} \times \frac{\tilde{G}_l^{(k_l-m-2)}(-\beta_l)}{b_l^{k_l-m-2}} + \begin{pmatrix} k_l-m-1 \\ k_l-m-3 \end{pmatrix} \times \frac{g_3}{b_l^3} \times \frac{\tilde{G}_l^{(k_l-m-3)}(-\beta_l)}{b_l^{k_l-m-3}} + \cdots + \\ &\begin{pmatrix} k_l-m-1 \\ 2 \end{pmatrix} \times \frac{g_{k_l-m-2}}{b_l^{(k_l-m-2)}} \times \frac{\tilde{G}_l^{(2)}}{b_l^2} + \begin{pmatrix} k_l-m-1 \\ 1 \end{pmatrix} \times \frac{g_{k_l-m-1}}{b_l^{(k_l-m-1)}} \times \frac{\tilde{G}_l^{(1)}}{b_l} \\ &+ \begin{pmatrix} k_l-m-1 \\ 0 \end{pmatrix} \times \frac{g_{k_l-m}}{b_l^{(k_l-m)}}\Bigg] \end{aligned} \tag{4.24}$$

Note

$$g_i = (-1)^i(i-1)! \sum_{\substack{a=1 \\ a\neq l}}^{K} k_a \left(\frac{b_a b_l}{b_l - b_a}\right)^i, \quad \text{for } i \geq 1.$$

Let

$$g_i = (-1)^i(i-1)! \sum_{\substack{a=1 \\ a\neq l}}^{K} k_a \left(\frac{b_a}{b_l - b_a}\right)^i, \quad \text{for } i \geq 1.$$

Accordingly, Equation 4.24 can be written as

$$\begin{aligned} = \frac{1}{(k_l-m)}\Bigg[&\begin{pmatrix} k_l-m-1 \\ k_l-m-1 \end{pmatrix} \times g_1' \times \left(\frac{1}{b_l^{(k_l-m-1)}} \times \frac{\tilde{G}_l^{(k_l-m-1)}(-\beta_l)}{(k_l-m-1)!}\right) + \\ &\begin{pmatrix} k_l-m-1 \\ k_l-m-2 \end{pmatrix} \times g_2' \times \left(\frac{1}{b_l^{(k_l-m-2)}} \times \frac{\tilde{G}_l^{(k_l-m-2)}(-\beta_l)}{(k_l-m-1)!}\right) + \begin{pmatrix} k_l-m-1 \\ k_l-m-3 \end{pmatrix} \\ &\times g_3' \times \left(\frac{1}{b_l^{(k_l-m-3)}} \times \frac{\tilde{G}_l^{(k_l-m-3)}(-\beta_l)}{(k_l-m-1)!}\right) + \cdots + \begin{pmatrix} k_l-m-1 \\ 2 \end{pmatrix} \times g_{k_l-m-2}' \times \\ &\left(\frac{1}{b_l^2} \times \frac{\tilde{G}_l^{(2)}(-\beta_l)}{(k_l-m-1)!}\right) + \begin{pmatrix} k_l-m-1 \\ 1 \end{pmatrix} \times g_{k_l-m-1}' \times \left(\frac{1}{b_l} \times \frac{\tilde{G}_l^{(1)}(-\beta_l)}{(k_l-m-1)!}\right) \\ &+ \begin{pmatrix} k_l-m-1 \\ 0 \end{pmatrix} \times \frac{g_{k_l-m}'}{(k_l-m-1)!}\Bigg]. \end{aligned}$$

Solving combinatorial and factorial terms, we get

$$= \frac{1}{(k_l-m)}\left[\frac{g_1'}{0!} \times \left(\frac{1}{b_l^{(k_l-m-1)}} \times \frac{\tilde{G}_l^{(k_l-m-1)}(-\beta_l)}{(k_l-m-1)!}\right) + \frac{g_2'}{1!} \times \left(\frac{1}{b_l^{(k_l-m-2)}}\right.\right.$$

$$\times \frac{\tilde{G}_l^{(k_l-m-2)}(-\beta_l)}{(k_l-m-2)!}\Bigg) + \frac{g_3^{'}}{2!} \times \left(\frac{1}{b_l^{(k_l-m-3)}} \times \frac{\tilde{G}_l^{(k_l-m-3)}(-\beta_l)}{(k_l-m-3)!}\right) +$$

$$\cdots + \frac{g_{k_l-m-2}^{'}}{(k_l-m-3)!} \times \left(\frac{1}{b_l^2} \times \frac{\tilde{G}_l^{(2)}(-\beta_l)}{2!}\right) + \frac{g_{k_l-m-1}^{'}}{(k_l-m-2)!} \times \Bigg(\frac{1}{b_l}$$

$$\times \frac{\tilde{G}_l^{(1)}(-\beta_l)}{1!}\Bigg) + \frac{g_{k_l-m}^{'}}{(k_l-m-1)!}\Bigg].$$

