Message-Driven Dynamics

by

Richard Anton Lethin

M.S., Massachusetts Institute of Technology
(1991)
B.S., Yale College
(1985)

SUBMITTED TO THE DEPARTMENT IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS OF THE
DEGREE OF

DOCTOR OF PHILOSOPHY
IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
MARCH 1997

Signature of Author ____________________

Department of Electrical Engineering and Computer Science
March 18, 1997

Certified by ____________________

William J. Dally
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ____________________

Dr. Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Thesis Committee

William J. Dally
Professor of Electrical Engineering and Computer Science

Thomas F. Knight, Jr.
Senior Research Scientist

F. Thomson Leighton
Professor of Mathematics

The research in this thesis was supported in part by a fellowship from the John and Fannie Hertz Foundation

Message-Driven Dynamics

by
Richard Anton Lethin
Submitted to the Department of Electrical Engineering and Computer Science
at MIT on March 18, 1997 in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy in
Electrical Engineering and Computer Science

Abstract

Fine-grained message-driven parallel computers offer the promise of massive computational power with relatively low costs of logic and memory. The first such computer, the J-Machine, was constructed at MIT. This research critically examines this machine and finds that global flow control and load balancing were neglected in the J-Machine design and that these features must be considered for successor designs.

The fact that the J-Machine is unable to run effectively a heavy fine-grained synthetic workload demonstrates the importance of this issue. Queues overflow and divert the processors from performing useful computation. Instead, the processors spend most of their time servicing costly overflow traps.

This thesis analyzes the queueing dynamics of large heavy closed message-driven workloads and derives expressions for overflow rate as a function of the number of processors, the number of messages, and queue size. A threshold of tolerable imbalance is derived, beyond which no queue size will suffice. To model the J-Machine's routers, a mean value model is constructed and it shows that the router arbitration logic chosen for the J-Machine shapes the qualitative observed overflow behavior. Simply making the J-Machine queues larger is not a useful solution to preventing overflows: at saturation, the network imparts sufficient imbalance so that even balanced applied workloads exceed the tolerable threshold.

Active global flow control and load balancing mechanisms are needed. Using simulation, the thesis compares one mechanism that returns overflowed messages to the sending processor to a mechanism using a high-water-mark and a priority alternate network. The conclusion is that the high-water-mark based scheme is more effective in preventing overflows.

Keywords: J-Machine, message-driven, message passing, closed queuing networks, fine grain, heavy load, mean value analysis, wormhole networks, buffer sizing, queue overflow.

Thesis Supervisor: William J. Dally
Title: Professor of Electrical Engineering and Computer Science

Acknowledgements

Computer architecture research is a long process, particularly when it involves building hardware, and particularly in graduate school. The pure labor of forging specifications into silicon can approach being overwhelming. Stuart Fiske, John Keen, Peter Nuth, and I devoted large portions of our graduate careers to building the Message-Driven Processor for the J-Machine. As a member of the research staff, Mike Noakes designed logic, diagnostics, packaging, interfaces, and device drivers; Andrew Chang has maintained the J-machine since Mike left. We became and have remained friends.[1]

At Intel Corporation, Greg Fyler marshalled resources and made sure that the MDP passed through corporation and fabrication hurdles. Roy Davison's VLSI layout consulting services were invaluable. Duke Xanthopoulos, Eric McDonald and John Cha designed peripherals for the J-Machine.

Software for the J-machine came from brilliant and effusive code production by the CVA software team: Waldemar Horwat wrote the prototype CST Compiler, COSMOS, and the MDP simulator; Scott Furman, Mike Noakes, Debbie Wallach, Shaun Kaneshiro and Ellen Spertus made it work on the hardware. Todd Dampier wrote the first queue overflow handler routines. Our collaborators at Caltech, notably Daniel Maskit, ported the GNU C compiler to the J-machine.

Andrew Chien and Scott Wills provided early architectural vision for the J-Machine and CST. Soha Hassoun and Jerry Larivee designed early versions of the MDP. Martin Szummer's UROP work sparked my interest in queueing problems in the J-Machine.

I would like to thank Bill Dally for being an inspiring and devoted graduate advisor for me. Bill correctly positions his research projects aggressively beyond the mainstream; the J-machine is one of the most creative hardware projects in academia, and remains one of the largest academic parallel computers ever successfully completed (is there a larger one?). Bill's creativity, stamina, and high result yield became for me the benchmarks that drove me to improve my research. On the occasions when entrepreneurship and consulting threatened to divert me from graduate school, Bill has gone beyond the call of duty to keep me excited about MIT.

My graduate career has benefited substantially from associations with other MIT faculty members. Conversions with Tom Knight and course work with David Gifford and Ron Rivest provided inspiration for educational and enjoyable research distractions. Tom Leighton's mathematical skills helped me attempt more careful queueing analysis. I am also grateful to have had the opportunity to teach VLSI for Chris Terman, and thank him for his career assistance and advice. I would also like to acknowledge the rich experience that came from studying with Robert Goldberg when he visited to lecture at the MIT Sloan School.

[1]As to the J-Machine itself, the one in our lab still runs the diagnostic suite perfectly after more than two years since the last time Andrew and I reassembled it.

I've enjoyed working Steve Keckler, arguing about compilers and processor architecture, ducking under the boom (most of the time), deploying the spinnaker, and jaunting off to Japan. However, rebuilding Etchells 105 was NOT fun.

Kathy Knobe patiently helped organize and encouraged my ideas, diluted my trace-scheduling chauvinism with her *real* compiler knowledge, and provided thesis-related commiseration.

Jim O'Toole has been a great graduate school friend by pushing me to strive in many areas: among them skiing the Sunrise trail at Norsk, cranking out thesis text, and looking for business strategy angles.

I will miss lunchtime conversations with the CAG folks; a political conversation including David Shoemaker, David Kranz, and Larry Dennison smothers the spectrum, and one can't match a real computer architecture conversation with John Kubiatowicz, Kirk Johnson, Don Yeung, or David Chaiken. Any meal with with Nate Osgood epitomizes brainstorming: ideas and enthusiasm like mad. What great fun!

My appreciation for computer architecture (and juicy hardware bugs) grew from designing the world's first VLIW with the Multiflow team. Josh Fisher taught my first computer architecture course at Yale and in doing so determined my career. John O'Donnell assigned me the task of designing the TRACE 28/200 floating point unit on my first day of work, and to date has not stopped sending neat projects my way. David Papworth and Paul Rodman taught me how to design supercomputers and how to debug them with a logic probe. Bob Colwell introduced me to computer architecture research and encouraged my decision to go to graduate school. Since Multiflow, it's been a pleasure working on additional VLIW projects with my friends including Cindy Collins and Ben Cutler at Equator.

I am grateful to Stuart Fiske, Kathy Knobe, Mike Noakes, Jim O'Toole, Teri Mendelsohn and Robert Parker for careful and constructive readings of this thesis text.

For their good friendship and support of my studies, I thank Stuart Fiske, Julia and Peter Nuth, Elliza McGrand, Dan Slavin, Cindy Collins, Michelle Nicholasen, Kathy Knobe, John Redford, Marco Fillo, Jane Murray, Peter Kindlmann, Jodeane Pringle, Stefan Freudenberger, Amy Bruckman, Trevor Darrell, Kathleen Smith, Nathaniel Kahn, Christine Langer, and Dan Simon. I also thank my siblings Carol, Peter and Douglas.

Teri Mendelsohn, of course, deserves special mention as my strongest inspiration and love. We can go to Paris, now.

Finally, as thanks for all that they are and all that have done for me, I would like to dedicate this thesis to my parents, Maria and Jay.

Contents

Chapter 1

Introduction

"Fine-grained message-passing applications running on fine-grained hardware" is the computer architecture philosophy that rallied the construction of the J-Machine at the Massachusetts Institute of Technology (MIT) from 1986 to 1993. The goal of this research effort was to demonstrate that a machine designed to be consistent with this philosophy would lead to greater computer processing throughput for lower cost by using an array of small "Jellybean" chips aggregated into a parallel computer.

When physicists invest years in the design and construction of a synchrotron, they do so with the expectation that the product of the effort will confirm or disprove some theory, uncover some previously unknown anomaly requiring a theoretical revision, or perhaps give evidence for the existence of a new fundamental particle. The investment in the J-Machine was fraction of the magnitude of the investment in a synchrotron, yet the same question forms in both cases: what was *learned* after building the machine?

This thesis provides one answer: despite the attraction of fine-grained message-passing parallel computing, and some attention to architecting a machine for a fine-grained workload, the J-Machine does not run fine-grained message-passing parallel programs well. This is not merely a problem where the design has some performance bug, or is uncompetitive with industry because of more constrained academic budgets and staffing. Instead, the problem is that the most important challenges to the design of such a machine – the scheduling, flow control, and load balancing of fine-grained computations – remain unsolved. This thesis explains why the simple load balancing and flow control methods used in the J-Machine do not suffice. Someone designing another fine-grained parallel computer may wish to read this thesis, to avoid some of the mistakes made in the J-machine's architecture, and overall to better understand the inherent problems with fine-grained architecture.

1.1 Overview

The J-Machine (Figure 1-1) is an experimental fine-grained parallel computer prototype that the Current VLSI Architecture Group built at MIT during the years 1986-1993. This thesis examines the performance of the J-Machine running synthetic workloads and finds that overflows in the processors' message buffers can severely limit the J-Machine's performance. With heavy fine-grained loads, the message buffers overflow on the processors in the center of the J-Machine's mesh.

To analyze the observed behavior, this thesis constructs models of the machine and solves them using queueing theory: one idealized model using classic product-form-solution closed networks, and another incorporating more realistic architecture features with mean-value solutions. The analysis finds that simply using queues to load balance the machine and avoid flow control requires queue sizes larger than those that the J-Machine provides, and that larger queues are insufficient with even a slight amount of imbalance.

Finally, the thesis presents the simulated behavior of some simple hardware load balancing mechanisms that are designed to avoid queue overflows. This work finds that one simple improvement in the J-Machine architecture can help avoid overflows when running workloads that are problematic for the current machine.

The following are among the contributions of this research.

- Methods for estimating and calculating the overflow rates and residencies for closed queueing networks for balanced (Section 3.3) and imbalanced (Section 3.4) random workloads.
- Methods for determining the required hardware buffer size as a function of the number of processors and the number of tasks for closed uniform random queueing workloads (Section 3.3.4).
- A formula for the degree of load imbalance that a closed queueing network can tolerate as a function of the number of processors and tasks (Section 3.4.2).

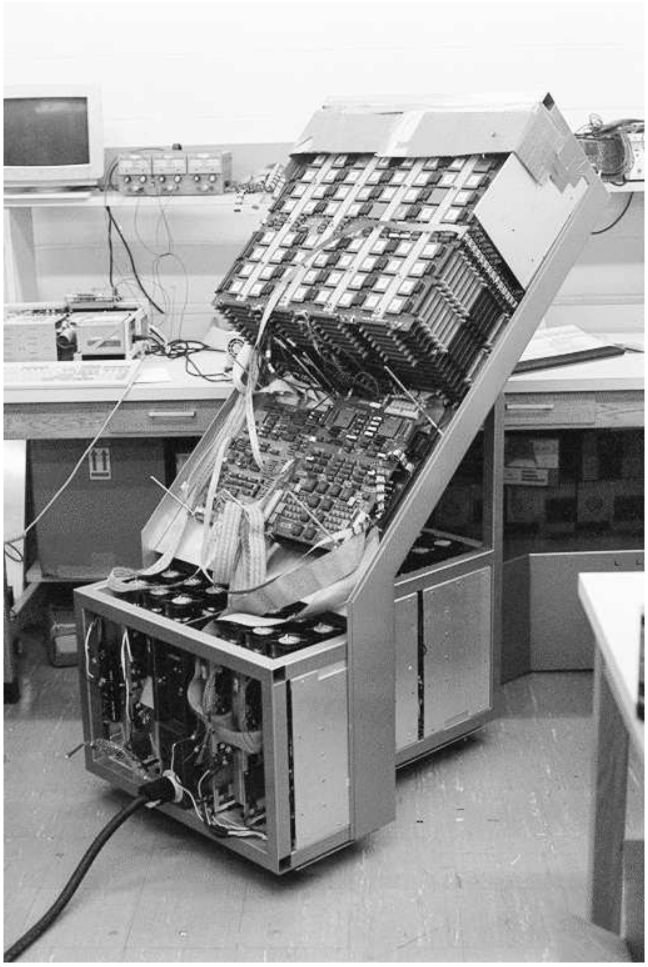


Figure 1-1: Photograph of the MIT J-Machine with the cover removed. The processor are arranged on a stack of 16 printed circuit boards, with each board holding an 8 by 8 array of Message-Driven Processors. You can see this stack in the upper portion of the pictured machine.

- The observation that a shared-memory workload where every random task request is answered with a task reply is substantially more stable than a workload where the tasks are walking the machine randomly (Sections 3.5.3, 4.10, 5.4.1, and 6.3.1).
- Measurements of the performance of the J-Machine running synthetic models of its intended fine-grained random workloads, and determination of the load parameters where the machine performs predictably and well. In general, best performance is achieved only when the network is lightly loaded (Chapter 4).
- The determination that the workload parameter region where the J-Machine is unable to perform well is coincident with the workload parameter regions where processor message buffers overflow (Chapter 4). In other words, message-queue overflows cause poor J-Machine performance.
- The observation that the location of the queue overflows within the J-Machine is in the center processors (Section 4.7).
- Modifications for a published mean value queueing model for heavy load traffic (Appendix A) and results from this model, showing that the particular choice of unfair arbitration within the J-Machine routers is responsible for the queues overflowing in the center of the machine (Chapter 5).
- Presentation of the simulated behavior of simple hardware load balancing modifications for the J-Machine. These simulations demonstrate that a high-water-mark based traffic escape for the J-Machine processors performs much better than other load balancing mechanisms which are used in existing parallel computers (Chapter 6).

1.2 Background

The following section is an overview to the context, goals, and rationale for this research. It begins with a description of the philosophical context that drove the J-Machine architecture: the notions of fine-grained hardware and fine-grained software, and then describes some of the architectural features of the machine which are important and novel. Then the section describes some of the observations that drove this thesis research: anomalous and unanticipated behavior when the machine runs heavy fine-grained workloads. Next, the section describes the queueing models that the thesis deploys to explain the observations. Finally, the section describes the evaluation of some of the load balancing attempts to solve the observed problems.

1.2.1 Philosophy

Fine Grained Hardware

The J-Machine [DFK+92, DCF+89, Dal90a, DKN93, Dal90b, DK85, NWD93, ND90, ND92, Cha87] is a fine-grained parallel computer, meaning that the amount of memory on each of its processors is small. In fact, the name "J-Machine" is an abbreviation for the full name of the computer: "Jellybean Machine." On a system with a small amount of memory per processor, the processors can be small, cheap, and plentiful, like jellybean candies. Consequentially, a parallel computer built from these "jellybeans" should be cheap and therefore more likely to be cost-effective [Dal90a].

Fine Grained Software

The intended software model for the J-Machine is also fine-grained. Tasks running on the machine would be quickly handled and quickly retired, perhaps taking as few as 10-100 cycles [Dal87, p.135]. The logic behind this short task model is that dividing an application into many small tasks exposes as much parallelism as possible. Furthermore, many algorithms can easily be structured into these small tasks. The J-Machine, being a massively parallel computer, would have the computational power and memory bandwidth required to execute those tasks in parallel.

Wire Efficiency

Another component of the J-Machine's philosophy is wire efficiency. Some interconnection networks scale the amount of wire faster than the number of processors (e.g. hypercube); a goal of the J-Machine architecture was to avoid this. This drove the decision to use a

3D topology for the network: one indicative characteristic of this network is the fact that messages traveling from one place in the machine to another never need to "double-back" in space: they can travel by the shortest distance to their destination [Dal87]. However, to allow unstructured workloads (see below) it was decided to provide the abstraction at the lowest levels that the machine is fully-connected.[1]

Unstructured Workload

From the moment of their genesis, some parallel computing architecture and programming model design choices are extremely centered on the topology of the interprocessor communication network and on mapping the structured and regular data flow of some applications directly onto the regular network topology. A systolic array is an example of such an architecture. One strength of this topology-centered approach is the mathematical certainty of the associated algorithm analysis. A weakness is that it is limited to specific data flows which are known before runtime.

The J-Machine was architected to try to address this weakness. Rather than constrain traffic to a particular structured flow, the primitive operation on the J-Machine is the object-oriented method invocation. The intended workload is a distributed object-oriented program. To initiate remote methods quickly and efficiently, this operation is mapped directly to a hardware primitive: sending an interprocessor message (Figure 1-2). The J-Machine hardware was designed to send messages quickly with high throughput and thereby effectively run parallel objected oriented workloads of unstructured and arbitrary traffic.

Target applications for this sort of machine might be an AI application of agents or actors [Dal92] communicating among themselves, or a virtual reality model with concurrent world modeling and rendering, or a speculative concurrent distributed discrete event simulation. For these applications, asking the programmer to map their applications all as systolic

[1]This doesn't seem controversial today: most parallel computer networks provide the same abstraction, that is, the network inputs accept messages from any input for delivery to any output. But there's a tension here: another desirable property of the abstraction is equality (or isomorphism) between all network inputs. The J-Machine's 3D network provides a connectivity equivalence. However, it does not provide equivalence in performance, and this stems from the fact that the mesh, as the only topology that does not require doubling back is also inherently non-isomorphic. Note that there are other networks with provably efficient space-mapping (but don't have the no-doubling-back property) but can provide isomorphic interfaces (e.g., Leiserson's Fat Tree [Lei85, L+92]).

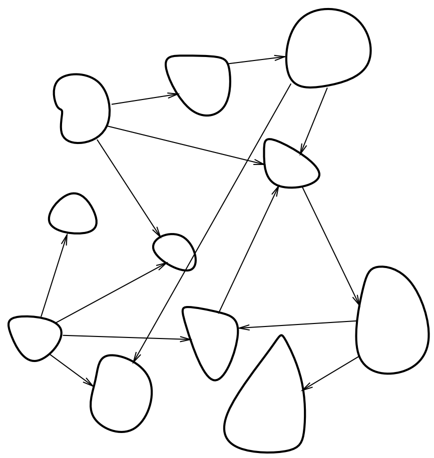


Figure 1-2: Programming model for the J-Machine: a cloud of distributed objects, communicating via message passing.

flows would be too burdensome. As a consequence, we tried to build a machine that would be able to run an unstructured workload well.

1.2.2 Hardware

Wormhole Network

The network of the J-Machine was designed to support the philosophical goals. It is a wormhole network with excellent load vs. latency characteristics, as demonstrated analytically and with simulation, running various random and other traffic flows [Dal90c]. The routers are simple with little multiplexing and buffering. This makes them fast and cheap. Dally showed that wormhole routers guarantee progress (freedom from deadlock and live-lock) because the link-level blocking within the network cannot form cycles. As long as the destination processors obey the *consumption assumption*, that is, the processors guarantee that they will remove any arriving message from the network, the acyclic blocking dependencies can not form deadlock (Figure 1-3).

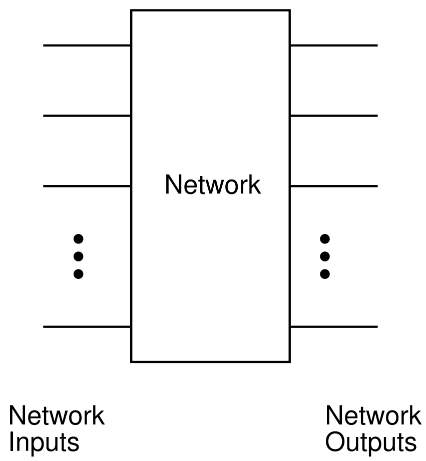


Figure 1-3: The wormhole network presents an abstraction of progress-guaranteed message-delivery from inputs to outputs, as long as the outputs obey the consumption assumption.

Message-Driven Processor

The processors in the J-Machine were also designed to support the goal of running unstructured fine-grained software well. The processors are different from conventional processors: they are Message-Driven Processors (MDPs) [Dal87, p.183]. The difference is in the way that tasks are initiated. While most current parallel computers are fast conventional commodity CPUs that are connected with some communication network (in many cases, mesh-based networks which are siblings to the J-Machine's network), the MDP is a hardwired custom processor, dedicated to its tasks. With the conventional processors in most parallel computers, the initiation and management of tasks is mostly under program control; occasionally, it is interrupt-driven. In the MDP, the management of tasks is hardwired as First-In-First-Out (FIFO) service of the message queue (Figure 1-4). The MDP is designed to repeat the sequence of initiating, executing, and suspending tasks, and it is only this task handling scheme which works well on the MDP. Fixing this task loop into hardware shaves a couple of instructions off the initiation and suspension of tasks – which is important if tasks are themselves only tens of cycles long.

In addition to being designed to run fine-grained software well, the MDPs are themselves

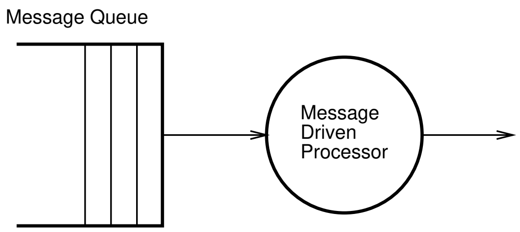


Figure 1-4: In the MDP, there is a direct correspondence between tasks and messages. The MDP hardware implements FIFO service of an incoming message queue. It is overflows of this pictured message queue, upstream of the processor, which are the subject of scrutiny in this thesis. These queues are fixed size and 512 words long.

fine-grained: they are fine-grained processors.[2] A fine-grained processor is one with a small amount of local memory; each MDP has about 10 KByte of memory on-chip and 1 MByte of slow off-chip DRAM. The rationale behind making a fine-grained machine is that the marginal cost of additional processors is low, so they should be scattered liberally around the machine. Note, though, that this fine-grained architectural choice comes at a cost; it implies[3] that memory-implemented hardware features *and particularly the message queues* will be small. This results in another burden or constraint for flow control, load balancing and task scheduling. This thesis will show that in the J-Machine, this additional burden becomes the most critical performance issue.

J-Machine

The J-Machine was built by a team of MIT graduate students and staff working with Intel Corporation engineers to produce the MDP chip [DFK+92]. This team completed a 1024-node machine at MIT in 1992 and brought up the system's micro-kernel, called COS-

[2] Fine-grained software by our definition involves applications made up of short (10-100 cycles) tasks. A fine-grained parallel computer is one where the processors each have a small amount of memory; or in other words, where the memory is divided finely over the processors.

[3] "Implies" is perhaps too strong a word: rather it strongly biases (if one isn't awake at the wheel) the design choices in a way that makes memory implemented hardware features small. In the case of message queues, one could (really, should) design them so that their overflows smoothly flow into other nearby processors, so that capacity constraints do not impose such stark performance penalties.

MOS [HTD], and created the compiler for Concurrent Smalltalk (CST) [CD88a, CD88b, Hor89, HCD89], the system's programming language. Consistent with the philosophy of the machine, CST offered a programming abstraction far from the details of the machine,[4] focused principally on letting the programmer express fine-grained object oriented programs.

The effort was extremely successful. With 1024 nodes, the J-Machine is one of the largest academic parallel computers ever built. The machine actually runs applications and is extremely reliable.[5] With the success of the first prototype, two additional J-Machines were built: one for Steve Taylor's parallel programming research group at Caltech, and another for Argonne National Laboratories. Although some aspects of the machine, such as the clock speed, were not competitive with other commercial processors of its time, other aspects of the machine, such as the network, were competitive.

1.2.3 Anomaly

J-Machine Evaluation

Various applications were written for the J-machine. A number of synthetic benchmarks tested the throughput of the network, the agility of message passing, the ability to perform sorting, and the speed of barrier synchronization [NWD93]. For the most part, these synthetic benchmarks were hand-written, occasionally the result of the programmer working strenuously to overcome the MDP's hard FIFO coding model.

Two applications were written in CST: a ray tracer [Fat92], and a traveling-salesman-problem search [Kan93]. The Caltech research group produced a C compiler for the J-machine that includes message-driven language extensions, called Message-Driven C (MDC) [Mas94, MZT93, MT93] and ported a number of applications to the machine.

All these applications were coarse-grained, in contrast with the initial goals of the project to build a machine for fine-grained software. That is, the number of cycles that tasks in

[4] Later, the compiler team add programming primitives to allow the programmer control over the location of objects and method invocations.

[5] At the time of this writing, the current J-Machine at MIT, which consists of 512 processors, has been running without any hardware failures for over 2 years. This is great, because replacing a processor in the center of the J-Machine is as tedious (and unpleasant) as simultaneously replacing the head gaskets of eight automobiles [ND90].

the applications took to execute might be measured in the hundreds, rather than tens. The applications tended to use the network lightly. For the CST applications, it was surprising that the programmers chose to try to make the applications as MIMD-like as possible, trying to minimize communication.

For the MDC applications, Maskit observed that using the J-Machine, the overhead for communication in the network was an insignificant component of application run time [MT93]. Partly, this was because the MDC tasks were so heavy-weight that it was impossible to use the network heavily. But also, this was because the overhead for injecting and receiving a message on the MDP actually was low, as the J-Machine designers had intended.

Fine Grained Workloads

Still, though, with these application programs being coarse grained, the performance of the J-machine running fine-grained applications was not being studied.

> *One of the goals of this research has been to begin to study the ability of the J-Machine to run its intended fine-grained workload.*

Network Saturation

Waldemar Horwat, the author of the CST compiler, empirically determined a communication-to-computation ratio for some small microbenchmarks written in CST. This ratio is the average amount of network traffic produced for a unit amount of computation. Based on this ratio, he concluded that when the size of the J-Machine exceeded 300 processors (the "Horwat Limit" for the J-Machine), the amount of communication traffic produced by the machine would exceed the capacity of the J-Machine's bisection [Hor89, p.110]. That such a limit exists makes sense because with the parallel computer being cube-shaped, the bisection of the machine grows only as $N^{\frac{2}{3}}$, slower than the number of processors N. Horwat concluded that in order to use machines larger than 300 processors, the programmer or some system process would need to map the application to the machine in a way that exploited locality and reduced the bisection bandwidth requirements of applications.

However, this thesis takes the position that if the programmer does not explicitly reduce

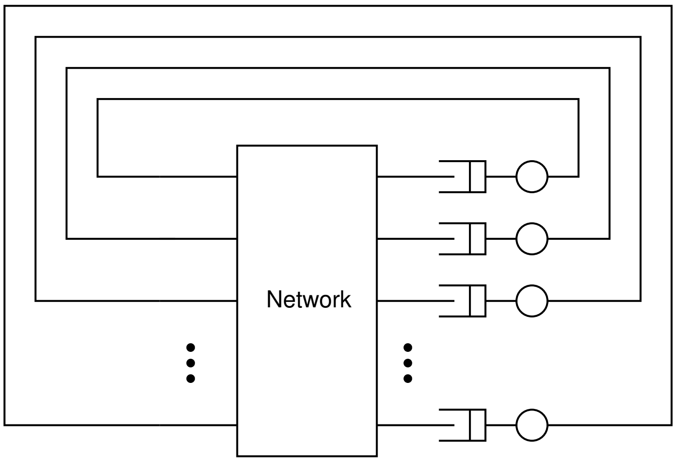


Figure 1-5: When the MDP's are combined with the network, the resulting system appears to have cyclic dependencies.

bisection pressure, the application should still be able to run well, albeit at the network-limited rate.

Our J-Machine laboratory prototype consisted of 1024 nodes, and the MDP has network addressing bits that would enable the hardware to scale to as many as 32K nodes. These machine sizes are larger than the Horwat Limit.

> *Another goal of this research has been to study how the machine responds to the situation when the network is saturated by the application, in the region beyond the Horwat Limit.*

Is the System Flow Dependency Graph Acyclic?

The wormhole network guarantees that messages make progress to their destination because of the lack of cyclic constraints in the link-level flow control dependence graphs and because of the consumption assumption. However, when one looks at the whole J-Machine system (including the MDPs) *the consumption assumption appears to be violated.* If the message queues filled, the processor would not be able to take in any new messages from the network

(Figure 1-5). Another way of looking at this is to note that the flow-dependence graph for the whole J-Machine (not the network in isolation) appears to include cycles of dependent and finite message buffering with non-adaptive routing. These dependency cycles run through the processors, into the network, and back into the processors.

> *Another goal of this research has been to investigate the consequences of this apparent violation of the basic network-progress assumptions.*

Processor Message Buffers

The designers of the MDP recognized the potential violation of the consumption assumption, and they tried to avoid it by making the message buffer on the MDP relatively large (512 words). Our expectation was that by making this queue large, it would almost always be empty. By being empty, the processor would implicitly obey the consumption assumption. But, while the buffer is large, it is still finite-sized. So, there is some chance that it can fill.

Processor Queue Overflows

If the message buffer fills, the consumption assumption is violated, and the machine can deadlock. To provide freedom from deadlock, the designers of the MDP added a message-buffer-overflow-trap feature for the MDP. The trap handler for message overflow empties the buffer to some alternate storage, to make room for more messages - so that the processor can comply with the consumption assumption. During the trap, the incoming network queue is temporarily blocked, so that the system does not lose any messages.

However, it turns out that the trap handlers for message buffer overflows is very slow, so much so that the performance penalty that an overflow trap exacts on the system is costly. With some design attention, the trap handler could have been much faster. However, the designers expected queue overflows to occur rarely, and so spent little attention on making the overflow handler fast.

Implicit in the inattention that this aspect of the architecture received–basically, making it *possible* for a processor with a full-queue to self-rescue itself, while not making it *fast*–is an assumption that overflows of these queues would be rare.

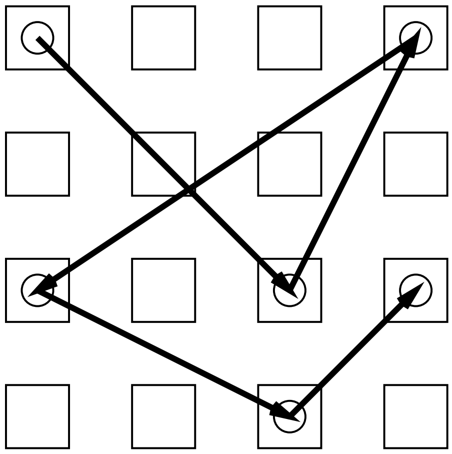


Figure 1-6: A task makes random jumps around the machine. A large number of these tasks are created in our simple workload.

But how rare is rare?

> *The final goal of this thesis research is to provide a method for predicting the frequency of queue overflows in the J-Machine and in similar systems.*

A Simple Experiment

I decided to try a simple experiment: measuring the performance of the J-Machine running a heavy-load, synthetic, fine-grained workload. This synthetic workload is simple: a number of tasks are created on the machine and distributed over the processors. Each task demands a small amount of computation from the processors (40 cycles or more, controlled by a spin-loop), picks another processor on the machine uniformly at random, sends a message to create a similar task on that random processor, and then suspends (Figure 1-6). In effect, the workload consists of a large fixed number of tasks making random hops over the processors in the machine.

One might reasonable intuit that this workload should perform well on the J-Machine: it is uniform and simple; the number of tasks in existence at any moment is constrained (there

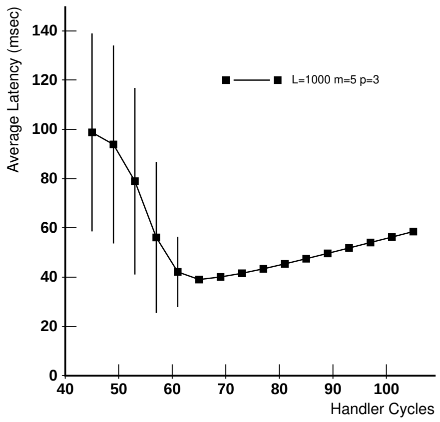


Figure 1-7: Moving into the fine grained domain (shorter task handlers), the average latency and variance of the simple benchmark increases sharply.

is no explosion of parallelism[6]), so it is a sort of best-case model of the type of oblivious parallel program for which the machine was designed. It this intuition correct?

Sharp Drop In Performance

The measured performance of this workload (see Figure 1-7 and Chapter 4) exhibited surprising results. Each point on the graph shows the measured latency of the workload averaged over several independent runs. As the message handler time is reduced (moving to the left along the curve) the workload is becoming more fine grained. And as the workload is becoming more fine-grained – the intended workload for the J-Machine – the latency increases sharply, with high variance in the measurements. This sharp increase in latency occurs at the point when the network saturates. Ideally, the performance would level off as the machine approached saturation. But this is not the case on the J-Machine. Why is this sharp performance loss occurring?

[6] See Blumofe and Leiserson [BL93] for more about the size problem.

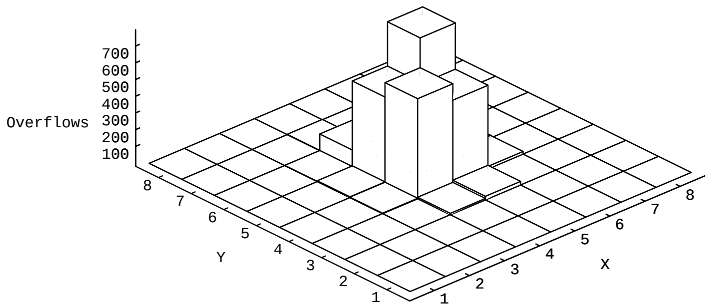


Figure 1-8: The cumulative number of overflows vs. processor X-Y position for the 4th processor board in the J-Machine after many runs of the synthetic workload. All of the overflows occur in the center processors.

Drop In Performance Due Queue Overflows

Further observation of the J-Machine showed that the reason for the drop in performance was due to the onset of processor message queue overflows.

> *In this heavy load region and running this simple synthetic workload, the implicit architectural assumptions that this queue would rarely overflow proves to be false.*

Another surprising observation is that for this workload, overflows occurred in the processors near the center of the machine (Figure 1-8). Overflows never occurred on the edge processors.

> *This thesis uses a mean value model to demonstrate that the overflows are occurring in the center of the machine because of unfairness in the machine's routers.*

1.2.4 Analysis

Observing the actual behavior of the J-machine raises some questions. How well does the J-Machine run its intended fine-grained workload? What happens when the network on the J-Machine becomes saturated? Is the implicit violation of the consumption assumption important? Why do queue overflows occur, and why do they occur in the center of the machine?

How can these questions be answered?

Using Queueing Theory

Classic queueing theory comes to mind immediately. This theory is "classic" because the body of this research is several decades old, well-established, and thoroughly deployed. The mathematical techniques of classic queueing theory rely largely on ergodic properties of continuous-time, discrete-state Markov models. The theory includes a number of results for queueing networks, which are systems of servers traversed by customers. By restricting the server and routing behaviors to particular mathematical ideals, e.g. FIFO non-preemptive exponential service disciplines, the queueing networks yield elegant solutions for the equilibrium probability distribution of the configurations of customers on servers. These probability distributions allow one to solve for throughputs, queue size processes, and so on.

One drawback of classic queueing theory is that it does not allow for blocking behaviors in the queueing networks. Because blocking turns out to be such an important effect in the J-Machine (and in any parallel computer), the applicability this theory is limited. Still, one can assume that network effects are unimportant, and study the dynamics of message traffic on large machines like the J-Machine. This is done in Chapter 3, constructing a model of this synthetic workload on the J-Machine as a classic closed queueing network, and generate techniques for determining the expected rate of overflows for this workload for both balanced and imbalanced workloads. Chapter 3 also derives relations for the required queue size as a function of the number of processors, the number of tasks, and the degree of imbalance. Based on these results, The chapter finds that when the workload is balanced and uniform, the size of the MDP's buffers is barely adequate for short messages (which consume buffer space slowly) and inadequate for long messages (which consume buffer space more quickly).

In addition, the chapter shows that for imbalanced workloads, well-known queueing dynamics lead to massive task accumulations at slow processors when relationships among the number of processors, the number of tasks, and the degree of imbalance exceed certain thresholds.

Mean Value Models

Unfortunately, because the classic queueing models do not account for network blocking, they cannot explain why overflows occur in the center processors of the machine. Fortunately, approximation techniques exist for modeling closed systems of processors connected by a deterministically routed wormhole network. These techniques are based upon mean value models, which are numerical methods which ignore the higher moments (e.g., variance) and solve for the mean (average) values of traffic rates, accumulations, and waiting times in queueing networks. Inexact approximations exist for adding blocking, and particularly wormhole routing, to mean value models. In Chapter 5 and in Appendix A, I use these approximations, implementing a solver and improving it for the particulars of heavy load traffic on the J-Machine. These techniques yield a model which is only qualitatively correct; the accumulation of errors from the many approximations and the particular choice of studying the heavy load domain make it difficult to get quantitatively correct traffic models.

The mean value model shows initially that overflows should occur on the edges of the machine. With some thought, it becomes apparent that this is due to a discrepancy between the fairness of the routers in the first version of the model vs. those in the J-Machine. A modification to the mean value model is developed for the J-Machine's unfair routers. This new model shows that queues overflow in the center of the machine.

1.2.5 J-Machine Improvements

One might question whether the J-Machine has fundamental design problems. It is incorrect to argue that the J-Machine is severely flawed: the fact that it runs applications and various small benchmarks counters such an argument. However, there clearly is room for improving the J-Machine, particularly for this heavy-load, fine-grained workload, to prevent the dramatic performance loss due to queue overflows. The "obvious" solution – simply increasing queue sizes – is not the best solution: this adds to the cost of the machine because the increased memory on each processor for the larger queue is multiplied by the large number of processors in the parallel computer. Furthermore, it does not help with the case of the imbalanced workload, with its large accumulation of traffic. So although adding memory might help, it does not completely address the problem.

Because queue overflows are not infrequent, it might help performance if queue overflow handlers were faster. However, any slowdown at all for overflow runs the risk of creating a local hot spot in the machine, triggering the bottleneck dynamics which lead to further massive traffic accumulations at that hot spot.

Request/Reply

If the traffic pattern for tasks is modified from a basic random walk to one following a request-and-reply pattern (similar to the traffic one might find in a shared memory machine), the queueing behavior of the workload is extremely stable. Each of the four core research chapters briefly discuss this, looking at this phenomenon using classic queuing models, J-Machine measurements, mean value analysis, and simulations. All find that request-and-reply is more stable than the random walk.

Since the request-and-reply routing is a characteristic of shared-memory machines, and since the J-Machine does not require messages to follow the request-and-reply pattern, in some sense the instability of the J-Machine workload stems from it not being a shared-memory machine.

High Water Mark, Alternate Output Priority

As an alternative to these solutions (increasing queue size, speeding the overflow handler, or changing to a shared-memory message pattern), Chapter 6 investigates methods in which the hardware performs some primitive heuristics to attempt to balance the load. One heuristic involves designating a high-water mark threshold in the message queue, and automatically switching the processor to send output traffic on a high-priority network when the quantity of accumulated traffic crosses that threshold. In this way, the hardware automatically responds to the danger of a buffer overflow by giving the processors output channel a boost in priority and potentially throughput, allowing the processor to reduce the size of its input queue by serving traffic faster.

> *In the heavy-load traffic domain, simulation shows that this technique is sometimes effective in preventing queue overflows.*

1.3 Related Research

1.3.1 Other Architectures

Alewife

Alewife uses a single network and faces the possibility of deadlock due to the protocol's backward-coupling of blocking from the network inputs to the network outputs [KJA+93]. Alewife prevents deadlock by using a countdown timer on each processor that resets any time the network output at a processor makes progress. If the counter reaches zero, the processor takes an interrupt and manually drains all arriving network traffic into memory, breaking deadlock. This timer-based solution is a different way of detecting deadlock than the J-Machine's queue overflow trap. Neither the J-Machine nor Alewife's solution actually detect the deadlock – instead, they both detect supersets of the deadlock condition. A solution with more detection precision might be a timer that resets any time the input or the output makes progress. Kubiatowicz has reported that the Alewife team did not find these deadman timeouts to be problematic [Kub95]. This is probably a result of a combination of factors. First, the Alewife machines that have been actually constructed have been small – 32 to 128 nodes; in contrast, the J-Machine prototype has 512 to 1024 nodes. This means Alewife is unlikely to saturate its network. Second, the Alewife group has concentrated on actual application benchmarks, while in this thesis the focus has been on synthetic benchmarks that try to characterize the J-Machine's overflow problem. Third, the Alewife machine mostly runs shared memory types of applications (though there are some exceptions, notably Chong's work [CA96] that contrast shared memory and message passing implementations of applications); as I report in Sections 3.5.3, 4.10, and 5.4.1, the request/reply sort of traffic is substantially more stable with respect to queue sizes than is the synthetic message passing workload (which I call "Snakes"). Despite this, for shared memory machines this thesis work is relevant (and the equations in Section 3.5.3 should be considered), particularly if they are large in size and definitely if they also support message-passing.

DASH

The architects of the DASH machine [LLJ+92] also recognized the potential for deadlock, and so partitioned their protocol into requests and replies, onto two decoupled virtual networks – a solution also used by von Eicken et. al. in Active Messages on the CM-5 [vECGS92]. In DASH, the protocol does allow for "request" messages to spawn other "request" messages - which then creates the possibility of deadlock. However, if the shared-memory protocol engines detect that they cannot make progress on their network inputs, they start to send negative acknowledgments back to the requesters, forcing those requesters to perform their traffic in a deadlock-free strict request/reply fashion. This thesis applies in the sense that for larger DASH-style machines which might saturate the bisection and have heavier and larger network traffic loads, one might ask how the queue sizes and possibilities of deadlock arise. The thesis provides some answers.

Cilk (CM-5)

Blumofe et. al.'s Cilk programming system [BJK+95] is more aggressive about using the two virtual networks that the CM-5 provides. This stems from a sophisticated approach to scheduling; see Blumofe's work [BL93] on scheduling multithreaded computations. Brewer and Kuszmaul also show [BK94] how to improve Cilk's substrate Strata, observing the formation of bad patterns of traffic and showing how to get good throughput on high-volume all-to-all types of communication patterns.

Cray T3D

Kessler and Schwarzmeier report [KS93] that the Cray T3D (and T3E) uses wormhole routing. The design is based upon the routing used in the J-Machine [Dal90a]; however, the T3D network has a torus topology so that it requires the extra deadlock-breaking virtual channels that Dally invented [DS87]. To fight the isomorphism-loss (noted by Adve and Vernon [AV91, AV94]) that the deadlock-breaking virtual channels cause, Scott and Thorson [ST94] demonstrate using simulated annealing to optimize the virtual channel routing assignments to balance channel traffic. The lack of isomorphism in the J-Machine's mesh topology is one of the causes of traffic imbalance; even Scott and Thorson's solution retains some nonisomorphism, however, it is so slight and the T3D network is so fast that it's hard to see the traffic imbalance being problematic.

With the T3D being a coarse-grained hardware implementation (in contrast to the J-Machine, which is fine-grained) the network queues are more than 10 times larger than on the J-Machine; the methods in Chapter 3 show that this is sufficient to prevent overflows in balanced loads. With imbalanced loads, however, this may not be sufficient.

Dally reports [Dal93] that the T3D network implements the return-to-sender algorithm (RTS) that Chapter 6 finds to be ineffective in balancing the load. RTS may be effective for load-imbalance due to burstier traffic. Demonstrating this would be an interesting result. Chapter 6 suggests that the RTS algorithm needs to be combined with some sort of backoff and flow control, but leaves the design of such a backoff scheme as further work.

Chaos Router

Konstantinidou and Snyder's Chaos Router [KS91] uses adaptive routing with random misroutes that is provably free of livelock and deadlock. Note, though, that their network analysis, like the ones for the J-Machine's network, is for an *open* network; that is, they study the network in a context where the consumption assumption is made. It is not immediately apparent how their network behaves in the closed (e.g, including processors in the system) context. Probably, with only a small modification, this can work as a seamless solution to dealing with queue overflows: a true fine-grained approach. Study of this network is likely to be fruitful, and the analytical modeling process aesthetically rewarding. As Karamcheti and Chien point out [KC94], one downside of the Chaos approach (and any adaptive network solution) is the loss of guarantees of message order arrival, but asking for this if the primary metric of quality is seamless overflow handling may be asking too much.

EM-X

Kodama et. al. report [KSS+95] that the EM-X parallel computer has transparent concurrent spilling and restoring of the network output buffer to and from an off-chip message buffer. This too is an elegant and seamless way to increase effective queue size; and its inclusion shows that the architects of the EM-X recognize the problem of queue capacity constraints. The size of the EM-X queues are not reported; the methods in Chapter 3 may add some insight into proper sizing of these queues.

Chien's CONCERT

Chien, one of the early J-Machine team members, has published a number of excellent papers closely examining interaction of network design and programming. This work comes from his implementation, with Karamcheti and Plevyak, of the Concert programming system [CKP93]. With Karamcheti, he highlights [KC94] the effect of network design decisions on end-to-end properties that programmers find valuable: flow control, in-order delivery, and reliable transmission. The J-Machine network provides in-order delivery and reliable transmission (though not fault-tolerant). However, it does not provide adequate flow control mechanisms. Karamcheti and Chien report [KC94] that providing these properties in software (rather than in the network) can be expensive. Aoyama and Chien [AC94] report the cost of implementing these features in hardware.

Kim, Liu and Chien propose [KLC94] that a good solution to this problem is "Compressionless Routing" - a way to speculatively inject wormhole messages, but with clean flow control feedback from the network and still allowing adaptivity. One nice observation is that the network can provide selectable in-order guarantees (by restricting the route) to allow the programming system to make the tradeoff between adaptivity and in-order delivery. Note that this study is with the network in an open context (in contrast to the research in this thesis). Within this context, the problem of deadlock formation is strictly within the network (rather than involving the processors or the application). They report that deadlocks actually occur infrequently. Developing an analytical reasoning behind this observation would be interesting.

Others

Brewer et. al. propose [BCL$^+$95] moving the management of the queueing to the application, proposing an application-network interface called RQ for "Remote Queues." This is in contrast to the J-Machine, which fixes queue and communication policy in hardware. Given that general universal flow control solutions are elusive, allowing the application more control with clean interfaces is a good approach. Wallach et. al. [WHJ$^+$95] propose another clean interface to the network called Optimistic Active Messages (OAM). In their scheme, the processor optimistically tries to send messages, but if network backpressure is encountered, the message is sent to memory. This differs from the J-Machine, where the processor gets caught by network backpressure, and is forced to do nothing but take SEND faults and spin until the backpressure is resolved. Wallach et. al.'s scheme has the desirable property of breaking the progress deadlock, and is probably the right model for designing a network interface (simply designing the network interface to allow the programmer to *choose* to use OAM is a giant-step from the J-Machine.)

