PERFORMANCE SIMULATION STUDY
OF AN APPLICATIVE MULTIPROCESSOR

by

M. K. Vernon

Computer Sciences Technical Report #536

February 1984

Performance Simulation Study of an Applicative Multiprocessor

M. K. Vernon

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

Abstract

As new complex distributed architectures, are proposed, there is an increasing need for performance estimation *prior* to system, or prototype, implementation. This paper describes the use of a Graph Model of Behavior to carry out a queueing simulation study of a proposed tightly-coupled local area ring network. The network design includes special-purpose hardware and software to execute applicative programs. The performance study was carried out prior to detailed design or prototyping of specialized VLSI hardware, in order to evaluate certain design decisions, and to gain confidence that system implementation would be worthwhile. In this case, performance bottlenecks were discovered which indicate design modifications are needed prior to prototype implementation. An understanding of the reasons for the performance problems, gained during highly interactive simulation experiments, points to directions for design improvements.

1. Introduction

Performance modeling and analysis tools are generally employed both during system implementation and after a system is built, in order to better understand observable behavior and to discover system modifications which will increase system performance. Most frequently, models are constructed after system implementation because at this point a model can be calibrated with measurement data, and then used to evaluate performance tuning options. This approach can be successful if the basic system structure remains fixed, and performance tuning is accomplished by minor changes in system architecture and/or by increasing the speed or capacity or various system resources.

For complex distributed system designs, significant effort can be involved in prototype implementation, combined with the substantial risk that acceptable performance will only be achieved by major restructuring of the system architecture. As new

complex distributed architectures are proposed, there is an increasing need for performance estimation *prior* to system, or prototype, implementation. It is our conjecture that performance tools which depend on statistical characterization of workload, and may only yield rough, "ballpark"-type performance estimates, can be useful during the early stages of system design. However it is important to have evidence with indicates that relative estimates for two or more alternative architectures correspond directly to what would be observed from relative measurements on the implemented systems.

This paper describes a queueing simulation study of a proposed tightly-coupled distributed system architecture. The architecture is a local area ring network designed with special-purpose hardware and software to execute applicative programs [BacJ78], through concurrent expression evaluation. There exists the potential for parallelism in the execution of applicative programs because the language encourages use of parallel constructs and because operations in the program can be applied to large arrays of data. Any operation can be executed as soon as its arguments are available, since no side-effects are allowed. In order for the potential for parallelism to be realized, a suitable multiple-processor architecture is needed. The ring architecture examined in this performance study has the main advantages of modularity, extensibility, and simplicity of the distributed control structure.

The performance study was carried out prior to detailed design or prototyping of specialized VLSI hardware, in order to evaluate certain design decisions, and to gain confidence that system implementation would be worthwhile. Furthermore, we relied on queueing models which characterize behavior at a more abstract level than generally used in the initial stages of system design and development [GriD80, GurJ83]. Characterizing system workload was a significant part of the performance evaluation effort. In the study, performance bottlenecks were discovered which indicate design modifications are needed prior to prototype implementation. An understanding of the reasons for the performance problems, gained during the simulation experiments,

points to directions for design improvements.

The tools used in the simulation study have two characteristics which are important to the credibility of the simulation results. The Graph Model of Behavior (GMB) used to represent the local ring network design, supports specification of queueing behavior, and the associated probabilistic view of system behavior, at the same time it supports specification of functional behavior in an arbitrary amount of detail [VerM83a]. Thus, we were able to add a minimum amount of detailed functional behavior to the performance model, which increased our confidence that significant data-dependencies in the proposed system were adequately represented. Furthermore, the GMB Simulator which was used to gather performance statistics, can be used interactively to observe modeled behavior [VerM83c]. Interactive observation of system behavior played a key role in debugging the model, determining performance improvements for the proposed architecture, and in discovering characteristics which would be important in alternative architectures.

The following sections describe the ring architecture (Section 2), workload characterization for the model (Section 3), and the model structure (Section 4). In Section 5 we describe our simulation results. Conclusions of this work are discussed in Section 6.