Let

$$g_1^{''} = (-1)^i \sum_{\substack{a=1 \\ a \neq i}}^{K} k_a \left(\frac{b_a}{b_l - b_a}\right)^i.$$

Accordingly,

$$= \frac{1}{(k_l-m)}\Bigg[g_1^{''} \times \left(\frac{1}{b_l^{(k_l-m-1)}} \times \frac{\tilde{G}_l^{(k_l-m-1)}(-\beta_l)}{(k_l-m-1)!}\right) + g_2^{''} \times \Bigg(\frac{1}{b_l^{(k_l-m-2)}} \times$$

$$\frac{\tilde{G}_l^{(k_l-m-2)}(-\beta_l)}{(k_l-m-2)!}\Bigg) + g_3^{''} \times \left(\frac{1}{b_l^{(k_l-m-3)}} \times \frac{\tilde{G}_l^{(k_l-m-3)}(-\beta_l)}{(k_l-m-3)!}\right) + \cdots +$$

$$g_{k_l-m-2}^{''} \times \left(\frac{1}{b_l^2} \times \frac{\tilde{G}_l^{(2)}(-\beta_l)}{2!}\right) + g_{k_l-m-1}^{''} \times \left(\frac{1}{b_l} \times \frac{\tilde{G}_l^{(1)}(-\beta_l)}{1!}\right) + g_{k_l-m}^{''}\Bigg],$$

which implies that for $1 \leq m \leq k_l - 1$,

$$H_l(m) = \frac{1}{k_l - m}\left[\sum_{i=1}^{k_l-m-1} g_i^{''} H_l(m+i) + g_{k_l-m}^{''}\right]. \quad \square$$

Proof of Lemma 3 To obtain a formal proof for Lemma 3, we first analyze the expression

$$R_l(n,m) = \frac{H[m, m-n, m+1, (b/b_l)]}{m\ B(m, n-m+1)}.$$

Since $1 \leq m \leq n$, the second term $(m-n)$ of the hyper-geometric function $H[m, m-n, m+1, (b/b_l)]$ either gets a value of 0 or a negative integer. Therefore, the summation representing the hyper-geometric function (see Equation 4.10) is truncated after $(n-m)$ terms. Based on this observation and the fact that

$$\frac{1}{m\ B(m, n-m+1)} = \binom{n}{m},$$

we can express $R_l(n,m)$ as

$$R_l(n,m) = \binom{n}{m} + \sum_{u=1}^{n-m} \frac{m(m+1)\cdots(m+u-1)}{u!} \times$$

$$\frac{(m-n)(m-n+1)\ldots(m-n+u)}{(m+1)(m+2)\ldots(m+u)} \left(\frac{b}{b_l}\right)^u \binom{n}{m}$$

$$
\begin{aligned}
&= \binom{n}{m} + \left[\frac{m \times (m-n)}{1!\ (m+1)}\left(\frac{b}{b_l}\right)^1 + \frac{m \times (m-n)(m-n+1)}{2!\ (m+2)}\left(\frac{b}{b_l}\right)^2 + \cdots + \right. \\
&\left. \frac{m \times (m-n)(m-n+1)\cdots(-1)}{(n-m)!\ (n)}\left(\frac{b}{b_l}\right)^{n-m}\right]\binom{n}{m} \\
&= \binom{n}{m} + \left[\frac{(-1)m \times (n-m)}{1!\ (m+1)}\left(\frac{b}{b_l}\right)^1 + \frac{(-1)^2 m \times (n-m)(n-m-1)}{2!\ (m+2)}\left(\frac{b}{b_l}\right)^2\right. \\
&\left. + \cdots + \frac{(-1)^{n-m} m \times (n-m)(n-m-1)\cdots(1)}{(n-m)!\ (n)}\left(\frac{b}{b_l}\right)^{n-m}\right]\binom{n}{m} \\
&= \binom{n}{m} + \left[\frac{(-1)m \times n(n-1)\cdots(n-m)}{!1\ (m+1)!}\left(\frac{b}{b_l}\right)^1 + \frac{(-1)^2 m \times m(m+1)}{2!} \times \right. \\
&\left. \frac{n(n-1)\cdots(n-m+1)}{(m+2)!}\left(\frac{b}{b_l}\right)^2 + \cdots + \frac{(-1)^{n-m} n!}{(n-m)!\ (m-1)!}\left(\frac{b}{b_l}\right)^{n-m}\right]
\end{aligned}
$$