The work in this thesis is less normative than the research in Brewer et. al. and Wallach et. al.; rather than proposing new solutions (other than the load balancing proposal in Chapter 6) this research reflects a preference and, with us having spent the time and money to build the machine, an *obligation* to analytically study of the existing J-Machine. Furthermore, the J-Machine architecture, by hard-wiring queueing and interface policy, is resistant to being used as a platform for normative research (Though Spertus [SD, S$^+$93, SD95] obtained some results). But the exclusion from the ability to do normative research is not just a question of hardwired policy, this is also a general downside of doing computer architecture research by building hardware.[7] Sigh. One can't be all things.

1.3.2 Queueing Models

Most queueing theory is based upon continuous time, discrete state Markov models; Kleinrock provides [Kle75] the classic introductory text. The tools for queueing networks is based up early work by Jackson and Gordon and Newell [Kle75] who gave now well-known product form solutions to equilibrium probability distributions for expected latency, queue size, and

[7] Things can get pretty bad: suppose the chip doesn't work!

throughput at the nodes of a network given exponential random service time distributions and random routing decisions. Basket et. al. [BCMP75] extended this work to a general solution for a more (though not fully) general set of service distributions. Chapter 3 primarily requires only the simpler Gordon and Newell solution for the closed queueing network and give ways, based upon that solution, to estimate overflow rates as functions of the number of customers and queue size. The challenge is in overcoming an intractably-large state-space distribution; Buzen [Buz72, Buz73] provided some recurrences which help in this problem.

Section 3.5.3 does use Basket et. al.'s solution (called BCMP) to attempt to study why the Request/Reply pattern of communication is so much more stable than the unconstrained "Snakes" traffic. Unfortunately, the formulation in terms of BCMP for Request/Reply does not have the simple solution as does the formulation in terms of the Gordon and Newell network for Snakes; and for this reason the analysis of Request/Reply remains for future work.

Determining the rate of queue overflow is essentially one of determining higher moments of the queue size random variable. McKenna and Mitra introduced [MM84] a method involving an integral representation of the BCMP solution that yields estimates of the higher moments. This method is not considered in this thesis; one possible limitation is that the method applies only to moderate-utilization domains and only in the case where each class of traffic includes one "think" node – perhaps implicitly making the network an open network. However, the very simple traffic patterns in this thesis might allow simple solutions using McKenna and Mitra, and shed some light on the Request/Reply vs. Snakes question. This remains for future work.

Stability

Note that the term "stability" in the context of this research is describing the volatility of the queue size process with respect to the buffer size, examining the implicit load balancing that comes with randomly selecting the destination node of communication patterns. In the context of open networks, stability refers to the long-term behavior of the queue size process, whether the achieved average service rate of all nodes is higher than the rate of incident traffic. Stability of this sort is trivial to determine in a classic exponential server network [Kle75] but in the context of routing and blocking decision processes is a subject of active research; see, for example Bertsekas et. al. [BGT95] or Dai and Weiss [DW94]. Coffman et. al. study [JGG+95] the asymptotic stability of first and higher moments for a rotating ring in an open-network context; the effort required for this result is impressive.

Queueing Networks with Blocking

Classic queueing networks (those solved by Jackson, Gordon and Newell or BCMP) do not allow for blocking effects. When these classic networks are applied to study the J-Machine in Chapter 3, the blocking and routing effects are being ignored. Others make similar assumptions; see, for example, Harchol and Black's application of queueing theory to study routing on square arrays [HB93].

There is a moderate sized literature on the incorporation of blocking models into queueing networks. Perros [Per94] provides a text for this topic, and Onvural provides a survey [Onv90] for closed networks. One of the items of note in Onvural's survey is a taxonomy of blocking mechanisms; the behavior of a queueing network varies a great deal with the mechanism. The J-Machine's link-level blocking is a simple Blocked-after-Service (BAS) mechanism; Onvural shows this mechanism is strictly free from deadlock if the total number of customers in the network is less than the smallest blocking cycle. On the J-Machine, a large parallel computer, the number of customers (which correspond to messages or tasks) far exceeds the size of the smallest cycle. This is why the J-Machine has mechanisms to detect potential deadlock situations and respond to them (in effect, this detection changes one of the servers in each cycle to be non-BAS and breaks the deadlock).

The techniques for analyzing blocking networks did not prove to be as useful for this thesis as did the Adve and Vernon analysis [AV91]; much of the technique in the literature appears to provide only small equivalences or solutions for small networks. This is not a condemnation of the skill of the researchers producing this literature, more an acknowledgment of the difficulty of the problem and idealism of the goal.

In one paper of note: Akyildiz [Aky88] considers BAS networks which meet the deadlock-freedom criterion (fewer customers than the capacity of the smallest blocking cycle) and notes that the throughput can be estimated for solving specially constructed smaller nonblocking networks which have comparable state spaces (though different structure). This resembles Chapter 3 analysis because both it and the chapter are concerned with state space sizes. However, Akyildiz is concerned with estimating throughput and considers blocking (i.e., queue-filling events) effects only for their brief effect on throughput. In contrast, Chapter 3 is solving for the rate of queue filling events, and is motivated by the extreme cost of these events on the J-Machine.

The result noted in Chapter 3 that all states have equivalent probabilities with a balanced workload resembles one of the working axioms of thermodynamics: that in a physical process all states that are accessible have equal probabilities [Rei65]. Furthermore, the reasoning that the chapter uses to conclude that overflows will be rare – comparing the relative size of state spaces – resembles reasoning in thermodynamics. This connection with thermodynamics breaks down with imbalanced or request/reply workloads because the state probabilities are no longer all equal. Still, the connection with physical reasoning is attractive, if for nothing other than aesthetics. Some literature analyzing blocking networks also uses thermodynamic analogies, for example, the entropy-maximization techniques proposed by Kouvatsos and Denazis [KD93]; also, Harrison and Nguyen [HN93] review Brownian approximations for heavy-load queueing networks. Perhaps some of these methods may apply to the characterization of parallel computers – this is left for others to pursue as future work.

Mean Value Analysis

Adve and Vernon's model of a shared-memory parallel computer with a wormhole-routed mesh network forms the basis of Chapter 5 and Appendix A. Their model differs from the J-Machine because it is for a shared memory architecture, and because their routers are different in structure. Furthermore, the fact that the J-Machine's routers are unfair substantially changes the results. The chapter and appendix largely consist of reimplementation of the model with these changes and then computation of the results; this was a nontrivial task, given the size and intricacy of the equations; it was necessary to examine Adve and Vernon's solver at length to determine the meaning behind their concise published explanation.

Adve and Vernon's model uses a technique called Mean Value Analysis (MVA) (which was invented by Reiser and Lavenberg [RL80]) to solve for the first moments (i.e., mean or average) of queue length, throughput, and waiting time within the network. The beginning of Chapter 5 reviews the fundamental equations within MVA. The classic Reiser and Lavenberg MVA also does not admit blocking; so Adve and Vernon's model incorporate many assumptions about independence to approximate the blocking behavior. With the many assumptions and with the large size of the J-Machine and the very heavy load points chosen, the model does not closely validate with simulated values; it only matches simulation and hardware measurements in a qualitative sense. Investigating the sources of the validation problems and incorporating fixes is likely to be a time consuming task.

Flow and Congestion Control

The challenges of flow and congestion control have been recognized for decades. Flow control refers to control of a source to match (and especially not exceed) the rate of the receiver. Link-level flow control, for example, is flow control over a single link where the receiver indicates when it is ready to accept more data. End-to-end flow control matches a source to a receiver which is several links away.

Congestion control refers to controlling several traffic sources to avoid loss of throughput, latency, and deadlock due to congestion (accumulations of traffic) within a network.

The J-Machine includes link-level flow control, but not end-to-end flow control, nor any congestion control. However, if there's any congestion in the network, it is quickly coupled backward via link-level blocking into the senders. The blocked senders spin and greedily get their packets into the network as soon as they can. One of the reasons that this works and that it's not necessary for the intermediate routers to discard packets when full is that the routing graph is free from deadlock. The volume of congestion forms at the input to the network. The input to the network is near the MDP, so it can stop sending as soon as congestion forms.

One of the most widely deployed congestion control algorithms is Jacobson's algorithm used in the Internet's TCP [JK88]. When the TCP algorithm detects congestion, the window size of the end-to-end connection is halved, and then as further packets are sent down the channel the window size is very slowly increased to probe for any additional available bandwidth. This algorithm is connection-oriented, and that makes it inappropriate for the message-oriented

J-Machine.

This thesis is flow control oriented in the following ways:

- It wants to know the frequency of buffer overflow for large clouds of traffic flowing over processors as in a parallel computer. This is less an effort of proposing ways to control the flow than it is an effort of trying to characterize the behavior of *uncontrolled* random flow. Because the flow is uncontrolled, the analysis in Chapter 3 admits standard queueing theory, but also requires ignoring the low level control (blocking, contention) that occurs in the communication network itself. The analysis in Chapter 3 is only for the uncontrolled flow over processors. In a sense, it is examining the control of flow via the "dispersion" that comes from choosing destinations randomly in the network. The dispersion technique is actually used by the MDP's runtime kernel called COSMOS.

- The analysis is performed to obtain the frequency of overflow because overflow is so terribly costly on the J-Machine. In a standard (Internet-like) communication network, the buffer overflows are relatively less costly, causing some packets to be discarded and later retried.

- The measurements in Chapter 4 show the effectiveness of the chosen J-Machine queue sizes running heavy traffic, and the failure of these queues to be adequate to prevent overflows.

- Chapter 5 studies the effect of the lower-level network link-level flow control on the average queue size on the processors using MVA techniques.

- Chapter 6 shows a way to balance the customers over the queues to prevent overflows, using high-water marks to switch congested nodes to using high-priority output channels. This is a combination load balancing and flow control algorithm.

1.3.3 Summary

This section has outlined the relationship of this thesis to other published research. The thesis touches issues in parallel computer hardware architecture, queueing analysis, and flow control.

1.4 Thesis Outline

The chapters that follow document the results and observations of the course of this research.

- Chapter 2 is a description of the elements of the architecture of the J-Machine and of the MDP which are essential to the thesis argument, focusing on the interface between the processor and the network.

- Chapter 3 is the first of the four core research chapters. It presents the classic queueing analysis for closed traffic running on large machines, and uses the models to relate the rate of overflow to the size of the machine, number of tasks, and degree of imbalance. The chapter documents the threshold behavior between the domains of operation where the degree of imbalance for a system is tolerable and where it becomes intolerable. Also, the chapter points out some of the obstacles to solving a queueing model for the stable request/reply traffic pattern.

- Chapter 4 presents the measurements for traffic running on the J-Machine, detailing the sharp drop in performance at network saturation with a fine-grained workload that is due to queue overflows. Evidence is presented showing that the variance in performance is due to variance the time to cross the queue size threshold. The chapter also shows that there is no avalanche of traffic onto overflowed nodes, that queues overflow in the center of the machine, and that when the machine is operating with long messages with this workload, it never achieves good performance. Finally, the chapter shows that the request/reply workload running on the J-Machine is extremely stable.

- Chapter 5 is the mean value analysis of the J-Machine. It reviews exact and approximate mean value analysis techniques, and extends the Adve and Vernon model (given in entirety in Appendix A) for heavy-load traffic. The chapter develops a new model of the channel for the unfair routers. Finally, it simplifies the mean value model to demonstrate some difference in stability of the snakes model and the request/reply model.

- Chapter 6 shows the behavior of simple load balancing heuristics which could be implemented in hardware. It finds that the "return-to-sender" heuristic (which has been implemented in the Cray T3D parallel computers) is not immediately effective in preventing loss of performance. Much better is a high-water-mark based heuristic which switches processor output traffic to a higher priority when the processor's input queue starts to grow. This hardware heuristic is effective in preventing overflows when the machine is operating at heavy load points with fine-grained traffic.

- Chapter 7 is the conclusion, which reviews my contributions, points out some of their limitations, and offers suggestions for further research.

Chapter 2

J-Machine Architecture

This chapter describes the communication features of the J-Machine and the Message-Driven Processor (MDP). The purpose of this description is to introduce the features that are modeled and examined in subsequent chapters, and to provide the architectural motivation for the models and examinations.[1]

The J-Machine is a parallel computer, built out of MDP chips. Each MDP chip integrates a simple CPU, network interfaces, network routers, external memory interface, and local memory (see Figure 2-1 for a die photograph of the MDP). These chips are full-custom VLSI, and were designed explicitly to make up the J-Machine. With everything integrated on-chip, the MDP is a one-chip building block. As a consequence, the J-Machine consists of little more than MDPs, some DRAM, and wire interconnect.

To work together on an application, the MDPs in the J-Machine must communicate with each other. This communication between MDPs is via the single communication mechanism that the chip provides: asynchronous remote method invocation.[2] That is, an MDP communicates by sending a message that schedules the execution of a method on another MDP. There is no other interprocessor communication mechanism. If other communication abstractions are required, they must be constructed from the remote method invocation. For example, a fetch of a remote memory location can be built with a method that reads the desired location and then sends a second method invocation back to store the data. For another more substantial example, see Wallach's [NWD93] implementation of a barrier with a coordinated butterfly

[1] For a comprehensive description of the MDP, see the Programmer's Manual by Noakes [Noa91]. A good general overview of the MDP is in [DFK+92]. The early rationale for the architecture of the J-Machine is in Dally's thesis [Dal87].

[2] The term "method" is interchangeable with "procedure."

of messages over the processors.

The J-Machine's sole reliance on the asynchronous remote method invocation for communication distinguishes it from other parallel computers. However, this mechanism corresponds directly to communication mechanisms in other parallel computer architectures, though at different levels of abstraction. For example, the Active Messages [vECGS92] software library running on a CM-5 provides a higher-level abstraction for remote method invocation. As another example, the protocol underneath Alewife's shared memory communicates with interprocessor messages which invoke hardware methods – a correspondence to a lower level of abstraction [KJA+93].[3] This correspondence between the communication method in the MDP and in other architectures imparts generality to the results in this thesis.

Memory buffer capacity and wire bandwidth limits constrain on the pattern of communication in any parallel computer.[4] Many times, these constraints are managed by blocking: when a buffer fills or a wire cannot handle the requested bandwidth, incoming subsequent messages are blocked. There are other ways of handling capacity and bandwidth constraints; adaptive routing schemes, for example, misroute messages to avoid the constraints. But blocking is a good solution when the blockage lasts only for a brief time, and it is attractive because it is simple to implement. The danger of blocking is that the network may deadlock. Care must be taken to avoid this. Care must also be taken to manage the constraints in a way to get good performance, for example by scheduling the computation and designing the network to avoid sending all messages over the same link. Management of these constraints is a central problem in the design of parallel computers.

Other architectures solve this problem in different ways. In Active Messages on the CM-5, a strict partition of the communication flow's requests and replies onto independent networks prevents deadlocks.[5] In Alewife, a deadlock is prevented with a deadman timer that invokes a processor interrupt when the machine seems to be making no progress. This interrupt is costly, so implicit in the Alewife design there is an assumption that these timeouts will be rare.

The J-Machine's implementation of its single communication mechanism also embodies a

[3] Alewife can also perform arbitrary interprocessor message sends.

[4] And, in fact, in any communication network, such as the Internet.

[5] Also, as the results in this thesis show, a request/reply communication pattern appears to be much more stable than more general communication patterns.

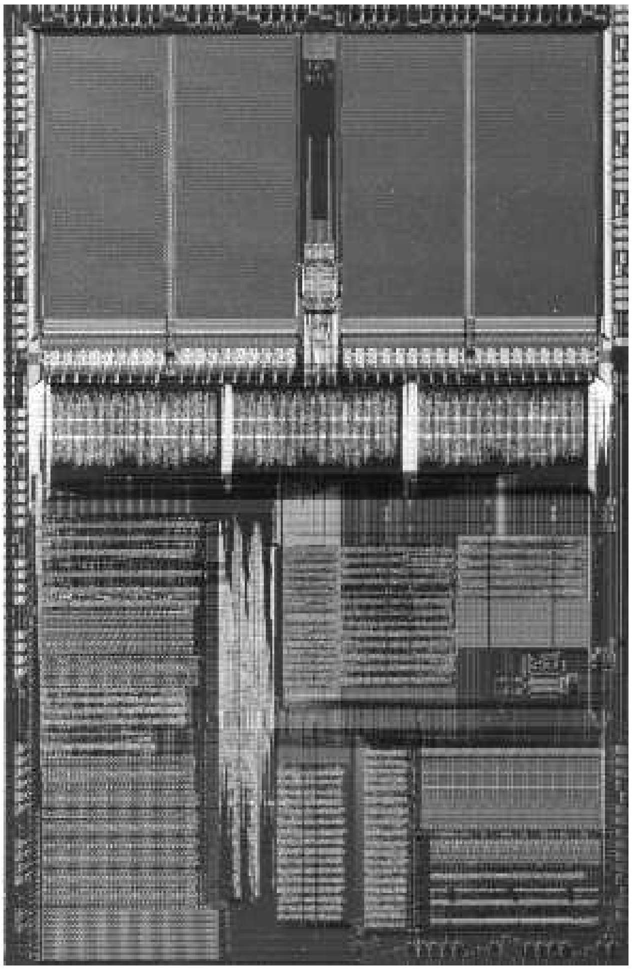


Figure 2-1: Photograph of the MDP chip. Local memory banks are in the upper portion of the chip. The three routers are below. The rest of the chip is the CPU, network interfaces, and addressing logic. Layout was produced with a structured custom methodology and standard cell place and route.

similar assumption. In the case of the J-Machine, it is the assumption that queue overflows, rather than deadman timeouts, will be rare. Exactly how the implementation does this is the subject of this chapter, and the consequence of this assumption is the subject of the thesis.

The assumption is embodied in the J-Machine implementation as follows:

- The communication mechanisms are structured in a way such that they could potentially deadlock. This is an inescapable consequence of choosing a non-adaptive network with fixed capacity buffers that block after service, where the total volume of outstanding traffic can exceed the capacity of cycles in the blocking-dependency graph [Onv90].
- As a result, it is necessary to implement an "escape." Instead of a deadman timer as in Alewife, the J-Machine implements the escape with a *queue overflow trap*: when the finite size input buffer to the processor fills, the processor traps and empties the buffer. The topology of the routers is such that all cyclic dependencies in the network include at least one of these buffers, thereby leaving the machine the machine free from the possibility of deadlock.
- However, these overflow traps are very slow. When they occur, they divert the processor for hundreds of cycles. Furthermore, the traps can lead to a complete traffic jam in the communication network which stops every other processor in the machine. Because they are so expensive, these traps must not occur if the machine is to achieve any level of performance.

How rare are these overflows? Chapter 4 demonstrates empirically, by running synthetic workloads, that they are not rare.

The rest of this chapter is structured as follows. Section 2.1 describes the MDP and its communication implementation. Section 2.2 describes how the machine can deadlock, and talks about the consumption assumption. Section 2.3 describes the queue overflow trap handler, and explains why it is so slow. Section 2.4 concludes the chapter.

2.1 MDP and Communication Implementation

Invocation of a remote method on another processor (the J-Machine's sole communication mechanism) is a MDP hardware function provided by several components: the CPU, the communication network, and in the processor-network interfaces. This section describes these components.

2.1.1 CPU

The CPU in the MDP is simple: it consists of a small set of arithmetic and address registers, ALUs for manipulating the registers, and some control logic. A simple instruction set drives the small CPU datapath. The point of making the processor simple has to do with trying to make its construction require less silicon area, and thereby to be consistent with the jellybean philosophy: make the processors inexpensive and get performance by using many of them.

Support for Dynamic Workload

A complete enumeration of the MDP's instruction set can be found in the Programmer's Manual [Noa91]. Here, it is appropriate to point out a few of the distinctive features of the instruction set:

- The MDP has extensive support for type tags. Every word of data in the machine includes a four-bit type tag – the word size of the MDP is 36 bits. Among the benefits of type tags is support for "object oriented" operator overloading; simple primitive methods like ADD can be compiled into higher-level methods as opcodes, and thus avoid the overhead of a complete method invocation for the simple primitive. Yet the method can be used for non-integer types: the opcode, when it encounters a non-integer type tag, traps to the runtime system which calls the appropriate method.

- Type tags also support "futures" which are place holders for results which have not yet arrived. When an opcode accesses a future, the thread suspends, adding itself to a list of tasks to be activated when the result arrives.

- The MDP has special opcodes to manipulate memory table which act like the directory of translation look-aside buffers (TLBs). This can aid in method or object lookup.

The details of these features (available in Noakes [Noa91]) are not of interest here. What is of interest is that these features reflect the intention of the J-Machine architects to design a machine to run an unstructured, object oriented workload that unfolds dynamically at run time. For example, support for tags to overload operators reflects anticipation of a workload with indeterminate handler latency, and futures support indeterminate remote method latencies. Translation support anticipates a workload where objects can relocate at run time.

This intention is also reflected in the communication features of the CPU:

- Computation is message-driven; that is, the processor starts executing instructions only when a message arrives. Instruction execution stops when the method suspends.

- The processor has a set of SEND instructions that inject data into the network input interface, directly from the CPU's register file. SEND instructions allow for the possibility that the network will be unable to accept the message (due to network blocking and capacity constraints); when this possibility happens, the SEND instruction faults (a SEND fault) to a handler that can back up the IP and retry the SEND instruction.

- The processor supports an EARLY fault: processing of a message can begin even before that message has completely arrived into the message buffer. The processor's addressing unit detects references to parts of a message which have not yet arrived, and traps with an EARLY fault when they occur.

The inclusion of these three communication features anticipated a varying workload that unfolds on-line, or at runtime. In the J-Machine, computation is not pre-scheduled; it occurs upon demand. The hardware is greedy: hardware circuits alter the order and rate of progress (the schedule) of the computation's tasks on-line (mostly via blocking) so that the order obeys the machine capacity and bandwidth constraints.

The network can exert backpressure to the CPU (in the form of SEND faults); this allows for the CPU to attempt greedy consumption of network capacity based on runtime-determined availability. Finally, the CPU can speculatively read from arriving messages. This allows for the case when those speculations fail due to network constraints.

Reliance on Only One Hardware Priority

The CPU includes three hardware threads, with two ("Priority 0" and "Priority 1") that correspond to the network priorities 0 and 1 and a third that runs as a background process when no messages are present in the message queues. Messages that arrive from the Priority 1 network are handled by the Priority 1 thread; messages that arrive from the Priority 0 network are handled by the Priority 0 thread. A hardware thread is a distinct set of data and control registers including instruction pointer (IP). The distinct hardware register set for each thread speeds thread-switching which occurs when a Priority 1 message preempts the processor from serving a Priority 0 message. This is the extent of preemption on the MDP - within a thread, execution is never preempted by a message at the same priority.

Priority 1 was included to allow the MDP to serve high priority messages even when the MDP is blocked or busy with regular messages. This is mainly for queue overflow handling, described below in Section 2.3. However, despite its intended usefulness, Priority 1 was almost completely unused by J-Machine programmers. The runtime kernel COSMOS and Concurrent Smalltalk (CST) compiler did not use Priority 1. All system traffic and application traffic went on Priority 0. This is significant because, as Section 2.2 will explain later, the use of a single priority raises the problem of potential blocking-dependency deadlock.

As mentioned, within a priority, threads are atomic. The processor serves messages in its queue in FIFO order. The first word of every message that arrives contains an instruction pointer to the start of a handler routine; for example, this handler might locate and dispatch to to an application method.[6] The last instruction in every method is a SUSPEND; when executed, it deallocates the storage for the message from the queue and allows the hardware to dispatch to the next message's handler.

[6] For more details, see Horwat's thesis [Hor89] which describes the CST implementation.

2.1.2 Network

The J-Machine communication network is implemented with routing logic blocks ("the routers") which are located on the MDP chip and the wires that connect them. The network has the following characteristics:

- Three-dimensional mesh topology.
- Deterministic e-cube routing.
- Wormhole channel allocation.

The next sections explain these characteristics.[7]

Three Dimensional Mesh Topology

Each MDP (excepting the surface processors) in the J-Machine is wired to and communicates with six neighbors: two in the X direction (horizontally on the printed-circuit board), two in the Y direction (vertically on the board) and two in the Z direction (vertically, between boards, to the processors immediately above and below). This is a three-dimensional topology, and has the attraction that it is easily packaged, because the world is three-dimensional. Figure 2-2 shows the corner of the J-Machine's packaging. Processors that are not on the sides of the array each talk to two processors in each dimension X, Y, and Z.

The J-Machine is topologically a "mesh" rather than a torus. This means that there are no wrap-around connections in the J-Machine's network. The processors that are on the sides of the mesh have "dangling" links; that is, they are not connected to other processors (though they may be connected to I/O devices.)

A mesh topology is inhomogeneous: it is possible that any two processor nodes within the mesh may "see" a different environment looking into the interconnection graph. For example, a processor node at the edge of a mesh "sees" that some of its links do not connect to other processors, while a processor node at the center of a mesh "sees" that all of its links talk to neighboring processor nodes. In contrast, when each processor node in a torus network

[7] For more details about the J-Machine network, see Nuth's paper [ND92].

"looks" into the network, it sees a view that is identical to the view seen by every other processor node in that torus network – the torus is a homogeneous network.

The inhomogeneity of the mesh network means that workloads that are balanced with respect to the J-Machine's processors are imbalanced with respect to the communication network. In a mesh, with uniform applied traffic (i.e., each processor sending traffic at the same rate and uniformly to every other processor), the wires in the center of the mesh are more heavily loaded than are the wires at the edges. This has observable consequences on the J-Machine: Chapter 5 shows that the mesh topology strongly determines the observed overflow behavior.[8]

The degree of inhomogeneity in a mesh can be determined simply. Consider a one-dimensional mesh that is X processors in a row. There are only links between adjacent processors in the row, so a message from processor i to processor j may have to travel over several links to get to the destination. Assuming that X is even and that each processor is sending at rate λ uniformly distributed over the all the processors, then the total load on the center link is $\lambda X^2/2$, and the total load on the two end links is $2\lambda(X-1)$. After dividing, one can see that the traffic load on the center link is $\approx X/4$ times greater than the traffic load on the edge link. This ratio increases with X: as the length of the mesh becomes greater, the degree of imbalance becomes larger. Calculating the degree of imbalance in a three-dimensional mesh is a simple extension of this. In a three dimensional mesh with e-cube routing, the load on each dimension distributes as if each were an isolated one-dimensional network, where X corresponds to the length of that dimension. In each dimension of the 512-processor J-Machine mesh, $X = 8$ and the center link is twice as heavily loaded as the end links.

Were the links to have unbounded transmission capacity, this would not be a problem. It is because the links have fixed capacity that the uneven applied load becomes manifest as a workload imbalance.

With inhomogeneity being inherently problematic, why was the J-Machine built as a mesh? The reason is that the mesh topology provides "dangling" network connections on the edge of

[8] In his work that led to the J-Machine's architecture, Dally contributed a now well-known solution for building networks with a torus topology [DS87]. However, the torus topology is not used in the J-Machine: the J-Machine is a mesh because the mesh provides the edge-channels for connecting I/O devices. A simple torus is homogeneous, and although the network design contributed by Dally in [DS87] breaks this homogeneity (a result noted by Adve and Vernon [AV94]), the deadlock-broken inhomogeneous torus would still have imparted less imbalance than the mesh.

the cube for connecting I/O devices [Nut]. Had the machine been built as a torus, with wrap-around connections, adding I/O devices to the machine would have required splicing special channels in. Furthermore, making the network toroidal would probably have necessitated adding additional deadlock-breaking virtual channels to the routers (Dally's result [DS87]), doubling their silicon area. Because the routers take up a substantial portion of the MDP area, doubling their size would have significantly increased the cost of the MDP chip.

Deterministically E-Cube Routed

Although each MDP chip in the J-Machine is not directly connect to every other MDP, each processor can send a message to every other. The MDP chip's routers forward messages from chip to chip along the links of the mesh to allow messages to reach their destinations. The route that the messages follow is deterministic, so that the path taken by every message from some processor A to some other processor B is always the same. This route is an "E-cube" route: a message from a processor at coordinates (x_a, y_a, z_a) to a processor at coordinates (x_b, y_b, z_b) in the cube first travels along the links of the X dimension to position (x_b, y_a, z_a), then follows links in the Y direction to coordinates (x_b, y_b, z_a), and then follows the links in the Z direction to the destination.[9]

The mesh links exert backpressure when there is contention for the immediately downstream router message storage or network wires. A message that encounters backpressure becomes blocked, and it waits for the downstream congestion to clear before it moves forward again. In the process, the blocked message may exert backpressure to other upstream messages.

Blocking of this sort, when used in a network with arbitrary networks or routing paths, can potentially deadlock: a cycle of unresolved blocking relationships can form, and with fixed storage, no message will be able to move. However, with the mesh topology and restricting the routing to e-cube, there are no cycles in the routing dependency graph. Because of this, e-cube routing within a mesh network is free from the possibility of deadlock.

[9]An unrelated point: although the design choice is arbitrary for the order in which the three dimensions are routed, it is not arbitrary for I/O, and we got it wrong. The J-Machine's packaging allows for I/O devices to be connected to the network only on X and Y channels. This was a blunder: when addressing I/O devices at a mesh coordinate with some z_1, an MDP with Z coordinate $z_2 \neq z_1$ must send through a proxy processor with coordinate z_1; this because otherwise the message would be routed X first, Y next, and Z last and thereby "fall off the edge of the board" before going to the correct Z coordinate.

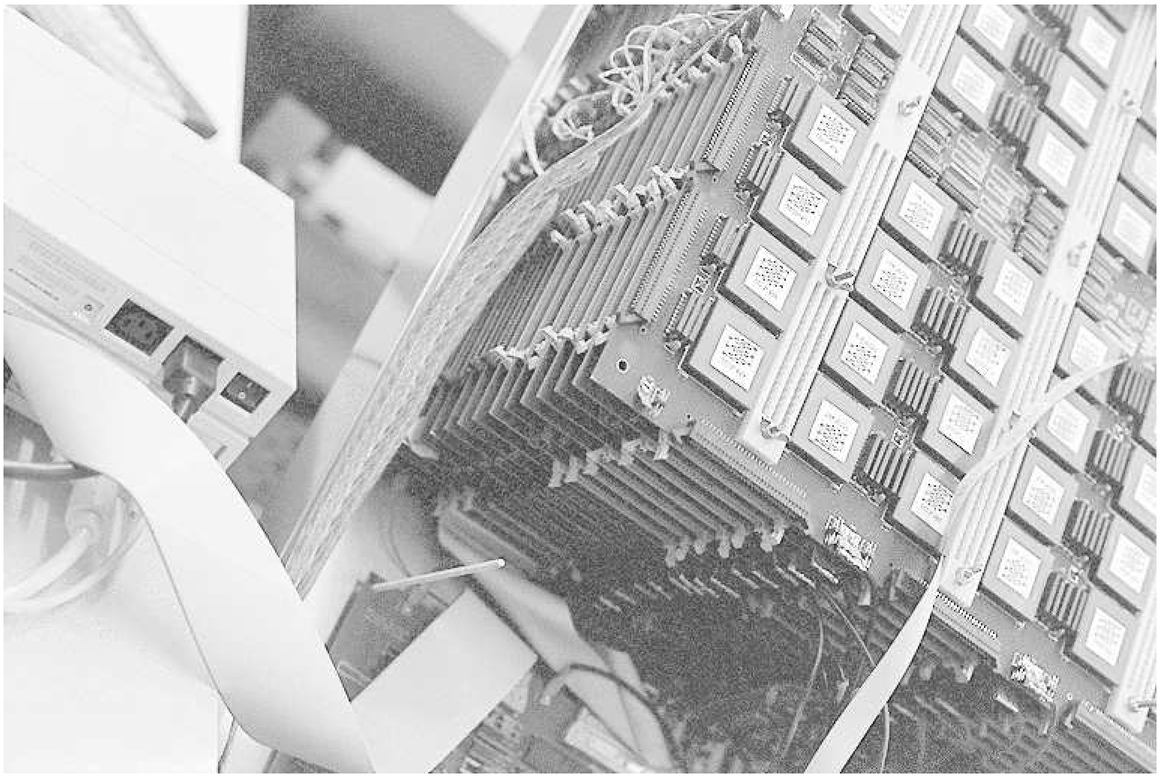


Figure 2-2: Detail photo of the J-Machine with the cover removed which shows some of the individual processors arranged in an array on the top board. Three DRAM chips are next to each processor and are the external local memories; otherwise all of the functional modules of each MDP – the processor, internal memory, queues, routers, control – are on-chip. Processors are wired to their nearest neighbor in the X and Y dimensions through the printed circuit board. The connection in the Z dimension is through elastomeric connectors which are compressed by tightening the pictured wing-nuts onto the shafts which skewer the stack of boards. The unwired network connections on the edge of the cube go to the connectors on the edges of the boards; a ribbon cable from one of these connectors to the host interface card appears at the bottom of the stack.

Note that this guarantee of deadlock freedom holds only with the network in isolation, where messages at the output of the network are always immediately extracted.

Wormhole Channel Allocation

Messages in the communication network are transmitted in a wormhole fashion;[10] that is, the messages are pipelined with flits[11] being contiguous[12] through the network routers to their destinations. Pipelining reduces the latency of message delivery. Keeping the sequential flits of the messages contiguous in the channels simplifies the control logic in the network switches; otherwise the switches would have to track multiple in-progress messages. To keep the flits contiguous, the first flit of the message (the "head") opens a virtual circuit to the destination. As the tail flit follows, it closes the virtual circuit behind itself. If a switch is in use with an open virtual circuit (because a message is passing by), other incoming messages are blocked from using that switch, and so the router switch is passing only one message at a time.

The wormhole network imposes no restrictions on the length of messages. However, in the J-Machine, messages must be smaller than the size of the MDP's message buffer (512 words).

Because e-cube routing is deadlock-free, the amount of storage in the J-Machine routers can be, and is, small: the router queues are only two flits long.

The router in the J-Machine is designed so that the head and subsequent flits can make progress through the switches of the J-Machine making one step per clock cycle, where a "step" potentially includes both the time through a switch and a chip-to-chip wire crossing. Dropping between dimensions in the tiered router structure also takes one clock cycle.

[10]See [DS87] for a more detailed description of wormhole networks.

[11]The word *flit* means flow control unit. It is the portion of the message that is transmitted and flow controlled as a unit, or in other words, the unit of pipelining.

[12]Contiguous, but not necessarily adjacent; for example, the head of a message can pass through a congestion point and race ahead of the subsequent flits.

Tiered Structure

The MDP's routers form a tiered structure. See Figure 2-3. Each MDP contains three routers, one for each dimension X, Y, and Z. Messages enter the router structure from the top, and travel along the X dimension routers until they reach the correct position. Then they drop into the routers for the Y dimension, travel along the Y channels until they reach the correct Y position, and then they drop into the Z channels, and so on. Following the e-cube routing, messages never need to go up to a higher dimension. Building the router in this manner simplifies the multiplexors in the router switches; with e-cube routing, each multiplexor has only two inputs.

Wire Arbitration

To allow the bus-width between the MDP chips to be larger, the wires are shared between the two opposing directions. The same wires that are used to forward messages from an MDP and a neighbor in the X dimension are the same wires used to forward opposite-direction messages from that neighbor to that MDP. This is not simultaneous bidirectional electrical signaling; the wires are allocated between the two directions as needed. If there is contention for the wires, the ownership is switched between the two directions on a cycle by cycle basis.

Multiple Priorities

Although the use of extra virtual channels to break deadlock in interconnection networks was popularized in the context of wormhole networks [DS87], the MDP routers do not have multiple virtual channels to break deadlock. Extra virtual channels are unnecessary because the e-cube routing on the J-Machine mesh is, by itself, deadlock-free.

However, the J-Machine does have two independent virtual networks designated "Priority 0" and "Priority 1." The networks are "virtual networks" because messages in the virtual networks make progress independently even though the networks share the physical wires of the machine. To decouple progress of any message on Priority 0 from any message on Priority 1, the two priorities have separate switching, separate control, and separate buffering. When there is contention between the two priorities for a wire, ownership is allocated strictly to

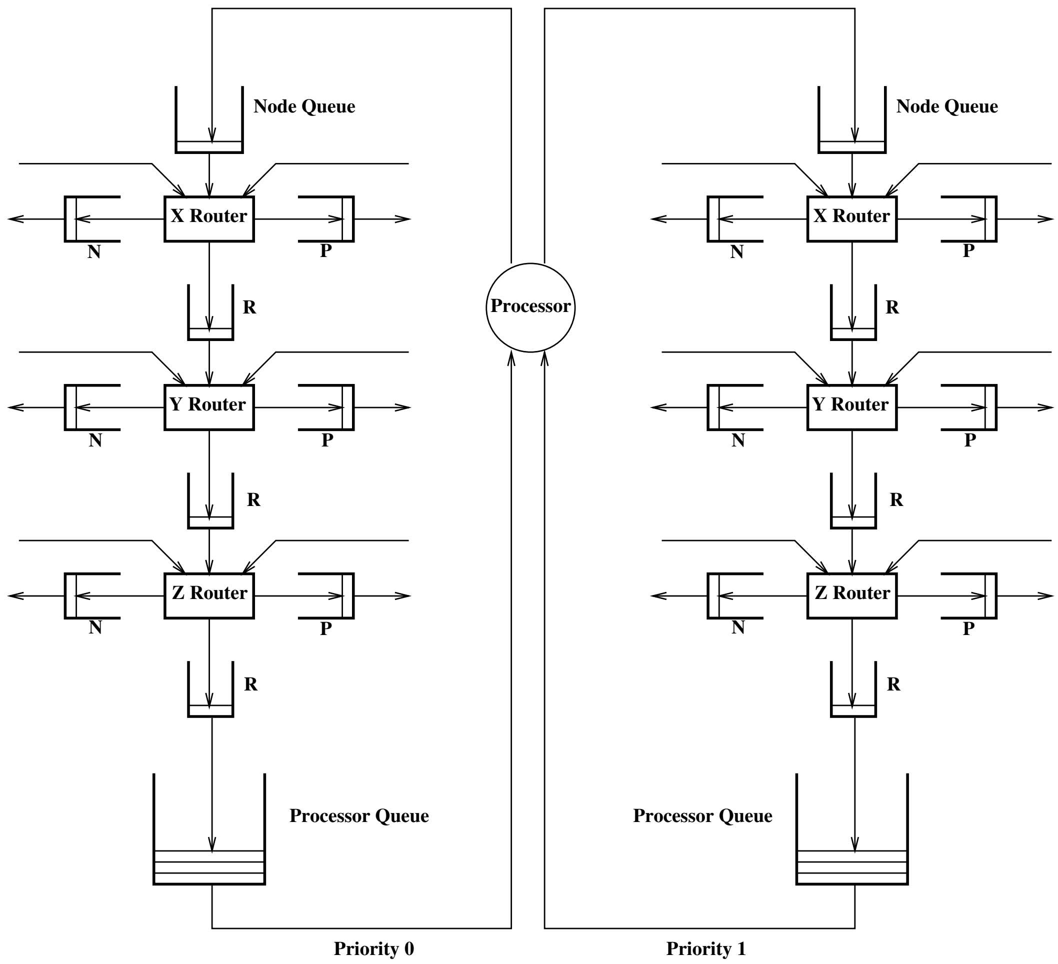


Figure 2-3: Message datapath in each MDP. The Priority 0 network is on the left and Priority 1 is on the right. The routers are tiered; that is, message drop from X to Y to Z routers as they travel to their destination. The "dangling links" go to the neighbor MDPs. For example, the outgoing link for Priority 0 labeled "N" in the X level goes to the incoming link labeled "P" on the neighboring MDP in the negative X direction. The incoming link labeled "N" comes from that neighbor's outgoing "P" link, and so on.

Priority 1.

The reason for adding Priority 1 to the MDP was to allow software to make communication progress in emergency or special situations – particularly such as a queue overflow – and also to allow for the kernel to send higher-priority traffic independently from the applications. As mentioned earlier and in fact, though, COSMOS never used Priority 1 on the J-Machine, nor did any applications.

One result of this thesis research is to show (Chapter 6), that using this network with a high water mark threshold is a heuristic that allows the J-Machine to actively balance the processor queues and thereby help avoid queue overflows. Unfortunately, making this change to the J-Machine would be impractical; the logic is hard-wired into the chip.

2.1.3 Processor-Network Interfaces

Receiving Messages

The network interface hardware automatically places message that arrive from the network into the processor's message buffer. See Figure 2-3. This buffer is a fixed-size memory including head and tail registers to make it a circularly allocated FIFO queue.

This buffer, when it overflows on an MDP, imparts a substantial performance penalty on the J-Machine; this makes it the focus of much of the research here.

Compared to the two-flit capacity of the buffers in the wormhole routers, the capacity of this buffer is large. It can be configured as any power of two in length up to 512 words (1024 flits) long. Usually, it is configured to the maximum size.

The purpose of the message buffer is to guarantee the consumption assumption (see Section 2.2.1), and also for the purpose of improving performance. With the buffer not full, the network interface hardware can quickly move the newly-arrived message out of the network and reduce the amount of blocking in the network. Having hardware manage the extraction of the message from the network makes this a fast process.

What happens when the message queue fills? See Section 2.3 which describes the queue overflow trap.

The hardware maintains head and tail pointers into the buffer, which are accessible in a processor register. In addition, the instruction set facilitates access into the buffer by providing a memory addressing mode for fixed-offset references from the head of the queue. Using this addressing mode, tasks initiated by messages can quickly fetch method-call arguments appearing at fixed positions in a message.

The CPU's primary task is to serve this buffer, and it does so strictly in FIFO order. When the queue is empty, the processor is idle. As soon as the first few words of the message arrive to an empty queue, the processor initiates an operation whereby it jumps to the instruction pointer (IP) which is always stored in the first word of the message.

Sending Messages

Hardware threads inject message data from the register file into the network by executing a succession of SEND instructions. The message is implicitly marshalled and formatted in this process: the arguments are read from the specified registers, and placed in the message in order.

There are 8 different SEND opcodes available, distinguished by three options: the priority level of the new message, the number words that the instruction reads from the register file and injects (one or two words can be injected), and whether the words injected are the last ones in the message. For example, a SEND2E0 instruction reads two words from the register file, and injects them at priority 0 with the last word marked as the end of the message. A SEND1 message injects one word into the currently outgoing priority 1 message which is not the tail.

The interface defined by the SEND instruction allows messages to be of any length. The first word sent is the destination processor number; this travels with the message to control the routers, but is stripped from the actual message delivered into the destination FIFO. The second word is interpreted by the destination dispatch hardware as the dispatch IP. Any number of words can follow the first two SENDs, with the last being marked by a SENDE instruction. The minimum message length is 2 words long (the address of the destination followed by a dispatch handler), which can be constructed by a single SEND instruction. Messages with payloads take a couple more instructions to inject.

Note that there is no hardware management of the recipient's buffer space at this level. The hardware just jams the traffic into the network.

The SEND interface is *modal.* What this means is that the semantics of the SEND instruction depend upon the state of the MDP. If no message is in the process of being injected, the SEND is interpreted as starting a new outgoing message, and specifies the destination address for that new message. If a message is in progress, the SEND instruction simply supplies body or payload flits.

For modal interfaces, it is good design practice to include a polling mechanism to determine the mode of the interface; and also it is good practice to include a reset mechanism. Neither mechanism is available on the MDP. As a practical consequence, the processor is committed to completing the injection of the entire message and must simply block if any backpressure is encountered after the first word is sent.

This is unfortunate, because the effective commit point happens before the processor knows whether the network can completely accept the message. Any of the individual send instructions injecting the message can fail with a SEND fault that indicates that the network is blocked. It is not legal to terminate the message after the first word is sent – it is possible that the destination has already received and started execution of the specified message.

The network uses SEND faults to couple network backpressure into the execution of the processor; but still, the network is obliged to eventually accept the message. The processor, because it is committed to completing the outgoing message, must retry the SEND instruction that faulted. As a result, the only action that the SEND fault hander can take is to back up the instruction pointer, and jump back to the thread to retry the SEND instruction. The handler cannot context-switch threads when the network output is blocked, and cannot allow the processor to make useful progress on other threads because those threads will not be able to detect the in-progress message.[13]

[13]One thing the SEND fault handler might do is spin with increasing amounts of time in an (exponentially?) increasing backoff to try to scale back the load on the almost-full network. The SEND fault handler also *must* check for and handle a queue overflow, because the synchronous SEND fault (incorrectly) has higher priority than the asynchronous queue overflow, and will otherwise mask the need to take a queue overflow. It would be legal to suspend the thread if it were the first word injected that caused the SEND fault; conventions on the use of send could be established which have the thread mark signal this opportunity to the SEND fault handler. However, these conventions would not help in the case of mid-injection SEND faults.

The MDP's SEND interface is different from the interface in other parallel computers, and the complexities associated with the modal interface, send commit point, and non-atomicity are inconvenient.[14] Why then was it designed this way? The answer is that the interface was shaped by the wormhole routing algorithm used by the MDP implementation to deliver messages. By accepting the words of the message from the processor as they are ready, the network injection unit and all of the intermediate routers between the source and destination can deliver the first part of the message before the processor has even completed injection (though in the network input logic, there is a small 8-word speed matching buffer). When there are no complications, this reduces the effective latency of message operations. This SEND interface limits the amount of memory copying: words go directly from the register file to the communication network. Furthermore, this interface supports arbitrary-length messages. However, the loss of an atomicity and clean backpressure from the network has severe consequences. The processor is forced to block on a message SEND, and this leads to the following circular dependencies.

2.2 Deadlock Possibility

2.2.1 The Consumption Assumption

The foundation for the guarantee that the communication network always makes progress delivering messages is the *consumption assumption*: when a message arrives at the output of the network for the processor, the processor (or its message buffer) *always* immediately consumes it.

The consumption assumption implies that the communication network queues at the network's output get emptied. By induction, because the communication network's blocking dependencies are acyclic, this means that all of the network queues are eventually emptied.[15]

[14] In contrast, on the CM-5, the entire message is constructed before it is dispatched into the network. The dispatch operation is atomic; it fails cleanly if the network declines to accept the message. This loses the ability to for the network and destination to get started on the message as it is being constructed (unless the network implements a feature to cancel an in-progress message). And the simple interface on the CM-5 restricts the length of messages to a fixed size. The CM-5 design limits the consumption of network resources and blocking by sending the message in a "burst".

[15] This inductive step requires that the routers be fair. On the J-Machine, they are not. See Chapter 5 for more about unfair routers.

By this reasoning, the conclusion is that all of the messages in the communication network are guaranteed to always make progress; that is, the communication network will not deadlock. This is a correctness guarantee.

Were the processor not to consume arriving messages, link-level backpressure in the network would form a tree of contention back from that processor, eventually blocking network input channels and preventing progress.