2. Proposed Ring Architecture: Overview

The ring architecture we are interested in evaluating is illustrated in Figure 2.1. The system consists of a host computer, and several hundred Processing Elements (PEs) arranged in a ring structure, interconnected by FIFO queues. An applicative program is represented in the system as a string of characters which are shifted around the ring. More than one program may be on the ring at any given time, each delimited by an "end-of-string" (EOS) character. Applicative programs and their input data are downloaded from the host computer to any one of the PEs via the shared bus. The program is then executed by the ring of PEs as its textual representation is shifted around the ring.

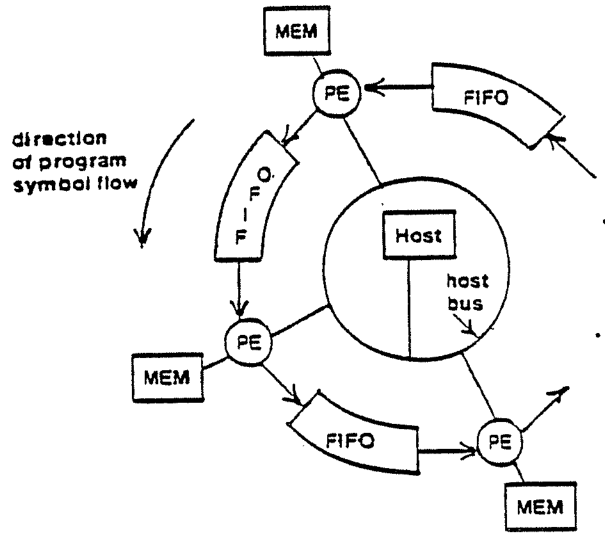

Figure 2.1: Proposed Architecture for Functional Programming

Programs are executed using a *reduction* strategy [TreP82]. A *reducible expression* (RE) in the program is an expression for which all operands are available. Using the reduction strategy, a reducible expression is evaluated by a PE on the ring, and is textually replaced by its results. In some cases, the replacement string will contain multiple new REs to be evaluated. Eventually, due to function composition, RE results become available as operands in other expressions, until the final result for the program is calculated.

As illustrated in Figure 2.2, each PE consists of two processors: a specialized i/o processor (ioP), and a more general-purpose reduction processor (RP), plus some local memory. Communication with the host machine via the shared bus is not illustrated in this figure.

The i/o processor performs all i/o with the host, all communication on the ring, and makes reducible expressions available to the attached reduction processor. The reduction processor executes reducible expressions and makes the results available to the i/o processor. Ring communication consists of reading characters from the input

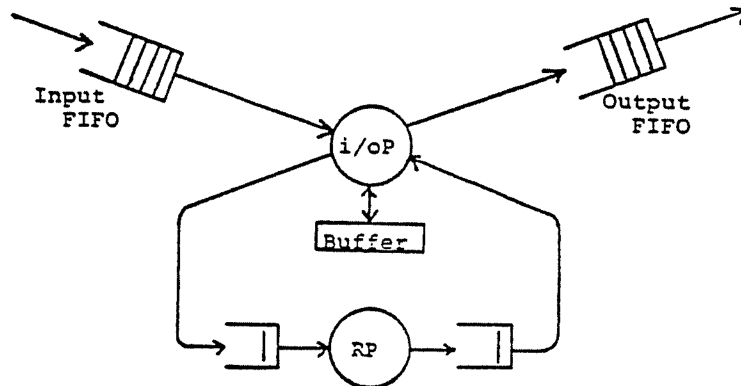

Figure 2.2: Organization of the Processing Elements (PEs)

FIFO, and writing characters to the output FIFO. The FIFO queues are specially designed for high-speed access (5 nanoseconds/character). An ioP buffers small amounts of program text in local memory during the process of recognizing and selecting REs for its attached reduction processor. Due to its important role in communications, the ioP will be implemented in custom VLSI so that it will also be extremely fast.

