Proceedings of the 1997 Winter Simulation Conference
ed. S. Andradóttir, K. J. Healy, D. H. Withers, and B. L. Nelson

A HYBRID TOOL FOR THE PERFORMANCE EVALUATION OF NUMA ARCHITECTURES

James Westall
Robert Geist

Department of Computer Science
Clemson University
Clemson, SC 19634–1906, U.S.A.

ABSTRACT

We present a system for describing and solving closed queuing network models of the memory access performance of NUMA architectures. The system consists of a model description language, solver engines based upon both discrete event simulation and derivatives of the Mean Value Analysis (MVA) algorithm, and a model manager used to translate model descriptions to the forms required by the solvers.

A single model description file is used to describe the essential elements that characterize the NUMA system and its workload. During a single simulation or MVA modeling run it is easy to dynamically vary elements of the system model, such as mean device service times, elements of the workload model such as cache miss rates, or both. Use of the extremely fast, but approximating, MVA solvers to interpolate between design points computed by the slower simulator allows the analyst to obtain detailed and accurate results in minimal time.

Keywords: modeling methodology; discrete event simulation; mean value analysis; NUMA architectures.

1 INTRODUCTION

While small-scale shared memory multiprocessors, consisting of tens of processors attached to a single shared bus, have been available for many years, large-scale systems with shared memory and scalable interconnection structures are relatively new (Lenowski and Weber, 1995). Currently available systems of this type include the Cray T3D, HP-Convex Exemplar SPP 1200, and Encore GigaMax. In these large-scale systems, memories are logically shared but physically distributed. Thus, the traditional assumption of constant access time to main memory is no longer valid, and so performance prediction tools must be adjusted to account for non-uniform memory access (NUMA) times.

Because of hardware imposed requirements for simultaneous possession of multiple resources, queueing network models have not been well suited for the performance evaluation of traditional shared memory multiprocessors. The use of split transaction bus architecture with queueing in the interconnection network makes NUMA systems somewhat more amenable to the use of queueing network based performance tools. However, some significant obstacles to their use must be overcome.

While queuing network models (Lazowska, Zahorjan, Graham, and Sevcik, 1984; Trivedi, 1983) can provide extremely fast performance evaluation, analytic solutions of such models are provided under an assumption that network devices are stochastically equivalent to first- come, first-served (FCFS) servers with exponentially distributed service times. *Mean-value analysis* (MVA) (Lazowska, Zahorjan, Graham, and Sevcik, 1984) is an easy to implement and widely used technique for solving closed queuing networks that satisfy these assumptions.

The MVA algorithm works as follows. For a closed network of M servers, let $q_{i,j}$ denote the probability that a request leaving server i next visits server j. If $Q = (q_{i,j})$ is the $M \times M$ matrix of such probabilities, then any vector solution, λ, to

$$\lambda = \lambda Q \tag{1}$$

contains the relative number of visits to each node in the steady-state. It is only relative because (1) contains 1 degree of freedom. The mean network performance measures are then completely determined by the vector λ and the mean service times at the servers.

It turns out that expected response time at server i with n customers in the network, $E[R_i(n)]$, is related to expected customer population at that node, $E[N_i(n)]$, and expected service time at the node, $E[S_i]$, by a simple formula:

$$E[R_i(n)] = E[S_i](1 + E[N_i(n-1)]) \tag{2}$$

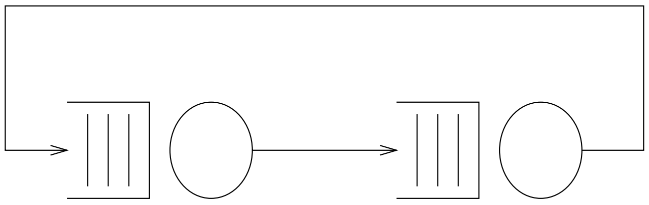

Figure 1: A two-server network.

This is the key observation of the MVA technique. Two applications of Little's theorem relate expected node throughput, $E[T_i(n)]$, to each of these measures:

$$E[T_i(n)] = \frac{n}{\sum_{j=1}^{M} \frac{\lambda_j}{\lambda_i} E[R_j(n)]} \tag{3}$$

$$E[N_i(n)] = E[T_i(n)] \times E[R_i(n)] \tag{4}$$

The MVA technique starts with $E[N_i(0)] = 0$, and then repeatedly applies equations (2) - (4) until network customer population, n, reaches the desired level.