The consumption assumption is also a foundation for a performance guarantee. Performance comes from the assumption that every arriving message will be *immediately* consumed. Immediate consumption reduces the message's occupancy and blocking in the network channels; this increases network throughput and reduces network latency.

2.2.2 Processor's Message Queues Help with Consumption Assumption

How does the CPU of the MDP guarantee the consumption assumption to the communication network? With message queues: each of the two logical communication networks (Priority 0 and Priority 1) is served at each MDP by a large message buffer. When the buffer has room, messages that have arrived at a processor can be immediately moved out of the network. This message buffer is relatively large - 1024 flits or 512 words - compared to the size of the buffering within the communication network itself. Those communication network buffers each hold only two flits.

However, using large message buffers to guarantee the consumption is only a "first line of defense." Because these message queues are fixed in size, they can fill. When they do, they block the network and violate the consumption assumption. Further lines of defense are required.

2.2.3 Processor Progress Determines Message Queue Progress

Progress in the processor's message queues would be guaranteed if the processor always made progress serving its messages. Unfortunately, processor progress is not guaranteed by the architecture. It is assumed to be true. This assumption is based on the fine-grained philosophy (and is implemented in CST): that methods are short and take only 10-100 cycles to execute. Long, non-terminating methods cannot be used on the J-Machine. Methods must be short. As long as this is the case, the processor works its way through the messages in the queue, serving them, and making progress.

However, there is a larger potential problem: the processor can block, and this blocking is caused by the inability of the processor to inject new messages into the network.

2.2.4 A Cyclic Blocking Dependency?

Recall that when a processor wants to send a message, it uses a SEND instruction to copy data from the register file into the network input. This is OK, as long as the network accepts the data. But the network may decline to accept the data because it is full and blocked. If the network declines to accept the data for the message, it causes the processor to take a SEND fault.

The limitations imposed by the modal interface for SEND restrict the processor's behavior at a SEND fault: all that an MDP can do in response to a SEND fault is back up the IP and retry the instruction. Although the CPU is executing instructions, it is blocked from making progress on the method that it is executing.

Note the cyclic dependency (see Figure 2-4):

- Progress in the network requires progress in the message queue.
- Progress in the message queue requires progress by the processor.
- Progress by the processor requires progress in the network.

Does this mean that the J-Machine can deadlock? If it weren't for the queue overflow trap (which occurs when the message queue fills) the answer would be yes.

Note that the cyclic dependency is related to the use of a single network for communication traffic. The J-Machine has two logical communication networks (Priority 0 and Priority 1) but only one is used in practice.

Some other architectures guarantee progress by using two logical networks. For example the CM-5 running Active Messages guarantees progress by requiring that the application partition its messages into two classes: requests and replies. Requests are mapped onto one logical network (the request network) and replies are mapped onto the return network. Furthermore, the methods initiated by request messages can only send reply messages, and methods initiated by reply messages can send no additional messages. As a consequence, the progress of the request network depends upon network progress - but there is no cyclic dependency because it depends on a separate and distinct logical network (the reply network). Progress on the reply network does not depend on either network, because the reply-initiated methods are not permitted to send new messages.

The architecture of the J-Machine is much less restrictive than Active Messages - there is no explicit architectural restriction partitioning the traffic into two networks. Nor is there any restriction or warning for programmers about avoiding this danger. Instead, there is an assumption that things will be "all right" and the programmer can let methods fire messages into the single network as they wish. The lack of restriction comes at a cost, and that cost is that potential deadlock situations might occur, and these situations get resolved with a queue overflow trap.

Queue overflow traps are the second line of defense. When the message queue fills, queue overflow traps prevent deadlock. Incoming message traffic is disabled, and the overflow trap handler is required to clear out the queues to make room for additional messages to enter the processor. Thus the deadlock is prevented.

2.3 Queue Overflow Handler

The designers of the MDP planned two possible implementations for the queue overflow handler. This handler has to move the contents of the queue memory into some other memory on the system, and there are two places to find that memory: on-node and off-node. Moving the queue contents to on-node memory requires only memory copying. Sending queue content off-node (i.e., to another MDP) requires communication, which might be impossible, were it not for the priority 1 network, because handling the priority 0 queue overflow potentially (and almost always) blocks the entire priority 0 network. This is the

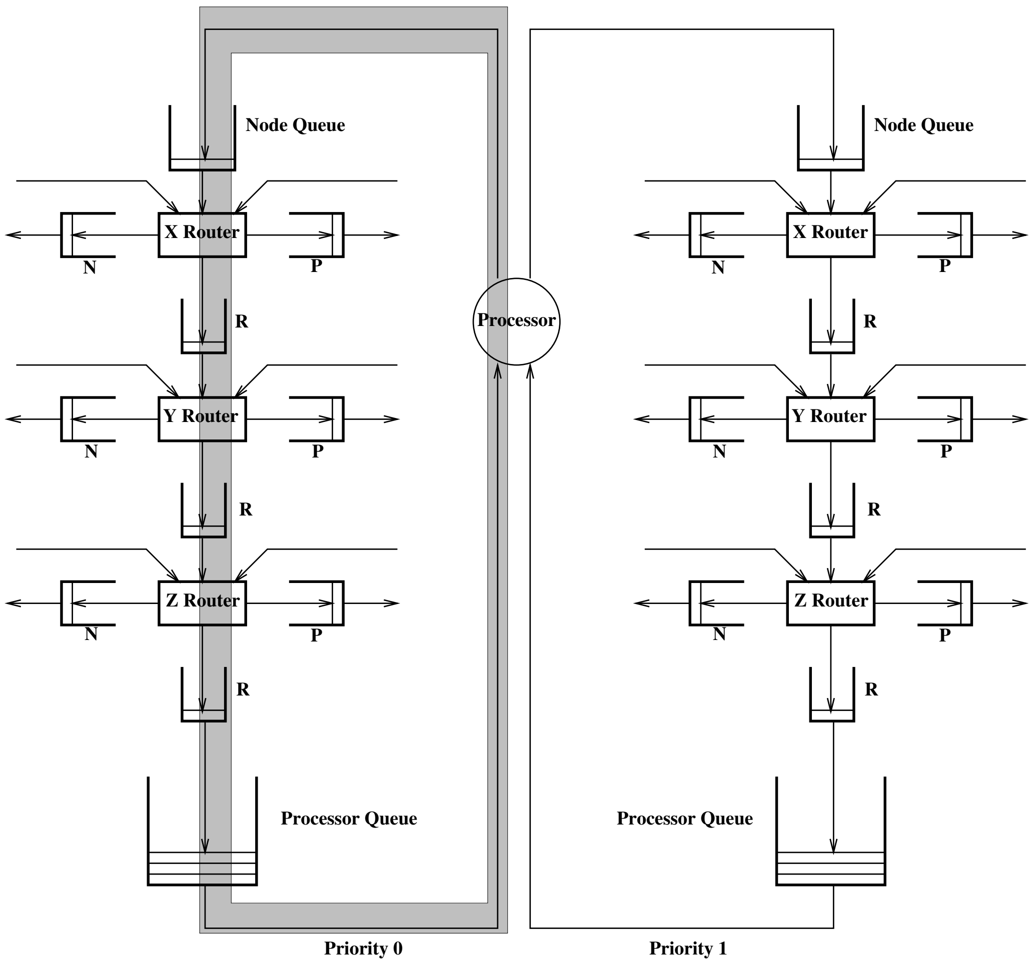


Figure 2-4: Finite capacity in all the message queues and within the routers, and the fact that processors block if they cannot send messages on their output, would lead to deadlock were it not for queue overflow traps. If all of the storage in the network along the shaded path fills, messages along the loop cannot make progress.

principal reason why the priority 1 network was included in the Message Drive Processor: to handle the emergency situation of a queue overflow.

It was originally intended that the response would be layered: initial overflows would go to on-node memory, with the use of the priority 1 network to send off-node if the on-node memory filled. This was never implemented. The COSMOS queue overflow handler only overflows to local memory, and not to other nodes on the priority 1 network. This happened for three reasons. First, the on-node (external) DRAM has lots of space for emptying the queue. Second, sending the queue content off-node would potentially not preserve message order. Third, late-discovered hardware bugs accidentally coupled progress of the priority 1 network to priority 0.[16]

Without the escape to off-chip processors, the remaining queue overflow handler (which only copies to local memory) is slow, for several reasons:

- All of the other MDP communication facilities, meaning the wormhole communication network routers and buffers, preserve message order, so the overflow handler is designed to do so as well. Doing this requires more copying than a non-order-preserving handler would require.

- The overflow buffer and overflow pointers are kept in the local DRAM memory that is external to the MDP chip. Even though it is local to the processor, access to this memory is slow, taking 5 cycles per load or store operation, mostly because package pin limitations on the MDP narrowed the DRAM interface.

- The action of copying the queue to memory is more complicated than simple loads and stores. The copying must track the beginnings and ends of messages. With a limited number of general CPU registers, tracking adds significantly to the number of instructions required to copy each word from the queue to the overflow buffer.

In other words, the loop copying queue contents to the large external memory is more complicated and slower than a fast DMA or something similar.

Figures 2-5 through 2-8 illustrate an example of the process by which the overflow trap handler empties the queue. The process is simple, but worth illustrating to explain why it is

[16]It Is possible, but inconvenient, to work around this bug.

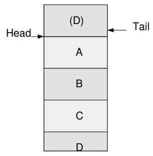


Figure 2-5: Example of queue overflow handling. The queue shown is full, since the tail has reached the head.

so slow, because this explains the poor performance of the J-Machine when queue overflows occur. Figure 2-5 shows a full queue, with 4 messages in it labeled A, B, C and D. D wraps around the circularly managed memory. The queue is full because the tail has reached the head of the queue.

The trap handler copies the contents of the queue into the overflow buffer. Message A, which is at the head of the queue, is not copied because the handler that Message A invoked was the one interrupted by the trap. A special message, the queue overflow token, is inserted into the queue in the point where the contents of the overflow buffer need will be inserted in order to preserve the original message order. This relationship between the position of the overflow token and the contents of the overflow buffer is an invariant.

The trap handler then resumes the execution of the handler for message A. This is the situation that is illustrated in Figure 2-6.

When A's handler finishes and suspends, the overflow token is the next message, and it points to a special token handler (see Figure 2-7). This handler first copies the newly arrived message E to the overflow buffer. Then, it moves a few messages from the overflow buffer to the queue – in practice an amount to fill half the queue size. Then, the token handler creates a new token, suspends, and the handler for the next message resumes (Figure 2-8). The costs of overflow, in cycles, for the MDP are summarized in Table 2.1. Note the extremely high per-word cost for copying - 22 cycles to move a word of the message into the overflow

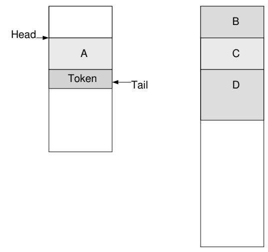


Figure 2-6: Example of queue overflow handling. The queue has been copied to an overflow buffer, and a special message, called the Token, has been inserted to mark the position of the copied queue contents.

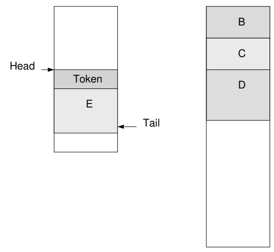


Figure 2-7: Example of queue overflow handling. Message A has suspended, so the token has reached the head of the queue. A new message, E, has arrived while A was executing.

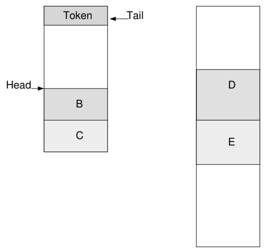


Figure 2-8: Example of queue overflow handling. E was placed in the overflow queue, and two messages, B and C, have been copied from the overflow buffer into the queue. A new token is inserted to mark the position of messages D and E.

per	*cycles*
Queue overflow trap preamble	150
Queue to buffer subroutine	50
Word to buffer	22
Queue overflow token	52
Message from buffer to queue	23
Word from buffer to queue	26

Table 2.1: Cycles to handle functions of the queue overflow trap and refill. For example, the penalty to handle the first overflow trap, with a 256 word queue, is $150+50+256*22 = 5832$ cycles.

buffer, and then 23 cycles per word to move it out. This implies that for a small 8-word message, more than 360 cycles of processing time are consumed just for copying it.

In order to allow the handler to safely manipulate the contents of the queue and its hardware pointers without interference, the logic which would during normal operation autonomously fill the queue is disabled during overflow handling and refill. As a consequence, during this time the consumption assumption, which assures progress in the wormhole network, is temporarily violated - the processor exerts temporary backpressure into the network. This can and does fanout into a tree of contention and jam the entire network. This can impose a substantial penalty – thousands of cycles for the overflow and refill – on the entire J-Machine, not just on the processor experiencing an overflow.

With overflow costs so high, one is naturally led to consider questions such as, "Why is this so broken?" The reason is that we did not anticipate that queue overflows would not be as rare as we'd hoped.

2.4 Conclusion

This chapter has introduced the communication architecture of the J-Machine, focusing on the consumption assumption and the way that the queue overflow traps are used to break a potential deadlock situation.

Note that although the focus is on the specific architecture of the J-Machine, this problem of deterministic routing (which comes about because of the desire to build very simple and consequentially very fast routers) and blocking and the potential for deadlock in the closed system is extremely general.

In isolation, the J-Machine's communication network, with its mesh topology, e-cube routing, wormhole channel allocation, and link level blocking, is guaranteed to make communication progress: as long as the consumption assumption is true. In isolation, the processor is guaranteed to make progress serving its message queue, because the fine-grained software philosophy limits the application to short, finite-length messages. However, when the network and the processors are combined the possibility of deadlock arises. This possibility is accounted for in the architecture by the existence of queue overflow handler trap. However, this trap is extremely slow. To avoid queue overflows - and the possibility of a deadlock - the queues are made large. However, although the queues are large, overflows still occur.

Should they be larger? What is the relationship between queue size and the rate of the occurrence of overflows on the J-Machine? This turns out not to be an easy question. The next chapter considers this question using queueing theory.

Chapter 3

Queuing Model

Each message buffer overflow event on the J-Machine inflicts a severe performance penalty. Obtaining an estimate of the expected rate of the occurrence of these overflows is the necessary first step in estimating performance. Multiplying the rate by the cost per event gives the expected overhead of buffer overflows. This chapter gives a method for estimating the rate of overflows.

Overflow rate is determined by the specific workload and this chapter focuses on one particular synthetic workload called "Snakes." Snakes consists simply of tasks migrating randomly about the machine via message-passing and is a coarse model of real applications.

The analysis in this chapter employs a a classic closed queueing model. Employing this model entails making the following assumptions: ignoring the effects of the communication network, such as blocking, delay, transition time, contention, finite capacity, pipelining, and routing. It also assumes that tasks are transitioning independently and that service time distributions are random independent exponential processes. Implicit in the structure of the model is the additional assumption that there is no active balancing process.

With these assumptions, the chapter makes the following contributions.

- It derives expressions for the expected rate of of message buffer overflows for both balanced and imbalanced workloads, as a function of the size of the machine, the number of messages on the machine, and the size of the message buffers. For a given "acceptable" rate of overflows, this allows one to choose a buffer size which is expected to achieve that rate.

- It derives an approximate relationship for the degree of imbalance in an imbalanced workload that still allows the expected queue size to be smaller than some fixed size and thereby allow the system to avoid large message accumulations at bottleneck processors.

These contributions provide a technique for choosing the hardware buffer size for message-passing parallel computers that avoids queue overflows.

The modeling effort begins in Section 3.1 by reviewing the classic closed queueing network and its well-known solution. Queueing theorists speak in terms of *customers* and *servers*; here, these correspond to the tasks and processors in a parallel computer. The *queue* is the accumulation of customers waiting for the server; it corresponds to the tasks waiting for service by the processor.

The Message-Driven Processor's architecture creates an equivalence between tasks and messages: both correspond to queueing network customers. The CPU of the MDP corresponds to the queueing network server.

Modulo the assumptions, the model is generally applicable to many parallel computers; for example, Alewife running message-passing traffic.

This chapter is structured as follows. Section 3.1 defines the classic model. The analytical power of queueing networks comes from the fact that with certain probability distributions for the service time of customers at servers, the system is a continuous-time, discrete state ergodic Markov process and therefore has an equilibrium probability distribution over the states which is independent of the starting state. The beauty of queueing networks comes from the fact that this probability distribution has a simple product form. Section 3.1 gives this solution.

Section 3.2 gives definitions for queue overflow within this model, and expressions for both the expected equilibrium rate of overflows and the expected fraction of time that the system spends overflowed. However, directly computing these rates for large systems involves computing weighted sums over intractably large state spaces of configurations of tasks on processors. To overcome this obstacle, the chapter considers two special cases. First, Section 3.3 considers the case where the workload is balanced and uniform. In this case, the model predicts that all states have equal probability, so the queue overflow rates are merely proportional to the sizes of their corresponding state subspaces. An expression for the sizes of these spaces is derived and is used to estimate queue overflow rates.

Section 3.4 considers the expected rate of queue overflow when the load is imbalanced. Central to this section is the derivation of an expression estimating the point at which imbalance leads to severe congestion at bottleneck servers. Derivations follow for the overflow rate in imbalanced systems.

The chapter concludes with a discussion of the limitations of the model and some rules related to permitted queue size.

3.1 Basic Closed Queuing Model

The well-known classic closed queuing model consists of N servers traversed by K identical customers. After receiving service at server i, a customer moves to server j with probability r_{ij}. Because the system is closed, $\sum_{j=1}^{N} r_{ij} = 1$ for all i.

Servers are Markovian; that is, their service time distribution is exponential.[1] The service rate of server i is μ_i. Modeling the servers as exponential servers causes the system to be a finite continuous-time Markov process. The states of this process, denoted $\vec{k} = (k_1, k_2, \ldots, k_N)$, are arrangements of the K customers over the N servers. The number of these states, $\mathrm{s}(N, K)$, is:

$$\mathrm{s}(N, K) = \binom{N + K - 1}{K}. \tag{3.1}$$

This can be derived by choosing one of the N servers for each of the K customers with repetitions allowed [Liu68, p.13]. Another way of thinking of this is in terms of K white stones and $N - 1$ black stones placed in some order along a line. Moving along the line, the positions of the $N - 1$ black stones partition the K white stones into N groups, where a group has zero white stones if two black stones are adjacent. The number of possible ways of placing these stones is the number of ways of choosing K out of the $N + K - 1$ positions for

[1] Using a distribution of service times rather than a fixed service time captures the notion that the computation required by message handlers within an application will vary. The Markovian assumption is useful because it results in a product form solution (Equation (3.6)). However, the Markovian distribution usually best models an arrival process composed of many independent arrival subprocesses, which is not really the case for the service time for a J-Machine message handler. Section 3.5.2 discusses the implications of this assumption for this study's results.

the white stones. There is a one-to-one correspondence between the ways of ordering these stones and the unique states $\vec{k}$ of placing customers on servers.

For large parallel computers, $\mathrm{s}(N, K)$ can be a very large number. For example,

$$\mathrm{s}(512, 1024) \approx 2.9 \times 10^{422}. \tag{3.2}$$

The size of this state space is an obstacle to using queueing theory to model a parallel computer. This chapter is largely devoted to overcoming this obstacle.

Given the system is in state $\vec{k_0}$ at time $t = 0$, $\mathrm{P}(\vec{k}, t)$ is the probability that the system will be in state $\vec{k}$ at time t. Because this Markov chain is ergodic[2], a limiting probability distribution $\mathrm{P}(\vec{k})$ exists:

$$\mathrm{P}(\vec{k}) = \lim_{t \to \infty} \mathrm{P}(\vec{k}, t). \tag{3.3}$$

This probability distribution has a simple form, first given by Gordon and Newell and simplified here from Kleinrock's text [Kle75, p.150]. Its derivation proceeds from the balance equation,[3]

$$\mathrm{P}(\vec{k}) \sum_{i=1}^{N} (k_i \geq 1)\mu_i = \sum_{i=1}^{N} \sum_{j=1}^{N} (k_j \geq 1)\mu_i r_{ij} \mathrm{P}(\vec{k} - \vec{e_j} + \vec{e_i}). \tag{3.4}$$

which holds, because for a Markovian system at equilibrium, the expected rate at which the system leaves some state $\vec{k}$ must be equal to the expected rate at which other neighbor states – in this case those one customer movement away – enter this state.

The solution to this balance equation is in terms of the values $\{x_i\}$, defined as solutions to these traffic equations[4] for $1 \leq i \leq N$:

$$\mu_i x_i = \sum_{j=1}^{N} \mu_j x_j r_{ji}. \tag{3.5}$$

The solution is:

$$\mathrm{P}(\vec{k}) = \frac{1}{G(K)} \prod_{i=1}^{N} x_i^{k_i}. \tag{3.6}$$

[2]A Markov chain is *ergodic* if there is a zero probability that some state will never recur.

[3]In this notation, a relation such as $(k_i \geq 1)$ in an equation is defined to be 1 if the relation is true and 0 if it is false. The vector $\vec{e_j}$ is the unit vector in the direction j, which means the state $\vec{k} - \vec{e_j} + \vec{e_i}$ has one more customer in queue i and one less in queue j than does the state $\vec{k}$.

[4]The values x_i combine the information in the routing matrix, r_{ij}, and service distributions, μ_i, and so are a sort of "load factor" that modulates the service rate so that arrival (the right side of Equation (3.5)) and departure (the left side of Equation (3.5)) rates at each server are equal.

The normalization constant, $G(K)$, ensures that the probabilities sum to 1.

$$G(K) = \sum_{\vec{k} \in A} \prod_{i=1}^{N} x_i^{k_i} \tag{3.7}$$

where the state space

$$A = \{\vec{k} : K = \sum_{i=1}^{N} k_i\}. \tag{3.8}$$

One can check that Equation (3.6) solves Equation (3.4) by substitution. For each vector $\vec{k} \in A$,

$$(\prod_{i=1}^{N} x_i^{k_i}) \sum_{i=1}^{N} (k_i \geq 1)\mu_i = \sum_{i=1}^{N} \sum_{j=1}^{N} (k_j \geq 1)\mu_i r_{ij} (\prod_{l=1}^{N} x_l^{k_l}) \frac{x_i}{x_j}. \tag{3.9}$$

Cancelling from both sides and rearranging,

$$\sum_{i=1}^{N} (k_i \geq 1)\mu_i = \sum_{i=1}^{N} \sum_{j=1}^{N} (k_j \geq 1)\mu_i r_{ij} \frac{x_i}{x_j} \tag{3.10}$$

$$\sum_{i=1}^{N} (k_i \geq 1)\mu_i = \sum_{j=1}^{N} (k_j \geq 1) \frac{1}{x_j} \sum_{i=1}^{N} \mu_i x_i r_{ij}. \tag{3.11}$$

Equation (3.5) can be substituted for the final terms of Equation (3.11), leaving an identity.

Expected properties of the system can be computed using a sum of that property weighted by the equilibrium probability distribution (Equation (3.6)). Buzen developed techniques for efficiently computing the expected mean values of queue size, throughput, and waiting time [Buz73]. However, queue overflow is a dynamic event. The next sections develop estimates for the equilibrium dynamics of the queue size processes that determine the rate of queue overflow.

3.2 Queue Overflow

Definition of Queue Overflow

If we designate a threshold Q for the capacity of the queue on each server, then the event "a queue overflow" is a transition of the system from non-overflowed state to an overflowed state, or more precisely, from a state in

$$A_{\text{nov}} = \{\vec{k} : k_i \leq Q\} \tag{3.12}$$

to one in

$$A_{\text{ov}} = \overline{A_{\text{nov}}}. \tag{3.13}$$

Rate of Queue Overflow

When the system is at equilibrium, the expected rate at which queue overflows occur is given by the following expression that sums, for states in A_{nov} but at the border with A_{ov}, the rate of customer transition into servers with a queue size of Q:

$$\lambda_{\text{Overflow}} = \sum_{\vec{k} \in A_{\text{nov}}} \mathrm{P}(\vec{k}) \sum_{i=1}^{N} \sum_{j=1}^{N} (k_i = Q)(k_j > 0)(j \neq i)\mu_j r_{ji}. \tag{3.14}$$

Overflow Residency

Another figure of interest is the expected fraction of time that the equilibrium system resides in states $\vec{k} \in A_{\text{ov}}$. This value is designated $\mathrm{P}(A_{\text{ov}})$. Similarly, the fraction of time the system spends in a state $\vec{k} \in A_{\text{nov}}$ is designated $\mathrm{P}(A_{\text{nov}})$. These values are given by the equations:

$$\mathrm{P}(A_{\text{ov}}) = \sum_{\vec{k} \in A_{\text{ov}}} \mathrm{P}(\vec{k}) \tag{3.15}$$

$$\mathrm{P}(A_{\text{nov}}) = \sum_{\vec{k} \in A_{\text{nov}}} \mathrm{P}(\vec{k}). \tag{3.16}$$

$\lambda_{\text{Overflow}}$ and $\mathrm{P}(A_{\text{ov}})$ are different measures. It is conceivable that the Markov chain defined by this closed queuing process might spend most of its time in an unoverflowed state so that $\mathrm{P}(A_{\text{ov}}) \approx 0$, but still have a high $\lambda_{\text{Overflow}}$ because it experiences a large number of transitions between A_{ov} and A_{nov}. Because of this, a cost model for overflows used for performance estimation should charge separately for $\lambda_{\text{Overflow}}$ and A_{ov}. The charge associated with $\lambda_{\text{Overflow}}$ is for the traps that the system takes when a queue overflows; the charge for A_{ov} is for the penalty for operating in the states in which a queue is overflowed.

Unfortunately, computing values for $\lambda_{\text{Overflow}}$, $\mathrm{P}(A_{\text{ov}})$, and $\mathrm{P}(A_{\text{nov}})$ for a large general closed queuing model is difficult, because the number of states is so large. But for balanced and uniform workloads, which are treated next in Section 3.3, the equations simplify.

3.3 Balanced Workload

The closed queuing workload on the system is defined to be a *balanced workload* if all of the values x_j defined by Equation (3.5) are the same for all j. A *uniform workload* is one that is balanced because $\mu_i = \mu$ and $r_{ij} = 1/N$ for all i and j; that is, all servers serve at the same rate and the transition probability among all servers is uniform. If a workload is uniform, it is balanced. However, it is possible for a balanced workload to not be uniform.

It is helpful to make a distinction among a *balanced workload,* a *balanced state,* and a *balanced process.* The terms balanced state and balanced process describe how the workload runs on the machine. A balanced state $\vec{k}$ is one where the size of the queue on each server is less than or equal to some threshold Q. A balanced process is one that stays in the balanced states for long periods of time. In Chapter 4, when we run balanced workloads on the J-machine we will see that they often run as imbalanced processes.

3.3.1 All States Equiprobable at Equilibrium

Because the x_j are all equal to each other when the workload is balanced, Equation (3.6) predicts that each unique configuration $\vec{k}$ has an equal equilibrium probability, equal to $1/|A|$. Calculating configuration probabilities becomes equivalent to counting numbers of states with that configuration.

Intuitively, with the traffic choosing destinations randomly, one would expect the random process to disperse traffic on the machine. With this in mind, the prediction that each configuration $\vec{k}$ in a balanced workload has equal probability seems strange. It means that an imbalanced state

$$\vec{k_a} = (K, 0, 0, \dots 0),$$

where all the customers are concentrated on a single server is as probable as a balanced state

$$\vec{k_b} = (\lfloor \tfrac{K}{N} \rfloor, \lfloor \tfrac{K}{N} \rfloor, \dots, K - (N-1)\lfloor \tfrac{K}{N} \rfloor)$$

where the customers are more evenly distributed.[5] This seems to be inconsistent with the

[5] This selection and definition of $\vec{k_b}$ is good as an example for this argument only where $K \geq N$. If $K < N$, then some other selection of an example balanced vector, such as $\vec{k_b} = \sum_{i=1}^{K} \vec{e_i}$, makes the same point. $\vec{e_i}$ is the unit vector with a 1 in the ith position.

intuition that the random transitions in this queuing system would tend to disperse any traffic accumulations.

But the predictions of the model are correct. The intuition that the traffic will be spread evenly is supported by the model, because of the combinatorics. Roughly, $|A_{\text{balanced}}|$, the number of states that one might consider "balanced" is much larger than the number of states that one might consider "imbalanced," so stationary probability $\text{P}(A_{\text{balanced}})$ is high, even though the stationary probability $\text{P}(\vec{k_b})$ of a particular example $\vec{k_b} \in A_{\text{balanced}}$ is small. In other words, the probability of an individual state $\vec{k_b}$ might be low, but there are many more states like it.

Because there are many more transitions into a balanced state like $\vec{k_b}$ than there transitions into imbalanced states like $\vec{k_a}$, one might have another intuition: that $\text{P}(\vec{k_b}) > \text{P}(\vec{k_a})$, rather than the model prediction, which is that $\text{P}(\vec{k_a}) = \text{P}(\vec{k_b})$. The number of ways of getting into $\vec{k_b}$ is equal to the number of neighbor states of $\vec{k_b}$, of which there are $N^2 - N$. In contrast, state $\vec{k_a}$ only has $N - 1$ neighbors. But this does not mean that the probability of state $\vec{k_a}$ is N times higher than state $\vec{k_b}$: although there are N times more ways in to $\vec{k_b}$, there are also N times more ways out of $\vec{k_b}$. In state $\vec{k_b}$, more servers have customers, so transitions happen faster out of $\vec{k_b}$ than $\vec{k_a}$. It is true that $\vec{k_b}$ gets visited more often, but the time spent there is shorter.[6]

3.3.2 Counting States

For a balanced workload, because all state probabilities are equal, the stationary probability $\text{P}(A_x)$ of an the system being in one set of states $A_x \subset A$ is the ratio

$$\text{P}(A_x) = \frac{|A_x|}{|A|} \tag{3.17}$$

In other words, computing probabilities is done by counting states. Because the number of states is large, we must find an efficient way of computing $|A_x|$

Define a function $\text{t}(n, k, q)$ that counts the number of ways of placing k identical customers on

[6]If one considers a system operating in discrete time with servers having geometric service time distributions all with rate $1/\tau$, vectors like $\vec{k_b}$ *do* have higher probability than vectors like $\vec{k_a}$. This comes from allowing simultaneous customer transitions, which is not possible when the servers are exponential. In a parallel machine with $N > \tau$ the chance of simultaneous transitions is high.

		k							
		0	1	2	3	4	5	6	7
	0	1	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1	1
	2	1	2	3	4	5	6	7	8
n	3	1	3	6	10	15			
	4	1	4	10	20	35			
	5	1	5						
	6								
	7								

Table 3.1: Number of ways of arranging k items on n servers.

n unique servers so that there are never more than q customers on any one server. Therefore:

$$\mathrm{t}(N, K, Q) = |A_{\mathrm{nov}}|. \tag{3.18}$$

With an expression for the value of $\mathrm{t}(n, k, q)$ it is possible to determine the residency in non-overflowed states for a balanced load:

$$\mathrm{P}(A_{\mathrm{nov}}) = \frac{\mathrm{t}(N, K, Q)}{\mathrm{s}(N, K)}. \tag{3.19}$$

($\mathrm{s}(N, K)$ was defined earlier in Equation (3.1).) An algorithm for a recursive calculation of $\mathrm{t}(n, k, q)$ is obtained by following the pattern used to recursively define $\mathrm{s}(n, k)$.

The recurrence for s is:

$$\mathrm{s}(n, k) = \begin{cases} 0 & n = 0 \wedge k > 0 \\ 1 & k = 0 \\ \sum_{i=0}^{k} \mathrm{s}(n-1, k-i) & \text{otherwise} \end{cases} \tag{3.20}$$

The first two terms are self-evident; the third term sums the sizes of $k+1$ mutually exclusive subsets. A subset number i puts i customers on the first server, then puts $k-i$ customers on the remaining $n-1$ servers; the size of this subset is $\mathrm{s}(n-1, k-i)$. This recursive definition can be further simplified to eliminate the $\sum$; see Table 3.1. The value for an item (n, k) in this table is the sum of the values in columns 1 to k from row $n-1$. This table is Pascal's Triangle, with its peak at $(n, k) = (1, 0)$ and its rows running diagonally. This is correct because Pascal's Triangle is a tabulation of the binomial, and $\mathrm{s}(N, K)$ is defined by a binomial, Equation (3.1). It suggests simplifying the recurrence $\mathrm{s}(n, k)$ above to:

$$\mathrm{s}(n,k) = \begin{cases} 0 & n = 0 \wedge k > 0 \\ 1 & k = 0 \\ \mathrm{s}(n-1,k) + \mathrm{s}(n,k-1) & \text{otherwise} \end{cases} \tag{3.21}$$

Another way of expressing the recurrence for s is as its generating function $\mathrm{S}(y, z)$ defined as:

$$\mathrm{S}(y,z) = \sum_{n,k} y^n z^k \, \mathrm{s}(n,k) \tag{3.22}$$

The recurrence in Equation (3.21) implies the following relationship within the generating function:

$$\mathrm{S}(y,z) = y\,\mathrm{S}(y,z) + z\,\mathrm{S}(y,z) + (1-z). \tag{3.23}$$

The final term $(1 - z)$ accounts for the base cases. Rearranging Equation (3.23) yields:

$$\mathrm{S}(y,z) = \frac{1-z}{1-y-z}. \tag{3.24}$$

We can convert this back and forth between the generating function and the closed form using standard generating function transformations [GKP94, pages 334-335].

The derivation for $\mathrm{t}(n, k, q)$ proceeds similarly, beginning with the recurrence:

$$\mathrm{t}(n,k,q) = \begin{cases} 0 & (n = 0 \wedge k > 0) \\ 1 & k = 0 \\ \sum_{i=0}^{q} \mathrm{t}(n-1,k-i) & \text{otherwise} \end{cases} \tag{3.25}$$

This is nearly identical to the recurrence for s, in Equation (3.20), except it constrains the sum index i to only go to q, the maximum queue size. Tabulating the recurrence for a given q illustrates a pattern that helps to simplify it. For example, if we use $q = 1$ (i.e., each queue can hold at most one customer), we get Table 3.2, which again is Pascal's triangle. This is not surprising: with a queue restricted to one customer, placing K customers on N servers is equivalent to choosing a subset of size K from N distinct servers. Furthermore, in the case with $q = 1$, the value for $\mathrm{t}(n, k, 1)$ is generally much smaller than $\mathrm{s}(n, k)$. For example, for $N = 5, K = 3$, and $Q = 1$, the proportion of non-overflowed states

$$\frac{\mathrm{t}(5,3,1)}{\mathrm{s}(5,3)} = \frac{10}{35} \approx 0.29 \tag{3.26}$$

		k							
		0	1	2	3	4	5	6	7
	0	1	0	0	0	0	0	0	0
	1	1	1	0	0	0	0	0	0
	2	1	2	1	0	0	0	0	0
n	3	1	3	3	1	0	0	0	0
	4	1	4	6	4	1	0	0	0
	5	1	5	10	10	5	1	0	0
	6	1	6						
	7								

Table 3.2: Tabulation of the number of combinations for (n, k) with all queues not exceeding one customer. This contains Pascal's triangle.

		k							
		0	1	2	3	4	5	6	7
	0	1	0	0	0	0	0	0	0
	1	1	1	1	0	0	0	0	0
	2	1	2	3	2	1	0	0	0
n	3	1	3	6	7	6	3	1	0
	4	1	4	10	16	19	16	10	4
	5	1	5	15	30	45	51	45	30
	6	1	6						

Table 3.3: Tabulation of the number of combinations for (n, k) with all queues not exceeding two customers.

is less than half. This supports intuition that if the queue size is 1, it is likely that the system would be overflowed more often than not.

When $Q > 1$ the tabulation is more complex. Table 3.3 plots $\mathrm{t}(n, k, 2)$. This table suggests that the recurrence to compute this combination is

$$\mathrm{t}(n,k,q) = \begin{cases} 0 & (n = 0 \wedge k > 0) \\ 1 & k = 0 \\ \mathrm{t}(n-1,k,q) + \mathrm{t}(n,k-1,q) - \mathrm{t}(n-1,k-q-1,q) & \text{otherwise} \end{cases} \tag{3.27}$$

It might be preferable to have an asymptotic definition for $\mathrm{P}(A_{\mathrm{nov}})$, which also might not be too complex. I have not had much luck getting one from Equation (3.27).[7] An algorithm text such as Cormen, Leiserson, and Rivest's [CLR90] gives methods for obtaining asymptotic bounds for one-parameter recurrences, but the recurrence of interest here is over two parameters. Perhaps using its generating function:

$$\mathrm{T}(y,z,q) = \frac{1-z}{1-y-z-yz^{q+1}} \tag{3.28}$$

followed by the Taylor expansion of $\mathrm{T}(y, z, q)$ about y and z, picking off coefficients of the terms $y^n z^k$ might be a way to derive $\mathrm{t}(n, k, q)$? Unfortunately, the form of $\mathrm{T}(y, z, q)$ makes this a difficult process, with the results quickly buried in combinations generated by messy chain rule derivatives.[8]

Fortunately, for small values of n, k, q, it is possible to automate the computation of $\mathrm{t}(n, k, q)$ directly using Equation (3.27). A nested loop over n and k gives the result in $O(n^2k)$ time, using $O(qn)$ space. The numbers are large, necessitating care to avoid limitations of common floating point number representations; but this is easy to do, using an efficient large integer [Len89] or extended range floating point package. Computing $\mathrm{t}(512, 512, 64)$ requires approximately a minute on a contemporary SPARC workstation.

3.3.3 Balanced Workload Overflow Residency

Figure 3-1 plots $\mathrm{P}(A_{\mathrm{nov}})$ versus the ratio $m = K/N$ for $N = 512$ and two queue sizes $Q \in \{64, 128\}$. These values are chosen to correspond to a 512-processor J-machine, 4-word

[7] In Section 3.5.2, I explore why an asymptotic relation might not be useful.

[8] Maybe binomial bounding arguments similar to those used in Section 3.4.3 would help?

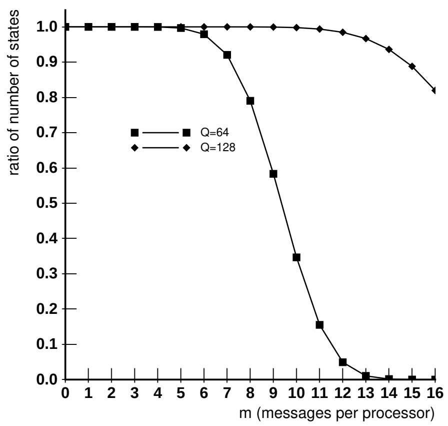


Figure 3-1: $\mathrm{P}(A_{\mathrm{nov}})$ versus $m = K/N$ for $N = 512$ and two queue sizes.

messages and a 256-word and a 512-word queue. For low values of m, the ratio $\mathrm{P}(A_{\mathrm{nov}})$ is indistinguishable from 1. This means that the system in the stationary state is likely to be in a state in A_{nov}.

As m increases, the graph starts to break away from 1 at $m = 6$ when $Q = 64$ and $m = 12$ when $Q = 128$. This suggests a simple linear relationship.

3.3.4 Leighton's Method

Leighton [Lei97] has suggested another approach for estimating $\mathrm{P}(A_{\mathrm{nov}})$, based on the white stone and black stone analogy mentioned in Section 3.1. Reiterating that analogy here: the states $\vec{k} \in A$ correspond to the arrangements of K white stones and $N-1$ black stones along a line. The black stones partition the white stones into N groups. The number of these constrained arrangements is

$$\begin{pmatrix} N+K-1 \\ K \end{pmatrix} = \begin{pmatrix} N+K-1 \\ N-1 \end{pmatrix}$$

and is is equal to $|A|$.

Let w_i denote the color of stone i; if the color is white, $w_i = 1$, otherwise the color is black and $w_i = 0$. Leighton's argument begins by noting that for any given stone i, the probability that a stone is white can be estimated:

$$\begin{aligned} \mathrm{P}(w_i = 1) &= \frac{K}{K+N-1} && (3.29)\\ &\approx \frac{m}{m+1} && (3.30) \end{aligned}$$

Assume that each stone chooses its color independently according to this probability. The resulting configuration resembles the constrained arrangements, though there may be more or fewer black stones, and as a result more or fewer partitions. That is, the dimensionality $\dim(\vec{k})$ of the corresponding configuration may be unequal to N, and the $\sum_i k_i$ may be unequal to K. As $N + K$ grows, though, according to laws of large numbers, the number of black stones will approach the expected values, and consequently the unconstrained configurations will resemble the constrained configurations.

In the unconstrained configuration space, because the stones are choosing color independently, the probability that the size of the first partition exceeds the buffer size Q is:

$$\mathrm{P}(w_1 = 1, w_2 = 1, \ldots, w_{Q+1} = 1) = \left(\frac{m}{m+1}\right)^{Q+1} \tag{3.31}$$

If this probability applies to the other queues, and if those queues were independent, then the probability that no queue exceeds threshold Q is:

$$\begin{aligned} \mathrm{P}(\forall i, k_i \leq Q) &= \left(1 - \left(\frac{m}{m+1}\right)^{Q+1}\right)^N && (3.32)\\ &\approx e^{-(N(\frac{m}{m+1})^{Q+1})} && (3.33) \end{aligned}$$

For parallel computers where the cost of an overflow is high, this probability should be close to 1; that is, it is desirable that:

$$\mathrm{P}(\forall i, k_i \leq Q) > 1 - \epsilon. \tag{3.34}$$

ϵ is a small number. With this in mind, one can solve for the estimated necessary buffer size Q_{required} to achieve this necessary probability:

$$\begin{aligned} Q_{\mathrm{required}} &> \log_{(\frac{m+1}{m})} N - \log_{(\frac{m+1}{m})} \ln \frac{1}{1-\epsilon} && (3.35)\\ &= \frac{\ln N - \ln\ln\frac{1}{1-\epsilon}}{\ln\frac{m+1}{m}} && (3.36) \end{aligned}$$

As $\epsilon \rightarrow 0$, the value $Q_{\text{required}} \rightarrow \infty$. This contradicts the fact that if $Q > K$, then the system will never experience overflows. The contradiction arises from the various approximations. Still, for moderate values, one can get a good estimate of Q.

One can also use Equation (3.33) to solve for the estimated largest value of m_{required} that achieves this necessary probability.

$$m_{\text{required}} < \frac{1}{\left(\frac{N}{\ln\frac{1}{1-\epsilon}}\right)^{\left(\frac{1}{Q+1}\right)} - 1} \tag{3.37}$$

Equation (3.34) is an estimate for $\mathrm{P}(A_{\text{nov}})$. Predictions from this equation are consistent with Figure 3-1. For example, with $N = 512$, $Q = 64$, and $\epsilon = 0.5$, the value $m_{\text{required}} = 9.3$, which is the center of the breakpoint in the figure.

The explicit computation of the recurrence (Equation (3.27) used to produce Figure 3-1 is too slow to estimate overflow residency for very large numbers of processors. In contrast, the Leighton method is direct, so with it, one can compute needed queue sizes for very large machines. For example, the MDP is designed with processor addressing bits to support a machine with up to 32,768 processors. With $m = 5$ messages per processor, the value of Q_{required} for $\epsilon = 0.5$ is 59. To achieve an overflow residency of 10^{-4} on this machine requires a queue size of 107 elements. The MDP has a queue that can hold up to 128 4-word messages; by this analysis, the MDP buffer size appears to be adequate.[9]

In general, Equation (3.36) suggests that the required queue size Q_{required} is logarithmic in the size of the machine and also logarithmic in the factor $\frac{1}{1-\epsilon}$. The key parameter, though, is m. Appearing in the denominator of Equation (3.36), as this m grows, so does Q_{required}. This growth is approximately linear in m, which makes intuitive sense, because m is the number of messages per processor on the system. However, as the analysis in Section 3.4.2 shows, as m grows, the sensitivity of the system to imbalance does as well.

The derivation of the estimate Equation (3.33) depended on assuming that all of the queue sizes are independent. Correcting this assumption gives indication of whether this is an upper bound or a lower bound. Equation (3.33) is based on independence which assumes

[9] However, note that as Section 3.4, with any nonuniformity or imbalance (the typical case!), the required queue size can be much larger.

	$k_1 \leq Q$	$k_1 > Q$
$k_2 \leq Q$	w	x
$k_2 > Q$	y	z

Table 3.4: Partition of configuration-space based on the sizes of k_1 and k_2 relative to Q.

that

$$P(\forall i, k_i \leq Q) = P(k_1 \leq Q)P(k_2 \leq Q)P(k_3 \leq Q)\cdots P(k_N \leq Q) \tag{3.38}$$

Instead, it should be based on this equation:

$$\begin{aligned} P(\forall i, k_i \leq Q) &= P(k_1 \leq Q)P(k_2 \leq Q|k_1 \leq Q)P(k_3 \leq Q|k_1 \leq Q, k_2 \leq Q)\cdots \\ &\quad P(k_N \leq Q|k_1 \leq Q, k_2 \leq Q, \ldots k_{N-1} \leq Q). \end{aligned} \tag{3.39}$$

What is the relationship between the corresponding unconditionalized and conditionalized probabilities in the terms of these two equations; that is, how does conditionalizing a term change it? In the case that $K > Q$ (if otherwise, this analysis would be of no interest), and examining the first conditionalized term, knowing that $k_1 \leq Q$ suggests some information about whether or not it is true that $k_2 \leq Q$. Table 3.4 shows a partition of the configuration space based on the sizes of k_1 and k_2 relative to Q. In the table, the event w is that $(k_1 \leq Q \wedge k_2 \leq Q)$, the event y is $(k_1 \leq Q \wedge k_2 > Q)$, and so on.