At any point in time, an applicative program may contain many reducible expressions. Minimum total execution time for the program will be achieved if the REs are evenly distributed among the reduction processors on the ring, and if REs are executed by the ring of PEs in order of decreasing "level number". Expressions can be tagged with a "level number", by the host machine or by the reduction processor, when they are created. The level number remains constant throughout the lifetime of an expression, and indicates the number of nested operators which are dependent on the results of the RE. Optimal scheduling of RE execution, using these guidelines, cannot be determined a priori by the host machine, since the structure of the program changes dynamically during the computation, depending on the input data. (See section 3). Instead, each i/o processor must use locally calculated information to decide which REs to select for reduction by its attached reduction processor, and which to pass on

to downstream PEs on the ring.

The following initial design decisions were made for the ring architecture in order to achieve sub-optimal execution times:

Selection Strategy

(1) The ioP selects a *priority* RE for execution without constraints. Priority is determined by operator type. For the purposes of this study, operations which expand to yield multiple new REs for execution have priority, whereas operations which yield data values do not.

(2) The ioP selects a non-priority RE for execution if its assigned level number is maximum relative to immediate neighboring REs in the program (i.e. if the number of nested expressions waiting for the result is maximum relative to the REs preceding and following it in the program text).

Load Balancing Policy

Only one RE is selected by the ioP for execution, per program visit. The first RE satisfying the above selection strategy rules is routed to the attached reduction processor. All other REs are passed to the downstream PEs, until the program returns from its travel around the ring.

Replacement Policy

After an RE is executed by an RP, its result must be inserted in the program text. The proposed replacement policy, implemented in the ioP and the reduction processor, is as follows:

(1) A priority RE selected by the ioP is executed in the "foreground". It is executed immediately by the reduction processor, and its results are placed in the output FIFO before subsequent program text is read from the input FIFO by the ioP.

(2) Non-priority REs selected by the ioP are executed in FCFS order in the "background". The ioP writes a labeled "token" in the output FIFO, which serves as a place-holder for the result of the RE. The ioP continues processing input text from the input FIFO while the reduction processor executes the non-priority RE. When the token returns to the ioP, it is replaced by the result. A token never traverses the ring more than once.

Summary

We have outlined the system architecture which is the subject of this performance evaluation study. More detail, including use of simple analytic equations to predict

speedup of this architecture relative to sequential machines for certain constructs, can be found in [PatD80]. The goal of the current study is to develop an abstract performance model which can be used to study some initial performance trade-offs relatively quickly.

3. Workload Characterization and Model Parameters

A fundamental decision to be made in constructing a performance model for the proposed architecture is how to characterize the system workload. The decision is guided on the one hand by *the need to estimate job characteristics with confidence, and the need to capture a significant amount of functionality in the PE submodel* (i.e. service and routing based on types of operators and availability of operands.) The decision is guided on the other hand by a desire to characterize program behavior statistically, in order to evaluate the performance over long-term operation of the ring relatively quickly.

In the implemented system, the customers on the ring are the characters which comprise the program text. Since the details of the algorithm to recognize REs in the program text are not important to the initial performance model, a more abstract representation of the customers is needed. We are not aware of any results which characterize functional program (or related dataflow program) behavior for the purposes of defining the statistical workload in a queueing model. Thus, we begin by examining the dynamic structure of applicative programs.

The structure of an applicative program can be represented by a tree. Each node in the tree represents an operator or user-defined function which is bound to operands that will be received from descendent nodes. Leaves in the tree represent reducible expressions. The initial structure of an applicative program is always a linear composition of functions or operators applied to the input data, as illustrated in Figure 3.1a. Execution of an RE may replace the node with a sub-tree of new operations to be performed (Figure 3.1b and c) or with an operand for its parent operator in the tree

(Figure 3.1d). Eventually, operands are available for the root of the tree, and the final result of the program can be evaluated. Nodes in the initial execution tree are generated at the host machine. Expansions and reductions in the tree are performed by the PEs on the ring.