However, most devices in the NUMA architecture (memories, buses) have deterministic (constant) service times rather than exponential, and thus violate the assumptions implicit in MVA. MVA solutions of queuing networks having servers with deterministic service times overestimate device contention and so underestimate architecture performance.

As an example, consider the 2-server cycle shown in figure 1. Each server has a mean delay of $E[S]$. Here $\lambda = (1, 1)$, and so we obtain, for each node,

$$E[R_i(1)] = E[S]$$

$$E[T_i(1)] = 1/(2E[S])$$

$$E[N_i(1)] = 1/2$$

and so

$$E[R_i(2)] = (3/2)E[S]$$

Note, however, that if a 2-customer, 2-server cycle had constant service times with value $E[S]$ at each server, the customers would ultimately move in tandem between the servers, each experiencing no queuing delay at all, and so we would have

$$E[R_i(2)] = E[S],$$

that is, 2/3 of the MVA value given above. This is the essence of the inaccuracy in using standard analytic solutions of closed queuing network models when representing networks with deterministic servers.

Discrete event simulation can, of course, provide extremely accurate estimates. However, it is usually too slow to allow a system designer to explore the relative merits of a vast range of potential hardware and workload configurations.

An analytic approximation technique that we term "mean-based iterative estimation" has been used by several authors. A general model of multiprocessor systems with multiple memory modules was built by Vernon and Holliday (1986) using a timed Petri net, where the transition holding times were deterministic or geometric. A more recent variation on mean-based iterative estimation, due to Sevcik and Zhou (1993) appears to be extremely accurate in estimating mean network cycle time for a wide range of potential NUMA architectures.

In (Geist, Smotherman, and Westall, 1996) and (Geist and Westall, 1996) we describe an alternative analytic technique for estimating performance indices of NUMA architectures. Our technique is based on modifications to the Mean Value Analysis (MVA) technique for solving a closed, multi-chain queuing network model. It employs an iteration to convergence to obtain node utilizations and thus draws from both the Sevcik-Zhou approach and the theory of queuing networks.

Our technique, called Deterministic Service Approximation (DSA) has been shown to provide very accurate results over a range of problems, but we have not been able to establish bounds on its error. Thus, the technique is particularly appropriate for inclusion in a hybrid solution system that employs simulation to establish foundation data points and the fast analytic solver to interpolate between them.

A considerable amount of data is required to define a model of a NUMA system and the characteristics of its workload. When a hybrid solution system is employed it is particularly desirable that:

- the model description be completely de-coupled from the model solution
- both the simulation and the analytic solvers be driven by the identical model description.

In the remainder of the paper we describe an approach to satisfying those objectives. In section 2 we review closed queuing networks and describe how transaction routing is specified in multi-chain models having intra-chain routing requirements that are dependent on the present state of a transaction. We describe the structure of our model description language in section 3 and show how it is used by the model manager in section 4. In section 5 we provide an example application to a hypothetical 8-processor architecture. Conclusions follow in section 6.