One can reason that $P(w) < P(x)$, since if it is known that $k_1 \leq Q$, then there is less chance that $k_2 \leq Q$. Similarly, $P(z) < P(y)$. This implies that:

$$P(w)P(z) < P(x)P(y). \tag{3.40}$$

The value

$$P(k_2 \leq Q) = \frac{P(w) + P(x)}{P(w) + P(x) + P(y) + P(z)},$$

and the conditionalized probability

$$P(k_2 \leq Q|k_1 \leq Q) = \frac{P(w)}{P(w) + P(y)}.$$

Assume (for contradiction) that:

$$\mathrm{P}(k_2 \leq Q) \leq \mathrm{P}(k_2 \leq Q | k_1 \leq Q). \tag{3.41}$$

This leads to the conclusion:

$$\begin{aligned}
\frac{\mathrm{P}(w)+\mathrm{P}(x)}{\mathrm{P}(w)+\mathrm{P}(x)+\mathrm{P}(y)+\mathrm{P}(z)} &\leq \frac{\mathrm{P}(w)}{\mathrm{P}(w)+\mathrm{P}(y)} && (3.42)\\
(\mathrm{P}(w)+\mathrm{P}(x))(\mathrm{P}(w)+\mathrm{P}(y)) &\leq (\mathrm{P}(w))(\mathrm{P}(w)+\mathrm{P}(x)+\mathrm{P}(y)\\
&\quad +\mathrm{P}(z)) && (3.43)\\
\mathrm{P}(w)^2+\mathrm{P}(w)\mathrm{P}(y)+\mathrm{P}(x)\mathrm{P}(w)+\mathrm{P}(x)\mathrm{P}(y) &\leq \mathrm{P}(w)^2+\mathrm{P}(w)\mathrm{P}(x)+\\
&\quad \mathrm{P}(w)\mathrm{P}(y)+\mathrm{P}(w)\mathrm{P}(z) && (3.44)\\
\mathrm{P}(x)\mathrm{P}(y) &\leq \mathrm{P}(w)\mathrm{P}(z) && (3.45)\\
& && (3.46)
\end{aligned}$$

Equation (3.45) contradicts the known fact Equation (3.40), so the assumption Equation (3.41) is false. By this reasoning, Equation (3.38) is an upper bound on Equation (3.39). This limits the usefulness of the method because Equation (3.34) requires a lower bound estimate. Still, this estimate appears to be a useful rule of thumb that can be checked by using the exact recurrence.

3.3.5 Balanced Workload Overflow Rate Estimation

When the workload is balanced, and particularly if it is uniform with all $\mu_i = \mu$ and $r_{ij} = 1/N$, the expression for $\lambda_{\text{Overflow}}$ in Equation (3.14) can also be simplified and approximated. The goal of this simplification is to eliminate the $\sum$ over the A_{nov} space.

$$\begin{aligned}
\frac{\lambda_{\text{Overflow}}}{\mu} &= \frac{1}{|A|N} \sum_{\vec{k}\in A_{\text{nov}}} \sum_{i=1}^{N}\sum_{j=1}^{N}(k_i = Q)(k_j > 0)(j \neq i) && (3.47)\\
&< \frac{1}{|A|N} \sum_{\vec{k}\in A_{\text{nov}}} \sum_{i=1}^{N}(k_i = Q)\sum_{j=1}^{N}(j \neq i) && (3.48)\\
&= \frac{N-1}{|A|N} \sum_{\vec{k}\in A_{\text{nov}}} \sum_{i=1}^{N}(k_i = Q) && (3.49)\\
&= \frac{N-1}{|A|N} \sum_{i=1}^{\lfloor K/Q \rfloor} i \binom{N}{i} \mathrm{t}(N-i, K-Qi, Q-1) && (3.50)
\end{aligned}$$

Equation (3.47) normalizes the overflow rate to the server rate on all servers, and moves the state probabilities and transition probabilities out of the sum. For each state at the border of A_{nov} and for each server within that state with a full queue, its sum terms correspond to each possible other queue that could send to the nearly full server to put the state of the system into A_{ov}. Then, Equation (3.48) drops the relation that ensures that the other server has something to send, which is justified if the number of customers is large. Finally, Equation (3.49) results from recognizing that $\sum_{j=1}^{N}(j \neq i)$ is equal to $N-1$, leaving a sum in Equation (3.50) that counts, for each state at the border of A_{nov}, the number of servers in that state that have Q customers.

Recognizing this, Equation (3.50) creates a sum that uses the function $\mathrm{t}(n,k,q)$ developed in the previous section. In it, the sum is over independent sets defined by having i with Q customers, and all other servers having less than Q customers. There are $\binom{N}{i}$ ways of choosing the filled servers. Equation (3.49) counts a state with i filled servers i times, so Equation (3.50) multiplies each term of the sum by i. With this equation simplified, it is possible to compute the values for a machine the size of the J-machine.

One final simplification is helpful. Often the $i = 1$ term of the sum in Equation (3.50) is much larger than any other. All other terms in the sum can be dropped, resulting in the approximation for the normalized overflow rate:

$$\frac{\lambda_{\text{Overflow}}}{\mu} \approx \frac{1}{|A|} N\,\mathrm{t}(N-Q, K-Q, Q-1) \tag{3.51}$$

Interestingly, this leaves the form of the overflow rate in a simple ratio like Equation (3.19). The program that approximates Equation (3.50) only sums terms that significantly affect the ratio.

This approximation is plotted in Figure 3-2 for several queue sizes. Consider the overflow rate when $Q = 64$, which corresponds to the effective size of the J-machine queue with 8-word messages. This figure shows that when $m \leq 2$, the overflow rate is low, less than 10^{-9}. As m increases and more customers are added to the system, the normalized overflow rate starts to grow, peaking at ≈ 0.04. This implies an overflow approximately once every 25 messages. After peaking, though, the overflow rate starts to drop, even though there are more customers in the system: the number of non-overflowed states is small, and as a consequence the system spends much more time in the states $\vec{k} \in A_{\text{ov}}$. Therefore, the area

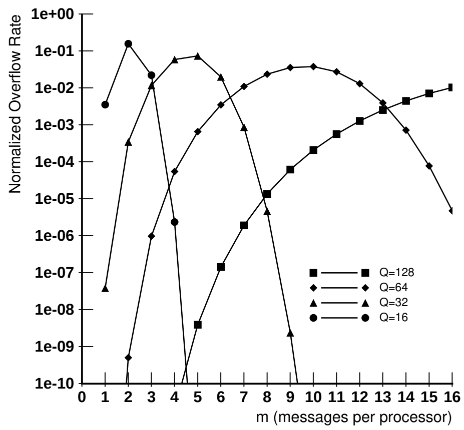


Figure 3-2: For uniform workload, $\frac{\lambda_{\text{Overflow}}}{\mu}$ (logarithmic scaling) versus $m = K/N$ for $N = 512$ and several queue sizes.

to the right of the peak represents a region of operation where the machine is spending most of its time overflowed.

Figure 3-3 plots Equation (3.50), but normalizes the abscissa to be in terms of the fraction of queue capacity consumed by total number of customers, that is, $K/(QN)$. The plot shows that curve shapes are roughly similar. Combining the information in this plot with a desired acceptable rate of overflows allows a machine architect to choose the appropriate amount of buffer storage. For example, if one chooses a maximum acceptable over flow rate of 1%, the model suggests that the buffer storage capacity should be 10 times larger than the number of customers.

This rule of thumb applies to machines sized from 256 to 1024 processors.[10] Figure 3-4 shows how the number of servers affects the normalized overflow rate. The overflow rate does increase as machine size increases, but the overflow rate continues to reach 1% when $K/(QN) = 0.1$.

[10] I did not run the program for larger machines because my program for computing $t(n, k, q)$ used long integers and so is too slow. The values could be computed by changing the program to use an extended floating point representation.

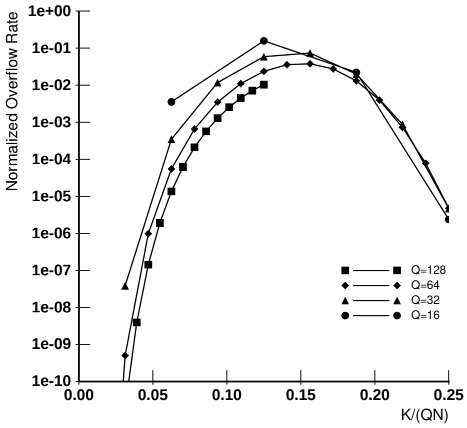


Figure 3-3: For uniform workload, $\frac{\lambda_{\text{Overflow}}}{\mu}$ (logarithmic scaling) versus $K/(QN)$ for $N = 512$ and several queue sizes.

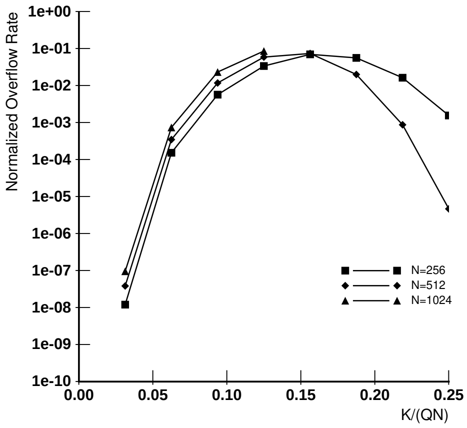


Figure 3-4: For uniform workload, $\frac{\lambda_{\text{Overflow}}}{\mu}$ (logarithmic scaling) versus $K/(QN)$ for $N = 512$ and several machine sizes.

Figure 3-4 shows past the peak rate of overflows and when the machines are mostly resident in the overflowed state, the smaller machine with 256 processors is maintaining a higher rate of overflows.

3.3.6 Balanced Workload: Conclusion

Section 3.3 used the fact that the load was balanced to simplify the large state equations in Equation (3.6). The model relates the fraction of the system queue capacity that is consumed to the normalized rate of overflows. For a particular "acceptable" normalized rate of overflows, it defines a limit to the number of customers that one can permit in the system or suggests the required queue size for a given number of customers. For example, for a normalized overflow rate of 10^{-6} (meaning, that one overflow occurs system-wide for every 1,000,000 messages that a *single* processor handles), at most 5% of the queue capacity of the system can be consumed by customers.

3.4 Imbalanced Workload

The previous section used the assumptions that the workload was balanced, with all x_j equal to each other, and uniform, with all $\mu_j = \mu$ and $r_{ij} = 1/N$. These assumptions simplified the state probability equations to produce relationships between machine size, load size, and overflow rates. But, these assumptions are limiting: in practice the workload is usually imbalanced, even if only slightly. The imbalance occurs if traffic is directed unevenly (r_{ij} unequal) or if servers work at different rates (μ_i unequal). Both lead to unequal values of x_i from Equation (3.5).

How do the dynamics of the imbalanced load differ from the balanced load?

The values x_j can have a substantial effect on the equilibrium probability distribution for customer configurations. As the number of customers increases, states in which the "bottleneck server" are filled have a dominant probability. Kleinrock [Kle75, p.152] formulates the well-known known behavior as follows: with some particular server number l having $x_l > x_i$ for all $i \neq l$, in the limit as $K \to \infty$, the probability $\mathrm{P}(\vec{k}) \to 0$ for any state in which $k_l \neq \infty$.

K	$E[k_1]$
512	25
768	257
1024	513
2048	1537

Table 3.5: Expected queue size for server number 1 when it is 2× slower than the others in a closed network of 512 exponential servers. This is computed using Buzen's convolution algorithm.

This is predicted by Equation (3.6) in which the weight of probability $P(\vec{k})$ is proportional to

$$x_1^{k_1} x_2^{k_2} \cdots x_N^{k_N}.$$

The probability of one state relative to others is exponential in the k_i. On parallel machines, K can be large; states where the largest value x_j are raised to K will dominate the probability distribution.

3.4.1 Simple Examples

An extreme form of bottleneck behavior is as follows. Consider a system with $N = 512$ servers with uniform customer transitions $r_{ij} = 1/N$. All the servers except number 1 serve at the same rate μ, while server number 1 works at the slower rate $\mu_1 = 0.5\mu$. Server number 1, therefore, is the bottleneck. Table 3.5 shows the expected queue length[11] for different values of K. The table shows that as the number of customers grows,

$$E[k_1] \rightarrow K - N.$$

As more traffic is added, nearly all of the additional traffic accumulates *in that single bottleneck server.*

Figure 3-5 shows that it is possible to have a moderate level for the expected queue size even with a large number of customers on the system, as long as the imbalance is only slight. For

[11][Buz73] gives a simple algorithm for computing this.

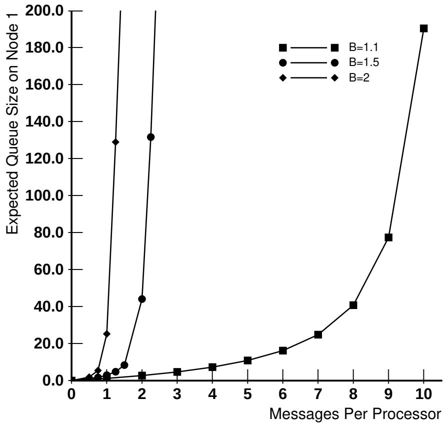


Figure 3-5: Expected queue size for server number 1 when it is B times slower than the others in a closed network of 512 exponential servers, versus the number of customers per server K/N. The figure demonstrates a threshold at $K/N > 1/(B-1)$ predicted by Equation (3.65) beyond which expected queue size grows rapidly.

example, with the imbalance of only $\frac{\mu}{\mu_1} = 1.1$, it is possible to operate with 2048 customers on the 512 server system, with the expected queue size on the bottleneck server of only 7. Beyond a threshold of a particular threshold of the number of customers, the queue size grows rapidly. The next section derives this threshold.

3.4.2 Allowed Degree of Imbalance

Consider the case where one server is slower than the others by a factor B. That is, $x_j = 1$ for all $j \geq 2$ and $x_1 = B$. Using Equation (3.6), the probability of configurations where the number of customers on server number 1 is Q is

$$\mathrm{P}(k_1 = Q) = \frac{1}{G(K)} \sum_{\{\vec{k}:\vec{k} \in A \wedge k_1 = Q\}} B^Q \tag{3.52}$$

$$= \frac{1}{G(K)} B^Q \,\mathrm{s}(N-1, K-Q). \tag{3.53}$$

The derivation eliminates the sum by counting the number of ways of placing $K-Q$ customers on the other $N-1$ servers.

For this to be unlikely, we want this probability to be less than the probability of reference "good" states. We can choose those reference "good" states arbitrarily; a convenient choice is the states where $k_1 = 0$, whose probability is

$$\mathrm{P}(k_1 = 0) = \frac{1}{G(K)}\,\mathrm{s}(N-1, K). \tag{3.54}$$

To solve for an approximate value of B where

$$\mathrm{P}(k_1 = Q) < \mathrm{P}(k_1 = 0) \tag{3.55}$$

first approximate the function s using Stirling's approximation[12] for factorial.

$$\begin{aligned}
\mathrm{s}(N, K) &= \binom{N+K-1}{K} && (3.56)\\
&= \frac{(N+K-1)!}{(N-1)!K!} && (3.57)\\
&\approx \sqrt{\frac{N+K-1}{(N-1)K}}\,\frac{(N+K-1)^{N+K-1}}{(N-1)^{N-1}K^K} && (3.58)\\
&\approx \frac{(N+K-1)^{N+K-1}}{(N-1)^{N-1}K^K} && (3.59)\\
&\approx \frac{(N+K)^{N+K}}{N^N K^K} && (3.60)
\end{aligned}$$

This derivation substitutes in Stirling's approximation, drops the root (which is small relative to the exponentials), and then drops the 1's because K and N are much larger than 1 in the parallel machine.

Plugging this into Equation (3.55) and using logarithms to approximate,

$$\begin{aligned}
B^Q\,\mathrm{s}(N-1, K-Q) &< \mathrm{s}(N-1, K) && (3.61)\\
Q \log B &< \log \frac{\mathrm{s}(N-1, K)}{\mathrm{s}(N-1, K-Q)} && (3.62)\\
Q \log B &< (N+K)\log(N+K) - N\log N - K\log K\\
&\quad -(N+K-Q)\log(N+K-Q)\\
&\quad +N\log N + (K-Q)\log(K-Q). && (3.63)
\end{aligned}$$

Because Q does not have to be much smaller than K for $\log(K-Q) \approx \log K$, Equation (3.63) simplifies to:

$$Q \log B < Q(\log(K+N) - \log K) \tag{3.64}$$

[12] $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$

$$B < \frac{N+K}{K}. \tag{3.65}$$

This is an interesting result. The thesis refers to it as the the *imbalance threshold.* The final result is independent of Q, to the extent that the approximation holds. If B exceeds $\frac{N+K}{K}$, then the expected bottleneck queue size grows sharply. Or, when the number of customers $K > \frac{N}{B-1}$, the queue size on the bottleneck server will grow sharply. This is consistent with Figure 3-5; for example, the graph shows a sharp increase in expected queue size for $B = 1.1$ at the predicted value of

$$K/N > \frac{1}{B-1} \tag{3.66}$$

$$K/N > \frac{1}{0.1} \tag{3.67}$$

$$K/N > 10 \tag{3.68}$$

This result is confirmed by the previous figures, and by simulation, but I do not have a good qualitative explanation why this threshold exists, and why it exists at this value. I suspect that there is some important relationship among the variance in the service times at the processors, the amount of clumping in the configurations, and the imbalance threshold. Consider the case of a network of servers with deterministic service time distributions (i.e., they all serve in constant time). In this case, the amount of clumping in the equilibrium expected customer configurations will probably be smaller than the amount of clumping observed here for the exponential distribution. Because the clumping is less in the deterministic system, there probably will be fewer processors which have empty queues. The flow rate onto the bottleneck processor will probably be heavier. Therefore, the system will probably be more sensitive to smaller degrees of imbalance.

3.4.3 Allowed Degree of Imbalance (Careful Analysis)

The analysis in Section 3.4.2 determines a threshold of imbalance by comparing the probability of a single "good" state (with the size of the queue on the bottleneck server empty) to the probability of a single "bad" state (where the queue size on the bottleneck server is Q). It is possible to perform the analysis comparing the probability of all "good" state to the probability of all "bad" states, as follows. The conclusions of this analysis are consistent with the "uncareful" conclusion (Equation (3.65)) in the previous section.

A single server is a bottleneck with imbalance B. From Equation (3.6) the normalization constant $G(K)$ in this situation is :

$$G(K) = \sum_{i=0}^{K} B^i \binom{N+K-2-i}{N-2}. \tag{3.69}$$

The good states have probability proportional to the sum of the terms above where $i \leq Q$. The bad states have probability to the sum of the terms where $i > Q$. The threshold corresponds to the conditions on N, K, Q, and B such that

$$\mathrm{P(good)} > A\mathrm{P(bad)}.$$

In the above equation, A is an "assurance factor"; assume $A > 1$.

For a lower bound on P(good):[13]

$$\begin{aligned}
\mathrm{P(good)}G(K) &= \sum_{i=0}^{Q} B^i \binom{N+K-2-i}{N-2} && (3.70)\\
&= \binom{N+K-2}{N-2}\left[1 + B\frac{K}{N+K-2} + B^2\frac{K^{\underline{2}}}{(N+K-2)^{\underline{2}}} + \cdots\right.\\
&\qquad \left. + B^Q\frac{K^{\underline{Q}}}{(N+K-2)^{\underline{Q}}}\right] && (3.71)\\
&> \binom{N+K-2}{N-2}\sum_{i=0}^{Q}\left(B\frac{K-Q+1}{N+K-Q-1}\right)^i && (3.72)\\
&= \binom{N+K-2}{N-2}\left[\frac{\left(B\frac{K+Q-1}{N+K-Q-1}\right)^{Q+1} - 1}{\left(B\frac{K+Q-1}{N+K-Q-1}\right) - 1}\right] && (3.73)
\end{aligned}$$

This bounds the exponentially modulated binomial series by a geometric series.

An upper bound on P(bad) can be similarly obtained:

$$\begin{aligned}
\mathrm{P(bad)}G(K) &= \sum_{i=Q+1}^{K} B^i \binom{N+K-2-i}{N-2} && (3.74)\\
&= \binom{N+K-2}{N-2}\left[B^{Q+1}\frac{K^{\underline{Q+1}}}{(N+K-2)^{\underline{Q+1}}} + B^{Q+2}\frac{K^{\underline{Q+2}}}{(N+K-2)^{\underline{Q+2}}} + \cdots\right.\\
&\qquad \left. + B^K\frac{K^{\underline{K}}}{(N+K-2)^{\underline{K}}}\right] && (3.75)
\end{aligned}$$

[13]The notation $x^{\underline{y}}$ means $x(x-1)(x-2)\cdots(x-y+1)$.

$$= \binom{N+K-2}{N-2} B^{Q+1} \frac{K^{\underline{Q+1}}}{(N+K-2)^{\underline{Q+1}}} \left[1 + B \frac{K-Q-1}{N+K-Q-3} \right.$$
$$\left. + B^2 \frac{(K-Q-1)^{\underline{2}}}{(N+K-Q-3)^{\underline{2}}} + \cdots + B^{K-Q-1} \frac{(K-Q-1)^{\underline{K-Q-1}}}{(N+K-Q-3)^{\underline{K-Q-1}}} \right] \quad (3.76)$$
$$< \binom{N+K-2}{N-2} \left(B \frac{K}{N+K-2} \right)^{Q+1} \sum_{i=0}^{K-Q-1} \left(B \frac{K-Q+1}{N+K-Q-1} \right)^i \quad (3.77)$$
$$= \binom{N+K-2}{N-2} \left(B \frac{K}{N+K-2} \right)^{Q+1} \left[\frac{\left(B \frac{K-Q+1}{N+K-Q-1} \right)^{K-Q} - 1}{\left(B \frac{K-Q+1}{N+K-Q-1} \right) - 1} \right] \quad (3.78)$$

In the above analysis, the term chosen to make an upper bound is the same as the one chosen in Equation (3.72) so that the solved geometric sums match in Equations (3.73) and (3.78) match.

Now, solving for

$$\mathrm{P(good)} > A\mathrm{P(bad)} \quad (3.79)$$

There are two cases, depending on the sign of

$$\left(B \frac{K-Q+1}{N+K-Q+1} \right) - 1 \quad (3.80)$$

First case: assume $B > \frac{N+K-Q+1}{K-Q+1}$; therefore, the sign is positive. In this case, Equation (3.79) is implied true by (cancelling various terms and dropping the 1's in a number of places):

$$\left(B \frac{K-Q}{N+K-Q} \right)^Q - 1 \;>\; A \left(B \frac{K}{N+K} \right)^Q \left[\left(B \frac{K-Q}{N+K-Q} \right)^{K-Q} - 1 \right] \quad (3.81)$$
$$\left(B \frac{K-Q}{N+K-Q} \right)^Q \;>\; A \left(B \frac{K}{N+K} \right)^Q \left[\left(B \frac{K-Q}{N+K-Q} \right)^{K-Q} \right] \quad (3.82)$$

Note, however, that Equation (3.82) cannot be true because (even allowing $A = 1$),

$$\frac{K}{N+K} > \frac{K-Q}{N+K-Q}.$$

Therefore it contradicts the assumption in the first case.

Second case: assume $B < \frac{N+K-Q+1}{K-Q+1}$; therefore, the sign of Equation (3.80) is negative. In this case, Equation (3.79) is implied true by an equation identical to Equation (3.81) except that the relation is reversed:

$$\left(B\frac{K-Q}{N+K-Q}\right)^{Q}-1 \;<\; A\left(B\frac{K}{N+K}\right)^{Q}\left[\left(B\frac{K-Q}{N+K-Q}\right)^{K-Q}-1\right] \tag{3.83}$$

$$-1 \;<\; -A\left(B\frac{K}{N+K}\right)^{Q} \tag{3.84}$$

$$B \;<\; \frac{N+K}{K}A^{\frac{-1}{Q}} \tag{3.85}$$

The simplification leading to Equation (3.84) comes from the assumption; $B\frac{K-Q}{N+K-Q}$ is less than 1; raised to a power it becomes negligible. This result, Equation (3.85) is consistent with (and less strict) than the assumption.

Also, it matches the previously derived threshold (Equation (3.65)) based on simple assumptions and improves that result by including the assurance factor A.

Bounding the assurance factor:

$$A < \left(\frac{N+K}{BK}\right)^{Q} \tag{3.86}$$

The derivation of this equation assumed that B was below threshold. This upper bound on A is not as useful as would be a lower bound; note, though that this upper bound increases exponentially with queue size.

3.4.4 Multiple Bottleneck Servers

When more than one server is a bottleneck, then the excess traffic which would have accumulated on the bottleneck server spreads over the two bottleneck servers. For example, if $x_1 = B, x_2 = B, x_3 = 1, \ldots x_N = 1$, then servers 1 and 2 will divide the excess traffic.

In the J-Machine, the processors near the center of the 3D mesh were slower than the others, and consequently suffer more overflows (see Chapter 4). Because there are several equivalent processors, they divide the traffic evenly, reducing the severity of the bottleneck. It might be possible to do an estimate, similar to the one in the previous section, for the allowed degree of imbalance in the case in which there are multiple bottleneck servers. Intuition is that this would not be independent of Q.

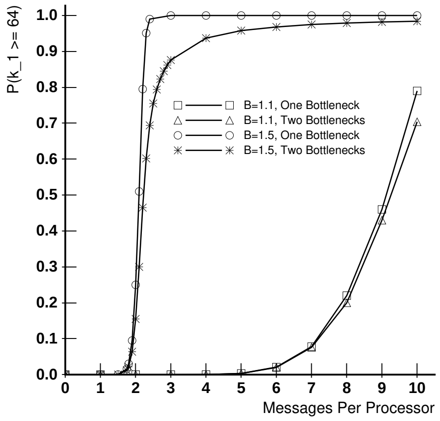


Figure 3-6: Equilibrium $\mathrm{P}(k_1 \geq 64)$ for 512 servers with one or two bottleneck servers that are B times slower than all other servers. With two bottleneck servers, we see the same threshold, but slightly smaller probability of overflow.

3.4.5 Imbalanced Overflow Residency and Rate

This section estimates the overflow rate and overflow residency in a manner that is similar to the analysis for the balanced case in Sections 3.3.3 and 3.3.5.

Residency

Because the system is imbalanced, the probability that the system is in a non-overflowed state cannot be computed as a state ratio, as was done in Equation (3.17). In order to estimate the probability that the system is overflowed, this analysis needs an additional assumption: that the overflow mostly occurs only on the server that is the bottleneck. With this assumption, estimating the probability that the system is overflowed becomes the problem of estimating the probability that the bottleneck server is overflowed. From Buzen [Buz73]:

$$\mathrm{P}(k_1 \geq Q) = {x_1}^Q \left(\frac{G(K-Q)}{G(K)}\right). \tag{3.87}$$

The value $G(K-Q)$ is the normalization constant (Equation (3.7)) that one would obtain for the same system of servers, but with only $K-Q$ customers.

Figure 3-6 plots Equation (3.87) for a 512-server system with queues holding 64 elements versus the number of customers per server. There are two curves: one for degrees of imbalance $B = 1.1$ and another with $B = 1.5$.[14] As expected, the probability of finding the system in a state with server number 1 overflowed increases sharply at approximately the same point as shown in Figure 3-5 and predicted by Equation (3.65). For example, for $B = 1.5$ the probability of being in an overflowed state increases sharply at 2 customers per server, the same as in Figure 3-5.

Figure 3-6, which describes an imbalanced system, is comparable to Figure 3-1 which describes a balanced system. Figure 3-6 plots the probability of finding the system in an overflowed state rather than the probability of finding the system in a non-overflowed state. Also, Figure 3-6 gives the probability that only one of the servers is overflowed, while Figure 3-1 considers all of the servers in the system. The balanced case starts transitioning from being completely non-overflowed with 6 customers per server and becomes mostly overflowed with 13 customers per processor. With $B = 1.5$, Figure 3-6 shows the system making this transition much earlier and much faster, centered at 2 customers per server. With $B = 1.1$, the transition occurs at around the same spot as the balanced case. One might expect that with an even lighter degree of imbalances (such as $B = 1.001$) Equation (3.87) would predict a transition at a much larger number of customers per processor. However, in this case the assumption that led to Equation (3.87), that overflows are only likely on the bottleneck server, does not hold. The balanced model would predict overflows much earlier.

[14] Figure 3-6 also plots the overflow residency on server $i = 1$ when there is an additional bottleneck server with the same degree of imbalance (i.e., $x_1 = B, x_2 = B, x_3 = 1, \ldots, x_N = 1$). The figure shows that the overflow residency on server number 1 is lower; this makes sense because one would expect that traffic accumulation on server number 2 would reduce the traffic accumulation on server number 1. However, this reduction is only slight, and it is interesting that the two-bottleneck residency tracks the one-bottleneck residency so closely.

Rate

It is also possible to estimate the rate of overflows on the bottleneck server, starting with Equation (3.14):

$$\begin{aligned}
\lambda_{\text{Overflow}} &\approx \sum_{\{\vec{k}:k_1=Q\}} \mathrm{P}(\vec{k}) \sum_{j=1}^{N} (k_j > 0)(j \neq 1)\mu_j r_{j1} && (3.88)\\
&\approx \sum_{\{\vec{k}:k_1=Q\}} \mathrm{P}(\vec{k})\mu && (3.89)\\
&\approx \mu \mathrm{P}(k_1 = Q). && (3.90)
\end{aligned}$$

Equation (3.88) assumes that the only significant terms in Equation (3.14) are those where the bottleneck server is near overflow. Equation (3.89) only considers the case where the load is such that $\mu_j = \mu$ and $r_{j1} = 1/N$ for customer movements from non-bottleneck servers to the bottleneck server, but only for $j \geq 2$, because server $j = 1$ is a bottleneck. This simplification is similar to my estimate for the uniform load case. Equation (3.90) is an algebraic simplification that results in the use of a value, $\mathrm{P}(k_1 = Q)$ which can be calculated using [Buz73]. Therefore,

$$\frac{\lambda_{\text{Overflow}}}{\mu} \approx \mathrm{P}(k_1 = Q). \tag{3.91}$$

Figure 3-7 plots Equation (3.91) and includes, for reference, the plot for the balanced case from Figure 3-2. As with a balanced load, the rate of overflows with an imbalanced load does not increase indefinitely with load, but instead reaches a peak and drops. This reflects the fact that eventually the system spends all of its time in the overflowed states, and makes few transitions from non-overflowed states.

As one would expect, for severe imbalance (in this case $B = 1.5$), the rate of overflows peaks rapidly.

For the moderate imbalance case, the rate of overflows on server number 1 stays below (though tracks closely) the rate of overflows on all servers for the balanced network. Partly, the rate for the imbalanced case is below the balanced case because this imbalanced estimate neglects the rate of overflows on the non-bottleneck servers (the simplification leading to Equation (3.88)). This suggests that if the imbalance is light, then an estimate based on the balanced overflow rate is a better approximation.

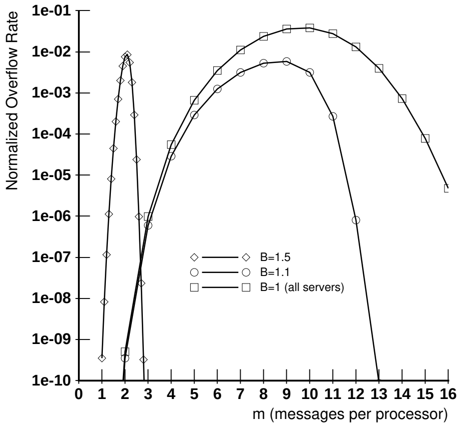


Figure 3-7: Equilibrium $\frac{\lambda_{\text{Overflow}}}{\mu}$ for imbalanced 512 server system with queues holding 64 elements. Logarithmic scale. Curves are shown for $B = 1.5$ (severe imbalance) and $B = 1.1$ (moderate imbalance). For reference, the curve for the balanced and uniform case (from Figure 3-2) is included and labeled $B = 1$; note, though, that this reference includes the rate of overflow over all servers, while the $B = 1.5$ and $B = 1.1$ case only consider the rate of overflows for the single bottleneck server.

As mentioned above, past the threshold where in Figure 3-6 the overflow residency goes to 1 (the imbalance threshold), the overflow rate drops sharply. Figure 3-7 shows the overflow rate peaking for a lower number of customers: near the point where the Figure 3-6 shows the overflow residency is 0.5. This seems intuitively reasonable: at the point where the system is spending half of its time overflowed, the rate of overflows is highest.

3.4.6 Imbalanced Workload: Conclusion

Beyond a particular threshold of imbalance, an imbalanced queuing network risks a large accumulation of customers on the bottleneck servers. Estimates of the overflow residency and rate can be made by ignoring all but the bottleneck server; these estimates show a sharp increase in the residency and rate at the threshold.

3.5 Discussion

The following subsections discuss limitations of this queuing model.

3.5.1 Open Queuing Networks

This chapter has considered closed queueing networks, where customers do not enter or leave the network. Formulating the model as an open network, with customer arrivals and departures, gives straightforward expressions for the probability that any given server's queue will exceed a threshold [Kle75]. Although this formulation cannot be used for the Snakes benchmark (the benchmark that is run on the J-Machine and measured in the next chapter), for other applications to which it can be applied the calculations are simpler.

An open queuing network is distinguished from a closed network by its allowance for customers to enter the network from an external source and to depart to an external sink. Formally, the source and sink are designated server number 0. Customers arrive to the network from server 0 with rate γ, and choose to go to one of the network servers, number i, with probability r_{0i}. A customer served at server i leaves the system with probability r_{i0}. Again, all service distributions and external arrival distributions are exponential. The routing matrix elements r_{ij} and external arrival rate determine an arrival rate at each server

$$\lambda_i = \gamma r_{0i} + \sum_{j=1}^{N} \gamma_j r_{ji}. \tag{3.92}$$

Jackson's theorem [Kle75] says that the probability distribution for the configurations of customers over servers (now an infinite space, for the number of customers in the system is unbounded) is the product of individual distributions for each server, where each server acts like a simple M/M/1 server with arrival rate λ_i and the given service rate μ_i.

Open queuing networks are also distinct from closed queuing networks in having a notion of stability. An open queuing network is defined to be stable if the queues do not grow without bound. This is guaranteed if

$$\lambda_i < \mu_i \tag{3.93}$$

for all servers. For closed queuing networks, the number of customers is fixed, so they are always stable by this definition.

Snakes is better modeled with a closed queuing network because the number of messages on the system is fixed. A closed queuing network also is a better model for a shared memory system where the number of outstanding memory requests per processor is fixed. For other J-machine applications, the open queuing network model is preferable. For example, an open model is preferable for a program that handled requests from an external source arriving via the network channels on the surface of the cube. It is preferable for a program that spontaneously generates new messages from the processors. With these applications, the question of stability is critical, because it defines the maximum applied message rate that the J-machine can handle before queues start to overflow. In practice, this question is also particularly interesting because Equation (3.93) may not (and probably does not) hold if the network is not a "pure" exponential open queuing network (and the J-machine is surely not pure). Algorithms for determining the stability of queuing networks with varied service disciplines are an active area of operations research [BGT95, JGG+95, DW94], and they are not addressed in this thesis.

The question that *is* addressed in this thesis is the behavior of the workload with respect to small, local capacity constraints imposed by the fixed size hardware message queues: how often do queue overflows occur?

This question is actually easier to answer for an open queuing network than it is for a closed queuing network. Because the individual servers in the open Markovian network act as M/M/1 servers (Jackson's Theorem), one can use the simple birth/death process, with birth rate λ_i and death rate μ_i, to get the probability that a server will be found overflowed. For each server, the overflow residency (the probability that server i will be found in an overflowed state) is

$$\mathrm{P}(k_i > Q) = \rho_i^{Q+1} \tag{3.94}$$

where

$$\rho_i = \frac{\lambda_i}{\mu_i}. \tag{3.95}$$

Because the servers are independent, the probability of no overflow is

$$\mathrm{P}(A_{\text{nov}}) = \prod_{i=1}^{N}(1 - \rho_i^{Q+1}). \tag{3.96}$$

The rate of overflow on server i is

$$\lambda_{\text{Overflow}\,i} = \lambda_i(1 - \rho_i){\rho_i}^Q. \tag{3.97}$$

Because these overflows are independent Poisson processes, their rates sum to an overflow rate of $\lambda_{\text{Overflow}} = \sum \lambda_{\text{Overflow}\,i}$.

These expressions are nice and simple, but again, they do not apply to the closed queuing network because the fixed number customers renders the individual queue size processes dependent.[15] An attempt to "force fit" the open model to the Snakes process would fail, because the arrival rates λ_i would be undefined, due to the lack of the open model's external driving rate γ. The solution to the closed model, Equation (3.6), is more difficult to use but accounts for the closed model's self-equilibration or self-regulation.

One final note is that if the network is sufficiently imbalanced, with a single bottleneck server, and if it has a large number of customers, then the closed queuing network starts to behave like an open queuing network where the bottleneck server is the external source. In that case, Equation (3.97) is a good approximation for the overflow rate on the non-bottleneck servers. Equation (3.91), derived for the closed imbalanced load, can be used to estimate the rate of overflows on the bottleneck server.

[15]In the closed queueing network, the output processes are also not exponential. This can be illustrated by a simple example in which a single customer bounces back and forth between two servers. The distribution of departure times from the servers is the sum of two exponential distributions: an Erlang distribution. The output processes in the open queueing network are exponential.

3.5.2 Exponential vs. Deterministic Service Distribution

This chapter has used the solution to a closed queuing network of servers with an exponential service time distribution to estimate the overflow rate and residency. For the Snakes benchmark, the service time distribution is for the most part deterministic; that is, the handlers for the Snakes message all run in a constant time. There is some variation for send faults and memory references.

If the service time is deterministic, then the system is no longer a continuous time Markovian process. The state transition probabilities depend not only on the current state, which is consistent with a Markovian process, but also on the amount of time that the system has been in that state, which is not consistent. Not only does the system not have a product form solution, but it also is not ergodic.

BCMP [BCMP75], the "power tool" of product form queuing network solutions, handles a number of different service time distributions, but it does not handle the deterministic service distribution. BCMP handles service distributions that have a property that tends to de-correlate the traffic within the network, called the M→M property [Mun72]. The deterministic service time distribution does not have this property. In fact, it tends to correlate traffic actively.

Balanced Traffic

Harchol-Balter and Wolfe proved that the waiting time in a network of exponential servers is an upper bound, though not tight, on the waiting time on a network of deterministic servers [HBW95]. The key to their proof was the observation that the deterministic network does not form "clumps" of traffic as the exponential network does. Their proof was for an open queueing network.

Less is known about deterministic servers in a closed network. One can form a closed Markov process based upon the deterministic closed network. However, the balance equation for the network is much more challenging to write, because one must consider simultaneous transitions. This Markovian process is ergodic, but no longer are the probabilities of all states equal. Instead, states in which the customers are more dispersed can have higher probability.[16] For this reason, and for the case with deterministic servers, overflow rate and residency estimates based on Equation (3.14) will be high when using the solution to the closed Markovian network. Nevertheless, for a "real" application with a spread in service time, the exponential server model might be acceptable.

In Snakes, asynchronous events, like SEND faults and EARLY faults spread the service time. In this sense, an exponential model might be a more accurate model for the service time of processors.

Imbalanced Traffic

When the traffic is imbalanced on the machine, the equilibrium mean queue size on the bottleneck server can be much larger with deterministic service times, than with exponential service times. This appears to be due to a tendency of the deterministic network to have higher overall throughput, which increases the arrival rate at the bottleneck server, increasing its queue size. In a test simulation with $N = 512$, $K = 2560$, and $B = 1.25$, the observed mean queue size for the bottleneck server is 512, which is consistent with the analytical predictions using Buzen's algorithm. However, when the service time is deterministic (with a small bit of "noise" added), the observed mean queue size on the bottleneck server is approximately 1200. And, it reaches this mean value quickly.

Asymptotic Overflow Probabilities

There is a potential danger of error when using the Markovian network to estimate overflow probabilities asymptotically with N. The Markovian balance equation (Equation (3.4)) uses the Poisson assumption that two customers never move simultaneously. For large machines (large N) this assumption is challenged. With fine-grained applications, where the average service time $1/\mu$ approaches the clock cycle time, the probability of simultaneous transitions increases. Still, this should be only a small effect.

[16]This can be demonstrated by analyzing a simple network of three deterministic servers.

3.5.3 Analysis of Request/Reply Variation

Chapters 4 through 6 observe that the Request/Reply type of workload (in which messages are sent out randomly but return to a home processor, as in a shared memory machine) has fewer overflows and smaller queues than does the completely unconstrained Snakes workload. The Request/Reply workload is able to tolerate a much higher load on the machine (many more outstanding messages and more imbalance) without getting overflows.

Intuitively, this seems like a reasonable observation, but explaining why it is true using queueing theory is challenging. Even though it is a closed workload (each processor has a fixed number of messages) the routing behavior of Request/Reply cannot be captured in the classic closed queueing model of Section 3.1. To capture the routing behavior of Request/Reply, one must use a more elaborate closed model. This more elaborate model has a product form solution for the equilibrium probability distribution. Unfortunately, it is not structured in a way that allows one to compute overflow rates by counting states, as in Section 3.3.2.

The key to representing Request/Reply in a queueing model is to use multiple customer classes to represent the different routing patterns of the customers at each processor.

With multiple classes, the closed queueing network is formulated as follows. There are N servers and C classes of customers. Each server i services with an exponential distribution for all classes at rate μ_i. After a class c customer completes service at server i, it moves to server j as a class d customer with probability $r_{i,c:j,d}$. Because the network is closed, $1 = \sum_{j=1..N,d=1..C} r_{i,c:j,d}$.

This multiple class network has a product form solution [BCMP75]. The states of the process are designated $Y = (\vec{y_1}, \vec{y_2}, \ldots, \vec{y_N})$ where $\vec{y_i} = (k_{i1}, k_{i2}, \ldots, k_{iC})$ and k_{ic} is the number of class c customers at server i in that state. The probability of a feasible state Y is

$$\mathrm{P}(Y) = \frac{1}{G} g_1(y_1) g_2(y_2) \cdots g_N(y_N). \tag{3.98}$$

G is a normalizing constant so that the probabilities over all of the feasible states sum to 1. The individual terms of the product are each of this form:

$$g_i(y_i) = k_i! \left\{ \prod_{c=1}^{C} \left(\frac{1}{k_{ic}!}\right) [e_{ic}]^{k_{ic}} \right\} \left(\frac{1}{\mu_i}\right)^{k_i} \tag{3.99}$$

where $k_i = \sum_c k_i c$. The values e_{ir} are any set of solutions to the equation:

$$\sum_{(i,c)} e_{ic} r_{i,c:j,d} = e_{jd}. \tag{3.100}$$

Note the similarity of Equations (3.98) through (3.100) to the Gordon and Newell solution (Equation (3.6)); when $C = 1$, these equations reduce to the Gordon and Newell solution.

Adapting these equations to the case of a Request/Reply traffic pattern is easy. Assuming that processors do not send requests to themselves, the transition probabilities are as follows:

$$r_{i,c:j,d} = \begin{cases} 0 & c \neq d \\ \frac{1}{N-1} & c = d \wedge c = i \wedge i \neq j \\ 1 & i \neq c \wedge j = c \wedge c = d \\ 0 & \text{otherwise} \end{cases} \tag{3.101}$$

A solution to Equation (3.100) for this transition matrix is:

$$e_{ic} = \begin{cases} N-1 & i = c \\ 1 & i \neq c \end{cases} \tag{3.102}$$

These equations formulate the Request/Reply workload workload in terms of a closed multi-class queueing network, and give the straightforward solution to the equilibrium probability distribution for configurations of customers over servers.

One can use the equilibrium probability distribution to compose expressions for the rate of overflow and overflow residency, like Equations (3.14) and (3.16). From these expressions and for a small system, a rate of overflow could be calculated. However, even for moderately larger systems, the calculation becomes difficult. As with the single class workload, the number of states becomes large, so it's necessary to simplify or approximate this expression. In fact, the size of the space of feasible states Y is *larger* for the multiclass case than for the single class case, because the states Y distinguish between the classes of customers at the node.

Beyond this point, things get trickier. Even for the balanced workload, the problem of determining state probabilities is harder than just counting states, because not all states are equal: some states are weighted more heavily in Equation (3.99) than others.

Another immediate problem is that the state probabilities in Equation (3.98) are for states which are more elaborately specified than is necessary for the purposes here: if one cares

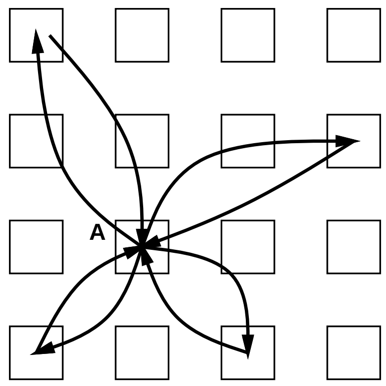


Figure 3-8: Traffic flow in Request/Reply. Eight message hops. Processor A sends out to random processors, but the message always returns home.

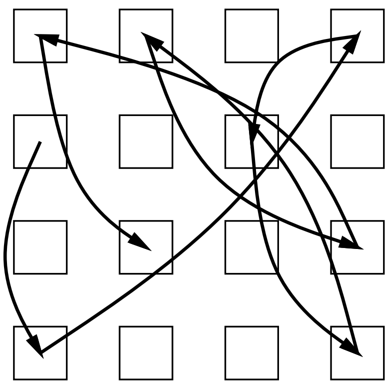


Figure 3-9: Traffic flow in snakes. The message makes a random walk of 8 hops on the grid.

only about overflow, all that is necessary is the total number of customers at each node. As Basket et. al. [BCMP75] note, the product form solution is ideal for getting marginal distributions (i.e. excluding so much state information) because one can simplify the $g_i()$ terms independently. However, a strategy for simplifying Equation (3.99) proved resistant to analysis. Perhaps using a recurrence, like Buzen's algorithm, will lead to a simplification.

The goal of doing this would be to better understand why the closed workload behaves so much better in simulation when using Request/Reply. Intuitively, one expects that returning to the home would tend to act to spread the traffic. But note that every feasible configuration in the one-class Snakes case is feasible in the multiclass case (e.g. it is possible in both systems to have all of the traffic aggregated on one node).

3.5.4 Other Dynamic Questions

There are other limitations of this model which merit mention.

Overflows change dynamics

In this model, processors do not slow when an overflow occurs, as they do in the J-machine. The model assumes that the overflows have no effect on the rates and transition probabilities. If overflowed servers get slower, the overflow residency is higher because the overflowed servers become even more of a bottleneck – and as such, would accumulate customers even more rapidly. One would expect this effect to push the system strongly into overflowed states. This behavior might be modeled using queuing theory techniques for state-dependent service rates. This is not explored in this thesis.

Overflows can jam the network

On the J-machine with random traffic patterns, queue overflows, quickly jam the communication network and stop all other servers, by blocking incoming traffic to the server. This blocking effect tends to prevent avalanches of traffic to the blocked server, though this is not guaranteed for all traffic patterns. For example, a ring pattern which was placed on the mesh in a manner that avoided contention would not benefit from this throttling behavior. This is discussed further in Section 4.4. It may be possible to design a closed system consisting of two coupled exponential queueing network models, one with a balanced workload, the other with the slow server due to bottleneck. This is left as future work.

Approach to equilibrium

This model does not address the rate of approach to equilibrium. In the actual benchmarks that we run on the J-Machine in the next chapter, the machine starts with the workload evenly placed across all of the processors. As the benchmark runs, the traffic redistributes according to the random processes in the machine. This happens at some rate determined by the degree of imbalance; the model in this chapter only studies the equilibrium probability distribution, and not the approach to that distribution. This is also of interest.