For performance modeling purposes, it is important to capture the dynamic behavior of the program execution tree. Thus, we choose to have customers in the modeled network represent nodes in the dynamically changing execution tree (i.e. REs or operators waiting for operands). Relevant information for a customer includes the type of the operator, its level in the tree, the textual size and data values of available arguments, which operands it is waiting for, if any, and whether is is a place-holding "token". The customers must be generated and characterized in a way that reflects the program behavior outlined above. Accordingly, the ordering of customers generated in the initial program by the host, and the sub-trees generated by the reduction

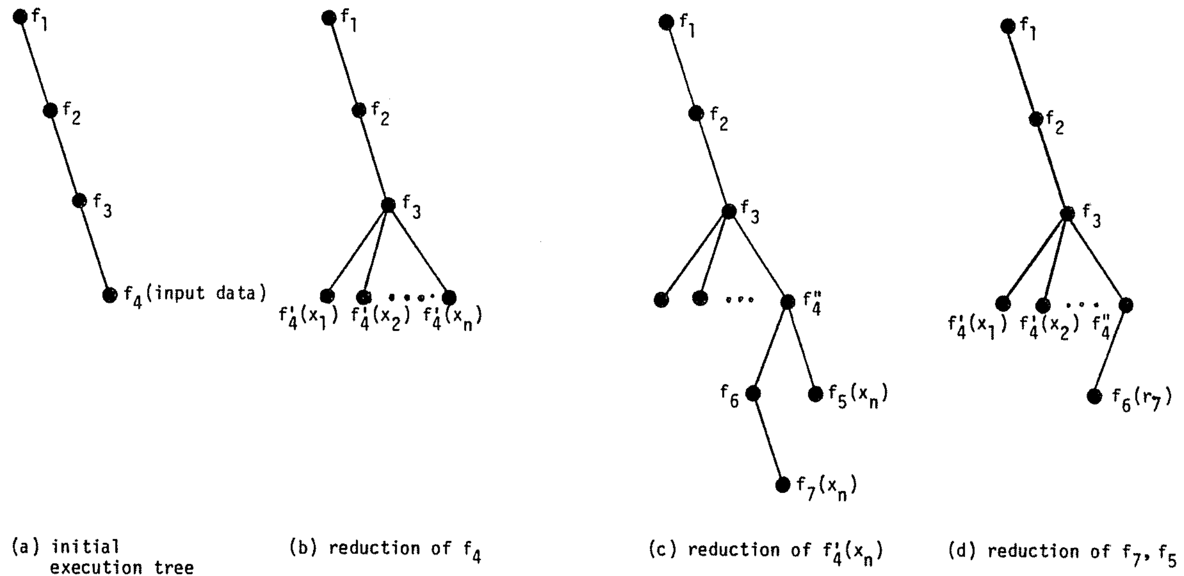

Figure 3.1: The Dynamic Structure of Functional Programs

processors, should correspond to a pre-order traversal of the execution (sub-)trees. Furthermore, we need to decide which information about each customer should be represented explicitly in the form of attributes or data values, and which information can be determined by probability distributions as needed. In the absence of better information, we are interested in defining separate workloads that should yield optimistic, and pessimistic, performance estimates.

In this study, we decided to represent the level number, the number of operands not yet available for execution, and the place-holding token indicator explicitly, since probabilistic estimation of these quantities would not yield a reasonable representation of the structure and behavior of the execution tree. The type of the operator (priority or non-priority) is decided probabilistically when the node is generated. All other quantities are determined by (optimistic or pessimistic) probability distributions when needed. We note that the explicitly represented information is used primarily for customer routing decisions in the network.

The following parameters are associated with each customer in the model in support of the explicit information required:

(1) A pair of indices identifying the program containing the customer, and the node in the program that the customer represents. This pair of indices is used to access data which defines how many operands the customer is waiting for (0 for REs).

(2) The level number of the customer.

(3) The index of the parent node of the customer. This index is used to update the number of operands the parent node is waiting for after the customer is executed by a PE.

(4) A value which can be set to indicate that the customer is a priority RE, an end-of-string (EOS) marker, or a place-holding token.