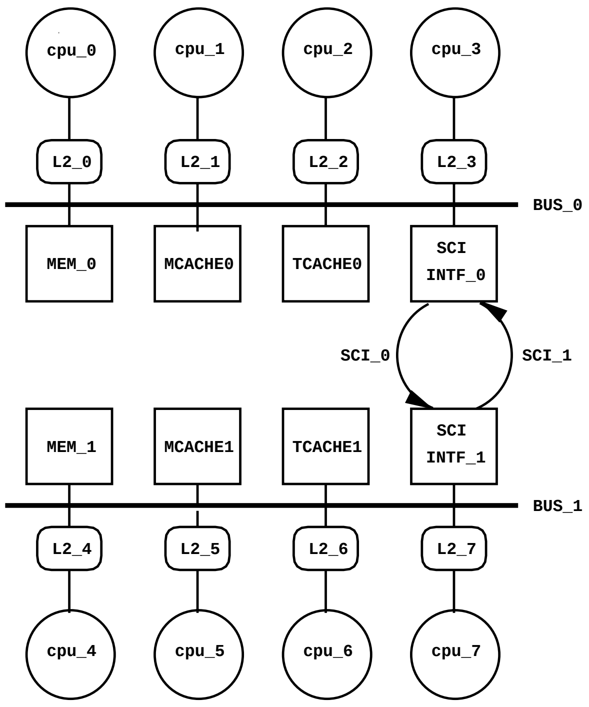

Figure 2: A hypothetical NUMA system

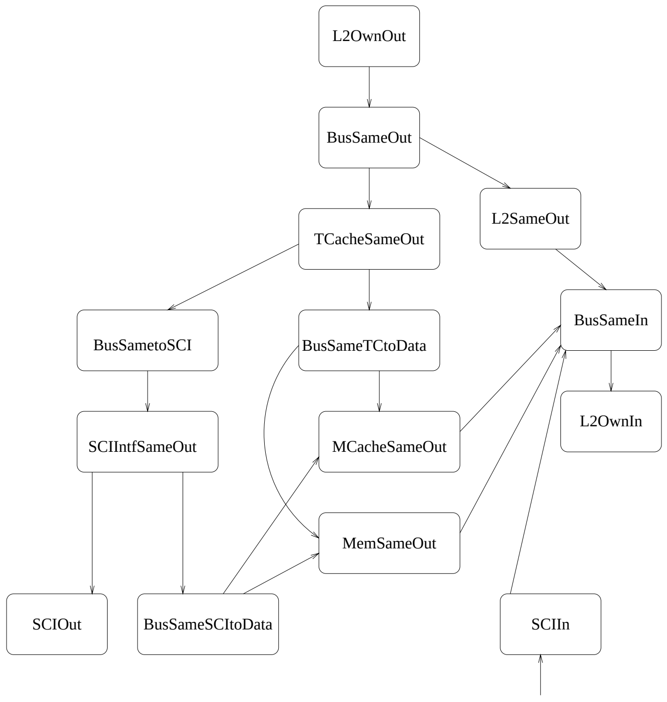

Figure 3: Routing classes for chain 0

2 QUEUING NETWORK MODELS

A closed queuing network consists of a collection of servers along with a fixed number of transactions that flow among the servers. An example of such a model is shown in figure 2. We use the term system model to refer to the part of a model specification that defines the servers and their associated service characteristics. Service characteristics include queuing discipline, server capacity, and service time distribution. We assume FCFS servers with capacity one transaction. We support only exponential and deterministic service times.

The term workload model refers to the part of a specification that defines the number of transactions and their routing as the flow from server to server. In simple CQN models, the routing of transactions flowing among the servers is defined by a single matrix Q of branching probabilities. The value of $Q(i,j)$ specifies the probability that a transaction completing service at server i will proceed to server j. Such queueing networks are referred to as *single chain* networks because all transactions are routed in a stochastically identical way.

In contrast, transactions (memory references) issued by one CPU in a NUMA system migrate through the memory subsystem (tag and data caches, busses, and memories) but always return to the CPU that issued the transaction. Thus, the routing behavior of the system cannot be specified by a single matrix of branching probabilities. In the standard terminology of CQN's, a unique routing chain must be associated with each processor. In multi-chain queueing network a unique matrix of branching probabilities, $Q_c(i,j)$, must be specified for each independently routed chain c. Thus, in a NUMA model, one would expect to have to specify a routing matrix for each processor.

Further complicating the issue is the fact that the routing behavior of a transaction within a given routing chain of a NUMA model cannot be specified by a single matrix that specifies the branching probabilities among devices. Each memory access transaction proceeds outward to the storage element that contains the target data item, but then returns directly to the CPU that issued it. Therefore the outbound path of the transaction is stochastic, but the return path is deterministic. In the example system of figure 2, when an outbound transaction associated with *CPU_0* completes service at *Bus_0*, it might proceed to *Mem_0*, *MCache0*, *TCache0*, *SCIIntf_0*, or another L2 cache. However, an inbound transaction associated with *CPU_0* must necessarily proceed to *L2_0* after receiving service at *Bus_0*.

If each physical server is viewed as a collection of logical service classes, it is possible to specify the routing behavior of a given chain c with a single matrix Q_c. The service classes used in routing chain 0 transactions in our example system are shown in figure 3. *Bus_0* server classes include logical service classes *BusSame_In* and *BusSame_Out*. A chain 0 transaction leaving *BusSame_In* proceeds to server class *L2Own_In* with probability 1.0, but a chain 0 transaction leaving *BusSame_Out* proceeds stochastically to a class associated with one of the devices *Mem_0*, *MCache0*, *TCache0*, or *SCIIntf_0*. When this approach is used, the value of $Q_c(i, j)$ specifies the probability that a transaction completing service at logical service class i will proceed to logical service class j.