3.6 Conclusion

The model assumes that the service distribution for servers is exponential which results in a product form solution (Equation (3.6)) for the probability of configurations of customers over servers and defines exactly the overflow rate and residency for the model. The challenge that this chapter overcomes is eliminating the summations over large state spaces in order to make numerical estimates for these exactly defined values.

How large should the queues on the servers of the J-machine be? With the cost of overflows high, the queues should be made large enough so that overflows should be very rare. Section 3.3.6 indicates that if the workload is balanced, the total queue capacity should be able to hold at least ten times the number of customers on the system to hold the normalized overflow rate below 1%.

If the workload is imbalanced, the degree of imbalance determines the required size of the queues. For slight imbalances ("slight" being below the threshold in Equation (3.65)), the results of Section 3.4.6 suggest that queue sizes similar to that of the balanced case are reasonable. Beyond the threshold, though, the bottleneck server's queue should be able to hold all K customers.

These results suggest that some balancing process is required to address the case of the imbalanced closed workload. One simplistic way would be to adjust the service rates and transition probabilities to make the workload balanced. In this case, the workload then acts as a balanced workload, and the 10x rule mentioned above would apply.

A more active balancing process would note the size of queues that approach some threshold and actively work to shrink them, by diverting traffic away. This chapter does not consider this kind of active load balancing at all; instead, it investigates the queue sizes needed to support the statistical load balancing technique of randomly spreading the customers around – which is, in fact, the technique implemented by the J-Machine's COSMOS runtime when it selects a node to handle a method application to a distributed object or an unanchored object type such as an integer.

The message queues on the J-machine are up to 512 words long, and the minimum message size is 4 words. This means that each message queue can hold 128 customers. The analysis in this chapter suggests, then, that the COSMOS statistical load balancing algorithm will only be successful if the number of messages on the system remains below 12 messages per processor for four-word messages. In this case, "success" is a 1% normalized overflow rate.

This limit is optimistic. In practice blocking effects within the network unbalance the workload. Furthermore, the cost of overflow handling on the J-Machine is so high that a 1% normalized overflow rate is unacceptable. This sets a much lower limit on the allowed number of messages on the machine. Chapter 4 demonstrates this limit.

Chapter 4

J-Machine Experiments

This chapter reports measurements of the performance of the J-Machine running a synthetic workload called "Snakes," a closed random heavy message-passing workload. In contrast to the previous chapter, which considered a mathematically tractable and idealized queuing network, this chapter considers the actual hardware running in the lab with all of its glorious complexity.

The chapter makes the following observations:

- At heavy fine-grained load, the J-Machine's performance drops sharply, due to queue overflows.
- The point at which the machine becomes imbalanced is when the network saturates.
- High variance in the performance of the machine at this load point is due to variance in the time to reach equilibrium and hit the queue capacity barriers.
- Overflows do not avalanche. This is an inadvertent consequence of the network traffic locking-up the machine when an overflow occurs.
- The overflow state is effectively "sticky." That is, once one processor becomes overflowed, the machine is biased to stay overflowed.
- Though the synthetic workload is itself uniform, the machine imbalances it; this is one of the reasons that we see overflows.
- The overflow behavior is strongly determined by the number of tasks on the system.
- Overflows occur exclusively on the center processors in the machine.

- The J-Machine is unable to get good performance on this workload with long messages.
- A simple load balancing scheme, called Request/Reply, in which the tasks return home every other excursion results in a well-behaved workload.

The chapter is structured as follows. Section 4.1 describes the synthetic workload, workload parameters, and derived metrics and parameters, and describes the Snakes benchmark. Section 4.2 describes a typical measurement and its sharp drop in performance. Section 4.3 describes how one can conclude that queue overflows cause the sharp drop in performance. Section 4.4 observes that the overflows do not avalanche. Section 4.5 presents a graph showing snapshots of the maximum queue size on the system over time, starting at the moment when the workload is started, and concludes that the observed variance in performance stems from variance in the time for the system to move away from the balanced configuration and allow a queue to hit the capacity threshold. Section 4.6 presents the result of adjusting the queue size on the MDPs, demonstrating that queue size strongly influences performance. Section 4.7 shows that overflows occur exclusively in the center of the machine. Section 4.8 presents the result of varying the length of messages in the workload. It shows that for this workload, the processor is unable to get good performance with long messages. Section 4.9 shows the result of varying the number of messages in the system. Queue overflow behavior becomes a large problem when the number of messages per processor reaches five per processor. Section 4.10 presents the result of a simple experiment where the messages return home after every random excursion. When the workload follows this routing pattern, it is extremely well-behaved, and exhibits few queue overflows. This proves to be a simple way to load balance, though at the cost of doubling the amount of message traffic. Finally, Section 4.11 discusses the results and closes the chapter.

4.1 Snakes

This chapter evaluates how the J-Machine runs a program called Snakes that creates a heavy closed message passing workload. The name alludes to wormhole routing [Dal87], the algorithm in the J-Machine that delivers the messages between processors. Messages in the J-Machine route like worms through holes in the sense that they have a head that opens a channel of pipeline stages between processors, and a tail that closes that channel after the message has passed. In Snakes, the entities live longer than a J-Machine message: they are tasks that exist for the duration of the workload and move around the machine. They are larger animals than worms, and hence are called snakes.[1]

In other words, the "snakes" are the tasks of a random workload on the machine. The workload is nearly closed, in the sense that it runs for a long time with tasks only exiting from the system at the end of the benchmark.

When it starts, the Snakes program creates a fixed number of tasks and distributes them evenly among the J-Machine's processors. Each task queues on its current processor in FIFO order. When it reaches the head of the queue, it invokes a short task handler that randomly selects a next destination processor. The task handler sends the snake to selected processor via the wormhole network and then suspends.

The primitive sequence:

- take the task from the head of the local queue,
- invoke a handler,
- send new messages, and then
- suspend

is a hardware-implemented architectural primitive that is the distinguishing feature of the J-Machine, and the agility with which the J-Machine performs this is what is intended to make it ideal for fine-grained applications. This thesis is not testing the hypothesis that agility is good for fine-grained applications. The thesis is, however, detailing the consequences for heavy-load throughput of the architecture that provides this agility. Snakes does no more than bounce messages around the machine and, as such, serves only as a basis for measuring throughput. To the extent that throughput is important for fine grained applications, then, this gives some prediction of the performance of the J-Machine running fine-grained applications.

[1]Interestingly, a snake can wormhole not only through the network but also through the processor, in the sense that the MDP will initiate a task before a message has completely arrived at the processor, and even allow new messages to be injected before the whole message has arrived.

Parameter	Description	Unit
N	Number of processors	
m	Number of snake tasks created, per processor, at start	
L	Number of hops that each snake performs	tasks
p	Message payload size	words
h	Task handler duration	cycles
Q	Processor queue size	words
c	Clock rate of the processors	cycles/sec

Table 4.1: Parameters used to describe a Snakes workload

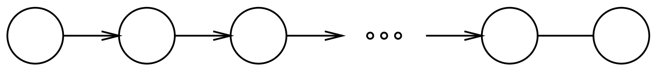


Figure 4-1: Task graph for a single Snake. Each task can take place on a different processor.

The task graph for the path of a single snake appears in Figure 4-1. The arrows represent messages sent on the hardware to move the snake from processor to processor. Many snakes run in parallel, with their creation and destruction (after each snake has managed to make a particular number of hops) bracketed by the barrier subroutines written by Wallach and documented in [NWD93]. The larger picture is in Figure 4-2.

4.1.1 Workload Parameters

Snakes is not a single workload. The parameters given in Table 4.1 select a particular load point for a run. Furthermore, each individual run of the workload is different because the snakes' paths are random.

Rather than give an exhaustive overview of the the behavior of the J-Machine running Snakes for all of the entire large parameter space, this chapter focuses on a particular region of op-

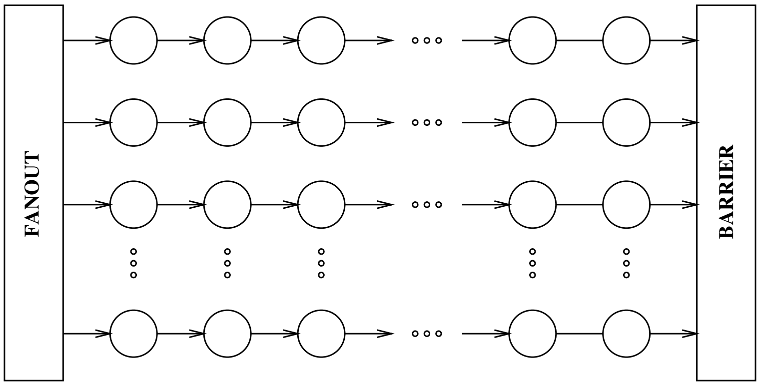


Figure 4-2: Task graph for the Snakes workload. Multiple snakes run in parallel, initiated by a fanout operation and terminated by a barrier.

eration – with many short-duration tasks – where queue overflow effects become substantial. One can argue that this is the type of load that would be presented by a fine-grained parallel application.

All measurements in this chapter are from our prototype J-Machine, with 512 processors.[2] Smaller machines are less interesting because they operate more predictably (they are harder to saturate).

The queue size of the MDPs can be configured to any size that is a power of two, up to 1024 words. Unfortunately, the design of the MDP makes it difficult for the queue overflow handler to distinguish overflowed states if the queue size is 1024 words. Therefore, the largest working queue size on an MDP is 512 words.

[2] At the time these measurements were taken, the 1024-node version of the machine was unavailable due to a power-supply failure. This doesn't substantially affect the results; we are able to saturate the bisection with a 512-processor machine.

4.1.2 Metrics

The primary metric in Snakes is the makespan,[3] denoted T. This measures the time from the initiation of the first of the tasks to the completion of the last. There is no hardware in the MDP to support timers, so the J-Machine's front-end workstation makes this time measurement, using messages sent from within the barrier code through the J-Machine's I/O interface. These messages mark the beginning and end of the run. The front-end is running UNIX, so interrupts from other processes on the front-end limit the precision of this measurement to about 100 μs.

It is possible to halt the J-Machine from the front-end – destructively, with limited time accuracy, and no repeatability – to get a snapshot of the configuration of the run at intermediate points. Section 4.5 uses this capability.

Finally, the workload and various fault handlers do count some other metrics about the run, for example, the number and the location of queue overflows.

4.1.3 Derived Parameters and Metrics

The following parameters and metrics can be derived from the primary metric T.

Average Makespan

Because there can be a large variation among different makespans for differently-seeded runs at the same load point, the measurements below are from many different runs at the same load point, using different seeds. One of the benefits of having real hardware is that it is easy to do multiple runs. The average makespan of these runs is denoted $\overline{T}$.

[3] The term *makespan* comes from the literature of scheduling research, and its use here reflects the fact that running Snakes quickly is at heart a scheduling problem. Seen in this light, this chapter examines how well the J-Machine's FIFO queueing and message delivery algorithms self-schedule this family of stochastically-generated workloads.

Bisection Utilization

Each time a snake sends a message to a successor, it chooses the destination randomly and independently. Therefore, the expected number flits of traffic transferred by Snakes across the bisection is:

$$v_{\text{bisection}} = \frac{2(p+1)mNL}{2}. \tag{4.1}$$

The factor 2 appears in the numerator because each word is two flits long. It appears in the denominator because only half the randomly-destined snakes are expected to cross the bisection.

The number of flits of bisection traffic volume available in time $\overline{T}$ is:

$$a_{\text{bisection}} = N^{\frac{2}{3}}\overline{T}c. \tag{4.2}$$

This is because the N processors of the J-machine are arranged in a 3D cube, resulting in $N^{\frac{2}{3}}$ channels crossing the bisection. Each cycle one flit can be transferred in each direction across each channel, and the clock frequency is c.

Dividing $v_{\text{bisection}}$ by $a_{\text{bisection}}$ gives estimated bisection utilization for a run of Snakes, as a function of the measured makespan $\overline{T}$:

$$u_{\text{bisection}} = \frac{(p+1)mN^{\frac{1}{3}}L}{\overline{T}c}. \tag{4.3}$$

Expected Processor-Limited Makespan

The total workload presented by Snakes is mNL tasks. Assuming that the workload is distributed evenly over processors, that those processors are working at some level of utilization $u_{\text{processor}}$, and that all processors work at the same rate of $\frac{cu_{\text{processor}}}{h}$ tasks/sec, the expected time to complete the workload in seconds is:

$$T_{\text{expected}} = \frac{mNLh}{Ncu_{\text{processor}}} = \frac{mLh}{cu_{\text{processor}}}. \tag{4.4}$$

Again, this holds only if one assumes that the workload is distributed evenly.

Expected Processor-Limited Bisection Utilization

Dividing the expected traffic volume by the expected time gives the expected traffic rate (flits/s) offered by the processors to the bisection:

$$t_{\text{offered}} = \frac{cN(1+p)u_{\text{processor}}}{h}. \tag{4.5}$$

The maximum throughput rate of the bisection is

$$cN^{\frac{2}{3}}. \tag{4.6}$$

Dividing Equation (4.5) by Equation (4.6) gives the expected processor-limited bisection utilization in terms of the handler duration:

$$u_{\text{bisection}} = \frac{N^{\frac{1}{3}}(1+p)u_{\text{processor}}}{h}. \tag{4.7}$$

Transition to network-limited domain

Noakes [NWD93, Figure 3] showed that the J-machine's network bisection saturated at some utilization $u_{\text{saturation}}$ of about 0.4. The results below also show saturation at this level; the limit seems to be related to the high penalty that the processor incurs when it takes SEND faults. This saturation level is therefore a limit on the throughput of the J-Machine. Running Snakes at some workload point, the system should not be network-limited so long as the expected bisection utilization needed is less than the saturation level:

$$u_{\text{bisection}} < u_{\text{saturation}} \tag{4.8}$$

$$\frac{N^{\frac{1}{3}}(1+p)u_{\text{processor}}}{h} < u_{\text{saturation}}. \tag{4.9}$$

In terms of the handler time h, this threshold is at:

$$\frac{N^{\frac{1}{3}}(1+p)u_{\text{processor}}}{u_{\text{saturation}}} < h. \tag{4.10}$$

4.1.4 Some Software Details

Snakes is hand-coded in MDP assembly language. This offers better performance than one can get with either of the programming systems for the J-machine. The increased performance is important to our efforts because the interprocessor communication network is difficult to saturate with short messages unless the processors are running at top speed.[4]

The assembly code for a short snake message handler is shown in Figure 4-3. The time to execute a snake message handler is 25 clock cycles for the instructions, plus a few extras cycles for prefetching. With the MDP's clock rate of $c = 10$ MHz, each snake handler runs in about 3 μs. Note, however, that the network can exert backpressure to the processor in the form of SEND faults. These send faults cause an increase the number of cycles that the processor takes to handle a message, and are overhead that decreases $u_{\mathrm{processor}}$.

The random number function that each message handler executes to determine the destination processor for its successor is nearly same as the one that COSMOS uses to apply messages to unplaced objects. This random number function is weak (all it does is add a random increment to a seed and mask to get a node address) but fast, and it is important to make handlers fast in order to pressure the bisection.[5]

The queue overflow handler constitutes a substantial portion of the snakes program text. It based on the overflow handler written by Todd Dampier for COSMOS. To use the handler for the measurements, a number of bugs were fixed and reentrancy was added.

[4]Does this mean that it unlikely that "real" J-Machine programs would have methods that operate with such short tasks? If one defines "real" programs as those produced by the CST compiler; the answer is "maybe." CST programs would incur a couple of extra cycles to do the method lookup; that overhead is not necessary in the Snakes benchmark. But CST produces relatively tight code, and if the methods are short, CST can produce a short task. On the other hand, tasks produced by the MDC compiler are much longer, because the compiler was less aggressive in optimization than was CST. However, this does not diminish the significance of my measurements. With longer tasks, some larger J-Machine would be required to saturate the bisection. So the short tasks and small machine measurements simply serve as a testbed for longer tasks and larger machine measurements.

Furthermore, the short tasks here are completely consistent with the original intended fine-grained software for the machine, that was for tasks that ran in tens of cycles. It is this low expected task length which provided the incentive for many of the other overhead-reducing features of the J-Machine.

[5]In order to get smooth measurement curves for this benchmark, it proved important to give each processor a different increment (COSMOS does not do that). When all increments are all the same, as they are in COSMOS, occasionally a couple of processors would have their random functions synchronized, and the correlation would increase the number of queue overflows.

```
Snake:
        ; Generate random next processor.
        move    [RandomSeed,a1],r2                          ;2
        add     r2,[RandomSeedInc,a1],r2                    ;2
        move    r2,[RandomSeed,a1]                          ;1
        and     r2,[RandomSeedMask,a1],r2                   ;2

        ; Check for end of snake.
        move    [1,a3],r3                                   ;2
        bz      r3,^SnakeDone                               ;1

; Clear retries so we don't get a send-fault timeout.
        move    0,r0                                        ;1
        move    r0,[Retries,a0]                             ;1

; Decrement hop counter.
        sub     r3,1,r3                                     ;1
        bz      r3,^SnakeLast                               ;1

        ; Send message to next processor.
        send    r2,0                                        ;1
        send2   [0,a3],r3,0                                 ;2
        sende   [2,a3],0                                    ;2
        move    [SnakesDid,a1],r1                           ;2
        add     r1,1,r1                                     ;1
        move    r1,[SnakesDid,a1]                           ;1
        suspend                                             ;2
```

Figure 4-3: The MDP assembly language code that handles a Snake. A random next processor is selected and a new message is sent to it (this version omits the spin loop). The right hand comment indicates the number of cycles to execute the instruction.

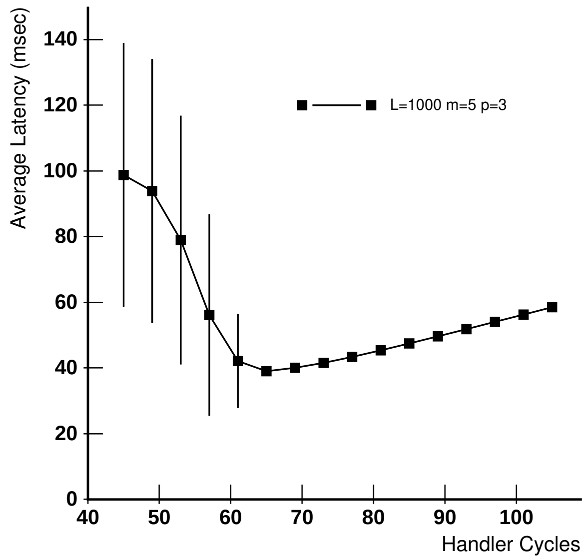


Figure 4-4: Average latency ($\overline{T}$) of the Snakes workload for an $8 \times 8 \times 8$ J-machine, varying the number of cycles spent in the message handlers, for $p = 3$ word message payloads, $L = 1000$ hops, $m = 5$ snakes per processor, and processor queue length $Q = 512$ words. Speeding the handlers (moving to the left along the cycles axis), a transition to much lower performance occurs at the point where the network would become the system bottleneck. Vertical bars show standard deviation of measurement.

4.2 Typical J-Machine Snakes Performance Measurement

This section presents Figure 4-4 a typical measurement of the performance of Snakes running on the J-Machine. To produce the plot, only the handler duration h is varied. The figure shows how $\overline{T}$ and its standard deviation vary with h.

This is interesting. This loss in performance is steep and undesirable. This loss in performance is in the very fine-grained workload region for which the J-Machine was built. Why is this happening?

This measurement uses the following workload parameters:

- The number of hops, L, is 1000. This allows plenty of time for the system to reach steady state and move away from the balanced configuration in which it starts. Although the time for the barriers at the beginning and end of the run is included in the measurement of T, making L large makes it an insignificant fraction of the time.[6]
- The number of snakes per processor, m, is 5, which means that with 512 processors, there are 2560 snakes running simultaneously.
- The message payload length p is 3. Messages also include one word for a header specifying the address of the first instruction of the message handler, so that in the message queue they consume $p + 1 = 4$ words in the 512 word queue.[7] The $m = 5$ messages per processor consume 4% of the queue storage in the machine; Section 3.3.6 suggested that overflows become prevalent at about 5% utilization.

The fixed parameters are chosen to produce a compelling demonstration of the sharp drop in performance transition moving from the processor-limited domain into the network-limited domain.

Equation (4.4) predicted that if the load was balanced and processor-limited, then the expected run time would be linear and proportional to h. This matches the right side of

[6] I verified this by making a differential measurement.

[7] When messages are en-route in the communication network, they also have a few flits prepended with the X,Y,Z coordinates of the destination processor. These flits are stripped as the message moves through the system and do not appear in the queue. We ignore them here.

Figure 4-4, with $h > 65$ cycles. But on the left side of the figure, the linear relationship does not hold, because the machine no longer is processor-limited; nor is the load balanced. In this region of operation, queue overflows are the limiting performance factor. The transition into the network limited region can be found by substituting into Equation (4.10).

$$h > \frac{N^{\frac{1}{3}}(1+p)u_{\text{processor}}}{u_{\text{saturation}}} \tag{4.11}$$

$$h > \frac{512^{\frac{1}{3}}(1+3)1.0}{0.4} \tag{4.12}$$

$$h > 80 \tag{4.13}$$

This is a bit higher than the transition point that the graph shows. This reflects the fact that processor utilization is not 1.0.

4.2.1 Typical Measurement as Bisection Utilization

Plotting the average makespan $\overline{T}$ shows performance explicitly. However, in subsequent plots within this chapter, it is occasionally preferable to show measurements as bisection utilization. This serves to check whether the particular measurements are plausible and also focuses attention on the efficiency of the utilization of the J-machine's wires to be consistent with the original J-Machine philosophy that wires are expensive.

Bisection utilization is computed from the $\overline{T}$ measurement using Equation (4.3). Figure 4-5 shows how Figure 4-4 looks when plotted this way. To the right of saturation, the relationship that appeared to be linear in Figure 4-4, predicted by Equation (4.4) is now hyperbolic. (The curvature is slight, so it appears linear in this figure). The hyperbolic relationship is consistent with Equation (4.7). The error bars are small because in this figure they show measurement confidence, rather than standard deviation.

4.3 Lower Performance is Due to Queue Overflows

Figure 4-4 showed that when the handler latency h was short, the performance dropped substantially and had a large standard deviation. This section shows that this drop is due to queue overflows.

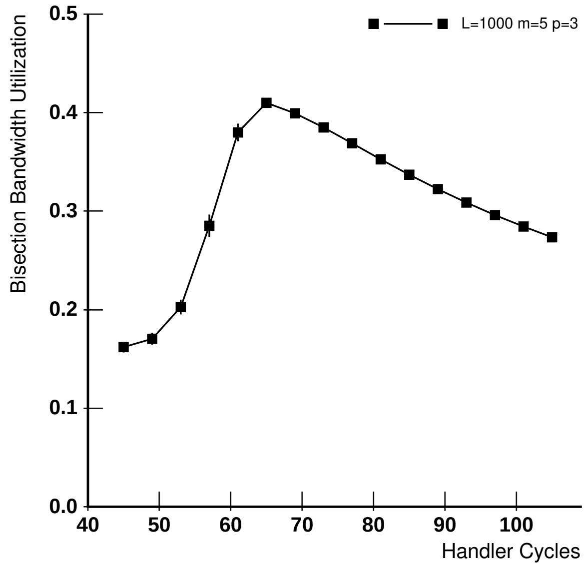


Figure 4-5: Same data as in Figure 4-4, plotted as bisection utilization. The error bars are smaller because this graph shows measurement confidence, rather than standard deviation.

The first thing that should be noted is that in all of the points in Figure 4-4 which had a low standard deviation, the system experienced no queue overflows. For all of the points which show a large standard deviation, the J-Machine experienced queue overflows.

Recall each point in the graph Figure 4-4 represents the average latency for many runs at that load point. Within these sets of runs longer latency measurements are correlated with more queue overflows. Consider the leftmost point in Figure 4-4, with $h = 45$. This point was the average of 150 different runs. Figure 4-6 plots the individual runs to show the relationship between makespan and queue overflows. The upper plot (A) is a scatter plot, and (B) is a distribution. At this load point, the system never experienced more than 3 overflows.

The first notable feature of the distribution is the specific time that the pattern took to run in the 16% of the cases that it ran with no overflows. The values of T for these runs cluster tightly at 38ms; so tightly that the individual points are indistinguishable in the scatter plot. This value, 38ms, is the time that Figure 4-4 approaches from the right as handler time h was adjusted, and is the lower limit that network throughput sets on the time to complete the pattern.

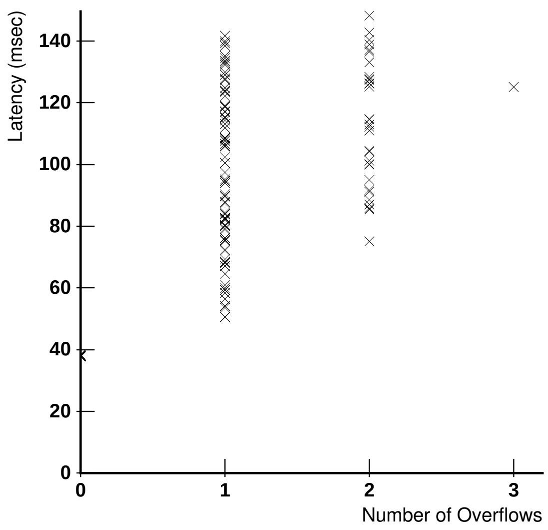
But when overflows occur, the machine required far longer to complete this pattern. In the 51% of runs in which a single overflow occurred, Figure 4-6 shows T ranging widely, from 50 to 140ms. When two overflows occurred, the makespan ranged from 75 to 196ms. This is a substantial slowdown, and a substantial range of slowdown. Section 4.5 will discuss why there is such a large variance.

In general, correlation does not mean causality. But in the case of the J-machine, we know (Section 2.3) that the overflow handler on the J-machine is slow, and that it stops the network. Combined with this knowledge, we can conclude from the data in Figure 4-6 that overflows cause the substantial drop in performance.

4.4 No Avalanches

It is surprising to see so few overflows in each run in Figure 4-6; never more than three in any run. I had expected many more, in an avalanche. This expectation arose from assuming that a processor handling an overflow trap would induce several other processors to take an

(A)



(B)

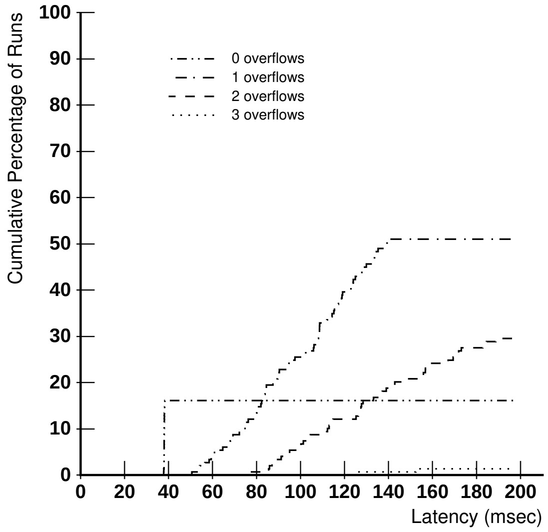


Figure 4-6: (A): T vs. number of overflows for 150 runs of the Snakes workload with $L = 1000, m = 5, h = 45, p = 3$, corresponding to the leftmost point in Figure 4-4. (B): The same data, plotted as cumulative percentage of runs distributed over T.

overflow trap, because the trapped processor would block any processor trying to send to it, as well as several others blocked by the contention tree it created; these would overflow, blocking other processors, until the entire machine avalanched with overflows. Apparently, this did not happen.

Looking at this process in the simulator, it can be observed that a single overflowed processor can stop all progress of all messages in the network. This ends up slowing the output side of the other processors. The reason that there is no avalanche is that a single overflowed processor stops *all* other processors. While all these blocked processors are not making any progress on their output side, there are no messages arriving on their input side from any other processors, so there are no immediately following overflows.

The single processor handling an overflow fault disables incoming messages from the network. This eventually jams all Z channels in the column leading to that processor. These jammed Z channels then block all Y messages in the Z-Y plane incoming to the processor's Z column. These blocked messages in turn block all messages in the X channels of the entire cube that lead to that Z-Y plane. As a consequence, all processors are eventually jammed.

The time that it takes for the formation of the avalanche-quenching contention tree can be estimated. At the instant the overflow occurs, messages are arriving at the processor at N times the average inter-arrival rate $\frac{T}{NLh}$, which at this operation point is 8 μs. An arriving message blocks traffic along the nearby Z channels; the message inter-arrival time along these channels varies from 2 μs to 8 μs, with the higher inter-arrival near the edge of the mesh. The overflowed processor tends to be at the center of the mesh, which is about $(4 + 4 + 4)$ links from every other processor, so the contention tree should form in approximately 100 μs.

This tree forms in only one column of Z channels, one plane of Y channels and all X channels, but in each case, only in channels leading toward the overflowed processor. Messages do then continue to transition among processors for a short time, but only if they are moving opposite the direction of the contention tree. So this means that processors near the edge of the mesh (in X) get few messages out (unless they are sending to a processor with the same X coordinate and bypassing X channels). Processors near the overflow point get messages out if they send in the unblocked X direction or bypass X (for which they have better odds than do X dimension endpoint processors) but because the direction in X is random, eventually these processors get caught up in the contention tree.

Avalanches are not impossible. In Snakes, the fact that the tree forms in a way that suppresses the avalanche seems to be coincidence; the communication pattern in Snakes happens to form the contention tree in a way that stops all message traffic. When message traffic ceases, overflows do as well. One can easily construct a different communication pattern on the network that would allow traffic to continue unimpeded after some processor gets an overflow. For example, all the snakes might visit the processors in a single tour that mapped precisely to the mesh topology of the network. In such a case, there would be no contention, so other communication would continue unimpeded after one processor overflowed, and the overflows would occur in a sequence upstream in the tour.

In fact, the existence of a pattern that can avalanche means there is some chance that a regular run of Snakes could avalanche if the random number generators chose exactly the correct pattern, for example, a tour of the processors of the type that we said that could get overflows. The chance of this happening is certainly small.

And even so, it is not certain that avalanches would even be detrimental, because the processing for simultaneous overflows can happen in parallel. Perhaps, if the overflows in avalanche were non-simultaneous, the processing would not overlap. This is easy to set up and test on the J-Machine, if one were interested.

Another insight is that if one were designing a new machine, and found that avalanches of overflows were problematic, then some sort of "brake" might be considered to try to duplicate the effect that Snakes gets here, where it stops the entire system on a single overflow. Some network designs – for example, some sort of adaptive network – might not have the avalanche-quenching behavior we see here.

Nonetheless, what is most important is to have described how intuition failed in expecting avalanches.

4.5 Queue Size Snapshots

Why is there such a large variance in T when overflows occur? It appears that this is due to variance in the amount of time that it takes the system to go from the initial evenly balanced

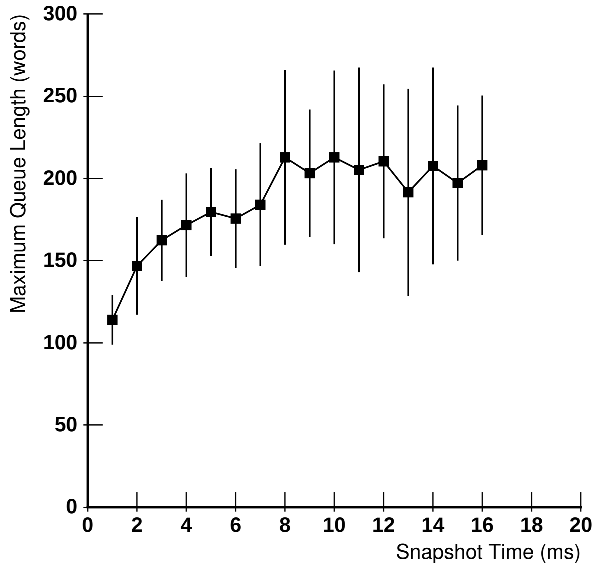


Figure 4-7: Snapshots of the maximum queue size as time advances. Load point is $m = 5$, $p = 3$, $h = 65$. Hardware queue size $Q = 512$. Vertical bars show measured standard deviation.

configuration to the equilibrium condition which is slightly imbalanced. Furthermore, it appears that the overflowed state is "sticky;" that is, once one processor becomes overflowed, the system stays that way.

Figure 4-7 plots average maximum queue size snapshots as a function of time. For each point, the average is taken over several runs. This plot runs at a slightly different load point than Figure 4-6. It has the handler time $h = 65$ rather than 45. This load runs unimpeded by queue overflows when the queue size is 512, as it is for this measurement. Though it would have been nice to snapshot the succession of queue sizes including queue overflow effects, this is not possible because the MDP will not halt when it is running the queue overflow trap handler.

Snakes starts with the load evenly balanced, in this case with 5 messages (20 words) in each queue. However, the figure shows the queue size starting at 100 words. This can be explained by the fact that the run cannot be halted soon enough after it starts. It only takes 1 ms for the largest queue to reach 100 words. How many messages have been processed to reach that point? Figure 4-4 showed that the whole pattern takes about 40 ms to execute when $h = 65$, so that means that each message is taking about 40 μs, of which only 6.5 μs is processing time, with the other portion being spent in SEND faults. In the 1 ms the system took to become unbalanced, each processor had executed about 25 messages. Within another 10 ms, the system appeared to have reached equilibrium, or after 250 messages have been handled.

The plot shows the average maximum queue size starting at 100 and growing to 200, with a standard deviation of 50. It shows clearly why a 256-word queue is inadequate: the equilibrium maximum queue size hovers close below 256, but only one standard deviation away, making queue overflows likely. With a 512-word queue, the larger margin prevents overflows.

4.5.1 Comparison to Model

At this load point, a 256-word queue is inadequate, while a 512-word queue is sufficient to prevent overflows. How does this result compare with the queueing analysis in Chapter 3? That queueing analysis didn't actually predict the average maximum queue size. Rather, the analysis estimated overflow rates in terms of the number of processors, customers, and the queue buffer size (measured in tasks or customers). However, the analysis did exhibit sharp increases in overflow rate at some workload points.

In the measurements in this section, messages are 4 words. An MDP queue that is 512 words long corresponds to a $Q = 128$ customers in the model. An MDP queue that is 256 words corresponds to a $Q = 64$ customer queue in the model. Is there a noticeable difference between $Q = 64$ and $Q = 128$?

The queueing analysis was split into two cases: balanced and imbalanced workloads. We can consider both cases here, because the Snakes workload itself is balanced, but the network acts to imbalance it.

Balanced Case

Considering the balanced case, the queueing analysis predicts that overflows should be extremely rare when the queue is 128 tasks long. See Figure 3-2, which predicts a normalized overflow rate less than 10^{-8}. This means the expected number of overflows should be one every 100,000,000 messages served by a processor.[8] Since the maximum queue size in Figure 4-7 is well below 512, this is consistent with our observation. If we consider the implications of making the queue 256 words long, the effective queue length is only 64 tasks. In this case, Figure 3-2 predicts an overflow rate of 10^{-3}, one every 1,000 messages served by a processor. Since the Snakes are $L = 1000$ hops long, this implies an expected rate of one overflow per run. Figure 4-7 shows the standard deviations, and the positive side of the standard deviation reaches 256 words. This means that there would several times where the queue size on the J-Machine would be greater than 256 words. Recall that Figure 4-7 only show the snapshot succession for the beginning of a run. So, during a $L = 1000$ run, there would be many times that the maximum queue size crossed 256 elements.[9] But the model predicts only one, for the whole system. The reason for the difference between the balanced model and the run on the J-Machine is that the run on the J-Machine is imbalanced.

Imbalanced Case

Applying the queueing analysis for the imbalanced workload is difficult because the factor B used in the queueing analysis is unknown for the J-Machine hardware. It might be derived as a working measurement by measuring the rate of overflows for a 256-word queue system, and using the observed overflow to estimate B.

[8] The overflow rates are normalized to the service rates on an individual processor, not for the overall service rate on the system.

[9] This is when the queue size 512, larger than the maximum ever encountered in the run. Section 4.5.2 shows that when the overflows do occur on the J-Machine, the machine "sticks" in the overflow state, and so only a few occur. However, this sticky effect is not captured in the queueing model. For the comparison, it is better to use the J-Machine with the queues running free within a larger buffer.

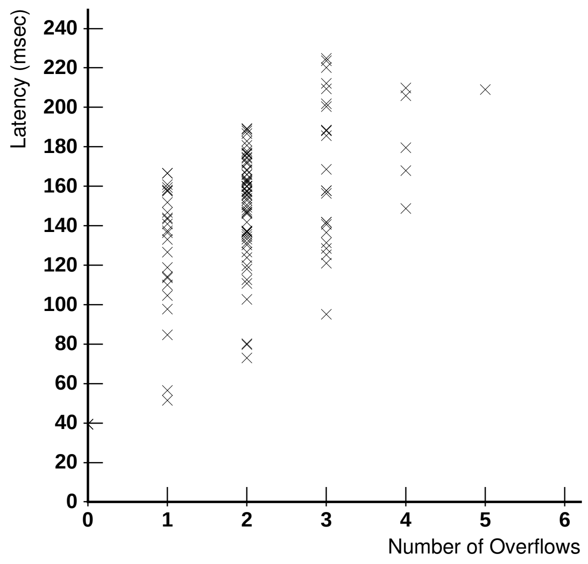


Figure 4-8: Latency vs. Overflows for $L = 1000$, $m = 5$, $p = 3$, $Q = 256$.

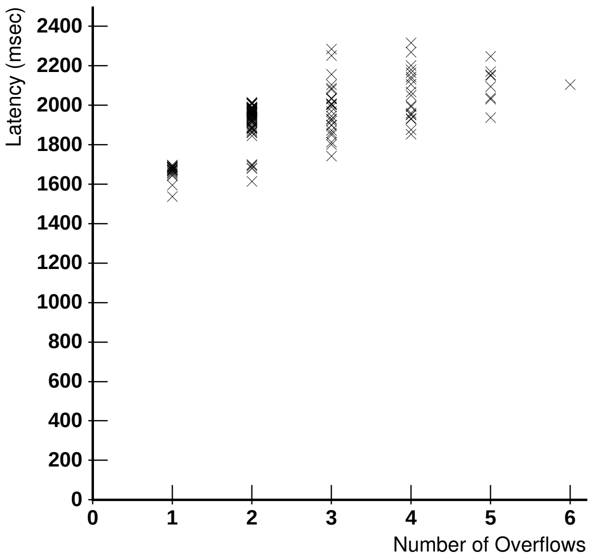


Figure 4-9: Latency vs. Overflows for $L = 10000$, $m = 5$, $p = 3$, $Q = 256$.

4.5.2 Sticky Overflows

What happens when the queue size is 256 for this load point? Figures 4-8 and 4-9 scatter-plot makespan vs. overflows for the previous section's workload point, but with a 256-word queue. Figure 4-9 runs for 10 times longer than Figure 4-8. From the plots, we can see that the relative variance is smaller when the run is longer. Furthermore, the number of overflows does not increase by a factor of 10.

Recall that Section 3.2 pointed out the distinction between overflow residency and overflow rate: overflow residency was the proportion of time that the system spent overflowed, while overflow rate is the rate at which the system made transitions between non-overflowed and overflowed states.

With Figure 4-7 showing the maximum queue size hovering below 256, we would expect the proportion of time that the system spends overflowed (the "queue overflow residency") to be low; that is, with the machine mostly staying a non-overflowed state. Occasionally, the variance in queue size would lead the system to make excursions into the overflowed state. Most of the cost is due to the costs for the trap into the states. Were this the case, increasing the run length from 1000 to 10,000 (as we did going from Figure 4-8 to Figure 4-9), we would expect to see an increase in the number overflows. But, we do not: both plots show about the same distribution of overflows, mostly from 2 to 5.

It is not surprising that simple expectations of the dynamics do not bear out when overflows occur. Expecting queue overflows to affect only the timing and not the trajectory through configurations of customers ignores the effects that the queue overflow handler has on the system when it disables its incoming queue and jams the network.

What appears to be happening is that the machine is sticking in the overflowed state. Here is the sequence of events:

1. The workload starts out balanced.
2. As the imbalance increases, the maximum queue size grows in a random walk with upward drift.
3. The maximum queue fills, and an overflow trap occurs.

4. The trapped processor disables its incoming network port which, within a small time jams the network and stops the other processors.

5. The trapped processor slowly copies queue contents to the overflow memory, and resumes operation when this is complete.

6. The machine resumes operation as before, until the overflowed processor reaches the token marking the position of data copied to the reserve queue.

7. The machine halts again. Any new arrivals to that queue are slowly copied to the overflow memory, and old messages in overflow memory are slowly copied to the working queue.

8. It is possible for the volume of new arrivals to be less than number of messages that the overflowed processor manages to handle. If this is the case, then overflow memory will be empty and no token will be written into the working queue, and the machine will leave overflowed mode.[10] Otherwise, a token is written to memory and the machine goes back to step 6.

If the number of messages in the overflow memory is small, as it would be just after an overflow occurred, then it seems plausible that the random walk would allow the overflowed queue to make an excursion below the overflow threshold, and therefore allow the machine to leave the overflowed states.

The overflowed processor has to expend substantial overhead to copy messages between the overflow memory and the working queue. However, because it disables incoming messages and jams the machine while doing this copying, it imposes an almost-equal amount of overhead on all of the other processors. The imposition of this overhead on the other processors is not by design: it is an unintentional characteristic of the uniform, heavy, random traffic in the Snakes workload that propagates the jam quickly via network backpressure to each processor in the machine. It is this same unintentional characteristic which seems to prevent avalanches (See Section 4.4).

Because it is unintentional, it also is not guaranteed to be equal. The overhead experienced by the overflowed processor could be slightly more or slightly less than the overhead it imposes on

[10]Though the queue on that previously overflowed processor might be poised just below threshold.

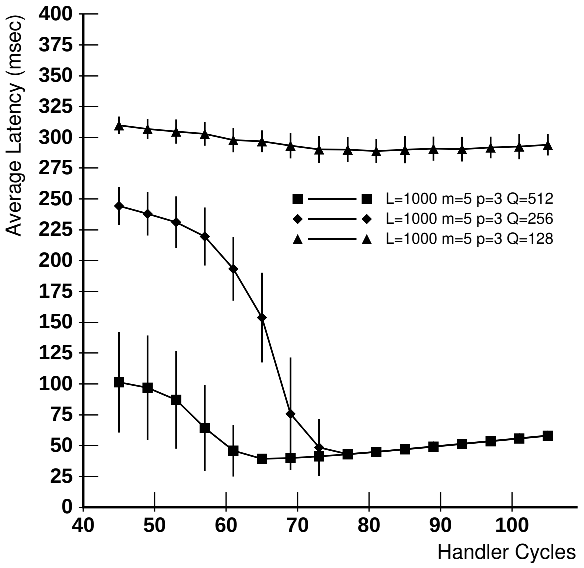


Figure 4-10: Effect of adjusting the queue size. Vertical lines show standard deviation.

the other processors. It seems likely that it would be slightly more. With a higher overhead, a slightly higher upward drift would be added to the queue size's random walk. This would tend to move the overflowed processor more unyieldingly into the overflowed states. The result is that the machine running Snakes takes a couple of overflows and then largely stays overflowed. At a load point such as that in Figure 4-7 where the natural maximum queue size seems to hover at 200, configuring the queue size to 256 is unacceptable. Overflowed states are sticky.

This evidence suggests that most of the variation in makespan arises from the variation in time that it takes for the machine to reach the overflowed threshold. A consequence of this is that the longer runs show a smaller percentage variation. The problem here differs from a common barrier crossing problem [Gal94, Chapter 7] because it involves multiple processes (the queue sizes on each processor) which are dependent, with each facing its own barriers.

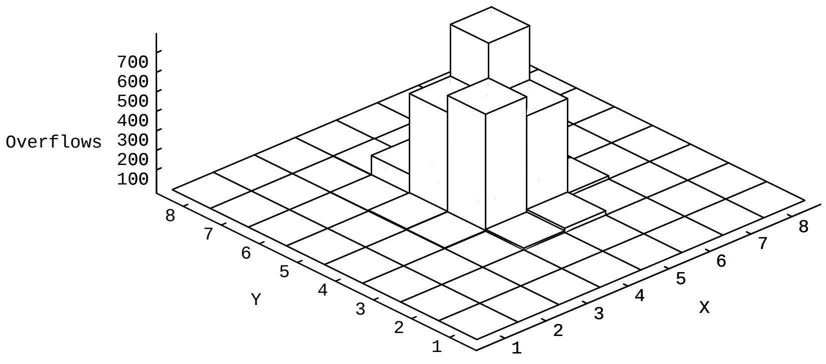


Figure 4-11: Plots of spatial distribution of the number of overflows in two middle planes of the 8x8x8 J-machine after running Snakes 4820 times at the load point $m = 5$, $L = 1000$, $h = 45$, $Q = 512$. All of the overflows occurred in the center of the machine. This is the $Z = 4$ plane.

4.6 Adjusting Queue Size

Figure 4-10 shows how changing the queue size affects performance. This figure includes the data given earlier in Figure 4-4 which was measured with a 512 word queue, and then also includes the performance plot for queue sizes 128 and 256 (note the larger vertical scale). Not surprisingly, a smaller message queue leads to a substantial increase in latency.

How large should the queues be on the J-Machine? In Chapter 3, Equation (3.65) suggested that the imbalance threshold was independent of queue size. However, the derivation for Equation (3.65) was for the case where there was only one bottleneck processor.

It might be possible to expand the argument in Section 3.4.2 to consider the threshold in the case where there are multiple bottleneck servers.

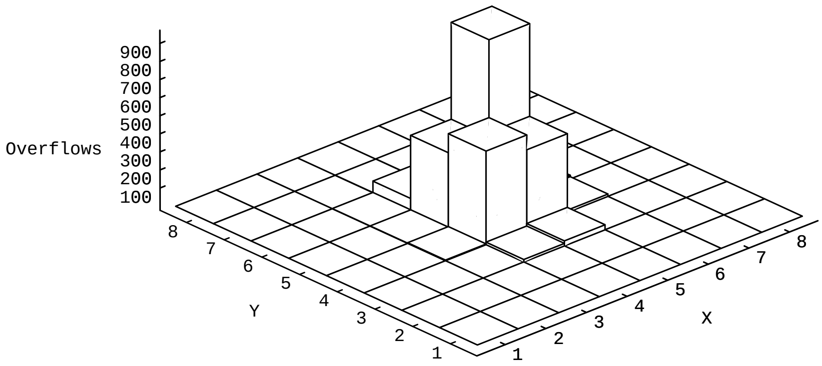


Figure 4-12: Plots of spatial distribution of the number of overflows in two middle planes of the 8x8x8 J-machine after running Snakes 4820 times at the load point $m = 5$, $L = 1000$, $h = 45$, $Q = 512$. All of the overflows occurred in the center of the machine. This is the $Z = 5$ plane.