Probability distributions chosen for the remaining workload characteristics are summarized in Figure 3.2. Parameters of the various distributions, reflecting optimistic or pessimistic workloads for the ring, can be specified at simulation time for the model. The parameters selected for the current study are discussed in section 5.

program arrival rate	exponential(*exprarate*)
initial program size (number of nested expressions)	uniform(*minPsize*, *maxPsize*)
probability RE will be expanded	expandP
RE expansion bounds	uniform(*minX*, *maxX*)

Figure 3.2: Probability Distributions for Ring Network Workload

4. System Configuration Submodels

The ring network model is composed from two submodels: a Host submodel (Section 4.1), and a PE with associated input FIFO submodel (Section 4.2). The configuration models define resource demands and routing of customers in the network, and support performance measurement. Performance measurements of interest include: 1) program response time, 2) utilization of PEs, including relative utilization of the i/o processor and the reduction processor, 3) queue lengths for the FIFO queues on the ring, and 4) queue lengths for processing at the reduction processor.

We use the Graph Model of Behavior (GMB) notation for defining the configuration models [VerM83a]. The Graph Model has primitives for defining control flow, data access paths, data transformations, and timing. In this paper, we show only the control graphs for the two submodels (Figures 4.1 and 4.2). Control graph execution is based on tokens (i.e. small black circles on the control arcs) that activate control nodes (named circles) which may have some associated functionality. When a control node terminates, tokens are distributed on output arcs, which may lead to new control node activations. A "*" operator in the graph indicates a logical "AND" condition on the input or output control arcs. Similarly, the "+" operator in the graph indicates a logical "OR" condition on the input or output control arcs.

For queueing simulation purposes, some of the control nodes represent system resources (i.e. servers), and some of the control arcs represent customer queues.

Other control nodes simply capture relevant system functionality. Control nodes are single-server by default, but can be defined to have infinite capacity instead. We are able to estimate utilization and throughput for control nodes, and queue length and waiting time distributions for control arcs in the model, using automated measurement facilities in the GMB Simulator. The functional specification domain includes capabilities to select values from probability distributions, to define delay for the node, and to perform arbitrarily detailed data transformations. The functionality of control nodes is discussed in the paragraphs below.

4.1. Host Machine Submodel

The model of the host machine, which generates applicative programs and uses the shared bus to communicate with PEs on the ring, is illustrated in Figure 4.1. Applicative programs are generated by PgmGen in the model. GetEntry sends out a broadcast message (token on *notifyPEs*) to initiate transfer of the program to the ring, and one PE responds to the broadcast message by placing a token on reqPgm. *InitExpr* selects the initial program length (according to the probability distribution in Figure 3.2, and *NodeGen* generates the customers in the initial execution tree. Each customer's parent node index is the index of the previously generated customer, and each customer except the final RE is initially waiting for one operand. The delay for downloading the program to the PE (shared bus transfer time in Figure 3.2) is represented by ioBus. The token output by ioBus on *PEixfer* represents the EOS marker for a program, and indicates that transfer of the program is complete.

The initial number of tokens on *SpaceOnRing*, specified at simulation time, controls the number of programs which can be executed concurrently in the network. One of these tokens is needed for each initiation of *GetEntry*, and will be returned when CheckResult terminates. Average queue length measurements for the SpaceOnRing arc will yield the average number of programs executing in the network.

Figure 4.1: Control Model of the Host Computer (HOST)

Figure 4.2: Control Model of the PE with Input FIFO (PEwFIFO)

The waiting time for tokens on WaitResult, which synchronize with the final results of executing the programs on the ring (on *ResultAvail*), can be be used to measure program response time. This is true as long as results are received in the same order that programs were generated. Due to the lack of better synchronization methods, *CheckResult* will determine if this assumption is ever violated during a simulation run. Alternatively, response times could be calculated for results received in arbitrary order by the function associated with CheckResult.

The only delays in the HOST model are for PgmGen (program interarrival times) and ioBUS (service delay on the broadcast bus). The parameters of these delay distributions may be changed before each simulation run (Figures 3.2 and 4.3). Parameter values *minPsize* and *maxPsize* (Figure 3.2) used by InitExpr in the HOST model support variation of initial program size.

4.2. PE with FIFO Submodel

The PE with input FIFO (PEwFIFO) submodel is illustrated in Figure 4.2. The submodel is designed such that any number of these building blocks can be connected together to form the ring of PEs. The port names at the top of the Figure ($request, $hostIN, and $result) indicate how the PE model is connected to the host model. A PEwFIFO submodel is connected to neighboring PE models via the two arcs that enter and exit on each side of the Figure.