For the eight processor system shown in figure 2, we use 27 different logical service classes. A subset of these is shown in figure 3. Since an inbound transaction associated with chain 1 proceeds to server *L2_1* instead of *L2_0* it would appear that each of the eight routing chains requires a unique 27 by 27 matrix of branching probabilities. However, only one such matrix is required if the workload possesses a sufficient degree of symmetry. In this case a single set of service classes can be used to characterize the routing of all chains and the Q matrix used for chain 0 defines the branching probabilities of the remaining chains. The requirement that, at the physical server level, routing be chain dependent is met through the use of a single mapping function that binds logical service class to physical server in a chain dependent way.

3 THE MODEL DESCRIPTION LANGUAGE

In this section we discuss the model description language in the context of the example system of figures 2 and 3. The system consists of two four-processor boards that are connected by an SCI (scalable coherent interconnect) type bus. Each processor has a private L2 cache. A local bus connects the L2 caches, the local memory, a level 3 memory cache, a tag cache, and a combined directory and SCI interconnect controller. The interconnect between the two boards is modeled as a slotted ring consisting of two unidirectional segments capable of overlapped operation.

3.1 THE SYSTEM MODEL

The first section of the model description file defines the physical device classes. Each physical device class has four parameters: the name of the device class; the number of instances; the service discipline (only E (exponential) and D (deterministic) are supported); and the mean service time in processor cycles.

```
Device Classes
{
   CPU␣       8␣ E␣  6.66667
   L2␣        8␣ D␣  6.00
   Mem␣       2␣ D␣ 20.00
   MCache␣    2␣ D␣ 10.00
   TCache␣    2␣ D␣  1.00
   Bus␣       2␣ D␣  4.00
   SCIIntf␣   2␣ D␣  4.00
   SCI␣       2␣ D␣  6.00
}
```

The CPU service time of 6 2/3 implies that there is an average of 6 2/3 cycles between level one cache misses. Service time values used here do not reflect the characteristics of any real system.

3.2 THE WORKLOAD MODEL

The first section of the workload model defines both the logical service classes used in routing and the function used to bind service classes to physical devices in a chain dependent way. Each physical device class must have at least one logical service class. Each service class has three parameters: the service class name (which by convention begins with the device class name); the associated device class name; and an expression that computes the relative device number within the physical device class as a function of the chain identifier. In the example shown below the class *BusSame_Out* maps to *Bus_0* for chains 0, 1, 2, and 3 but maps to *Bus_1* for chains 4, 5, 6, 7.

```
Service Classes
{
   CPU␣                  CPU␣   chainid
   L2Own_Out␣            L2␣    chainid
   L2Own_In␣             L2␣    chainid
   BusSame_Out␣          Bus␣   chainid / 4
   BusSame_In␣           Bus␣   chainid / 4
   BusSame_ToSCI␣        Bus␣   chainid / 4
   BusSame_ToData␣       Bus␣   chainid / 4
   MemSame_Out␣          Mem␣   chainid / 4
   MemOther_Out␣         Mem␣   1 - chainid / 4
   MCacheSame_Out␣       MCache chainid / 4
       :
       :
}
```

The remainder of the workload model definition is used to define the Q matrix for chain 0. It is here that “end user” measures such as miss rates must be mapped to service class branching probabilities. The mapping must be manually performed in C language statements as shown below. This is clearly the most tedious and error prone step in the model definition process, but we know of no general way to automate it. (It is possible to provide post- definition tests for inconsistent and incomplete specification, and we do so.)

The variables and parameters sections are used to construct expressions that define service class branching probabilities in the routing section, which follows. Variables are initialized once by the model manager, exported to the application, and are not touched by the model manager thereafter. They may be varied by the application during a modeling run. For example, in a given study *L2MissRate* might be varied from 0.02 to 0.20 in steps of 0.02.

```
Variables
{
   L2MissRate␣        0.50

   LocRef␣            0.50
      L2Local␣        0.10
      MCacheLoc␣      0.20
      MemLocal␣       0.70
   :
   :
}
```