4.7 Overflows Occur at the Center of the Machine

Figure 4-11 and Figure 4-12 plot where overflows occur in the J-Machine. The first plot is for the 64 processors in the $Z = 4$ plane, and the second plot, which is similar, is for the $Z = 5$ plane. The plots show that the overflows all took place in the processors near the center of the machine. No overflows took place in the processors at the edges.

Initially, when I was running this workload, this stark asymmetry in the location of overflows was unexpected, because the Snakes workload itself is inherently balanced and uniform. But from the plots, it is immediately obvious what is going on: the 3D mesh topology of the J-Machine imbalances the load. So while the Snakes workload is balanced over the processors of the J-Machine, it is not balanced over the network.

This can be illustrated by considering N processors sending uniform traffic to each other on a 1D mesh (a line) which is not constrained by capacity. Each node is sending at the same rate λ. A link next to the end node is going to carry traffic equal to $2\lambda(N-1)$. The center link[11] carries traffic equal to $\lambda N^2/2$, which is $N/4$ times as much traffic as the nodes near the edges. Processors near these center nodes "see" this, because the more heavily loaded channels have much greater occupancy. The greater occupancy is coupled to the center processors through network backpressure that leads to more SEND faults than nodes to the edges. Later, Chapter 5 will show that this particular result is due to unfair routers, but at this point, that is perhaps not obvious from the J-Machine measurements here.

Because they experience more send faults, they are slower than the other processors in the machine and therefore the workload becomes imbalanced.

4.8 Effect of Message Length

Figure 4-13 shows the behavior of the J-Machine with larger messages by varying p as well as h. Note that this figure is for the workload running with $m = 2$ snakes per processor. With fewer snakes running, the machine should be more tolerant of load imbalance. For short messages ($p = 3$ and $p = 7$), this appears to be the case, the machine does not ever

[11] Assuming that the number of nodes on the line is even.

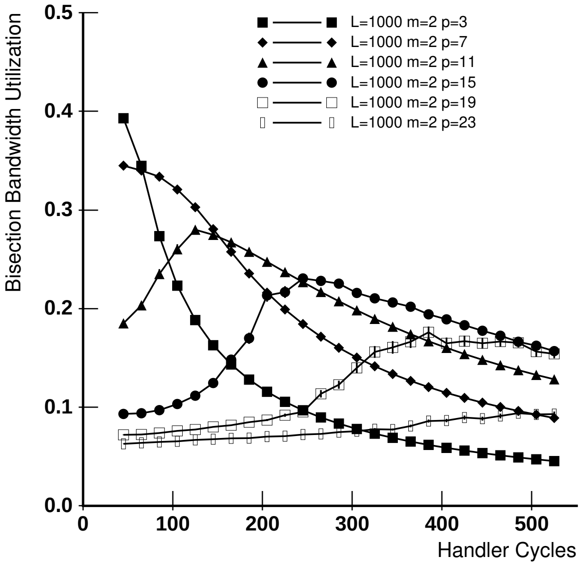


Figure 4-13: Mean bisection utilization for Snakes versus handler cycles, for several different message payload lengths p. For short messages, performance drops as the network starts to saturate. For longer messages, bisection utilization is always poor. Q=512. Note that the X-axis scale is larger than in Figure 4-5

have overflows, even with short handler times.

But with longer messages, even though the J-Machine is running with fewer snakes, the machine gets overflows. To some extent, poor performance with long messages is due to the fact that longer messages take more space in the queue and so they consume the queues more rapidly. If message payload length is $p = 11$ words, then with $m = 2$ messages per processor, the average space used by messages by itself consumes 5% of the queue capacity in the machine. Section 3.3.6 had suggested that queue overflows for a balanced workload do not become problematic until 10% of the capacity is consumed, so the baseline rate for balanced overflows looks acceptable. For this imbalanced load, using Equation (3.65) suggests that the imbalance is as much as $B = 1.5$ with short messages. That is, nodes in the center of the machine are taking more than 50% longer to handle messages than nodes near the edges.

Figure 4-13 shows that the performance drops at just about the point where Equation (4.10) predicts that the network reaches saturation, assuming that $u_{\text{saturation}} \approx 0.4$. For example, Equation (4.10) predicts that saturation occurs for $h = 320$ for $p = 15$ word messages; just below is where the performance drops in the figure.

The data indicate that the J-Machine works better with shorter messages.

4.9 Effect of the Number of Messages

Chapter 3, Section 3.3.6 suggested that for a balanced load, queue overflows become a significant factor when the number of customers approaches 0.1 of the system's total queue capacity. In the case of the J-machine where each processor's message queue is 512 words, this would indicate that one could permit 51 words of the queue capacity to be used without difficulty. If the payload length p is 3, then the footprint of a message in the queue is 4 words. This would suggest that the J-machine, for a balanced load, could operate with few overflows with up to 12 customers per processor. Of course, this is true only if the load is balanced.

Figure 4-14 shows the effect of changing the number of customers on the machine. Rather than being able to run with 12 customers, the system encounters difficulty when the number of customers per processor exceeds 5.

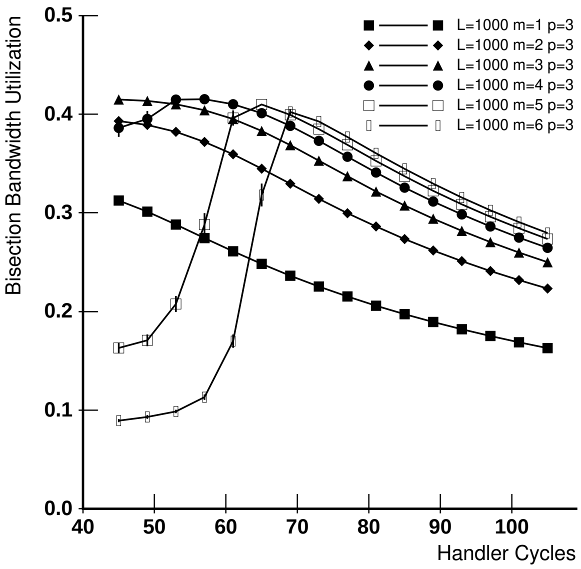


Figure 4-14: The effect of increasing the number of customers on the system, as a a function of handler duration. $Q = 512$.

This indicates that the balanced model is invalid for the J-machine. Previously, Section 4.7 showed that some processors (the ones in the center) get more overflows than other, despite the balanced workload. This indicates that these slow processors are a bottleneck, and that the imbalanced queueing model (Section 3.4) applies.

The key parameter of the imbalanced model is the value B, which represents the slowness of the bottleneck processors relative to the other processors. This factor B was used in Section 3.4.2 to determine a threshold in the number of customers where the imbalance led to large accumulations on the bottleneck.

The workload is balanced; the imbalance comes from the hardware. What should the value B be for the J-Machine? This is daunting to derive analytically, and perhaps not meaningful when derived that way. However, a value (below) can be derived from the observation of the threshold on the hardware. Because the threshold observed in Figure 4-14 is at $m = 5$, Equation (3.65) suggests the degrees of imbalance experienced on the machine is $B = 6/5 = 1.2$, or, that the bottleneck processors are about 20% more heavily loaded than the other processors in the machine at these load points.[12]

4.10 Request/Reply

Previous sections showed that the network topology of the J-machine unbalances the uniform load. This leads one to consider the effect of a primitive load balancing algorithm on the performance.

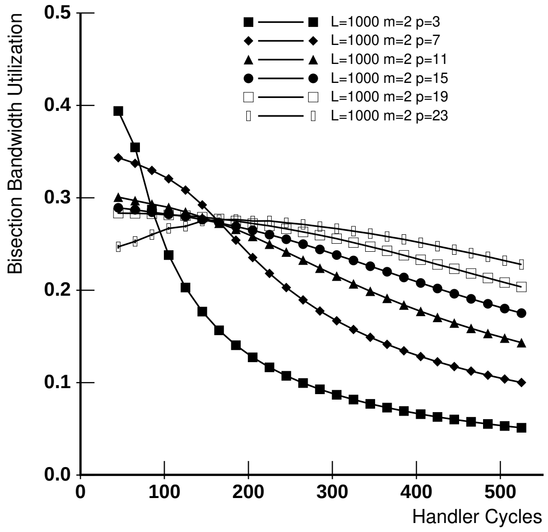
One simple load balancing algorithm requires that messages return to their home processor periodically.[13] Because the messages in Snakes include a counter, this algorithm is easy to add. After each message is handled, the counter is checked; and if it is evenly divided by the return period, the home processor is selected for the next processor, rather than some random destination.

The result is illustrated in Figure 4-15. This should be compared to Figure 4-13, which has

[12]Equation (3.65) was derived for a single bottleneck processor, but I am assuming that this doesn't matter here.

[13]This effort is different than the RTS algorithm in Chapter 6. There, RTS only returns when there is an overflow. Here, the messages return every period.

(A)



(B)

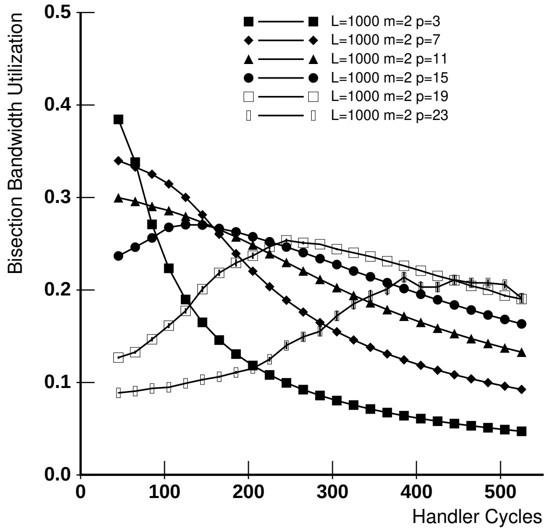


Figure 4-15: Mean bisection utilization for Snakes versus handler cycles, for several different message payload lengths p. In the upper graph (A) tasks return to their home processor every second message. In the lower graph (B) tasks return to their home processor every fourth message

the same load points, but does not have return-to-sender. The figure shows that adding simple load balancing improves the performance of the machine radically. In fact, it works amazingly well.

The thesis returns to this observation periodically, referring to the return-to-sender with period 2 as the "Request/Reply" workload. Section 3.5.3 considers the queueing analysis for this workload (but does not go beyond formulating the workload as a multiclass queueing network.)

It's worth mentioning that the return-to-sender with period 4 does show (see the lower part of Figure 4-15) some benefit in terms of balancing the load, but this benefit is not complete. In other words, a moderate degree of structure to the message passing pattern imparts a moderate degree of load balancing. This lends some suggestion as to how a "real" application - not as radically unconstrained as Snakes, but not as radically constrained as Request/Reply - would run on the machine. It will run with moderate imbalance.[14]

4.11 Discussion

This chapter has presented results of the synthetic, closed, uniform, message-passing benchmark called Snakes running on the J-Machine. When the load is heavy and the throughput of the synthetic benchmark is network-limited, the network imbalances the workload, and as a consequence queue overflows occur on the processors. These overflows limit the performance of the machine.

How meaningful are the results in this chapter? The results suggest that when the intended fine-grained software workload is run on a large J-Machine, queue overflows cause poor performance. Because this is only a synthetic workload, it is impossible to conclude that the performance *will* be poor for all or most other workloads - it is possible that some workload will be well behaved and will not overflow. In fact, such workloads exist, as Section 4.10 demonstrated: a Request/Reply workload is extremely stable. However, Snakes demonstrates that one of the most general workloads (simple random tasks walking the machine) is prone to cause queue overflows.

[14]Note that this period 4 return to sender could be captured by the BCMP [BCMP75] (see Section 3.5.3) but requires 4 classes per processor.

Although the queueing analysis in Chapter 3 suggests that the J-Machine's queues are adequate, this is only the case for small messages and for light loads. With small messages, the queue storage is effectively larger, and with light loads, the network does not strongly imbalance the processors. However, with longer messages, the queue is effectively smaller risking overflows due to simple dynamics. And at heavy loads, the network has substantial impact and can imbalance the processors.

Queues overflow in the center of the machine. This chapter does not establish why they overflow in the center rather than at the edges or somewhere else. Intuitively, one expects the load on the network channels near the center of the machine to be higher than near the edges, this because the machine topology is a mesh with no wrap-around connections. Next, Chapter 5 presents a mean value analysis which explains why the queues overflow in the center.

Chapter 5

Mean Value Model of the J-Machine

This chapter gives a model for a Mean Value Analysis (MVA) of message traffic on the J-Machine. The chapter shows that when running a heavy uniform and random workload, the equilibrium average queue size on processors near the center of the J-Machine mesh is much larger than on processors near the edges.

This J-Machine model is an adaptation of one by Adve and Vernon (AV) [AV91, AV94]. Their model incorporates wormhole routing into a closed queueing network that can be solved with Approximate Mean Value Analysis (AMVA). The chapter adapts their model for the J-Machine topology, the unfairness of the J-Machine router, and the Snakes workload.

Previous chapters in this thesis showed that message queue buffer overflows limited the performance of the J-Machine when it was running a heavy random workload. Chapter 3 predicted overflow rates and residencies for a closed queueing network and determined a threshold of imbalance at which any bottleneck led to severe congestion within the system. That analysis did not consider the communication network; the analysis in this chapter does.

Results

This chapter provides a working analytical model of the J-Machine, which determines expected queue sizes at heavy loads. The model makes several predictions which are qualitatively consistent with observations of the laboratory prototype J-Machine and with simulations running a heavy workload. These include:

- Mean processor queue sizes are largest near the center of the J-Machine mesh.

- The position of this accumulation is demonstrated to be a strong function of the choice of arbitration in the J-Machine's network router switch, particularly the unfair router arbitration.
- The fact that predicted accumulations are larger than available message buffer space on the J-Machine suggests that it will experience high rates of queue buffer overflow.
- The Snakes workload shows much larger predicted expected mean queue size than does a Request/Reply type of workload.

The model predicts that the mean processor queue size will exceed the size of the MDP's message buffer at the same points that were observed on the J-Machine.

Methodology.

The AV model is implemented on top of the J-Machine simulator. Some simple modifications are necessary to correct errors in the published version and to strengthen some equations that do not work well at heavy load. A large modification to their model is developed to account for the fact that the J-Machine's router is unfair. Without this modification (that is, with the original fair router model) the AV model predicts that queues are largest near the edges of the cube. This prediction is qualitatively inconsistent with the observations of the J-Machine in Chapter 4, where the greatest number of queue buffer overflows occurred in the center of the machine. Changing the simulator (since it is not possible to change the J-Machine hardware) so that the routers are fair results in mean queue sizes that are qualitatively consistent with the AV model. Changing the model for an unfair channel results in predictions that are qualitatively consistent with the results on the J-Machine and simulation.

Caveats.

This J-Machine model is an AMVA model. As such, it does not result in a simple expression for performance from workload parameters, unlike other system performance models. Instead, it is a large system of nonlinear equations that is solved with an iterative numerical

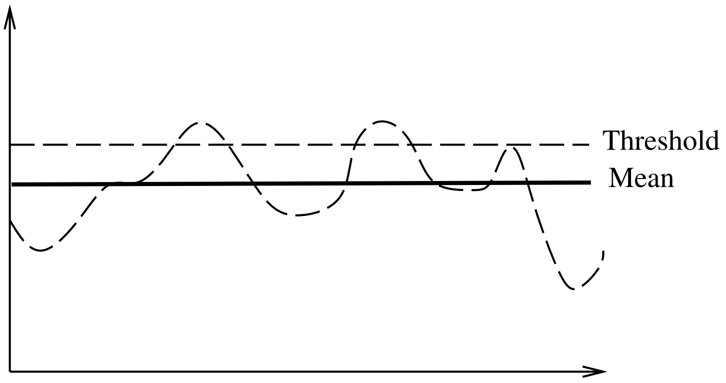


Figure 5-1: The rate at which queue size crosses the capacity threshold increases as the mean moves closer to that threshold.

relaxation algorithm. The current implementation of the model takes a long time to converge to a solution. It is sometimes faster to simulate the system than to solve the queueing model!

The MVA algorithms only solve for the equilibrium *mean* of the queue size in a queueing network. This, by itself, is insufficient for predicting the *rate* or *residency* of queue buffer overflow.[1] See Figure 5-1; if a buffer size determines a capacity threshold and the mean queue size is below the threshold, the rate of overflows is partly determined by the variance of the queue size process.

This would suggest an analysis using first *and* higher moments (e.g., variance). Higher moments would capture behaviors like the formation of convoys, which lead to a more "bursty" queue size process and a higher second moment. In general, higher moments are important to the analysis of any fixed buffer size scheme.

The literature contains some modeling predictions of second moments;[2] but this thesis does not make any analytical predictions of second moments. This does not remove all significance of MVA model results. Because queue buffer overflows are so costly on the J-Machine, the

[1]Section 3.2 defines residency and rate.

[2]See, for example, [MM94] which gives the moments of queue size in a simple windowing flow control scheme.

occurrence of even a single overflow causes substantial loss of performance, and in many cases, the mean queue sizes predicted by the MVA models are are *many times larger* than the buffer capacity threshold. So, the mean values alone imply that queue overflows are inevitable, with consequent loss in performance.

It is when the queue size predictions approach but remain below the buffer size threshold that the results are more ambiguous. However, if one is willing to assume a reasonably well-behaved[3] queue size process, then one can reasonably conclude that a processor with a mean queue size near the threshold would have a higher rate of overflows than a processor with a mean value well below the threshold. In other words, the mean value predictions do offer some indication of the expected overflow rate and residency.

If, on some hypothetical machine, the occurrence of a queue buffer overflow trap were less time-consuming, though still not necessarily free, a performance estimate might require a more accurate prediction of the cost of overflows for a workload. Predicting this would require a strong estimate of the rate of overflows, and therefore some good estimate of the variance. It is in this case that attention to the second moments in an analysis is worthwhile.

The MVA methods used in this chapter are approximations, in at least two senses. First, the basis of the MVA algorithm is the Arrival Theorem (Section 5.1.1), which holds for ideal Poisson queueing networks which have product form solutions. The J-Machine is not an ideal queueing network, so the Arrival Theorem is not strictly applicable. However, simulations show that the application to the J-Machine works reasonably well, perhaps because our workloads are random with a large number of customers. Second, the blocking model (Section 5.1.5) is based on independence assumptions which are only partially true; they work in practice; overall model conclusions are only empirically justified. Using independence assumptions of this sort has a long tradition in the application of Queueing Theory [Kle64].

[3]One can construct time processes where the moving the mean toward a threshold by adding a constant has no effect on the rate of threshold crossing. For example, consider a queue size process which jumps back and forth between +100 and -100; if the threshold is 10, moving the mean to 10 has no effect on the rate of that threshold crossing. This is not a well-behaved process. By saying that the processes is "well-behaved," I mean that the process is random with moderate sized peaks and valleys.

Chapter Outline.

This chapter has five major parts. Section 5.1 gives a short introduction to the MVA technique. These are well-known algorithms for solving for the mean values of waiting time, queue size, and throughput of closed product-form queueing networks. The section gives explanations of the exact and approximate versions. An advantage of MVA is its computational efficiency. Also, it has an intuitive basis which facilitates extensions with things like blocking models.

A reiteration of Adve and Vernon's published model is in Appendix A. This appendix corrects some typographic problems and bugs in the published model and includes some initial changes for the J-Machine. The largest of these changes is that the original AV model has a remote memory system to handle remote requests from the processors. The Appendix changes it so that remote requests are handled by the processors, as they are in the J-Machine.

Using the Adve and Vernon model, Section 5.2 shows that with fair routers and a Request/Reply communication pattern, queues are largest on the edges of the cube, rather than in the center where the J-Machine exhibited most of its queue buffer overflows. There is an intuitive explanation for this: messages from the edge processors face a larger number of traffic confluences than do messages from processors in the center. These edge processors then experience greater backpressure, so that they are the ones that get bottlenecked. This intuition is verified by modifying the simulator's router to run with fair arbitration.

Section 5.3 introduces changes to the model to account for the J-Machine's unfair router. These changes show accumulations which are qualitatively consistent with the observed behavior of the J-Machine and with simulations.

Section 5.4 changes the model for the Snakes workload. This entails substantially simplifying the original AV model, which distinguished among traffic classes in order to model a Request/Reply workload. In Snakes, all customers are equivalent and fall into a single class. The simplified model shows that this workload is significantly less well-behaved than a Request/Reply workload.

Section 5.5 discusses the results and limitations of the model, and summarizes the chapter.

5.1 Mean Value Analysis Review

MVA [RL80] is an algorithm that computes expected equilibrium mean queue size, mean throughput, and mean waiting time for the servers of a closed Poisson queueing network: N exponential servers each with service rate μ_i, fixed K identical customers, transition probabilities r_{ij}. The well-known product-form equilibrium probability distribution for the closed Poisson queueing network, Equation (3.6), can be used to compute the same results as MVA, but doing so is often impractical because of the sum over the large state space. Chapter 3 avoided the impractical sum by exploiting solution symmetry for uniform workloads and by using Buzen's algorithm [Buz73] for imbalanced workloads. MVA avoids the impractical sum by exploiting the Arrival Theorem.[4]

5.1.1 Arrival Theorem

Informally, the Arrival Theorem states that the view of the queueing system that a customer "sees" at the instants it arrives at a server is identical to the time-average-view of the same system, but with one fewer customer (see Figure 5-2). More formally, the equilibrium probability distribution for the configurations of customers over servers is identical for both of these cases:

- when the system is sampled at the instants of customer transitions and there are k customers,
- when the system is sampled randomly in time and there are $k-1$ customers

The Arrival Theorem allows one to compute mean values for the system with k customers by using the mean values for the system with $k-1$ customers. Thus the basic MVA algorithm is a recurrence. The base case that terminates the recurrence is with 0 customers and mean values that are 0.

The Arrival Theorem holds only for certain ideal networks, such as closed product form queueing networks. For non-ideal networks, it often works well as a good approximation.

[4] See [HP93, p.241] for a proof of the Arrival Theorem and a more detailed exposition of algorithms for determining mean values in queueing models.

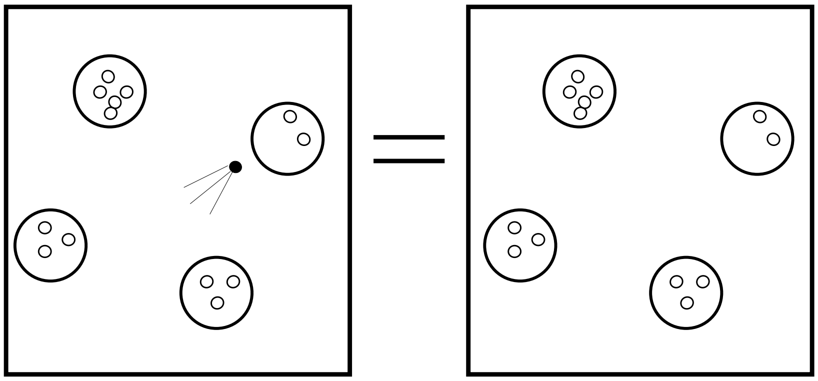


Figure 5-2: Illustration of the Arrival theorem. The probability distribution for the configurations of customers seen by a customer at the instant it makes a transition from one server to another (the black dot in the queueing network on the left) is the same as the probability distribution for the same system with one fewer customer when the samples are taken randomly.

This allows it to be used in otherwise inaccessible applications.

There are two variants of MVA: exact and approximate. Exact MVA starts with the base case of 0 customers and iterates up to k customers. Approximate MVA (AMVA) is an iterative relaxation technique that uses estimated mean values with k customers to guess the solution for $k-1$ customers, and then uses this guess to refine the solution for k customers. The next two subsections outline these variants.

5.1.2 Exact MVA

The central structure of MVA algorithms is a recurrence. It solves for the mean traffic metrics (queue size, throughput, waiting time) for the network when the fixed number of customers in the network is k by using the traffic metrics for the same network if it only had $(k-1)$ customers in it. Because of this recurrence, the variables representing network metrics are indexed by the number of customers in the network:

- $q_i(k)$ is the mean queue length at server i (including the customer in service) when there are k customers in the network, and

- $w_i(k)$ is the mean waiting time (including service time) for a customer at server i with k customers in the network.

The algorithm is restricted to ergodic networks (ergodicity being a property of the transition matrix elements r_{ij}) where all customers are guaranteed to keep revisiting all servers, including the reference server. A particular server is selected as a reference server; let it be server number 1. With it, the following throughput and visit ratios can be defined:

- $T(k)$ (throughput) is the rate at which the reference server handles customers, when there are k customers in the network.

- v_i is the visit ratio for server i. It is defined as the number of visits that a customer is expected to make to server i before it returns to server 1. The values $\{v_i\}$ are the unique set of solutions to the traffic equation:

$$v_i = \sum_{j=1}^{N} v_j r_{ij} \tag{5.1}$$

  with $v_1 = 1$.

The throughput of an individual server i is the product of its visit ratio and the reference server throughput:

$$T_i(k) = v_i T(k). \tag{5.2}$$

Little's Law[5] relates the queue size, waiting time, and throughput of individual servers:

$$q_i(k) = T(k) v_i w_i(k). \tag{5.3}$$

The individual queue sizes sum to the number of customers in the system:

$$k = \sum_{i=1}^{N} q_i(k). \tag{5.4}$$

[5] $q = \lambda w$: the mean queue size is equal to the mean arrival rate times the mean waiting time. This law is general and applies to both individual servers and networks of servers. This law also works for servers with a arbitrary service time distributions.

Substituting Equation (5.3) in Equation (5.4):

$$k = T(k) \sum_{i=1}^{N} v_i w_i(k). \tag{5.5}$$

The core recurrence in MVA is as follows. The Arrival Theorem says that the number of customers found on arrival is the mean queue size when there is one less customer in the system. Therefore, the mean waiting time that a customer experiences at server i is the waiting time to serve the customers that it finds on arrival, plus its own service time:

$$w_i(k) = \frac{1}{\mu_i} \left(q_i(k-1) + 1 \right). \tag{5.6}$$

Rearranging Equation (5.5) allows the system throughput to be calculated from the individual server waiting times:

$$T(k) = \frac{k}{\sum_{i=1}^{N} v_i w_i(k)}. \tag{5.7}$$

With the throughput, the queue size at the individual servers can be calculated:

$$q_i(k) = T(k) v_i w_i(k). \tag{5.8}$$

The base case, $q_i(0) = 0$, completes the recurrence.

5.1.3 Multi-Class MVA

A limitation of the simple closed product form queueing network is that it cannot represent the situation where some customers follow different paths through the network with different routing probabilities. To address this limitation, a standard extension of the simple closed product form queuing network is to partition the customers into distinct classes, where each class has a different routing behavior. In the original AV model, multiple classes are used to model the routing behavior of messages in a shared memory multiprocessor. For each processor in the machine, a separate class is defined, corresponding to the limited number of outstanding in-progress memory requests. The routing behavior of the classes differ because each returns to its own home processor.

Introducing multiple classes increases the modeling fidelity, but at a cost. Although product form solutions and MVA solutions exist for multiclass models, the computational complexity of the MVA solution increases substantially. Approximations must be introduced to address this.

For multiple classes, the K customers are partitioned into C classes, each class containing k_c customers so that $K = \sum_{c=1}^{C} k_c$. Each class c can have a distinct routing matrix, with elements $r_{ij,c}$. Multi-class closed Poisson queueing networks have a product form solution [BCMP75]. This solution can be obtained by using a multiple-class version of MVA, as follows.

For multiple class MVA, the parameters of the single class model are indexed by class; also, the recurrence is no longer over the total number of customers in the network, but over a larger space indexed by the population vector $\vec{K} = (k_1, k_2, \ldots, k_C)$:

- $q_{i,c}(\vec{K})$ is the mean number of customers of class c at server i when the class population vector is $\vec{K}$.
- $w_{i,c}(\vec{K})$ is the mean waiting time for a customer of class c that arrives at server i when the class population vector is $\vec{K}$.
- Instead of one reference server, there are C possibly distinct reference servers, one for each class c, denoted ref_c. $T_c(\vec{K})$ is the throughput of class c customers through its reference server number ref_c and $T_{i,c}(\vec{K})$ is the mean throughput of customers of class c at server i with population vector $\vec{K}$
- $v_{i,c}$ is the visit ratio for class c customers relative to visits at its reference server number ref_c. The values $v_{i,c}$ are the unique set of solutions to the traffic equations for the class routing matrix:
$$v_{i,c} = \sum_{j=1}^{N} v_{j,c} r_{ij,c} \tag{5.9}$$
with $v_{\mathbf{ref}_c,c} = 1$.

Using these definitions, the core multi-class MVA recurrence is:

$$q_i(\vec{K}) = \sum_{c=1}^{C} q_{i,c}(\vec{K}) \tag{5.10}$$

$$w_{i,c}(\vec{K}) = \frac{1}{\mu_i}\Big(q_i(\vec{K} - \vec{e_c}) + 1\Big) \tag{5.11}$$

$$T_c(\vec{K}) = \frac{k_c}{\sum_{i=1}^{N} v_{i,c} w_{i,c}(\vec{K})} \tag{5.12}$$

$$q_{i,c}(\vec{K}) = T_c(\vec{K}) v_{i,c} w_{i,c} \tag{5.13}$$

Equation (5.11) exploits the Arrival Theorem, which also holds for multiclass queueing networks. With it, the mean values for population $\vec{K}$ are calculated directly from the mean values for the population vectors $\{\vec{K}-\vec{e_1}, \vec{K}-\vec{e_2}, \ldots, \vec{K}-\vec{e_C}\}$. Starting at the base case with population vector $\{0, 0, \ldots, 0\}$, exact MVA must be performed for each of the population vectors in

$$\{0, 1, \ldots, k_1\} \times \{0, 1, \ldots, k_2\} \times \ldots \times \{0, 1, \ldots, k_C\},$$

so that computational complexity is $O(\prod_{c=1}^{C} k_c)$ for exact multi-class MVA. This is a substantial obstacle to using this algorithm for studying a system like the J-Machine, where there might be 512 classes (one per processor) with four customers per class, leading to the need to recurse over a space of 4^{512} different population vectors: clearly intractable. To overcome this obstacle requires the use of approximate MVA.

5.1.4 Approximate Mean Value Analysis

Approximate MVA (AMVA) reduces the computational complexity of exact MVA by trading away provable accuracy[6] and becoming an iterative approximate numerical algorithm. Rather than compute MVA solutions at all of the $\prod_{c=1}^{C} k_c$ intermediate population vectors, AMVA instead iteratively refines a queue size guess for population vector $\vec{K}$ by using this estimate to approximate the mean values for the neighbor population vectors $\vec{K} - \vec{e_c}$.[7]

Researchers have developed a variety of techniques for using the values $q_{i,c}(\vec{K})$ to estimate $q_{i,c}(\vec{K} - e_{c'})$. The best-known is the Schweitzer approximation (mentioned in [CN82]):

$$q_{i,c}(\vec{K} - \vec{e_{c'}}) = \begin{cases} q_{i,c}(\vec{K}) & \text{if } c \neq c' \\ \frac{k_c - 1}{k_c} q_{i,c}(\vec{K}) & \text{if } c = c' \end{cases} \tag{5.14}$$

The Schweitzer approximation is based on an assumption that the system is behaving "reasonably" in the following ways:

[6]Provable, that is, for a Poisson queueing network.

[7]Recall that the notation $\vec{e_i}$ denotes the unit vector with a 1 in position i.

- the number of class c customers at some server i is not affected by removing a customer of some other class $c' \neq c$ from the network,
- and when a customer of class c is removed from the system, the number of customers of class c at server i is reduced exactly proportionally to $\frac{k_c-1}{k_c}$.

This is an empirically good assumption that improves in accuracy as k_c grows.

With Equation (5.14), the MVA equations (5.10) through (5.13) form a large system of nonlinear equations in the variables $q_{i,c}(\vec{K})$. The AMVA algorithm solves for $q_{i,c}(\vec{K})$ via iterative refinement:

1. Guess initial values for $q_{i,c}(\vec{K})$.
2. Use Equation (5.14) with the current values of $q_{i,c}(\vec{K})$ to get approximate values for $q_{i,c}(\vec{K} - \vec{e_{c'}})$.
3. Using $q_{i,c}(\vec{K}-\vec{e_{c'}})$ from the previous step, compute new values for $q_{i,c}(\vec{K})$ and substitute them for the current value.
4. Repeat steps 2 and 3 until the $q_{i,c}(\vec{K})$ converge adequately.

Silva et. al. [dSeSLM84] proved that there is one unique meaningful solution to a multiclass AMVA model, which the iterative procedure is guaranteed to find.[8]

5.1.5 Blocking

To this point, the chapter has described only the basic exact and approximate mean value algorithms. Next, this section describes a way of extending them to include an approximate model of blocking. Blocking due to the wormhole network is a large portion of the AV model. This section introduces the basic concepts underlying the AV blocking model.

In the basic MVA algorithm without blocking, the waiting time experienced by a customer arriving at a queue is the sum of the time required to serve customers waiting in the queue

[8] It is wonderful that the equations have one and only one solution, and that the simple refinement of guesses is guaranteed to find it.

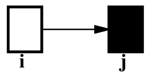


Figure 5-3: Server i feeds only one server j.

and the residual service time for the customer at the head of the queue, as in Equation (5.11). To incorporate blocking into the MVA model, this equation is modified so that the mean waiting time also considers the probability that the downstream queue is blocked and residual time until the blocked downstream queue clears.

It is easiest to allow only single-element queues when modeling blocking. This does two things: it eliminates the need to consider the time spent waiting for service, and it makes the probability that the downstream queue is blocked equal to the probability that the queue is occupied by one customer, rather than that the queue is filled.[9]

For basic (no blocking) MVA, Equation (5.11) says that the waiting time is the combination of the expected service time for queued customers and the residual time for the customer in progress. To model blocking, this is replaced with an expression that considers the residual time for the customer in progress, and the residual time for the downstream queue.

Consider first the case where some server j is always the output of server i for class c (i.e., $r_{ij,c} = 1$, in Figure 5-3). In this case, the waiting time for a customer of class c arriving at server i is:

$$w_{i,c}(\vec{K}) = \frac{1}{\mu_i} + \left(u_j(\vec{K}) \frac{w_{j,c}(\vec{K})}{2} \right). \tag{5.15}$$

This includes the waiting time $1/\mu_i$ to complete service at i, plus an approximation of the residual waiting time, $w_{j,c}(\vec{K})/2$, at j weighted by the probability $u_j(\vec{K})$ that server j is

[9]Note how this relates to the subject of this thesis. The thesis is interested in studying the "filling" events in the processor's message queue. Chapter 3 is devoted to estimating the rate of filling events in the processor queue. The frequency of blocking events is a function of the mean *and* the variance; but the methods in this chapter are concerned only with the mean values. How then can one incorporate blocking? The solution is to approximate. This is easier if the queues can hold only a single customer. When the capacity is only one, then the mean queue size *is* equal to the second moment (because $1^2 = 1$), so the variance can be estimated.

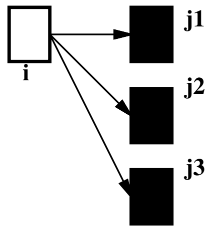


Figure 5-4: Server i feeds multiple servers j.

occupied.[10]

If server i has multiple outputs (Figure 5-4), the waiting time equation can be:

$$w_{i,c}(\vec{K}) = \frac{1}{\mu_i} + \sum_j r_{ij,c}\left(u_j(\vec{K})\frac{w_{j,c}(\vec{K})}{2}\right). \tag{5.16}$$

In this case, the residual time on each possible output j is weighted by both the probability that j selected and by the probability that j is occupied.

There are several assumptions of independence implicit in these equations; the strongest being that the occupancy of the downstream queue is independent of motion from the upstream queue.

Using blocking equations of this type within the model sets up many more constraints in the set of nonlinear equations that AMVA must solve. Results for convergence and existence of solutions with blocking equations, mentioned in Section 5.1.4, apply only to simple queueing networks. In future work, one might want to try to show that solutions exist when blocking equations are added.

[10]Dividing the arrival waiting time by two to approximate the residual waiting time is a rough cut which assumes that the motion from i to j is uncorrelated with motion out of j. This is reasonable if the service time probability distribution at i and j are spread out; if they were deterministic servers, one would expect this approximation to be inaccurate. In a synchronous network, the service distribution for flits in the communication network is highly deterministic (made random only by the sharing of the physical channel by the opposite direction). The AV model compensates for this with another trick: making the blocking time at i depend mostly on a spatially distant downstream server (where the head of the wormholed message resides); this probably reduces errors because the spatial separation reduces correlation and because there is no accumulation of blocking error working along the message.

5.1.6 Summary

This section reviewed MVA algorithms. The Arrival Theorem, which is the foundation of these algorithms, was introduced in Section 5.1.1. Using the Arrival Theorem, the basic and exact MVA algorithm was given in Section 5.1.2. This was extended for multiple classes in Section 5.1.3 in order to explain how the AV model incorporates Request/Reply workloads. Exact multi-class MVA is computationally expensive, so the iterative approximate MVA algorithm (AMVA) using the Schweitzer Approximation was given in Section 5.1.4. Finally, a modification to the MVA equations to handle blocking-after-service with single-element queues was given in Section 5.1.5.

5.1.7 Heads Up

This review served as a basis for understanding the Adve and Vernon wormhole model. This model appears in Appendix A. The next sections rely upon a detailed understanding of the notation and process flow for the model.[11]

5.2 AV Model Results: Fair Router

This section reports queue size results for the AV model with workload parameters given in Table 5.1. This places the system at a load point where the J-Machine experiences a large number of queue buffer overflows (see Section 4.2). Message handling time is short so that the machine is bisection-limited. With a moderate number of messages on each processor, imbalances in the service rate can lead to large accumulations (see Section 3.4).

As this section will show, the immediate results of the AV model do not conform qualitatively to the measurements on the J-Machine. The distribution of the location of queue buffer overflows on the J-Machine showed a high concentration in the center of the machine, but the result of the AV model is that the queue sizes are largest on the edges of the machine.

[11]The reason that the model is segregated into an appendix is because it is very long and also to make explicit my scholarly debt to Adve and Vernon; although I have made improvements (described in the appendix) and reimplemented their model for the J-Machine, the the model remains largely their innovation and should be considered such.

Term	Value	Explanation
N	512	Processors, in an 8x8x8 bidirectional 3D mesh.
F_{sd}	$\frac{1}{N-1}$	Uniform traffic.
L_j	8	All messages same length.
P_j	N/A	With all messages same length, this parameter is irrelevant.
N_{out}	5	Messages per processor, near the threshold at which overflows became frequent in Chapter 4.
τ	25	Fine grained software, with short handlers.

Table 5.1: Model parameters for simulation

The section describes how the cause of this discrepancy is the mismatch between the AV router model and the J-Machine's router: the AV router is fair, while the J-Machine's router is unfair.

This section is structured as follows. Section 5.2.1 contains and discusses a plot of the AV model's predicted equilibrium mean queue size. It shows the largest accumulations of customers at the edge of the machine. Section 5.2.2 shows that this is qualitatively different from the result obtained by simulating the J-Machine. However, Section 5.2.3 shows that when the simulator runs with fair routers, the mean queue sizes are qualitatively consistent with the results from the J-Machine. This helps motivate Section 5.3, which develops a model of the unfair routers.

5.2.1 AV Result

Figure 5-5 shows the mean node queue sizes predicted by the AV model for a 512-processor J-Machine. This is for the $Z = 3$ processor plane; the model predicts roughly the same distribution in other planes. The distribution shows accumulations of messages in the queues of processors near the edges of the machine, with the largest accumulation near the corners. Note that the variation between high and low is more severe in the X direction than in the Y direction; this is because X is first in the dimension order chosen by the router. The maximum mean queue sizes in Figure 5-5 is about 6.5. This is only 5% of the capacity of the processor queue in the J-Machine.

There are two discrepancies between this figure and the qualitative results obtained on the J-Machine:

1. The queue size distribution is largest near the corners of the plane, but in the J-Machine the number of overflows is largest near the center of the machine. As the section shows next, this appears to be due to the unfairness of the J-Machine's routers.

2. Even the largest predicted mean queue size (6.5 messages) at this load point is small in relation to the queue capacity (128 messages). One might expect that this would render the occurrence of overflows rare, but in fact at this load point on the J-Machine, queue buffer overflows were frequent.

5.2.2 Simulation with Unfair Routers

Chapter 4 showed that the queue buffer overflows occurred most often in the center of the cube. This observation alone does not necessarily imply that the mean queue sizes are largest in the center of the machine, but it is possible to show that this is the case by using the J-Machine simulator.

Figure 5-6 shows the measured average queue size for the $Z = 3$ slice for one run of a simulated J-Machine running the Request/Reply application for 10^5 clock cycles. The figure shows that, indeed, the mean queue sizes are largest near the center of the machine.

Note that in cases where center queues are largest, one would expect the overflow behavior to be more severe. The queue sizes in the simulator are measured in flits, and the peak queue size in Figure 5-6 is about 300 flits, or 74 four word messages; this differs from the prediction for the AV model, where the largest predicted mean queue size is 6.5 messages.

This large difference in magnitude can be explained by the fact that there is only one center of the machine, while there are eight corners. In queueing networks, equivalent bottleneck nodes divide the accumulation among themselves evenly. The eight equivalent cube corners will divide the traffic evenly among themselves, reducing the severity of the bottleneck compared to the severity of the bottleneck traffic accumulated in the smaller center.

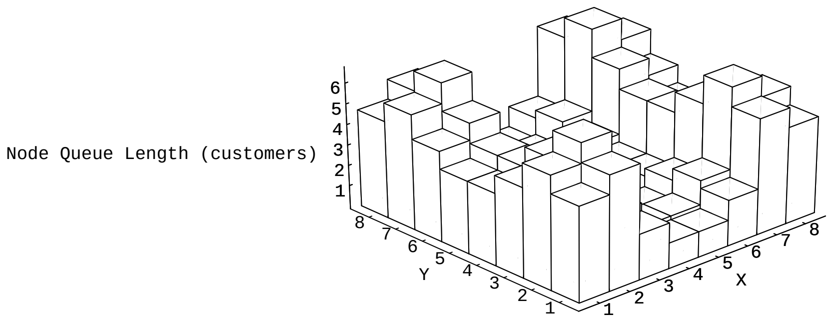


Figure 5-5: AV/fair predicted mean node queue size (in customers, not flits) over processors for the $Z = 3$ plane of a 512-processor J-Machine running the uniform heavy Request/Reply protocol.

If one multiplies the accumulation in the corners (6.5) by the number of corners (eight), the total accumulation (52 messages) is roughly close to the 74 messages in the center accumulation. This is consistent with the conclusion that there is excess traffic that in the fair case gets distributed over the corners, but in the unfair case gets packed into the center.

5.2.3 Simulation with Fair Routers

Figure 5-7 shows the mean simulated queue size for a 512-processor J-Machine, but this time with fair routers. In this case, the figure shows accumulations in the corners and edges of the plane, just like the AV model predictions. [12]

Why does the fairness of the routers change the location where traffic accumulates? See Figure 5-8. If the output of a switch at the confluence of two streams faces backpressure, and if it is fair, it should "evenly" direct backpressure to the two input streams. In the J-Machine, the bisection of the machine is the bottleneck and backpressure from this bottleneck is propagated back from the bisection to the processors. Were the J-Machine to have fair routers, this backpressure would be divided evenly backward from the bottleneck. Processors further from the bisection would have available only a small fraction of the bandwidth available to the center nodes. As a result, the service rate of the edge nodes is reduced and traffic accumulates in those nodes.

With unfair routers, the situation is as in the lower drawing of Figure 5-8. With traffic entering from the higher priority inputs from upstream processors, downstream processors are only allocated bandwidth not requested by the upstream processors. In fact, they can be completely blocked from entering, as in the hypothetical situation in Figure 5-8 where demand from upstream processors matches the available output bandwidth.

So, accurately modeling the J-Machine with MVA appears to require changing the equations

[12] The figure is not symmetric because of the particular way that the measurement was made: with only one run and with the average queue size calculated as the average of the queues sampled on each cycle. However, the values from cycle to cycle are not independent: the queue size on one cycle is likely to be close to the queue size on the previous cycle. With the actual J-Machine hardware not necessarily ergodic, the mean queue size resulting from a single run reflects the particular configurations that the particular run goes to. In the case of Figure 5-7, the queue sizes reflect a bias with higher traffic accumulation in the $Y = 1$ row of the plane.

for an unfair router. How can this be done? Next, Section 5.3 provides an unfair router model.

5.3 Unfair Router Model

The previous section showed that the fairness of the J-Machine's routers substantially modifies the qualitative distribution mean queue sizes predicted by simulation. This section develops my modification to the original AV model for unfair routers.

This is a modification to the equation that computes the waiting time that the head flit of a message incurs when it arrives at the input to a channel. With fair routers, the message need only wait for messages that are in progress, arrive simultaneously, or are present when the other message arrives. There is no need to wait for messages which have not yet arrived to the channel. The new message waits only for messages present on arrival.

When the router is unfair, this is not the case. After waiting for the three cases above, the new message may need to wait for messages which arrived from higher priority inputs during the wait. In other words, in a fair router, after the channel would have cleared for the waiting message, it still may need to wait for any newer high priority inputs. Under heavy load, a low priority message may need to wait for a long time.

The modifications to the waiting time equation simply insert into each of the three terms an estimate for the cost of this succession of higher priority messages. The cost of this succession is an estimate of the waiting time from the moment that the channel clears, which is the product of an estimate of the number of higher priority messages that pass and the expected residence time a higher priority message on the channel. To do this, the section now introduces the parameters given in Table 5.2 to the model.