At first glance, the control structure appears quite complex. However, most of the functionality of the FIFO queue, the i/o processor, and the reduction processor is represented in the control graph. Very little functionality is specified separately for each control node.

The PEwFIFO submodel is logically divided into four parts: 1) communication with the host computer is modeled by *GenReq*, *xferFin*, and *readInput*, 2) write and read operations on the input FIFO are modeled by *writeFIFO* and *readFIFO*, 3) the functionality of the reduction processor is modeled by *isubtree*, *csubtree*, and *RP*, and 4)

all other nodes model routing of customers in the i/o processor.

The initial control state for the i/o processor contains a token on *readExpr*, waiting for input customers. *readExpr* is a double-headed arc which will initiate node *GenReq* if input arrives from "notifyPEs" in the host, or will initiate node *readFIFO* if input arrives from the input FIFO. In response to input from the host, PEwFIFO generates a request for the new program and receives input customers on *HostInput* and a token on *EOS* (from PE*n*xfer in HOST).

Tokens on SpaceInFIFO model the fixed queue size of the input FIFO. In response to FIFO write operations from the PE connected on the left, FifoW generates acknowledgements on *ACKs*. Once input customers have arrived from the host or the FIFO, the i/o processor removes them one at a time from readInput or readFIFO, and routes them according to the functionality associated with the node *ioP*.

The complex routing functions of the i/o processor, and *all* i/o processing delays including FIFO read access times, are modeled in the *ioP* node. Customers are routed by ioP to the HOST submodel if the final result has been calculated, to ioReplace if the customer is a token place-holder inserted by the current PE, to the RP if it is a selected RE, and otherwise to its internal buffer (BFR) or to the output FIFO (ShiftOut), depending on the customer attributes. The RE selection, load balancing, and replacement policies are modeled in the interpretation for ioP.

When a priority RE is selected for execution it will be expanded by *isubtree* and *csubtree*, and then "executed" by *RP*. Priority REs are executed in the foreground, indicated by the non-preemptive priority scheduling at node RP (">"), and by routing them directly to ShiftOut after *RP*. In the initial performance model, csubtree expands the RE into a set of new REs which can all be executed concurrently (see Figure 3.1b). This expansion leads to the maximum amount of parallelism in the program, and should yield optimistic measures of performance. In a more accurate follow-on study, the expansions performed by csubtree would be determined by analyzing the fre-

quency of execution of various types of expansions in a large number of representative applicative programs.

A non-priority RE is executed by *RP* and then routed to *TokenReplace* to wait for its place marker. Replacement of the token by the result is modeled by updating the number of operands the parent node is waiting for, and then reading a new input customer.

All data processors in PEwFIFO except ioP have very simple interpretation including simple, if any, routing decisions. Two nodes (writeFIFO and RP), have non-zero service delays.

Summary

The probability distributions associated with the two configuration submodels are summarized in Figure 4.3. We chose uniform distributions when the customer includes a large array of data to be processed (estimated to be 5% of the total customer population on the ring, and corresponding to priority REs), and exponential distributions otherwise. Furthermore, we assume a 5 nanosecond FIFO access time, 8 characters per data array element, and a 30 nanosecond delay per character in the i/o processor due to the RE recognition and selection algorithms. Foreground service times in the reduction processor are faster than background service times because program expansion is done by csubtree. Thus, foreground service reflects the time to generate one of the new REs, and should be faster than executing the RE. Parameters for the distributions are again defined at the start of a simulation experiment, and are discussed in Section 5.