The values specified as parameters are recomputed by the model manager each time model initialization is performed. In the hypothetical study proposed above, as the application varied *L2MissRate*, the remainder of the values in the variables section would remain fixed, but the values declared below would be recomputed by the model manager at each initialization call.

```
Parameters
{
   RmtRef␣     1 - LocRef
   TCacheMissRate  0.50␣ /* Raw tag cache misses */

   L2OwnHit␣   1 - L2MissRate
   L2OwnMiss␣  L2MissRate

   L2SameHit␣  LocRef * L2Local
   L2SameMiss  1 - L2SameHit

   PctTCacheLoc LocRef*(MCacheLoc+MemLocal)/
                (RmtRef+LocRef*(MCacheLoc+MemLocal))
   TCacheHit␣   (1-TCacheMissRate)*PctTCacheLoc
   TCacheMiss  1 - TCacheHit
    :
    :
}
```

These parameter definitions are necessarily based on request routing protocols for the architecture under consideration. For the example architecture of figure 2, an L2 miss goes over the local bus either to the tag cache or to one of the other three local L2 caches. From the tag cache it may proceed either to the local memory, the level 3 memory cache, or the SCI controller which contains the complete tag directory. Thus an L1 miss can be satisfied locally in four ways: an L2 hit (*L2OwnHit*), by another L2 on the same bus (*L2SameHit*), a local memory hit (*MemLocal*), or a memory cache hit (*MCacheLoc*).

The service class branching probabilities for routing chain 0, ($Q_0(i,j)$), are defined in terms of the variables and parameters in the routing section. Each service class must have a line specifying the service class name and the number of associated *non- zero* branching probabilities. Following this line each service class fed by this service class must be defined by a line that contains the target service class and the branching probability.

```
Routing
{
   CPU␣          1
      L2Own_Out␣          1.0

   L2Own_Out␣    2
      BusSame_Out␣        L2OwnMiss
      CPU␣                L2OwnHit

   BusSame_Out␣  4
      L2Same0_Out␣        L2SameHit / 3
      L2Same1_Out␣        L2SameHit / 3
      L2Same2_Out␣        L2SameHit / 3
      TCacheSame_Out␣     L2SameMiss

   L2Own_In␣     1
      CPU␣                1.0

   BusSame_In␣   1
      L2Own_In␣           1.0

   BusSame_ToSCI 1
      SCIIntfSame_Out␣    1.0

}
```

The variable definitions and statements included in the variables, parameters, and routing section are used to construct a C-language function that is dynamically linked by the model manager and is used to construct the Q matrix at run time. This approach makes it easy to perform studies that require altering elements of the workload model without having to precompute, store, and reload a large collection of Q matrices.

4 THE MODEL MANAGER AND NETWORK SOLVERS

The model manager is responsible for converting a model description such as the one shown above into the input data required by the Mean Value Analysis

(MVA) and simulation solvers. The four MVA based solvers are discussed first.

Solving a closed queuing network refers to the process of determining the mean utilization, throughput, population, and response at each server. Input data required for a multi-chain MVA solution includes the mean service time for each server, the population of each chain, and for each chain and server the relative number of visits made by that chain to that server. Our chain populations are one and the mean service times are specified directly in the system model description.

The per chain device visit ratios may be computed as follows. If Q is the class branching probability matrix for chain 0, then the class visit ratios V for chain 0 satisfy the equation $V = VQ$. The system as specified possesses 1 degree of freedom and cannot be directly solved for V. However, if $V(0)$ is arbitrarily assigned the value 1.0, the system can be reduced in rank and solved for the remaining visit ratios. The model manager performs this step and assigns the class visit ratios to the array *chain0vr*. The device mapping function is then used to compute the device visit ratios for all chains in the following way (class 0 is assumed to be the CPU class):

```
for (ch = 0; ch < n_chains; ch++)
{
   dev = DevMap(ch, 0);
   d_vratio[ch][dev] = 1.0;
   for (cl = 1; cl < n_classes; cl++)
   {
       dev = DevMap(ch, cl);
       d_vratio[ch][dev] += chain0vr[cl];
   }
}
```

The MVA algorithm provides exact solutions for closed queuing networks of the type known as product-form. Product-form networks include those whose servers use a FCFS queuing discipline and exponentially distributed service times, but do not include networks having any FCFS servers with deterministic service times. If the service time distribution is assumed exponential when it is actually deterministic (as in most NUMA system components), throughputs are underestimated and queuing delays are overestimated. These observations motivated the development of our MVA-DSA (deterministic service-time approximation) algorithm. In MVA-DSA the service time used in the MVA computation is dynamically reduced as a function of server load. The MVA-DSA approach is shown to match the results of simulation much more closely than does true MVA in (Geist, Smotherman, and Westall, 1996) and (Geist and Westall, 1996).