These parameters are used to modify the fair model channel waiting time equation, Equation (A.7) repeated again here for reference:

$$\begin{aligned} w_{c|I} &= \sum_{i \neq I} \sum_{j} \sum_{k=1}^{L_j} u_{j,c,i}[k] \left(\frac{1}{2} r_{j,c,i}[k] + \sum_{l=k+1}^{L_j} r_{j,c,i}[l] \right) \\ &\quad + \sum_{j} \left(\frac{u_{j,c,I}[L_j]}{1 - \sum_{j'} \sum_{l=1}^{L_{j'}-1} u_{j',c,I}[l]} \times \frac{r_{j,c,I}[L_j]}{2} \right) \end{aligned}$$

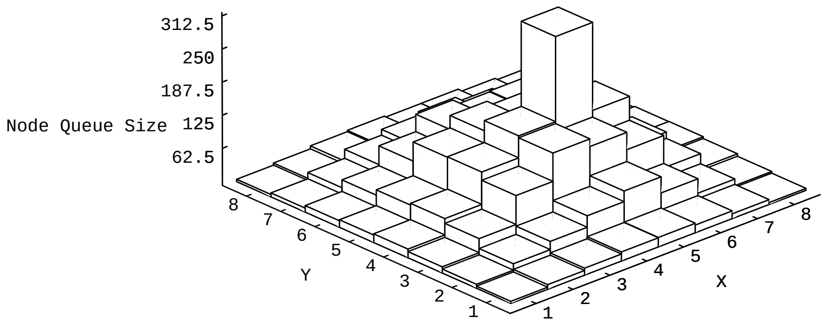


Figure 5-6: Simulated node mean node queue size (flits, not customers) on $Z = 3$ plane of a 512-processor machine. With unfair routers, traffic accumulates in the center of the machine.

Term	Definition
$u_{\text{out}\mid\text{clear},j,(c-1)_i}$	The probability that a type j header is on input port i to channel c at the moment that a tail flit leaves channel c.
$b_{c,I\mid\text{clear}}$	The probability that input port I to channel c will be blocked by other channels with higher priority, at the instant that channel c clears.
$w_{c,I\mid\text{clear}}$	The waiting time that a head flit on input I must incur for higher priority messages to go by, on channel c, from the instant that a tail flit clears channel c.
α	Hole compensation factor.

Table 5.2: New parameters for the AV model for unfair routers.

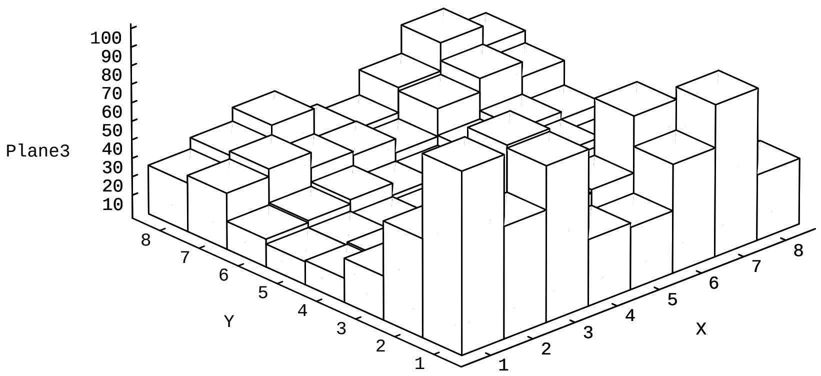


Figure 5-7: Simulated node mean node queue size (flits, not customers) on $Z = 3$ plane of a 512-processor machine. With fair routers, this matches qualitatively the shape of Figure 5-5. Quantitatively, this simulation shows accumulations of about 50 flits in the low corners, and 100 flits in the high corners. Figure 5-5 predicted accumulations of 7 customers in the near-corners; with messages being 8 flits long, we see that the model underestimates the results of simulation.

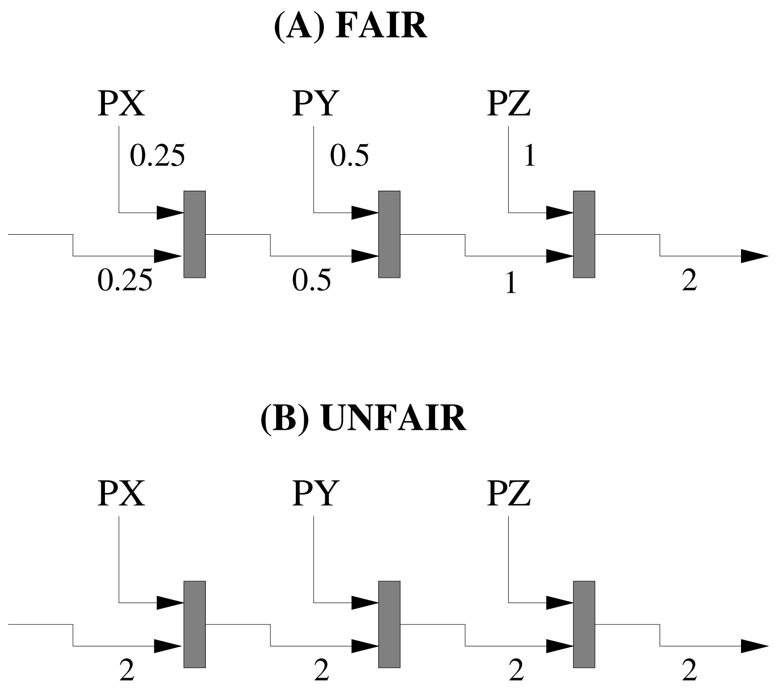


Figure 5-8: Confluences of traffic. Each processor is offering traffic at rate 1 and the final output can accept traffic at rate 2. Numbers indicate the accepted traffic rate on that wire. (A) Fair routers: fair switches in the routers divide the available outgoing traffic evenly between both inputs. Processor PX, facing more traffic confluences of traffic than PZ, gets a smaller fraction of the available bandwidth. (B) Unfair routers, with the "through" input favored over the entering input. In this case, we assume that there is incoming traffic of rate 2 entering from the left, so none of PX, PY, and PZ receive any bandwidth.

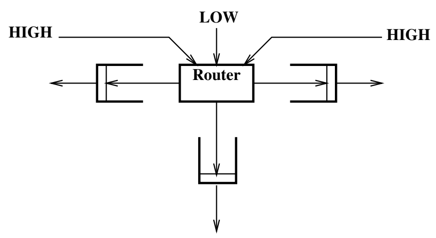


Figure 5-9: Example input priorities within the J-Machine router. The pass-through inputs to the router are given priority over the inputs that drop in from the processor or from higher dimensions. A message from LOW will not proceed to the output if there is any competing message on HIGH.

$$+\sum_{i\neq I}\sum_{j}\left(u_{\mathrm{out},j,(c-1)_i}[1]\sum_{l=1}^{L_j} r_{j,c,i}[l]\right).$$

See Section A.5.3 for a description of this equation. Equation (A.7) is fair because a message arriving on I need not wait for more than a single message on any other input. In the first part of the equation, there is no cost for higher priority messages that might arrive or be waiting on other channels after the in-progress message completes. Similarly, there is no cost imposed in the second part of the equation for higher priority messages which might intervene between the previous message on the same channel and the one that just arrived. Finally, the third part of the equation, for simultaneous arrivals, does not account for whether the message that arrived simultaneously is of higher or lower priority.

To model unfairness, first define a priority measure $\pi_{c,i}$ on the channels that is the priority of input i to channel c. In the J-Machine, the priority of the inputs to the channels is as in Figure 5-9. If channel c is free and if several inputs i to channel c have their head flit waiting, then the channel with the largest value of $\pi_{c,i}$ is always selected. The router is a wormhole router, so once a channel is selected by a message, it is never preempted. The selected message owns the channel until its tail flit clears.

The first step in modeling the unfair routers is estimating the probability that a head flit is present on some input i to channel c, sampled at the instants when the tail flit clears channel c. The AV model already has a variable, $u_{\mathrm{out},j,(c-1)_i}[k]$ that tracks the utilization of i by flit

k of messages of type j that are destined to channel c. If sampled at random instants, this is a good estimate for finding the designated flit on channel i for c. However, the sampling instants for unfair modeling are at the instants when the channel c clears, and these instants are not independent of the presence of a head on i. It is necessary to adjust $u_{\text{out},j,(c-1)_i}[k]$ to account for this non-independence.

Channel c will not clear at the instants that a non-head flit is present on channel $(c-1)_i$. The only times that $(c-1)_i$ could be sampled is when it is empty, or when the head flit is there. Therefore, the adjusted probability (indexed by message type j) is:

$$u_{\text{out}|\text{clear},j,(c-1)_i} = \frac{u_{\text{out},j,(c-1)_i}[1]}{1 - \sum_{j'} \sum_{l=2}^{L_{j'}} u_{\text{out},j',(c-1)_i}[l]}. \tag{5.17}$$

This adjustment is in the same spirit as the part of the second term of Equation (A.7) which compensated for the non-independence head and tail arrivals on the same instant.

In fact, this approximation turns out to be relatively poor. Basically, the problem is that when the system is at saturation, the spaces between messages are most often caused by a message that is upstream of input i turning away, creating a sort of "anti-worm" which is a succession of non-flits, or holes. Once a hole appears on a channel, it is likely to be followed by another hole, for the length of the message. In the time slots when the successor holes are on the channel, it is unlikely that the channel-clearing event occurs. This behavior results in the need for further adjustment to Equation (5.17).

The utilization of the input i to c by holes is

$$1 - \sum_{j'} \sum_{l=1}^{L_{j'}} u_{\text{out},j',(c-1)_i}[l].$$

The correlation in the succession of holes on an input is high; generally, a succession of holes is created when an upstream message turns away before reaching c; the length of this anti-worm is approximated well by the factor α:[13]

$$\alpha = \frac{4}{\sum_j L_j}. \tag{5.18}$$

[13] The fact that the model is ignoring send faults (see the discussion and footnotes for the Node Queue on page 242) is relevant. In particular, SEND faults due to node queue buffer overflows create holes which are longer than the duration of a single message. For simplicity, I am ignoring this effect (it requires introducing second moments again).

This is the inverse of the average message length, and it us used to reduce the hole utilization in Equation (5.17) as follows:

$$u_{\text{out|clear},j,(c-1)_i} = \frac{u_{\text{out},j,(c-1)_i}[1]}{\sum_{j'} u_{\text{out},j',(c-1)_i}[1] + \alpha(1 - \sum_{j'} \sum_{l=1}^{L_{j'}} u_{\text{out},j',(c-1)_i}[l])}. \tag{5.19}$$

The probability that a low priority input I will be blocked by higher probability channels i is estimated by subtracting from one the probability of the complementary event with no channels clear:

$$b_{c,I|\text{clear}} = 1 - \prod_{i:\pi_{i,c} > \pi_{I,c}} \left(1 - \sum_j u_{\text{out|clear},j,(c-1)_i}\right). \tag{5.20}$$

This is the probability of having to wait for another message when the channel clears. But when that message clears, another message might follow, and another after that. The low priority channel must wait for all of them. Each time the channel clears, it is as if a coin is flipped with heads-probability $b_{c,I|\text{clear}}$; and the expected length of a sequence of heads on these biased coin flips is:

$$\sum_{f=1}^{\infty} (b_{c,I|\text{clear}})^f = \frac{b_{c,I|\text{clear}}}{1 - b_{c,I|\text{clear}}}. \tag{5.21}$$

The expected waiting time for channel I to acquire c, measured from the instant that channel c clears, is given by the following expression:

$$w_{c,I|\text{clear}} = \left(\frac{b_{c,I|\text{clear}}}{1 - b_{c,I|\text{clear}}}\right) \times \left(\sum_{i:\pi_{i,c} > \pi_{I,c}} \sum_j u_{\text{out|clear},j,(c-1)_i} \sum_{l=1}^{L_j} r_{j,c,i}[l]\right) \tag{5.22}$$

This multiplies the expected number of messages by the expected cost of those messages.

This value gets inserted into the fair waiting time equation, to make it the unfair waiting time equation:

$$\begin{aligned} w_{c|I} \quad = \quad & \sum_{i \neq I} \sum_j \sum_{k=1}^{L_j} u_{j,c,i}[k] \left(\frac{1}{2} r_{j,c,i}[k] + \sum_{l=k+1}^{L_j} r_{j,c,i}[l] + w_{c,I|\text{clear}}\right) \\ & + \sum_j \left(\frac{u_{j,c,I}[L_j]}{1 - \sum_{j'} \sum_{l=1}^{L_{j'}-1} u_{j',c,I}[l]} \times \left(\frac{r_{j,c,I}[L_j]}{2} + w_{c,I|\text{clear}}\right)\right) \\ & + \sum_{i:\pi_{i,c} > \pi_{I,c}} \sum_j \left(u_{\text{out},j,(c-1)_i}[1] \left(\sum_{l=1}^{L_j} r_{j,c,i}[l] + w_{c,I|clear}\right)\right) \end{aligned} \tag{5.23}$$

In the first term, $w_{c,I|clear}$ has been inserted for the additional waiting time after a message that was in progress on arrival has cleared, and so on.

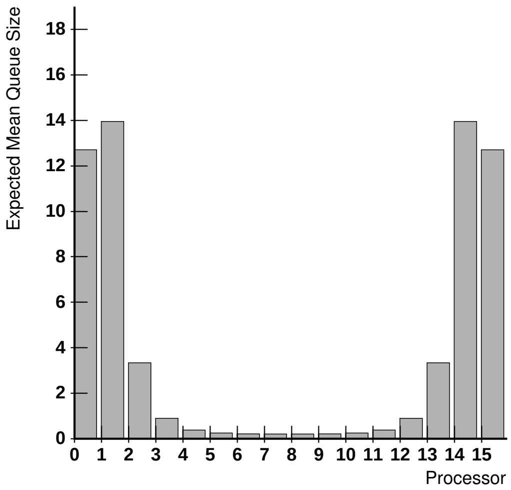


Figure 5-10: AV/fair mean queue size (messages) predicted for 16 processors running a heavy workload.

5.3.1 Queue Size for 16-processor Linear Array

To demonstrate how these equations alter the predicted mean queue sizes in the machine, see Figures 5-10, 5-11 and 5-12. These plots are for a linear array of $N = 16$ processors, with $\tau = 10$, $L_j = 8$, $N_{\mathrm{out}} = 5$, an extremely heavy load.

First, Figure 5-10 shows the mean queue size for the machine with the original fair router model. The figure shows severe accumulation at the end nodes. In Figure 5-11, with the unfair model but with no correction for the holes ($\alpha = 1$), there some change, but the model still shows accumulation in the queues near the endpoints. The queues at the end do not show accumulations. This makes sense, because they do not need to wait for messages from any other processors to pass into the network.

Setting $\alpha = 0.1$ causes the model to get the result shown in Figure 5-12, which shows the peaks moving toward the center. (Note that they don't move the peaks completely into the center.) This result is counterintuitive; I had hypothesized that the accumulation would show up in the center of the machine. However, simulation confirms that the mean queue size

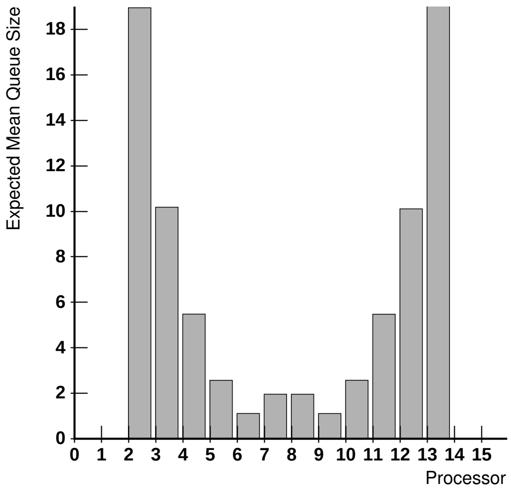


Figure 5-11: AV/unfair, with $\alpha = 1$. Mean queue size (messages) predicted for 16 processors running a heavy workload.

follows the pattern where there are two peaks moving toward but not reaching the center: the simulation results are in Figure 5-13. The strange model prediction looks qualitatively a lot like the results from simulation. The simulation results show peaks on the order of 100 to 150 flits. The model predicts accumulations of about eight messages; with eight flits per message the model predicts accumulations of 64 flits. So there is a quantitative difference in the size of the predictions; this is similar to the quantitative deviation with qualitative consistence in Figure 5-7.

We have the attractive result that the surprising qualitative analytical model prediction is matched by a surprising qualitative simulation result.[14]

Therefore, using the unfair router model can get qualitatively correct results for the J-Machine style unfair routers. What happens when we run this for the larger, 512-node, machine? The results are discussed in the next section.

[14] I have not yet investigated the quantitative deviation. My intuition is that this is not a bug. Instead, I suspect that the qualitative distinction is simply due to some of the more aggressive modeling assumptions (rigid messages, in particular) that do not hold as well at heavy processor loads.

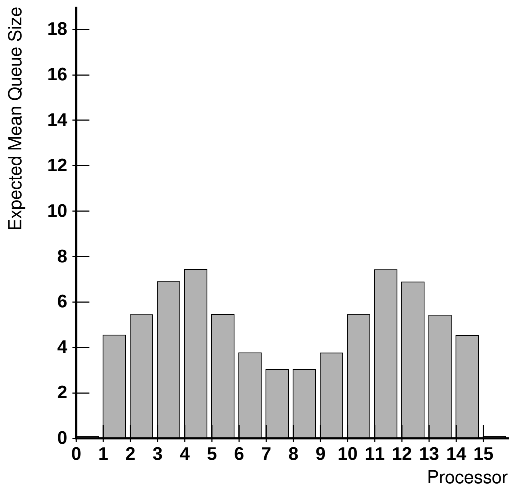


Figure 5-12: AV/unfair, but this time with $\alpha = 0.1$. Mean queue size (messages). This matches, qualitatively, the shape of the result obtained from simulation, Figure 5-13.

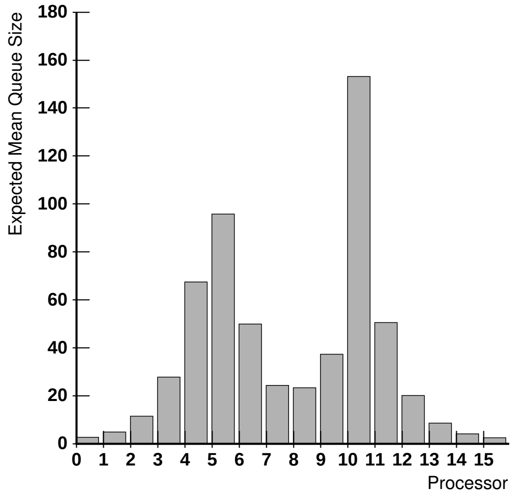


Figure 5-13: Simulated mean queue size (flits) for one run of the 16 processor linear array.

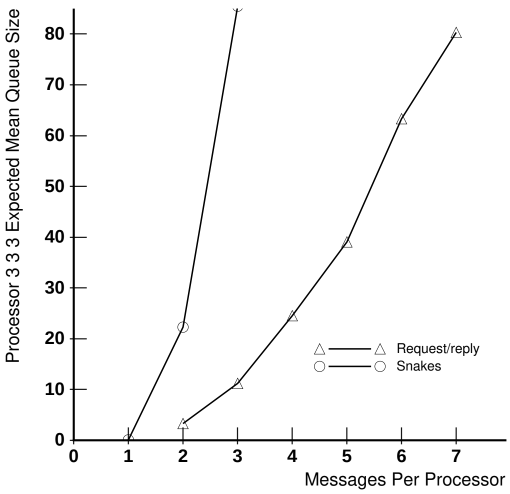


Figure 5-14: MVA predicted mean queue size (messages) on center processor (3,3,3) vs. number of messages per processor, running Request/Reply and the Snakes workload.

5.3.2 Unfair Queue Size Predictions for 512-node J-Machine

Figure 5-14 summarizes the relationship between maximum mean node queue size and number of messages per processor, with $L_j = 8$ and $\tau = 25$ for a 512-processor J-Machine. The plot shows that the mean queue size increases approximately linearly with N_{out}. Increasing N_{out} by one increases the number of messages in the system by 512; on average, about 16 more messages end up on the most heavily loaded server.

Figures 5-15 through 5-20 show the evolution in the spatial distribution (through one of the center planes of the processor cube) of mean node queue size as the number of customers on the system increases. When there are only two customers per processor on the system, as in Figure 5-15, then the node queues are nearly empty, with a slight accumulation in the center of the machine. As more customers are added, the mean queue size grows, but primarily for the center nodes.

Figure 5-21 shows how the maximum mean queue size falls as the processor service time τ grows. With $N_{\text{out}} = 5$, with just a few messages per processor, the queue size can be reduced

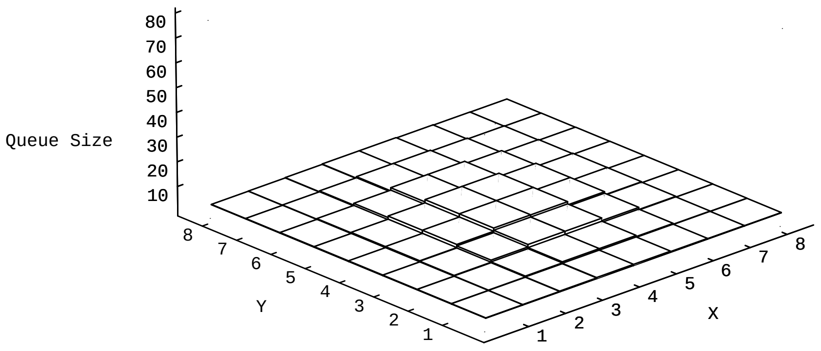


Figure 5-15: MVA predicted mean queue size (messages) for plane 3 of a 512-processor array with 2 messages per processor and service time 25, running Request/Reply workload.

almost to zero by increasing the service time so that the machine is not throughput-limited by the network. This is the behavior that was observed in Chapter 4. Figure 5-22 through Figure 5-26 show the evolution in the size of the queues as the handler time τ increases. With a short handler time of $\tau = 15$ cycles, the peak in mean queue size is similar to the peak with $\tau = 25$. The peak decreases to zero as the handler time increases.

5.4 Modifications to AV for Snakes Workload

This section simplifies the AV model, so that it models a Snakes workload, rather than a Request/Reply workload. This simplification is done by collapsing the multiclass Request/Reply workload into a single-class workload. The result of this is even more severe accumulation in the center of the machine.

In Chapter 4, the J-Machine exhibited a different behavior when running a Snakes workload than it did when running a Request/Reply workload. The Snakes workload showed

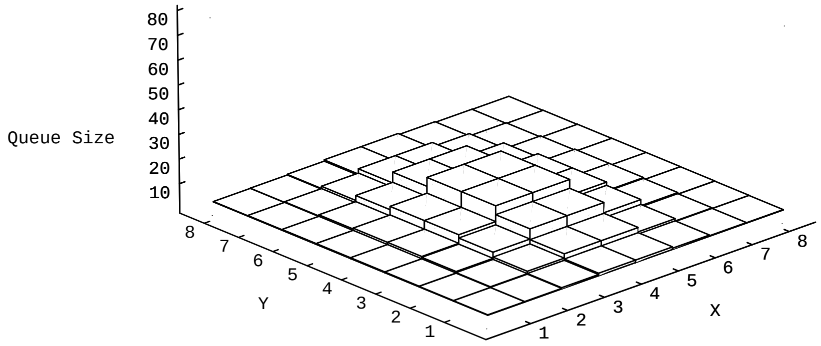


Figure 5-16: MVA predicted mean queue size (messages) for plane 3 of a 512-processor array with 3 messages per processor and service time 25, running Request/Reply workload.

a sharp drop in performance when the machine was network-limited. In contrast, the Request/Reply workload showed many fewer overflows and exhibited much better performance. These results are consistent with the predictions of this modified AV model.

5.4.1 Changes to the AV model for Snakes

To model the Request/Reply workload, the AV model uses multiple classes. Each processor has a corresponding class; the outstanding messages which return home to that processor are in that class. For example, on a 512-processor J-Machine with four messages per processor, there are 512 different classes, each containing four customers. Each class routes differently from each other because they route back (reply) to the own home processor after each outgoing message (request).

In the Snakes workload, customers do not return to their home processor.[15] Instead, all

[15]Though in the version that runs on the hardware, I have them return home after 100-1,000 hops so that the benchmark gives a metric, completion time, which can be used to determine throughput.

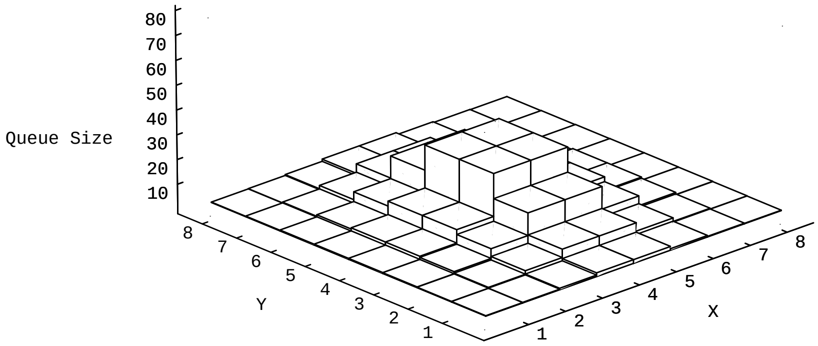


Figure 5-17: MVA predicted mean queue size (messages) for plane 3 of a 512-processor array with 4 messages per processor and service time 25, running Request/Reply workload.

customers route randomly through the network using the same random probabilities. Since all of the customers are identical, multiple classes are unnecessary. The changes to the model which follow are then mostly about stripping away multiple classes.

The routing probability matrix F_{sd} in the original AV model expressed common destination probabilities for requests in the multiple class model (Equation (A.1)). The single class general queueing model uses a simple routing probability matrix r_{sd}, which defines a set of visit ratios $\{v_i\}$ which are solutions to the traffic equations:

$$v_j = \sum_{i=1}^{N} v_i r_{ij}. \tag{5.24}$$

Let processor number 1 be the reference processor for the class; then $v_1 = 1$ by definition, to make the solution to the traffic equations unique.

The variables r_{sd} and v_i replace the routing matrix F_{sd}. This is a convenient way of identifying the model equations which need to be changed for Snakes: any equations which have $F_s d$ in them. This is intuitively satisfying, since it corresponds to those equations which incorporate information about the routing paths of customers.

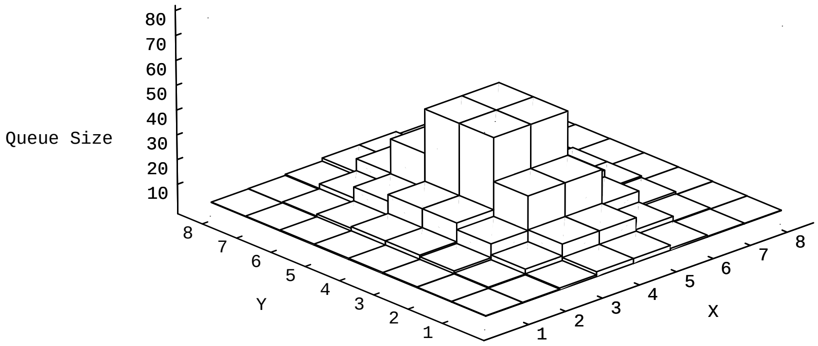


Figure 5-18: MVA predicted mean queue size (messages) for plane 3 of a 512-processor array with 5 messages per processor and service time 25, running Request/Reply workload.

Another change to the model involves reducing the number of message types. The original AV model allowed for four types of messages, each with different lengths. This was good for the model's original subject, shared memory systems with different lengths for read, write, data, and acknowledge messages. This modification for Snakes permits only a single type of message, which will change a number of equations. However, the analysis leaves the type designation (usually denoted j) on the model parameters, to maintain clarity with the original structure.

Most of the equations in the model – and particularly those that capture the behavior at individual channels – do not change. The ones that do change follow.

The core of all MVA algorithms is the round trip time to the reference servers. With only one class of message, only one round-trip time is needed.[16]

$$R = \sum_{i=1}^{N} r_{\text{proc}}[i] + r_{\text{network}}[i] \tag{5.25}$$

[16]Important: $r_{\text{proc}}[i]$ must be weighted by v_i. Equation (5.27) r_{network} incorporates this weight.

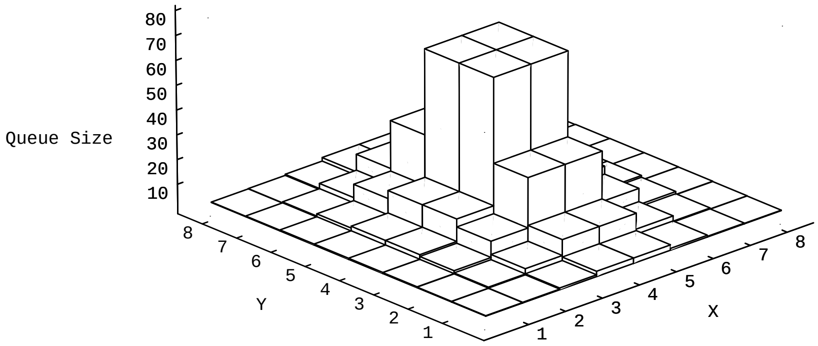


Figure 5-19: MVA predicted mean queue size (messages) for plane 3 of a 512-processor array with 6 messages per processor and service time 25, running Request/Reply workload.

The definition of $r_{\text{proc}}[i]$ and $r_{\text{network}}[i]$ is different for Snakes. Previously, they were the residence time at the processor and the residence time in the network for class i messages. Here, they are respectively: the residence time at processor i for our single class of message, and the residence time in the network when a message is sent from processor i. Both of these are weighted by visit ratio:

$$r_{\text{proc}}[s] = v_s r_{\text{proc},s}(N_{\text{out}}) \tag{5.26}$$

$$r_{\text{network}}[s] = v_s \sum_d r_{sd} r_{\text{msg1},sd} \tag{5.27}$$

The equation to weight the residence time is substantially simplified from Equation (A.8), partly because there is only one type j.

$$r_{j,c,i}[k] = \frac{\sum_{s=1}^{N} \sum_{d \in D_{c,i|j(s)}} v_s r_{sd} r_{j,c,sd}[k]}{\sum_{s=1}^{N} \sum_{d \in D_{c,i|j(s)}} v_s r_{sd}} \tag{5.28}$$

Channel utilization is also simplified. Note that N_{out} is now all of the messages on the system, rather than just the number of messages in each class. Also, the equation uses the

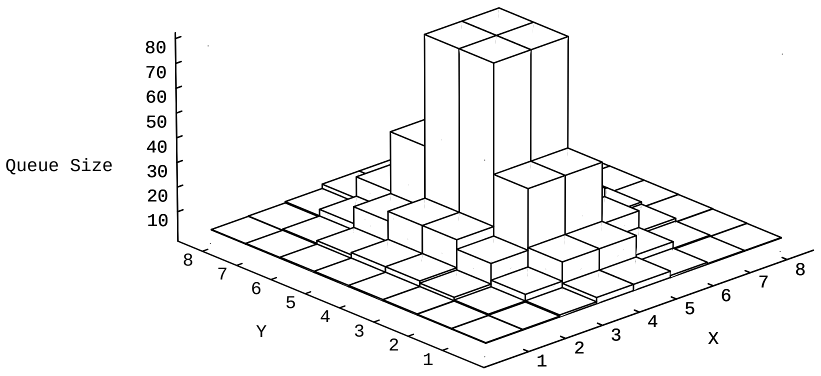


Figure 5-20: MVA predicted mean queue size (messages) for plane 3 of a 512-processor array with 7 messages per processor and service time 25, running Request/Reply workload.

single round trip time R.

$$u_{j,c,i}[k] = \sum_{s=1}^{N} \sum_{d \in D_{c,i|j(s)}} \frac{N_{\text{out}} v_s r_{sd} r_{j,c,sd}[k]}{R}. \tag{5.29}$$

Link utilization is calculated similarly:

$$u_{\text{link},c} = \sum_{s=1}^{N} \sum_{d \in \cup_i D_{c,i|j}(s)} \frac{N_{\text{out}} v_s r_{sd} L_j}{R}. \tag{5.30}$$

The value $w_{\text{node},s|q}$ is the waiting time at node s seen by an arrival of class q. There is only one class, but this does not necessitate changing Equation (A.14). Because there is only one type j of message, there's no need to change Equation (A.15).

The node queue size is substantially simplified because there is only one class and one message type. According to Little's Law and incorporating the Schweitzer approximation:

$$q_{\text{node},j,s|q} = \frac{(N_{\text{out}} - 1) v_s w_{\text{node},s|s}}{R}. \tag{5.31}$$

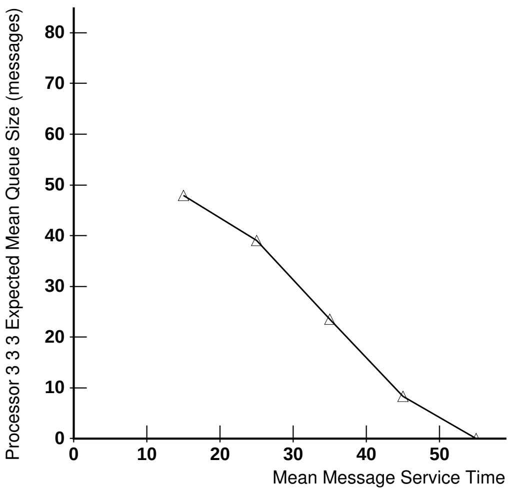


Figure 5-21: MVA predicted mean queue size (messages) on center processor (3,3,3) measured in messages vs. mean message service time τ, running Request/Reply (5 messages per processor).

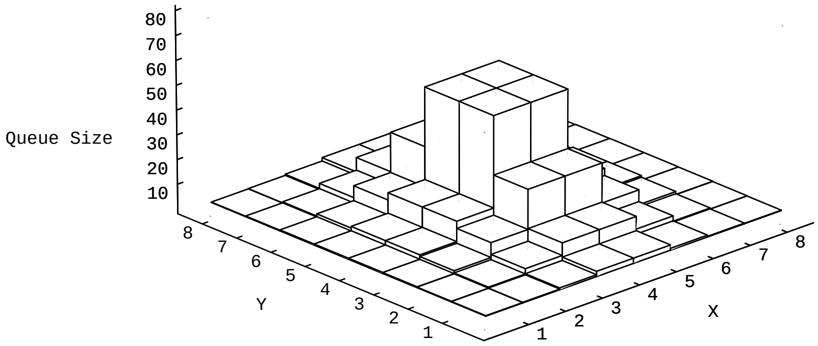


Figure 5-22: MVA predicted mean queue size (messages) for plane 3 of a 512-processor array with 5 messages per processor and service time 15, running Request/Reply workload.

The probability that the node is busy with a particular flit of a message is this weighted average:

$$b_{\text{node},j,s|q}[k] = \sum_d \frac{(N_{\text{out}} - 1) v_s r_{sd} r_{\text{node},j,sd}[k]}{R} \tag{5.32}$$

The node residence time is the weighted average.

$$r_{\text{node},j,s}[k] = \frac{\sum_{d=1}^{N} v_s r_{sd} r_{j,\text{node},sd}[k]}{\sum_{d=1}^{N} v_s r_{sd}} \tag{5.33}$$

Note that after cancelling v_s, the denominator is equal to 1.

As in the original AV model, processor queueing is handled with a subroutine AMVA, but with a single class, iterating between the approximations with N_{out} and $N_{\text{out}} - 1$ customers. The processor equations are as follows:

$$q_{\text{proc},i}(N_{\text{out}} - 1) = \left(\frac{N_{\text{out}} - 1}{N_{\text{out}}}\right) q_{\text{proc},i}(N_{\text{out}}) \tag{5.34}$$

$$r_{\text{proc},i}(N_{\text{out}}) = \tau \left(q_{\text{proc},i}(N_{\text{out}} - 1) + 1\right) \tag{5.35}$$

$$T_{\text{proc}}(N_{\text{out}}) = \frac{N_{\text{out}}}{\sum_i v_i (r_{\text{proc},i}(N_{\text{out}}) + r_{\text{network}}[i])} \tag{5.36}$$

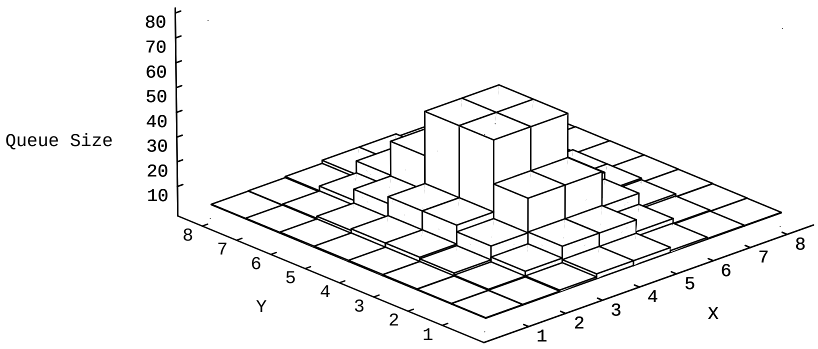


Figure 5-23: MVA predicted mean queue size (messages) for plane 3 of a 512-processor array with 5 messages per processor and service time 25, running Request/Reply workload.

$$q_{\text{proc},i}(N_{\text{out}}) = T_{\text{proc}}(N_{\text{out}})v_i r_{\text{proc},i}(N_{\text{out}}) \tag{5.37}$$

$$r_{\text{proc}}[i] = r_{\text{proc},i}(N_{\text{out}}) \tag{5.38}$$

5.4.2 Snakes Model Predictions

Not surprisingly, the mean values that the Snakes model gives are much more strikingly skewed than for the Request/Reply workload. See Figure 5-27, which gives the predicted mean queue size running Snakes, and compare it with Figure 5-15, which gives the expected mean processor queue size for the same workload when running Request/Reply. The later shows almost no accumulation in the center, while the former shows a substantial accumulation.

Figure 5-14 shows that this trend continues. With the Snakes workload, the mean number of messages that accumulate on the center processor is much higher than it would be with Request/Reply.

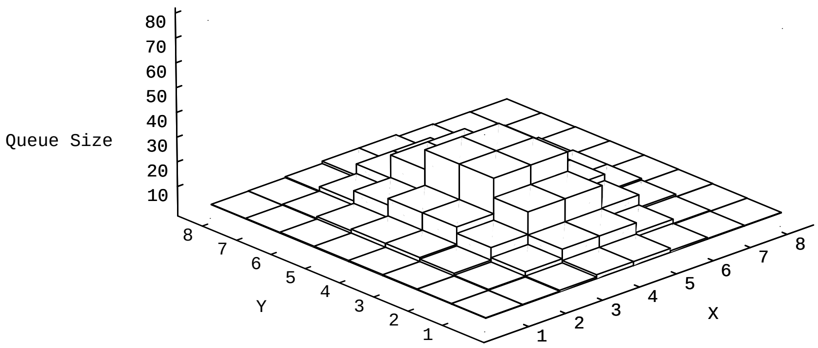


Figure 5-24: MVA predicted mean queue size (messages) for plane 3 of a 512-processor array with 5 messages per processor and service time 35, running Request/Reply workload.

The MDP processor message buffer can hold 512 words, room for 128 four-word messages. The model in Figure 5-14 suggests that the mean queue size exceeds this capacity threshold when there are between four and five messages per processor in the system.

This prediction matches observed hardware behavior. Figure 4-14 in Chapter 4 showed the measured effect of increasing the number of messages on the J-Machine. This figure shows a sharp drop in performance when the number of messages reaches four per processor.

5.5 Discussion

5.5.1 Why are the J-Machine's Routers Unfair?

This chapter shows that the particular heavy-load traffic dynamics that we experience are due to the unfairness in the J-Machine's routers. Why are the J-Machine's routers unfair? The designer of the J-Machine's routers answers this question:

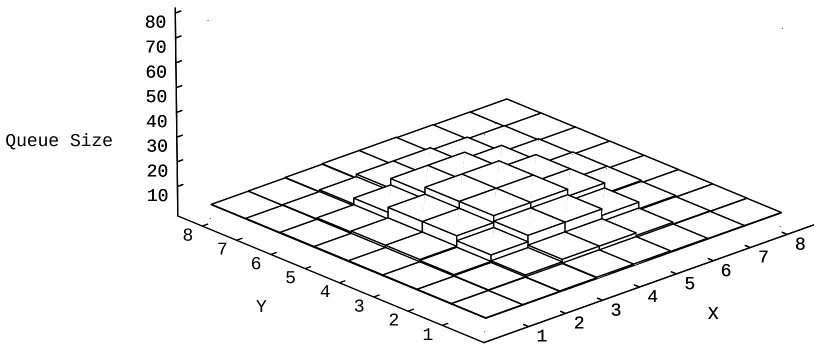


Figure 5-25: MVA predicted mean queue size (messages) for plane 3 of a 512-processor array with 5 messages per processor and service time 45, running Request/Reply workload.

> My argument for making the routers unfair was to try to drain messages as quickly as possible. The idea being that if two packets arrive simultaneously, the one closest to the destination (already within a dimension) should take precedence over the one entering the dimension. The simplest implementation was just a fixed, static priority. At one time, I expected to always have a cycle of idle time between packets on a dimension, so starvation would not occur. When that assumption changed, I thought about building a fairer priority scheme, but never bothered to build it [Nut].

This is reasonable. We designers never simulated a whole 512-processor J-Machine with routers (not enough simulation horsepower). Also, as far as I know, there has never been a case where the fairness has been observed (there are no heavy load fine-grained applications), other than at the heavy workload point studied in this thesis, though this is because no large fine-grained applications were written for them.

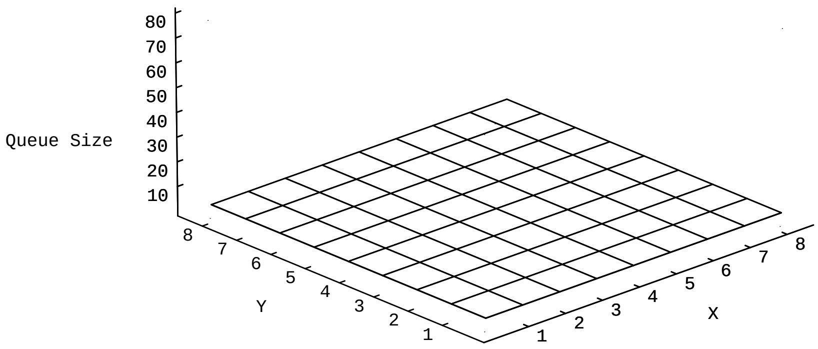


Figure 5-26: MVA predicted mean queue size (messages) for plane 3 of a 512-processor array with 5 messages per processor and service time 55, running Request/Reply workload.

Furthermore, it is not clear that the unfair routers are any worse than the fair routers. Suppose the routers were fair? In that case, simulation and the original AV model predict that queues would overflow on the edges of the machine, and performance would still be poor.

5.5.2 Architectural Implications

One goal of a computer architecture performance model is to facilitate the evaluation of different architectural options. What are the architectural implications of the results of this model?

The non-homogeneity of the J-Machine's topology – that is, the fact that a processor near the center of the machine looks into the network and sees a different environment than one on or near an edge, is clearly problematic. When faced with any degree of imbalance, a heavy random closed workload in a queueing network is prone to catastrophic accumulations.[17]

[17]The reasonable amount of imbalance is one of the results in Chapter 3.

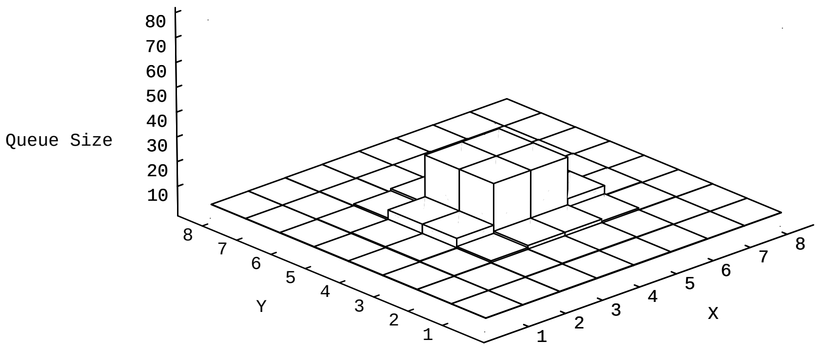


Figure 5-27: MVA predicted mean queue size (messages) for plane 3 of a 512-processor array with 2 messages per processor and service time 25, running the Snakes workload. Compare this with Figure 5-15, which for a Request/Reply workload did not show any substantial accumulation in the node queue.

Although the workload is balanced, the non-homogeneous topology of the J-Machine changes the relative achievable service rate of the processor, which unbalances the workload. This results in the severe accumulation of customers in the queues.

The fact that the J-Machine is a parallel computer makes this problem worse, because the tendency for the machine's programmers is to add more outstanding messages with more processors. On an N-processor J-Machine, even though the number of messages per processor might be a small number m, like 5, the total number of messages on the system is mN. Nearly all of this total is available to potentially accumulate on the bottleneck processors. If the size of the message buffer and m are held constant, one would expect that eventually the message queue size would set a limit on the number of processors in the machine.

This would suggest using a more homogeneous topology, like a torus. Looking into the network, each processor in a toroidal machine sees the same view as every other. There is a problem with toroidal wormhole networks, because non-homogeneity is introduced by the extra virtual channels that prevent network deadlock [AV91]. Still, this probably doesn't make that much of a difference in the size of the accumulation. It would be easy to test this hypothesis at another time.

The results of this model confirm the observation in previous sections (3.5.3, 4.10) that an unstructured workload like Snakes is going to fare more poorly than a structured workload like Request/Reply. This is unfortunate, because the message-passing philosophy that motivated the J-Machine's construction included the following claim: the message-passing style of communication was better (more wire-efficient) than shared-memory, because message-passing did not require "round-trip" communication. However, the message-passing style of communication (with Snakes as a representative) imposes a more stringent requirement for load balancing that is not present in a shared-memory style of communication (with Request/Reply as a representative). The J-Machine has not demonstrated the superiority of message-passing: rather, as documented in this thesis, it has demonstrated a weakness.[18]

One obvious architectural implication is that the buffers should be big. Although it runs against the fine-grained hardware philosophy, and ignores the speed/distance tradeoffs in a

[18]The religious debate between the shared memory advocates and the message-passing advocates has subsided somewhat; both camps have adopted the others' technologies: shared memory machines that can also perform message-passing (Alewife [KJA+93]), and message-passing machines that can also provide shared memory (M-Machine [Gur94, FKD+95]).

large-memory versus a memory hierarchy, it is simple and gets the job done – particularly since most modern parallel computers use relatively coarse-grained hardware where the extra cost of additional memory is a small fraction of the total node cost.

5.5.3 More Extensive Validation Needed

Although the model gives qualitatively correct results for the shape of the spatial distribution of customers over processors, there is room for reasonable concern because the MVA's predicted queue size values differ from simulations *by around a factor of two.* On the other hand, this might also be seen as being quite good given the size of the model, the number of approximations and assumptions that go into it, and the fact that the workload that examined in this thesis is heavy. Still, it would be better if the results matched more closely. Future research should include extensive validation of the model versus the simulator.

It is quite possible that the model is already pushing the MVA technique to the limit. Examination of the model in the course of this research has identified some discrepancies, as follows. One large problem is the rigid message assumption. When individual channel occupancies and residencies are examined, the rigid message assumptions lead the model to make different predictions for individual queue size residency than for residency in the two-flit router buffers in the J-Machine. Given this, it is impressive that the system level predictions are as good as they are and it is not clear how to improve the situation. Interestingly, changing the simulator so that the wormhole messages behave in the unrealizable rigid fashion[19] does not change the simulation results much.