$$
\begin{aligned}
R_l(m,n) = &\binom{n}{m} + \left[(-1)\binom{m}{1}\binom{n}{m+1}\left(\frac{b}{b_l}\right)^1 + (-1)^2\binom{m+1}{2}\right. \\
&\left.\binom{n}{m+2}\left(\frac{b}{b_l}\right)^2 + \cdots + (-1)^{n-m}\binom{n-1}{n-m}\binom{n}{n}\left(\frac{b}{b_l}\right)^{n-m}\right].
\end{aligned} \tag{4.25}
$$

Using Equation 4.25, now we present a lemma which will be used in writing the proof for Lemma 3.

Lemma 4 *Let* $y = (t/b_l)$. *For* $2 \leq m \leq n$,

$$
R_l(n,m) = \frac{(-1)^m}{(m-1)!}\frac{d}{dy}^{(m-1)}\left(\frac{F_l(n,1)-1}{y}\right).
$$

Proof: By induction on m.
Basic Step. Consider $m = 2$. From Equation (4.25),

$$
R_l(n,2) = \binom{n}{2} + (-1)2\binom{n}{3}y + (-1)^2 3\binom{n}{4}y^2 + \cdots + (-1)^{n-2}(n-1)\binom{n}{n}y^{n-2}. \tag{4.26}
$$

From combinatorial theory, we know

$$
(1-y)^n = \sum_{r=0}^{n}\binom{n}{r}y^r, \tag{4.27}
$$

which implies that

$$
\frac{(1-y)^n - 1}{y} = (-1)\binom{n}{1} + (-1)^2\binom{n}{2}y + (-1)^3\binom{n}{3}y^2 + \cdots + (-1)^n\binom{n}{n}y^{n-1}
$$

Differentiate both sides with respect to y,

$$\begin{aligned}\frac{d}{dy}\left[\frac{(1-y)^n-1}{y}\right] &= 0+(-1)^2\binom{n}{2}+(-1)^3 2\binom{n}{3}y+(-1)^4 3\binom{n}{4}y^2 \\ &= +\cdots+(-1)^n(n-1)\binom{n}{n}y^{n-2}\end{aligned}$$

$$=\binom{n}{2}+(-1)2\binom{n}{3}y+(-1)^2 3\binom{n}{4}y^2+\cdots+(-1)^{n-2}(n-1)\binom{n}{n}y^{n-2}. \quad (4.28)$$

From Equations (4.26) and (4.28),

$$R_l(n,2) = \frac{d}{dy}\left[\frac{F_l(n,1)-1}{y}\right].$$

Induction Hypothesis. Assume that for some $3 \leq i < n$,

$$R_l(n,i) = = \frac{(-1)^i}{(i-1)!}\frac{d}{dy}^{(i-1)}\left(\frac{F_l(n,1)-1}{y}\right).$$

Induction Step. Using Equation (4.25), we can write

$$\begin{aligned}R_l(n,i) &= \binom{n}{i}+(-1)i\binom{n}{i+1}y+(-1)^2\frac{(i+1)i}{2!}\binom{n}{i+2}y^2+(-1)^3(i+2) \\ &\quad \frac{(i+1)i}{3!}\binom{n}{i+3}y^3+\cdots+(-1)^{n-i}\frac{(n-1)\cdots(i+1)i}{(n-i)!}\binom{n}{n}y^{n-i},\end{aligned}$$

and

$$\begin{aligned}R_l(n,i+1) &= \binom{n}{i+1}+(-1)(i+1)\binom{n}{i+2}y+(-1)^2\frac{(i+2)(i+1)}{2!}\binom{n}{i+3} \\ &\quad y^2+(-1)^3\frac{(i+3)(i+2)(i+1)}{3!}\binom{n}{i+4}y^3+\cdots+(-1)^{n-i-1}(n-1) \\ &\quad \frac{(n-2)\cdots(i+1)}{(n-i-1)!}\binom{n}{n}y^{n-i-1}.\end{aligned}$$

Realizing from these two equations,

$$R_l(n,i+1) = \frac{(-1)}{i}\frac{d}{dy}[R_l(n,i)] = \frac{(-1)}{i}\frac{(-1)^i}{(i-1)!}\frac{d}{dy}\left(\frac{d}{dy}^{(i-1)}\left(\frac{F_l(n,1)-1}{y}\right)\right).$$

This proves,

$$R_l(n,i+1) = \frac{(i-1)^{i+1}}{i!}\frac{d}{dy}^{(i)}\left(\frac{F_l(n,1)-1}{y}\right). \square$$

Using Lemma 4, we now can prove Lemma 3.