The models support estimates of program response time (WaitResult in HOST), utilization of ioPs and RPs, queue length of the FIFOs (Fifo in PEwFIFO), and queue length and waiting times for RP input queues (foreground and background). In addition, the waiting times on waitToken in PEwFIFO, which reflect how long a background result is available before it is inserted in the program text, are of interest.

ioBus service times	exponential(10×*meanPgmSize*)
FIFO write distributions	0.95 : exponential(5×*meanREsize*) 0.05 : uniform(5×8×*minX*, 5×8×*maxX*)
ioP service distributions	token: 30 nsec PE token: 0.8 × exponential(50) 0.2 × uniform(750,1500) non-RE: exponential(30×*meanREsize*) priority RE: uniform(30×8×*minX*, 30×8×*maxX*) background RE: uniform(180,360)
RE execution times	foreground: exponential(100) background: exponential(1000)

Figure 4.3: Probability Distributions for Ring Network Configuration Submodels

5. Simulation Results

We begin the simulation studies by composing four PEwFIFO models plus the Host model. Accordingly, the program size parameters are are set to reflect the scaled-down size of the system. These workload parameters are summarized in Figure 5.1. The program expansion probability of 0.4 is higher than we expect in the implemented system, but should yield optimistic performance estimates for the ring. The limits for the size of program expansion, (50,100), are 2-3 orders of magnitude smaller than the size of data vectors that we expect to operate on in programs running on the system,

exprarate	1 second
minPsize,maxPsize	5,10
expandP	0.4
minX,maxX	50,100
meanREsize	8 characters
meanPgmSize	500 characters

Figure 5.1: Parameters Used for Initial Performance Simulations

due to the size of the ring model. The effects of scaling down system size may, in general, be significant when considering the accuracy of quantitative performance estimates. However, the extrapolation of qualitative performance results obtained in this study to larger networks, appears to be valid. We comment further on this issue below.

The initial performance measure of interest is the *processing power* of the ring network. In other words, we are interested in the average number of executing reduction processors for a given load on the ring. We calculate this performance quantity from estimates of the utilization of the i/o processors and the reduction processors in the model. After debugging the model, we obtained the following estimates:

$$\rho_{ioP} \approx 60\%$$

$$\rho_{RP} < 1\%$$

These results were estimated using 90% confidence levels within less than 10% of the stated values. The estimates indicate much poorer performance than anticipated for the multi-processor ring. Extrapolating the results for 200 PEs on the ring leads to the conclusion that we would have less than twice the power of a single processor. Even if the load on the ring (60% utilization) could be increased, the speed-up would be unacceptably low.

Interactive observation of system behavior during a follow-on simulation experiment, indicated that a significant cause of the poor performance was due to the overly conservative load balancing policy, whereby a PE selects only one RE per program visit. The conservative policy results in a large amount of unnecessary data traffic on the ring, as unselected REs travel around the ring of PEs more than once.

The load balancing policy was modified in the model such that each PE removes 1 in N REs selected according to stated selection rules, where N is the number of processors on the ring. Each RE will be removed during its first complete traversal of the ring according to this policy. The policy requires a small amount of global information to be

known by the ioP (i.e. N must be known to the PE, and should be modified when processors are added or removed from the ring). Two other minor changes were made in the load balancing and RE selection algorithms [VerM83b]. The overall goal is to remove REs from the input stream as soon as possible (i.e. in one pass around the ring) while still distributing the processing load.

Performance estimates obtained for the new load balancing policy were as follows (again within 10% of stated values using 90% confidence levels):

$\rho_{ioP} \approx 60\%$

$\rho_{RP} \approx 13\%$

These results show better than an order of magnitude increase in performance, and indicate that the power of a 200-processor ring might be on the order of 25 times a single processor. However, the results are still disappointing, particularly since we are modeling dynamic program expansions that should result in optimistic performance estimates (Section 4.2).

The results obtained from the performance model are qualitatively consistent with the lower level test-case simulation results reported in [PatD80]. Execution time for a particular parallel program construct in this model showed significant increase in value and in *slope* as the number of elements in the data array are increased to multiple times the number of PEs on the ring. Furthermore, results of our model were reconciled with performance estimates obtained using simplified analytic equations [PatD80, VerM83]. General agreement with lower-level simulations and the very approximate analytical equation, combined with interactive observation of system behavior increases confidence that the performance model captures relevant features of the proposed architecture.

Interactive observation of system behavior in the modified model indicates the remaining performance bottleneck is due primarily to unavoidable traffic on the ring.

For example, when tokens are replaced by results of expression evaluation, the data may remain on the ring for a significant amount of time before other results become available to form new REs.