The use of the multi-chain version of the exact MVA algorithm is restricted to models of small NUMA systems because both the space and time complexity of the algorithm grow exponentially with the number of chains (processors). We have found the practical limit for the use of the exact MVA algorithm (with or without the DSA approximation) to be 8 processors. For larger models or for very fast performance on small models the approximate MVA algorithm is a better choice.

Approximate MVA is an iterative algorithm, and incorporating DSA is straightforward. After each of the normal approximate MVA solution steps, the service times of deterministic servers are adjusted using the normal DSA approximation. In all cases in which we have applied the algorithm we obtain convergence within 20 iterations.

The simulation solver is a typical discrete event simulation in which events represent arrivals and service completions at server nodes. Server parameters include service distribution (deterministic or exponential) and mean service times. For efficiency in routing, the workload manager converts the Q matrix for chain 0 into a compact representation which contains for each source service class only the destination service classes for which the branching probability is non-zero.

A transaction record is associated with each routing chain and carries the chain identity in element *xachain* and current service class of the transaction in element *xaclass*. When a transaction completes service, the following algorithm computes the identity of the next server to be visited:

```
rnum = random();
sre = &simroute[xact->xaclass];
for (i = 0; i < sre->count; i++)
{
    if (rnum <= sre->prob[i])
    {
       nextclass = sre->target[i];
       break;
    }
}
xact->xaclass = nextclass;
nextserver = DevMap(xact->xachain,
                    xact->xaclass);
```

The pointer *sre* points to the compact representation of the distribution function for the branching probabilities associated with the current service class. The *DevMap* function performs its usual function of mapping a (chain, class) pair to the associated real device.

The use of service class based routing within the simulation is the key to the simplicity of the routing and the decoupling of the model description from the simulation solver. If device based routing were used, it would be necessary to carry explicitly in the transaction record the state information (e.g., outgoing or incoming transaction) that is implicitly carried in the service class representation. It would then also be necessary to imbed in the simulation solver the logic required to act upon this explicit state information.

5 EXAMPLE

The system described in the previous sections was used to drive both simulation and MVA solvers. The value of $L2MissRate$ was varied in steps of 0.05 from 0.05 to 0.65. The memory delay experienced by a transaction is the end-to-end time that elapses between the time a memory transaction leaves its CPU and the time it returns. The average memory delay reported by the exact MVA, approximate MVA with DSA, and the simulation solvers is shown in figure 4. Values reported by exact MVA with DSA are virtually identical to those of approximate MVA with DSA.

Simulation results are the average of 24 runs of 1,000,000 simulated processor cycles for each of the 13 data points (L2 miss rates) computed. Estimated confidence intervals range in size from 0.08 to 0.17 at a 90 % confidence level. The lack of smoothness in the simulation curve is a characteristic of deterministic service times and disappears, as one would expect, when exponential service times are used.

Elapsed time for the simulation was slightly more than 8 hours on a 486-66 PC. Elapsed time for computing the 13 data points with exact MVA was less than 10 seconds. Elapsed time for approximate MVA with DSA was less than 1 second.

6 CONCLUSION

The model description language provides a mechanism for describing a NUMA system and its workload in a way that is succinct yet quite readable. By using the model manager, the system designer can work from a single model description and conviently investigate the effects of varying both system and workload parameters. The extremely fast, but approximating, MVA-DSA solver can be used to interpolate between design points computed by simulation to provide detailed and accurate results in minimal time.

There are clearly a number of ways in which the existing tool might be improved. Included among them are the following: addition of a graphical front end for creating model descriptions; development of automated ways to derive branching probabilities from high-level workload characteristics; and, last but not least, verification of predicted results on a real NUMA system.

REFERENCES

R. Geist, M. Smotherman, and J. Westall. 1996. Performance evaluation of numa architectures. In *Proc.* 34^{th} *Annual ACM Southeast Conf.*, 78–85, Tuskegee, AL.

R. Geist and J. Westall. 1996. Performance and availability evaluation of numa architectures. In *Proc.* 2^{nd} *Annual IEEE Intl. Computer Performance and Dependability Symposium*, 271–280, Urbana, Il.

E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik. 1984. *Quantitative System Performance: Computer System Analysis using Queueing Network Models.* New York: Prentice-Hall.