Lemma 3 is proved in two parts. The first part proves the lemma for $m = 1$, while the second part proves it for $2 \leq m \leq n$.
(1) For $m = 1$: Using Equation 4.25,

$$R_l(n,1) = \binom{n}{1} + (-1)\binom{n}{2}y + (-1)^2\binom{n}{3}y^2 + \cdots + (-1)^{n-1}\binom{n}{n}y^{n-1}.$$

From Equation (4.13), we know that

$$\begin{aligned} F_l(n,1) &= 1 - yR_l(n,1), \\ &= 1 + (-1)\binom{n}{1}y + (-1)^2\binom{n}{2}y^2 + \cdots + (-1)^n\binom{n}{n}y^n. \end{aligned}$$

Using Equation 4.27, we can finally write

$$F_l(n,1) = (1-y)^n \ . \ \square$$

(2) For $2 \leq m \leq n$: By induction on m.
Basic Step. Let $m = 2$. Using Lemmas 4 and 5,

$$R_l(n,2) = \frac{1}{1!}\frac{d}{dy}\left[\frac{(1-y)^n - 1}{y}\right] = \frac{1 - ny(1-y)^{n-1} - (1-y)^n}{y^2}.$$

Using Equation (4.13),

$$F_l(n,2) = 1 - y^2R_l(n,2),$$

which proves

$$F_l(n,2) = ny(1-y)^{n-1} + F_l(n,1).$$

Induction Hypothesis. Assume that for $3 \leq i < n$,

$$F_l(n,i) = \binom{n}{i-1}y^{i-1}(1-y)^{n-(i-1)} + F_l(n,i-1).$$

Induction Step. Using Lemma 5,

$$R_l(n,i+1) = \frac{(-1)}{i}\frac{d}{dy}[R_l(n,i)].$$

Also, it is clear from Lemma 5 that $R_l(n,i)$ has a negative sign for odd values of i and a positive sign for even values of i. Keeping this fact in mind and using Equation (4.13), $R_l(n,i)$ can be expressed as

$$R_l(n,i) = (-1)^i\left(\frac{F_l(n,i) - 1}{y^i}\right).$$

Suppose i is odd, then

$$R_l(n,i+1) = \frac{(-1)}{i}\frac{d}{dy}\left[(-1)\frac{F_l(n,i)-1}{y^i}\right] = \frac{1}{i}\frac{d}{dy}\left[\frac{F_l(n,i)-1}{y^i}\right].$$

From induction hypothesis,

$$F_l = \sum_{k=0}^{i-1}\binom{n}{k} y^k(1-y)^{n-k}.$$

Accordingly,

$$\begin{aligned} R_l(n,i+1) &= \frac{1}{i}\frac{d}{dy}\left[\frac{\sum_{k=0}^{i-1}\binom{n}{k} y^k(1-y)^{n-k} - 1}{y^i}\right] \\ &= \frac{-\binom{n}{i} y^i(1-y)^{n-i} + 1 - \sum_{k=0}^{i-1}\binom{n}{k} y^k(1-y)^{n-k}}{y^{i+1}}. \end{aligned}$$

$$F_l(n,i+1) = 1 - y^{i+1}R_l(n,i+1) = \binom{n}{i} y^i(1-y)^{n-i} + \sum_{k=0}^{i-1}\binom{n}{k} y^k(1-y)^{n-k},$$

which gives,

$$F_l(n,i+1) = \binom{n}{i} y^i(1-y)^{n-i} + F_l(n,i).$$

Similarly, when i is even, then

$$\begin{aligned} R_l(n,i+1) &= \frac{(-1)}{i}\frac{d}{dy}\left[\frac{1 - \sum_{k=0}^{i-1}\binom{n}{k} y^k(1-y)^{n-k}}{y^i}\right] \\ &= (-1)\left[\frac{\binom{n}{i} y^i(1-y)^{n-i} - 1 + \sum_{k=0}^{i-1}\binom{n}{k} y^k(1-y)^{n-k}}{y^{i+1}}\right]. \end{aligned}$$

This gives

$$F_l(n,i+1) = \cdot\binom{n}{i} y^i(1-y)^{n-i} + F_n(n,i). \ \square$$