Further incremental improvements in the RE selection and load balancing policies at the expense of design complexity are possible, and can be studied with the model. We would also be interested in estimating other performance quantities for the system model, such as FIFO and RP queue lengths, as well as investigating the results for larger system models. However it appears more fruitful to examine alternative architectures before continuing the top-level design of the ring system. In searching for an alternative architecture, we would look for architectures which provide static storage for operands which have been calculated but are not yet part of a reducible expression.

6. Conclusions

This study was an experiment in applying queueing simulation techniques to predict the performance of a complex architecture prior to implementation. In particular, we have focused attention on the benefits which can be derived when standard queueing simulation techniques are supported by a Graph Model which also supports interactive observation of functional behavior. The methods employed are aimed at discovering potential performance problems *early* in the design process, before proceeding with detailed logical design of a system.

The level of behavior represented in the performance model is more detailed than an analytic queueing model can support, and more abstract than a standard functional model of the proposed system. In general for new architectures, and particularly for special-purpose applications where multiple processors cooperate in performing a computation, initial modeling of relevant functional behavior significantly increases confidence that a performance model is meaningful.

The effort involved in a performance study using Graph Model tools is considerably less than would be required for construction of a simulation model using a general pur-

pose simulation language. We estimate that a study comparable in complexity to the one we have described would take 2-4 weeks for an experienced analyst to design and devlop the performance model, including resolution of ambiguities in the design specification, two weeks to debug and enhance the model, and 2-4 weeks to run experiments on the model, depending on the results.

The level at which customers in the distributed system are modeled, and the characterization of customers for purposes of routing and service demands are the key decisions to be made during the performance model design. Using the Graph Model, the performance analyst is presented with essentially unlimited options for these decisions. Explicit representation of customer characteristics for routing and service demands, is favorable for accurately modeling unfamiliar system behavior. Stochastic characterization of customer behavior is favorable for simplifying the performance model, and is preferred whenever the probability distributions can be estimated with confidence.

A model, such as the one described in this paper, which contains explicit representation of complex behavior, can be used to validate analytic results, or to guide in constructing simpler analytic models for further study. The model can also be used to study performance trade-offs more quickly and easily than is possible in detailed functional models. If extensive performance results indicate the architecture is reasonable, a more detailed functional model can be used to study the correctness of the logical design. In the case of the proposed ring architecture, this would involve more detailed study of the custom i/o processor algorithm, and the design of the FIFOs.

As is the case in any modeling study, several ambiguities in the design specification we uncovered and resolved during the modeling process, and significant insight into the structure and behavior of the proposed design was gained during model evaluation. The characterization of functional program behavior required to abstract essential performance features, provides an initial step for future study.

References

[BacJ78]
Backus, J., "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", *CACM*, Volume 21, No. 8, August 1978, pages 613-641.

[PatD80]
Patel, D.R., "A System Organization for Applicative Programming", Report No. CSD-810302, Computer Science Department, University of California, Los Angeles, March 1981.

[GriD80]
Grit, D. H. and R. L. Page, "Performance of a Multiprocessor for Applicative Programs", *Proc. of Performance 80*, pages 181-189, 1980.

[GurJ83]
Gurd, J. and I. Watson, "Preliminary Evaluation of a Prototype Dataflow Computer", *Proc. of IFIP Congress*, Paris, September 1983, pages 545-551.

[TreP82]
Treleaven, P. C., D. R. Brownbridge, and R. P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture", *ACM Computing Surveys*, Volume 14, No. 1, March 1982, p. 93.

[VerM83a]
Vernon, M., "Performance Evaluation of Asynchronous Concurrent Systems: The UCLA Graph Model of Behavior", *The 9th International Symposium on Computer Performance Modelling, Measurement, and Evaluation*, College Park, Maryland, May 25-27, 1983.

[VerM83b]
Vernon, M., *Performance-Oriented Design of Distributed Systems*, Technical Report, UCLA Computer Science Dept., in publication.

[VerM83c]
Vernon, M., "GMB Simulator System Reference Manual", Computer Science Department, University of California, Los Angeles, California, December 1982.