D.E. Lenoski and W.D. Weber.- 1995.- *Scalable Shared-Memory Multiprocessing.*- San Francisco: Morgan Kaufmann..

K. Sevcik and S. Zhou. 1993. Performance benefits and limitations of large numa multiprocessors. In *Proc. 16th IFIP Int. Symp. on Computer Performance Modeling, Measurement, and Evaluation (PERFORMANCE '93)*, 183–204, Rome, Italy.

K.S. Trivedi. 1983 *Probability and Statistics with Reliability, Queueing, and Computer Science Applications.* Prentice-Hall, Englewood Cliffs, NJ.

M.K. Vernon and M.A. Holliday. 1986. Performance analysis of multiprocessor cache consistency protocols using generalized timed petri nets. In *Proc. Performance '86 and ACM SIGMETRICS 1986*, 9–17, Raleigh, NC.

AUTHOR BIOGRAPHIES

ROBERT GEIST is a Professor in the Department of Computer Science at Clemson University. He received a B.A. degree in mathematics and an M.A. degree in computer science from Duke University in 1970 and 1980 respectively, and he received M.S. and Ph.D. degrees in mathematics from the University of Notre Dame in 1973 and 1974 respectively. His research interests are in performance and reliability modeling of computer and communication systems and stochastic modeling in computer graphics.

JAMES WESTALL is a Professor in the Department of Computer Science at Clemson University. He received a B.S. degree in mathematics from Davidson College in 1968, a Ph.D in mathematics in 1973 and

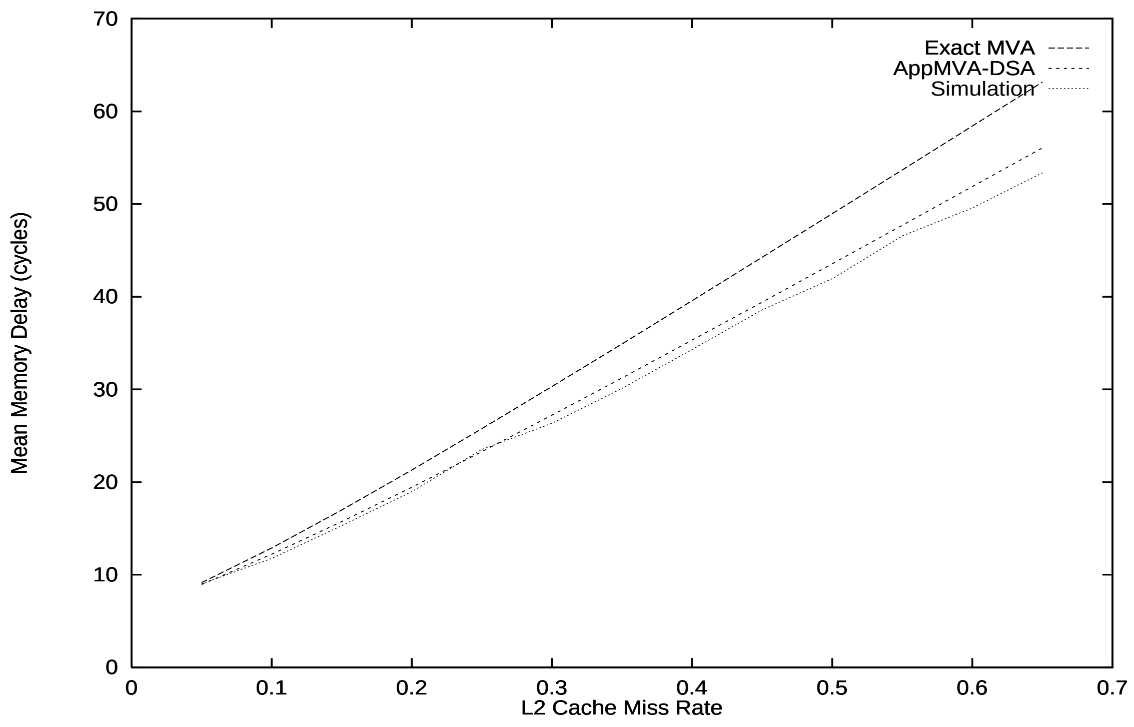

Figure 4: Comparison of analytic and simulation results

an M.S. in computer science in 1978 from the University of North Carolina at Chapel Hill. His present research interests include performance measurement and modelling of computer systems and networks and segmentation and recognition of of handwritten characters.