In addition to validating the model more extensively against simulation, it would also be of value to validate it more closely against the hardware. This is even more difficult than validating against the simulator, because the network signals are inaccessible on the MDP's VLSI chip, and because the MDP does not provide any kind of hardware statistics-gathering. But, it might be possible to infer internal MDP utilization and residency parameters from probes placed on the network pins.

[19]Which I suspect is how AV validated their model.

5.5.4 Explore More of the Parameter Space

This chapter examines only a small portion of the workload parameter space. Specifically, it examines the behavior running a uniform workload with one particular short message size. Some potentially interesting areas for examination in future research include imbalanced workloads and longer messages; the latter being particularly interesting because of the difficulty that the J-Machine exhibited in Chapter 4 handling long messages.

5.5.5 Larger Machines

The workload that examined here is heavy. The choice of this workload approximates the situation where a larger machine would saturate the machine bisection even with a longer message service time. It would be interesting to see how this model behaves for larger workloads. However, this is not practical at this time, because running a single AV solver for a 512-processor J-Machine takes about a day to run. Each solver iteration takes only 5 minutes, but the rate of convergence is low because the solver uses an under-relaxation technique (combining new values with old) with a small change coefficient (0.1). The under-relaxation technique is used to keep the solver stable; without it, the approximate solutions thrash wildly and do not converge. The problem appears to be due to the blocking model and the unfair router model. One way to simulate larger machines is to investigate ways of speeding the rate of convergence so that the model requires fewer iterations. Jacob White has suggested that various numerical techniques, like using Newton's method, might speed convergence [Whi97].

Another way of looking at the model for larger machines would be to simplify the model. If the flit-by-flit tracking of utilization or residency could be removed, the model would be much faster.

Ironically, this model is for studying a parallel computer architecture, but the numerical model runs only on a conventional serial computer – and getting solutions is difficult because the model runs so slowly. Parallelizing this model to make it run faster on a parallel computer would be easy.

5.5.6 Final Challenge

One prediction of the model is that in a 16-node linear array, a sort of wave appears, and this wave is related to the length of the messages. It is not possible to run a 1024-node J-Machine at the moment (due to obsolescence in the J-Machine). However, the model predicts that there will be two centers of overflows in such a machine, one in the upper stack and one in the lower stack.

5.5.7 Summary

This chapter began with a description of the basic MVA algorithm. The Adve and Vernon wormhole model is in Appendix A; this Chapter runs this model and finds that a large change to the waiting time equation is necessary to properly model the J-Machine's unfair router. This provides an analytical model which confirms the experimental observation in Chapter 4 that queues overflow in the center of the machine. Finally, changing the model for a Snakes workload showed that this makes the queue size problem even more severe.

Chapter 6

Architecture Variations

This chapter discusses an empirical study of the performance of two simple flow control and balancing variations for the J-Machine running the Snakes workload. The study examines the ability of these variations to load balance the processor's message queues, and to thereby prevent queue overflows while still maintaining high throughput.

The study suggests that a simple protocol using a high-water threshold within the processor's message queue and a high-priority alternate network can prevent queue overflows. Processors that are accumulating dangerous numbers of messages in their queue can use the high-priority network to speed up their message service and thereby reduce the size of their queues.

Methodologically, this chapter is limited to only presenting the results of architecture simulations at two load points. The results are only suggestions for future studies. In contrast to previous chapters, there is little analytical work here.

This chapter is structured as follows: Section 6.1 describes the problem that our proposed architectures variations are designed to solve. Section 6.2 describes the architecture variations. Section 6.3 presents the measurements and performance of these variations running the random workload. Finally, Section 6.4 discusses the conclusions and suggests further studies.

6.1 The Problem

Previous chapters in this thesis described the performance loss experienced by the J-Machine when it runs the Snakes workload. The MDP's message buffers are too small to hold the accumulations of messages. This results in queue overflows and causes traps to fault handers which run slowly on the MDP. The slow fault handlers disrupt the progress of the rest of the machine and therefore drastically reduce overall system performance.

How can this be fixed? There are two seemingly obvious and simple solutions. The first is to simply make the queues larger. The second is to make the overflow handler much faster so that it does not exact such substantial performance penalties. Unfortunately, these solutions are themselves problematic. Making queues larger imposes a cost that is multiplied by the number of processors in the machine; in a parallel computer this is undesirable. Although making overflow handlers faster might help, at any particular level of overflow speedup overflows will still exact some performance cost on the processors and run the risk of exceeding the imbalance threshold and cause severe accumulations of messages. Therefore, the simple solutions may not be acceptable.

These simple solutions are passive; an active mechanism to prevent overflows would attempt to detect and restore the balance in some way. This chapter looks at two active balancing mechanisms.

Overflows are caused by both temporary and long term imbalances:

- **Temporary Imbalances**. In the random workload, the service time of messages is random, as are the destination of messages. This leads to variation in the arrival and departure rate of messages from the processor, and thus to temporary variation in the number of messages in the message queue.

- **Long Term Imbalance**. With a quantity of closed traffic, any bottleneck in the queueing system can lead to large accumulations of messages (see Section 3.4). Recall that in the notation of classic queueing theory (Chapter 3), the network is defined by routing probability transitions r_{ij} and service rates μ_i. The bottleneck processors in the system are those with the largest values of x_i, where the x_i solve the traffic equations $\mu_i x_i = \sum_{j=1}^{N} \mu_j x_j r_{ji}$. This equation suggests two reasons why a processor in the J-Machine can become a bottleneck. First, other processors can direct a disproportionate share of the available traffic at that processor: the r_{ij} are imbalanced. Second, the service rates are unequal: the μ_i are unequal. Note that the two are not exclusive. For example, it is possible to have the r_{ij} themselves imbalanced, but the μ_i

selected in a way that compensates so that the x_i are equal. If the x_i are unequal, the equilibrium solution to the queueing equations yield unequal expected average queue size; temporary imbalances vary around these unequal averages.

In other words, the temporary imbalances are captured by to the second and higher movements of the probability distribution of the queue size process, while the long term imbalances are captured by the first moments.

Another important imbalance effect in the J-Machine is backpressure from the network into the processor that strongly affects the processor's service rate. This effect could not be captured in classic queueing models (Chapter 3), which do not easily admit blocking and flow control. The mean value model discussed in Chapter 5 does, and found that the effect of different amounts of network backpressure within the processor led to imbalance and queue overflows. Network backpressure in effect alters the classic model's processor service rates μ_i.

Ideally, one would like an active balancing mechanism to deal with all of these imbalances (temporary, long-term, and blocking). One of the variations presented in this chapter appears to be able to do this.

The analytical work in the previous chapters dealt with systems in which no active balancing processes were running: that is, passive random flows. Analytically capturing the flow dynamics of passive random flows is difficult; capturing the flow dynamics of actively controlled random flows is more difficult. Even more difficult is designing provably beneficial control algorithms for network flows. I do not tackle any such analytical challenges in this chapter. Instead, I provide simulation results which suggest promising candidate algorithms.

6.1.1 Metric

The architectural variations are evaluated here in terms of their ability to achieve good throughput while avoiding queue overflows, by simply measuring the time to complete the benchmark. This metric captures the throughput. And also, because the simulator imposes the J-Machine's severe costs for any queue overflows, it also captures the occurrence of any overflows.

6.1.2 Workload

The space of possible Snakes workloads is infinite. This study restricts itself to examining two instances of the Snakes workload: one that is balanced and one that is imbalanced. In both cases, there are 5 messages per processor (so that there are $K = 2560$ messages total on the system) and each message must run for $L = 500$ steps before returning to the home processor. The messages are short: 8 flits, or 4 words long.

In the imbalanced case, the simulator adjusts the workload transition probabilities r_{ij} so that processor (1,0,0) in the machine is a bottleneck with imbalance $B = 1.35$. The imbalance B is the ratio of the rate of incident traffic on processor $(1, 0, 0)$ relative to the rate of any other processor on the machine. This level of imbalance is above the bottleneck threshold $\frac{N+K}{K} = \frac{512+2560}{2560} = 1.2$, so classic queueing queueing theory predicts that processor (1,0,0) should exhibit a severe accumulation of messages in its queue.[1] However, this imbalance is not so severe that we would expect that all flow control algorithms would fail completely. Although its traffic is above the imbalance threshold, the bottleneck processor is not completely swamped with traffic – B is at least close to 1.

Clearly, there's a much larger space of possible workloads than these two. An analytical study of the flow control variations should consider the behaviors of the system over the entire space, or against worst case instances. One cannot expect or suggest that the algorithms here perform well over the entire space.[2] Empirical examination over the whole space and analytical studies are left as future work.

6.1.3 Machine

The experiments start with a simulation model of a $N = 512$ processor J-Machine, with its 3D mesh, deterministic dimension-ordered oblivious wormhole routing, small message queue sizes, and slow overflow handling. The experiments then add some variations to the machine which make decisions on directing the flow based on local knowledge. These variations are described in the next section.

[1] See Section 3.4.2 for the definition of the imbalance B and derivation of this threshold.
[2] Your mileage may vary.

Mnemonic	Description
REF	J-Machine reference.
REQREP	Request/Reply messages return to home processor every other message.
RTS	Return to sender: if queue is full, destination processor returns the message to the sender.
VCFC	Virtual channel flow control in the network; when blocked, messages move to alternate virtual channels.
HWM	High water mark; when processor queue fills, messages are sent out at higher output priorities.

Table 6.1: These are the simulator variations that I evaluate for queue overflow behavior.

6.2 Description of Variations

This section describes the different hardware variations (summarized in Table 6.1) that are evaluated. REF and REQREP are for reference, with REQREP being an ideal but limited to a constrained message flow. Of the three hardware variants (RTS, VCFC, and HWM), only RTS and HWM are evaluated later with simulation because with examination, it appears that VCFC is not a promising candidate.

6.2.1 Reference

A basic J-Machine simulation model is used for comparison. This model is designated REF (reference). It has a good model of the J-Machine's router and overflow handler timings, but otherwise it does not simulate the individual instructions on the processor. In this simulation model, the MDP's send faults are disabled.

6.2.2 Request/Reply

The next variation is the same J-machine as REF, but with the workload modified to follow a Request/Reply pattern rather than the Snakes pattern. In Request/Reply, messages are sent out randomly to a destination, but then return to their home processor. This model is designated REQREP. Note that this variation does not change the hardware, just the traffic flow.

This study includes the measurements for this workload since the workload is significantly more stable than Snakes running on the same machine. While REF shows the queue overflows when the handler duration is short, REQREP never shows queue overflows. This is consistent with observations of this workload running on the actual J-Machine described in Section 4.10. Section 3.5.3 formulates a queueing theory model to explain why this workload runs so much better than Snakes; however, the solution to this model remains a challenge and an interesting potential area for future work.

6.2.3 Return to Sender

In the return to sender (RTS) variation, full queues at the destination divert incoming traffic back to the sending processor. Transmission of the returned messages is done through an alternate virtual network. If the destination queues are not full, they accept the message and a short acknowledgment is sent back to the sender.

RTS is a variation on the window flow control algorithm [BG87, p.117] and ensures that there is room for the diverted message at the sending processor. This is necessary to prevent a situation in which the message cannot be returned to the sender, because the sender's incoming message queue was full. To prevent this, a separate memory area on each processor is designated for potentially returned messages. Before sending, the sending processor attempts to reserve space in this return memory to hold the returned message. If the space cannot be allocated, the sending operation blocks until some space becomes free in the sending memory. This memory becomes free as other processors acknowledge their acceptance of messages. Accounting for space reservations and acknowledgments is done with a counter.

RTS resembles window flow control, but there are some interesting differences. Although windowing flow control generally limits stream flow to prevent buffer overflow at the receiver, it is generally only used for long streams, because setting up a window flow controlled connection involves some initial handshaking to negotiate window sizes and to send the initial acknowledgments.

In a packet-oriented system like the J-Machine, any initial negotiation of the connection is undesirable. Because tasks are short, small amounts of packet overhead for negotiation can be significant. A fine-grained machine must be able to inject a single packet into the network for an arbitrary destination with little overhead.[3] Processors in this packet-oriented machine should be able to send a message quickly to any other processor, interleaving messages to many destinations. One could imagine that when the system is initialized, each processor might negotiate an open stream to every other processor, allocating an initial window on every other processor. Clearly, this requires $O(N^2)$ storage on the system to manage the N^2 streams. This is practical for small machines, but not for larger ones.

Another way to allow processors to send messages with little negotiation overhead would be to allow processors to *speculatively* inject their messages. The destination would acknowledge whether it accepted the message by sending a short packet back to the destination which says ACCEPTED or REJECTED. If the message were rejected by the destination, the sender would resend the message at some later time. The ACCEPTED/REJECTED acknowledgments simply tell the sending processor whether it needs to resend.

The ACCEPTED/REJECTED acknowledgments are not telling the sender that a message was lost. A characteristic that can distinguish networks in parallel computers from network in other systems (e.g., inter-networking) is the assumption that the parallel computers' wires are sufficiently reliable so as to make it unlikely that the network drops messages.[4] Also, it is possible to design the parallel computer network in a parallel computer so that the routers' response to a full queue is to block incoming messages, rather than to drop messages. This is one of the main features of the wormhole network. The wormhole network does not lose messages.

RTS is nearly identical to this ACCEPTED/REJECTED protocol. The message is sent speculatively, and an ACCEPTED message is returned if the receiver accepts the message. There is one main difference between RTS and ACCEPTED/REJECTED: in RTS, when the receiver cannot accept a message, it returns the whole message to the sender, rather than a REJECTED message.

[3] Were the consumption assumption always true, this would be easy, because the message would always be accepted by the destination. But as earlier chapter demonstrated, the consumption assumption is not always true.

[4] See [DDH+94] for a way to increase the reliability of wires.

This difference in RTS exploits the reliable message delivery in the parallel computer to reduce the amount of memory bandwidth required at the sender. Why is this? In the ACCEPTED/REJECTED protocol, the sender must create a local copy of the message. This is so that if a REJECTED message is returned, the sender has a copy of the message for the next attempt at resending. Making this copy requires memory bandwidth. Since the message will, when it is received, also be placed in the receiver's memory, in the normal case without rejections, each message will get written into memory twice: first, to make the backup copy for a possible resend, and second when the message is accepted. Most of the time this first backup copy is not used.

RTS does not require writing the backup copy. In RTS, the message is kept in the network routers. The sender reacquires the payload of the message for resend when and only if the message is returned. The message routers implicitly store the backup copy of the message. Because this backup is not written to memory by the sender, the overall memory bandwidth required by the network for the normal case where messages are all being accepted is one half the memory bandwidth required by the ACCEPTED/REJECTED protocol.

The cost of the RTS protocol is a possibly higher amount of memory bandwidth consumed when messages are rejected. In RTS, on return, the whole message is returned. With the wormhole network, the messages can be of arbitrary length, so the returned message may be long and consume more network capacity. In contrast, the ACCEPTED/REJECTED protocol needs to send only a short, fixed-size packet to reject a message. The overhead of returning a message in RTS can be substantially higher than in ACCEPTED/REJECTED.

In RTS, although the memory for the rejected message does not need to be written, it does need to be *allocated*. This can be easily implemented with a counter that tracks the quantity of memory for returned messages. The counter is initialized with the quantity of available memory. When messages are sent, the counter is decremented. When messages are ACCEPTED, the counter is incremented. If the counter reaches zero, the sender blocks.[5]

Although RTS allows processors to divert incoming traffic when their queues fill, RTS does not solve the problems of deadlock. For example: a particular processor has no free message-return space – all of the message-return space is reserved for messages that are outstanding.

[5]RTS's counter is actually easier to implement then the memory management needed by ACCEPTED/REJECTED.

This processor cannot be allowed to send any more messages, because there might not be any more space available. Because it cannot send, the processor cannot dispose of its current handler, and cannot serve any other messages in its input queue.[6] This input queue might fill or be filled, so that the receiver rejects all incoming messages, which blocks other processors from releasing their storage. One or more processors in this situation could be cyclically dependent and the system could therefore become deadlocked.

To prevent the deadlock, RTS can be enhanced by allowing preemption of some sort; perhaps with a deadman timeout. Although the simulator did not implement this deadlock, the experimental workloads did not experience any. However, this deadlock issue would need to be considered in any hardware implementation.

RTS is a small modification to the J-Machine architecture. It is currently used in a real commercial parallel computer, the Cray T3D, which has a deterministic wormhole-routed network based on the one in the J-Machine.

It appears that the RTS algorithm might be able to help with queue overflows. It allows for the receiver to divert incoming traffic back to the senders, so receivers never get overflow traps. It is for this reason that RTS is in the following experiments. For the experiments, RTS was added to the J-Machine simulator. To make a comparison that's more fair, it is necessary to take into account the fact that the additional virtual channel used to return messages to the processor requires its own input message queue. Rather than increase the amount of queue memory for this extra virtual channel, the experiments accommodate it by dividing the size of the queue used on the reference processor into two halves, with each half serving a different virtual channel. Note that this does not increase the number of queue overflow traps, because the RTS does not take queue overflow traps! It does make it more likely that queues will fill, but this only results in an earlier onset of the state where messages are being returned to senders.

[6]This is assuming strict, FIFO, non-preemptive service of this queue, the way that it is in the J-Machine. Actually, there's evidence that the processor ought to allow preemption in this situation. See [WHJ+95].

6.2.4 Virtual Channel Flow Control

Dally's [Dal90d] virtual channel flow control (VCFC) adds a degree of adaptivity within the network, rather than within the processors. With this adaptivity, there are multiple virtual channels within the network. If the head of a wormhole message encounters blocking, it can move to a higher-numbered (though not necessarily higher-priority) alternate virtual channel. Because buffering and switches of the alternate virtual channel are independent of each other, messages can make progress through blocking that is created by switch contention. The alternate virtual channels still must contend for physical wires, so the progress of network traffic is still constrained by wire bandwidth. The extra virtual channels only help prevent reductions in effective throughput that are due to switch and buffer contention.

In simulation, this does nothing to help the queue overflow problem. There's no particular reason it should. The adaptive aspect of VCFC – switching to a higher virtual channel – is at best only inadvertently coupled to the size of queues on the processors. Therefore, VCFC can do little to detect or to prevent processor queue overflows. Is there a way to design a feature which is similar to virtual channel flow control, but actually makes routing decisions based on the objective of reducing processor queue overflows? The answer is yes, and leads directly to the next hardware variation that is presented below.

No measurements for this option VCFC will be presented in this thesis, because it does not help at all in preventing processor queue overflows.

6.2.5 High Water Mark, Alternate Output Priorities

With VCFC, the head of any message in the network can move to an alternate priority as soon as it is blocked. But as explained above in Section 6.2.4, this adaptivity occurs only within the network itself. Although moving to a different virtual channel is beneficial[7] it does not prevent or reduce the impact of processor queue overflows.

In the next variation, the hardware has two independent alternate virtual networks. In this case, the two alternate virtual networks are prioritized: if a Priority 1 message and a Priority

[7]Dally's experiments demonstrated that VCFC conveys a benefit in terms of higher effective maximum network throughput [Dal90d].

0 message are contending for the same physical wire, the Priority 1 message is given the wire. This means that traffic applied to Priority 1 will consume network bandwidth completely at the expense of Priority 0 when there is any contention between the two for physical wires.

How does the processor choose the priority of a message? It selects the priority that helps to prevent queue overflows and only uses local information. The obvious piece of local information is the size of the local processor queue, and therefore the obvious way to choose the outgoing priority is to base it on the size of the local queue relative to some high water mark. When the local queue is full, sending on the higher priority channel reduces the amount of backpressure from the network, so that the processor can serve messages from its input queue faster than the message arrival rate. This reduces the size of that queue and helps to prevent overflows.

This variation is designated "high water mark" or HWM. The high water mark is a fixed threshold in the processor's normal message input queue. If the size of this queue is larger than the high water mark, hardware starts directing the processor to send its messages out at a higher priority. The experiments below present the simulated performance of the machine with three different high water marks and show that some work better than others.

Once in the network, messages do not change priorities. A separate processor message queue serves each different network priority. To be fair, the queue size used by REF is divided into two halves with one half assigned to each priority in HWM

6.2.6 Discussion

This section has presented the 5 machine variations that were selected for for this investigation. The next section, examines the performance of these variations at the two workload points.

6.3 Measurements

This section presents measurements from simulating the different hardware options. All simulations are performed with an $N = 512$ processor (8 by 8 by 8) mesh-topology machine.

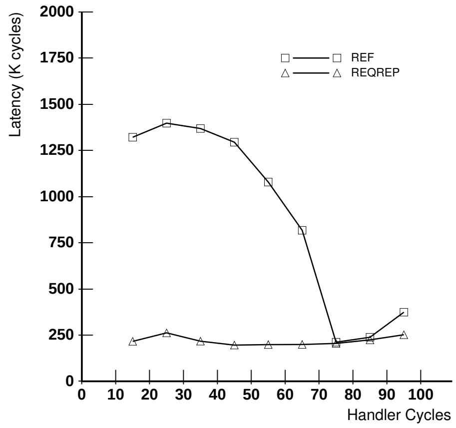


Figure 6-1: Number of cycles to complete a Snakes pattern versus a Request/Reply pattern, for a uniform workload running on 512-processor J-Machine simulation.

To give the machine enough time to reach steady state, the simulations are run with the snake length $L = 500$. The workloads start with $m = 5$ messages per processor, which is a workload point where the hardware proves to be sensitive to bottleneck dynamics. Finally, the simulations are run with the minimum J-Machine message size of 8 flits. The imbalanced workload runs with processor $(1, 0, 0)$ having a higher load with $B = 1.35$. This is higher than the imbalance threshold $B = 1.2$ expected for this value of $K = mN$ and N.

6.3.1 Measurements for REF and REQREP

Figure 6-1 presents the result of simulating REF and REQREP. On the Y-axis is the time to complete the benchmark. A shorter time is more desirable, because it means the machine is operating faster. On the X-axis is the handler time, which is the number of cycles that each message requires when it arrives at this processor. Increasing the handler time makes the workload relatively more demanding of the processors than of the network.

The behavior described by the REF curve in this figure is qualitatively consistent with the behavior observed for the J-machine in Chapter 4. In both cases, with long handlers, the J-Machine runs well. However, when the handler time dips below 70 cycles, the J-machine starts exhibiting queue overflows. Previously, Chapters 3-5 showed analytically that this is due to network-induced imbalances that transform the center processors into bottlenecks, and that these center processors are the ones that experience queue overflows. The trap handlers for the queue overflows substantially increase the runtime. For a similar graph showing this behavior on the J-Machine hardware, see Section 4.2.

Similarly, the REQREP curve shows behavior that is qualitatively consistent with observed behavior on the J-machine. REQREP presents on average exactly the same overall network traffic load on the machine. However, because this traffic makes regular returns to home processors, it stays stable – even at workload points for which the unconstrained network is impaired by queue overflows. REQREP is stable against these network-induced hot spots.

Figure 6-2 shows the behavior of this machine when the workload includes a hot-spot, with imbalance $B = 1.35$. The imbalance is designed so that REF will perform poorly. It does, and at all handler times: the REF curve in Figure 6-2 shows poor performance with any handler time. In contrast, REQREP is stable against this hot spot with any handler duration.[8]

This is the critical challenge in the design of a better J-Machine. REF is extremely sensitive to imbalance, whether network-induced or processor-induced. REQREP is not sensitive to imbalance, but it is following a less general flow pattern.

6.3.2 Measurements for RTS

One might expect RTS to work well. It does not have tremendous overflow penalties, and can nimbly divert traffic back to the sender. However, it does not improve the performance of the machine. See Figure 6-3. This graph shows the performance of RTS with the uniform and hot spot workloads. The performance plots of REF and REQREP (uniform workloads) are included for comparison.

When the handler times are longer than 70 cycles, and with the uniform workload, RTS performs no worse than REF. Because the workload is processor-limited, the cost of the acknowledgment messages is barely perceptible in the total run time. The RTS workload seems to take just slightly more time than REF.

[8]With REQREP, the a message is choosing a new destination only half of the time. The other half of the time, it is returning home. So the imbalance $B = 1.35$ refers only to the imbalance in the outgoing selection.

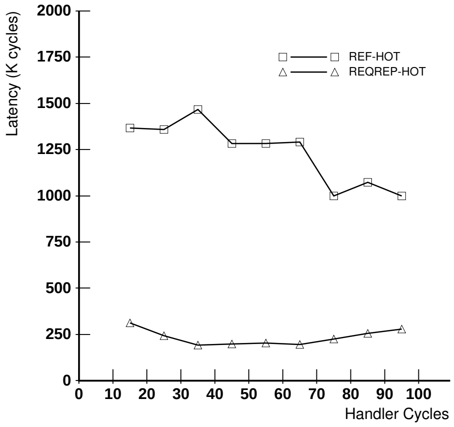


Figure 6-2: Number of cycles to complete a Snakes pattern versus a Request/Reply pattern, for a uniform workload running on 512-processor J-Machine simulation.

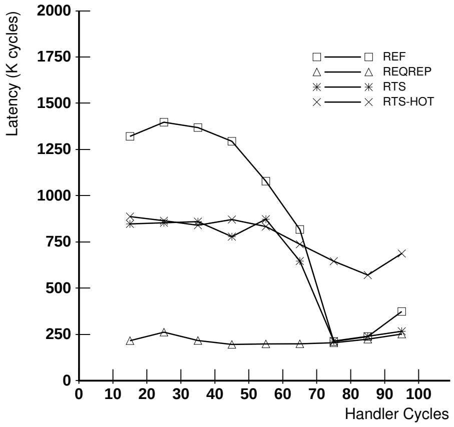


Figure 6-3: Number of cycles to complete a Snakes pattern vs. handler duration when the machine using the RTS flow control method for balanced and hot spot workload cases, compared to the reference machine and REQREP. When the handler duration is long, the RTS algorithm works as well as the reference machine. Unfortunately, like the reference machine, RTS is unable to achieve good throughput with short handler times.

When the workload becomes network-limited and where REF started showed a drastic overflow-induced increase in runtime, RTS also shows an increase in runtime, which is due to an increase in the number of returned messages. RTS is not improving machine performance.

With the network-limited workload, there is a 2x improvement for RTS over REF. This is not a meaningful improvement, for two reasons. First, RTS still is more than 3x slower than the ideal performance that REQREP exhibits. Second, the relative cost between REF and RTS in this region is determined mostly by the speed of the overflow handlers on the J-Machine. The J-Machine handlers are slow. Were they to be improved – perhaps by using a wider path to main memory – there would not be such a large difference between REF and RTS. In other words, RTS is not improving the situation in any systematic way; it's just changing the way in which the system incurs the penalties of overflows.

Also, RTS does not improve performance with the hot spots. In Figure 6-3 the behavior of this workload with the hot spot (RTS-HOT) shows no improvement, even with long handlers.

Why isn't the RTS algorithm taking care of queue overflows? Wouldn't an overflow be replaced by RTS by a quick message diversion and prevent the catastrophic costs experienced by the current J-Machine (and its simulation model, REF?)

Section 6.1 discussed the difference between short term and long term imbalance; the short term imbalance causes queues to get larger, perhaps because of a larger number of arrivals. In the longer term imbalances, the queues are large because of differences in load among the processors.

The problem is that the imbalances in the workloads that are tested here are long term imbalances.[9] The network-induced load imbalances are long term. The hot spot workload is long term imbalance. With a long term imbalance, the equilibrium state of the machine is one where some queues are full. With RTS, the queues can fill and the machine can proceed. But it does not *resolve* the problem. The queues can fill, and the algorithm can "kick in," but the workload still specifies that the same number of messages through the bottleneck processor, with the same large accumulation of customers on the bottleneck.

RTS diverts the accumulation away from the bottleneck, but the accumulation must reside somewhere, distributed over the following three places:

[9] Recall that Section 6.1 defined the difference between long-term and short-term imbalance.

- en-route back to the sender for a return.
- in the sender's return queue.
- en-route to the destination for a retry.

In the implementation of RTS on the simulator, the sender immediately retries sending the message. This forces as much of the accumulation as possible off of the sender's return queue, and back into the network. The extra traffic in the network serves little purpose. It does not help the hot spot processor serve messages faster. The extra traffic bouncing back and forth between the sender and the receiver just congests network channels. Meanwhile, the other processors on the system are still making progress on their workload.

Ideally, one would want that throttling to reduce the progress on the rest of the system, until the queueing dynamics should start showing some accumulations off of the bottleneck processor. However, there's nothing in the RTS algorithm to achieve this by design. The other processors in the system do not intentionally experience slowdowns due to retries.

In fact, while some of the congestion from retries does get inadvertently imposed as extra latency on other processors, some of that cost may also be inadvertently imposed on the hot spot processor itself. In fact, this is likely, because the output links used by the hot spot processor to return messages to the sender are the same links that the hot processor needs in order to make progress. And this is a problem, because slowdowns on the hot-spot processor will just increase the amount of imbalance. Fixing this is not simply a matter of adding an extra channel from the processor into the network. Even with an extra channel that reduces the congestion in the adjacent outgoing link, the returned retries also consume nearby network links. The extra load on the nearby links is still close to the hot spot processor, and can still inadvertently slow it down.

RTS: Summary

RTS, as implemented here, does not help the overflow problem. This is because it does not actively work to balance traffic. Rather, it diverts the expected accumulation on the bottleneck processors into the queues of nearby router channels. This actually could make the system worse off, because congestion in those channels from retries can actually increase the severity of the hot spot.

The simulator implements an immediate retry. It may be that some sort of adjustment to the rate of retires could make RTS work. However, any sort of throttling of this sort would require an analytical justification; this analytical justification is deferred as possible future work. One reason why this is deferred is because HWM in the next section works so well.

6.3.3 Virtual Channel Flow Control

Section 6.2.4 stated that the virtual channel flow control does not help the queue overflow problem. In simulations, this was the case. In fact, the splitting of the REF buffer size into two halves for VCFC proved to make VCFC even more vulnerable to queue overflows. The performance of this option proved uniformly poor, never even approaching the behavior of REF at any workload point.

6.3.4 Measurements for HWM

Figure 6-4 shows the performance of HWM with three different high water marks. With them are the measurements for REF and REQREP. This graph shows some interesting behaviors. In particular, examine the curve for HWM64. This is for HWM that switches the output to Priority 1 when the processor's input queue is greater than 64 flits. Note that the behavior of this variant shows the machine approaching the performance of REQREP, but with no overflows. This is excellent.

Unfortunately, for longer handler times, including those where REF seems able to work adequately, the HWM64 variant shows a drop in performance. Queue overflows are occurring; caused, probably, by the choice for fair comparison of queues 50% the size of those in REF.

Why is HWM64 able to balance the system when the workload is more network-limited (even with 1/2 size queues) but not when the workload is processor-limited? Simply: "leverage." When the system is network-limited, the variant works as designed. Processors that are potential overflow candidates (due to short or long term imbalance) with their queue size growing can switch the priority of their output messages and gain a substantial advantage

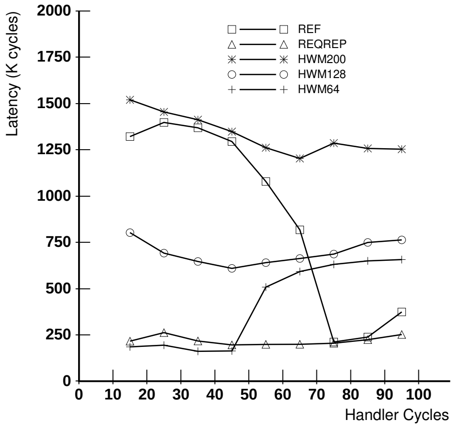


Figure 6-4: Number of cycles to complete the Snakes pattern with HWM variant of the J-Machine, with three different high water mark threshold sizes, compared to REQREP and REF.

over processors trying to send at Priority 0. Their advantage stems from backpressure at Priority 0; the high priority messages on the Priority 1 channels do not experience that backpressure. Processors that switch to Priority 1 can catch up to the other processors.

When the workload is not network-limited but when it is processor-limited, the processors sending at Priority 0 are not being blocked by the network. In this case, sending at Priority 1 offers no advantage: sending at Priority 1 is no faster than sending at Priority 0. So processors which have larger queues cannot catch up by making better relative progress at their output.

HWM128 and HWM200 do not perform as well as HWM64. This suggests that very aggressive response to queue accumulation works best at balancing the workload.

Figure 6-5 shows the performance of HWM running with hot spots. Note that HWM64 is able to balance this workload-induced imbalance – as long as the workload is network-limited. In this case, the leverage over the network imbalance is sufficient to overcome the workload imbalance. However, as the network becomes less heavily loaded, the RTS variant is less able to balance the workload.

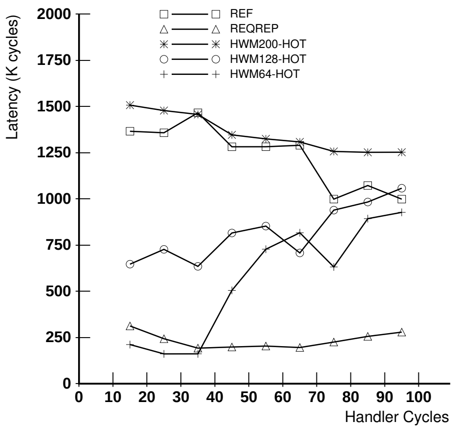


Figure 6-5: Number of cycles to complete the Snakes pattern with HWM variant of the J-Machine, with three different high water mark threshold sizes, compared to REQREP and REF, with hot spots.

6.4 Discussion

6.4.1 What's Wrong with HWM?

HWM showed some promise: at heavy load, HWM64 was able to balance the processor queues, even with smaller queue sizes than REF. If one were to build an improved J-Machine, would one want to include HWM? Yes. It is simple to implement, easy to understand, and remarkably effective. It exacts little cost on the architecture. However, there remain some weaknesses to this approach which are worth mentioning.

One of the weaknesses of HWM is that it does nothing to balance the processor unless the machine is heavily network-limited (this is the workload domain of study in the thesis). But there are other workload points of concern, none of which HWM is able to address. For example, with longer handlers so that the machine is processor-limited, HWM cannot correct any imbalance. There is no advantage to be gained in this by a processor moving to the higher priority output channel. It does not make the vulnerable processor any faster than the other processors. Processors which are accumulating traffic have no leverage against the network.

This argument suggests a simple variation which would work, but which is perhaps undesirable. That is, an improvisation on HWM which instead of speeding processors that are too slow, slows processors which are too fast. In this case, slowing the processor acts to capture messages on the nodes, and acts to balance the machine. There's no question of leverage here: it's always possible to slow a processor. At first, this seems to be a good solution, but the question arises: how slow and for how long? Were the processor to slow too abruptly, or stay slowed for too long, it could end up capturing too much traffic and experience overflows anyway. There's a control problem here.

HWM also faces this control problem. What should the high water threshold be? How long should the processor be able to send at Priority 1? Any implementation of HWM should be accompanied by an attempt to tackle this challenging analytical problem. This is deferred as future work.

6.4.2 Is RTS that bad?

Similarly, the measured failures of RTS do not prove that it's a hopeless algorithm. At the load points measured in this study, it fails. But, might it not work at other load points? Or, might some fix to the implementation of it make it work?

Clearly, some better understanding of RTS would help, including an exploration of RTS at other workload points and an analytical study. One key parameter of the algorithm for further study is the policy for processors with returned messages. Some sort of back off may be able to achieve a good balance between throughput and avoiding queue overflows. These questions are deferred for further research.

6.4.3 What Other Variations Are There?

This study examined two flow control variants. There are many more variants which might have better overflow behavior.

Using a torus network with wraparounds reduces the amount of mesh-induced inhomogeneity and can reduce the number of overflows. But this does not help with the problem of workloads that are by themselves imbalanced.

The Chaos Router [KS91] uses random misroutes to prevent deadlock and to adaptively route around traffic. It's possible that this algorithm could work well. This requires integrating the processor queues into the general chaos-routing protocol; in the original papers, the processor queues are attached to the network using specialized connections. The analysis of closed traffic within this modified Chaos network would be interesting.

In general analytical studies of heavy load closed traffic with realistic blocking and finite memories is difficult, and gets worse when active balancing mechanisms are added. The rate of balancing events within the system can be a function of the variance of queue size processes. Because variance is difficult to predict, the analysis is trickier.

6.5 Conclusion

This chapter has presented the result of a simulation study. It finds that a high water mark based balancing protocol is, at the two workload points measured, able to maintain load balance. This suggests ways for designing active control mechanisms to balance flow in parallel computers.

Chapter 7

Conclusions

7.1 Summary

A rule that's rewritten every time a new computer is constructed is: if a feature isn't simulated, it doesn't work.[1] Is that what happened with the J-Machine?

When the J-Machine was built, the wormhole network had been extensively simulated and validated at the gate level and at the behavioral level. The wormhole network was also scrutinized analytically to assess its performance – but these simulations and analyses were performed with the wormhole network in isolation; that is, simulated with traffic arriving from "somewhere else" and then being immediately consumed (somewhere else) after delivery at the destination. In this *open* context, the wormhole network hardware that we built works correctly and well.

The complete Message-Driven Processor was also extensively simulated – at the gate level and at the behavioral level, but these simulations were too detailed to model a large J-Machine. The instruction level simulator was too slow to simulate a 512 processor machine - and even it had been able, it had no accurate model of the wormhole network anyway. However, the MDP chip that we got back from the fab can put messages into the network and they are correctly delivered to the destination. Programs run correctly, and often they run well.

The J-Machine works.

It is when the J-Machine runs heavy traffic that things get interesting. For intense fine-

[1] Of course, it could be simulated and still be broken...

grained traffic running on a large J-Machine with an accurate network model, no simulations were performed before the machine was constructed. One of the things that this thesis shows is that these intense traffic flows, even if they are perfectly uniform, perform poorly on the J-Machine because of queue overflow traps. Overflow traps ensure that the machine runs correctly; but, the traps are so slow that if they occur at all, they destroy performance.

The irony of this performance drop is two-fold. First, it occurs precisely in the workload region where the machine is supposed to perform well: with a massive "cloud" of fine-grained tasks traversing a large machine in a message passing style. Second, it occurs largely because the fine-grained hardware philosophy leads to smaller message buffers at the processors, and it is these message buffers that overflow and cause performance loss.

Is this a bug in the MDP? No. This is not a simple bug in the sense that it renders the machine unusable. The machine runs a light workload perfectly well, and even runs some heavy workloads well if they are structured in just the right way, for example, as a request/reply pattern.

But it is a performance problem. Another contribution of this thesis is distinguishing the workload parameter region where the machine performs well from the region where the machine performs poorly. The machine performs poorly when the network is saturated, when the messages are long, and when there are many messages.

The classic queueing model in Chapter 3 offers some explanation why this is so. This model shows the relationship between the queue size and the expected rate of queue overflows. It offers a general rule of thumb: the rate of queue overflows grows sharply when the total traffic quantity exceeds 5% of the storage available in the queues - for uniform traffic. When the traffic pattern is nonuniform, or imbalanced, the expected queue size grows sharply as the degree of imbalance approaches the "imbalance threshold" defined by the number of processors and the number of messages. As the number of messages or processors increases, the ability of the machine to tolerate imbalance diminishes.

There's a satisfying beauty to the classic queueing model, but it does ignore substantial features of the J-Machine, like the wormhole network. As a consequence, by itself, it can't explain why the imbalance becomes severe at network saturation, or why the overflows occur only in the center of the machine. The first of these discrepancies is easy to explain with a little more information: at saturation, the inhomogeneity of the J-Machine's mesh topology creates a traffic imbalance that exceeds the imbalance threshold. This leads to strong bottleneck message accumulations on disadvantaged processors; the queues overflow, and the machine performance plummets. Explaining why the queues overflow only in the center of the machine requires the more substantial analysis that considers the fairness of the network routers.

The maximum message queue size of the MDP is too small. The message queue can hold only 128 of the shortest messages. A workload with only 5 short messages per processor is running too close to the 5% threshold where the severely expensive overflows become likely. Making the queues a few times larger will not solve the problem, though: if the network gets saturated, the machine will be operating in the imbalanced region, and the required queue size will grow proportionally to the total amount of traffic on the whole machine – and not just proportionally to the amount of traffic on that one node.

The flow of traffic needs to be controlled, and the load on the machine needs to be balanced. These two needs are closely related. Flow control and load balancing are classically difficult algorithmic challenges. Because the philosophy of the J-Machine puts so much emphasis on making the dispatch and delivery of messages have low overhead, there was no effort to implement a global flow control algorithm in the MDP's hardware. There's an assumption that it will be performed at a higher software level.

There are obstacles to doing this at a higher software level on the J-Machine.

1. If the computation really is fine-grained, then adding instructions to the short tasks and methods for low-overhead load balancing slows them down.

2. Second, the network interfaces on the MDP do not have useful features to help them implement a load balancing algorithm. The feedback from the network on a SEND instruction is not useful; the modal interface prevents making any change in the decision to send based on this feedback. Even if a method facing backpressure were to copy a message to memory rather than send it, the message-driven scheduling of the MDP raises the question of exactly how and when the MDP would get around to dispatching the stalled message.

3. The problem of load balancing and flow control is hard.

The MDP's operating system COSMOS and programming language CST include only a single load balancing algorithm: when there is a choice on where to initiate a method or create an object, they do so at random. In this thesis, the Snakes workload models this load balancing algorithm. The underlying principle of why this should work is simple: it will disperse the traffic. But as the thesis shows, the queues have to be large enough and the traffic has to be very well balanced for this to work on a large parallel computer.

Clearly, the load balancing can be performed at an even higher level, within the application, using code automatically generated by the compiler or perhaps requiring the programmer to implement load balancing. Again, the lack of good interfaces make it difficult for the programmer - for example, there's no programming interface within the CST language to get low-overhead feedback on the state of the network. Mostly, this is because there's no way for this information to be obtained from the hardware.

There are other ways that the application can keep the load balanced. CST programmers demanded a feature that allowed them to select the home node when they created an object. This breaks the CST abstraction which tried to hide all J-Machine details, but it gives the programmers some ability to control the load that gets imposed on the network – not necessarily in response to run-time conditions; like the dispersion method. This is still passive load balancing. Another passive method that seems to work well is using a Request/Reply type of communication pattern over the objects. This thesis shows that this communication pattern has much lower variation in queue size than does the general snakes workload.

It does not seem to be the case that Snakes is artificial with much more poor overflow behavior than a "real" application. Although some applications for the J-Machine were written, none of the large ones were really massive and fine-grained. Snakes models a real fine-grained algorithm with some generality to traffic flow. The Request/Reply pattern is more stable. However, measurements (presented in the thesis) show that the small degree of structure in a Request/Request/Request/Reply (representing a message-passing workload with traffic migration but still with some structure) is insufficient to impart stability.

Given this, unless an application program is using the network lightly, or the application is very strict in the communication traffic pattern that it imposes on the network, there's going to be a danger of the application causing or experiencing overflow problems on this machine.

Some active feedback and active load balancing is needed. And this runs into the fact that the general problem here is one of performing on-line resource- and precedence-constrained job scheduling. And this is a very hard problem [GJ91]. The problems inherent in this combined system of network and processors are the fundamental problem of parallel computer architecture.

Because the problem is so hard, it's tempting to have the programmer shoulder the responsibility for this load balancing. If the machine were modified to provide better interfaces that provide good feedback, this could work well. Of course, with the processors being fine-grained, and the application being fine-grained, the problem of scheduling could have very high overhead. Couldn't something be done in hardware to help?

The thesis examines two hardware solutions to help with load balancing/flow control, and finds that there are some simple solutions which can help, and some solutions that don't appear to help. In particular, the flow control feature included in the Cray T3D, returning traffic to the sender on an overflow does not help. A different mechanism, using a high-water-alternate-output algorithm gives a chance for recovery to processors that are struggling with overflows and does help. The advantage of one mechanism over the others is due to the particular choice of workload: with Snakes, the lack of correlation between successive messages makes return-message feedback less useful, and the network occupancy to deliver those return messages interferes with the ability of the struggling processor to recover. With some other workload that is stream-oriented, the return-to-sender method might help. It may be that some particular choice of flow control is universally better than others; or, some suite of more general interfaces and building blocks to allow applications to construct their own flow control is necessary. This determination is left as future work.

7.2 Contributions

The specific contributions of this thesis are previewed in the introduction.

In queueing theory, I give methods for computing the expected rate of overflows for closed message-passing traffic running on a large queueing network. This analysis is done for both balanced and imbalanced workloads. For balanced workloads, working rules herein and methods herein can help choose hardware queue size. For imbalanced workloads, the analysis derives a threshold of imbalance beyond which queue size on bottleneck processors grow very large.

Other researchers (e.g. [NWD93, S+93]) have published papers reporting the good performance of the J-Machine; I present some of the problems, which are more interesting from a research perspective. With heavy fine-grained workloads, the J-Machine performs poorly, because processor message queues overflow and reduce the hardware to servicing overflow traps. These overflows occur in the center processors of the machine. Much of the rest of this research is dedicated to understanding the observed behavior here. The queueing analysis explains some of the sensitivity to imbalance, and the poor performance with long messages.

More detailed understanding requires an analytical model which incorporates more features of the J-Machine. The mean value analysis is an extremely large model that models the behavior of the wormhole network. It is necessary to modify this model to incorporate the unfair routers; this unfairness is responsible for the queue overflows that occur in the center of the machine.

Finally, the thesis performs a study to determine if hardware mechanisms can serve as adequate load balancing and flow control to avoid queue overflows. It finds that a mechanism using high-water marks in the queues and an alternate-priority output network can serve to relieve some of the burden of overflowed processors.

7.3 Future Work

The thesis finds that Request/Reply communication patterns are substantially more stable than the snakes message-passing pattern. The analysis of this difference using queueing theory in Chapter 3 is incomplete. It may be that using McKenna and Mitra's [MM84] integral representation may shed some light on this difference. This representation may also be useful in determining expressions for higher moments of the overflow process and ultimately a better cost model for overflows using this passive dispersive flow control.

It may also be nice to examine in more detail and report the thermodynamic analogies with the queueing studies.

The flow control study in Chapter 6 should be substantially expanded, examining different heuristics and more workload points. Overall, this thesis does not include any work with applications - partly because of the lack of a good library of fine-grained message-passing applications. With these types of applications now becoming available, the issue of application flow control could be studied in the common benchmark-oriented empirical fashion.

The network architecture in this thesis is the one used in the J-Machine, but there are other network architectures which are of interest. It would be interesting to study the message-driven dynamics of a system using a chaos router [KS91], perhaps extending their architecture to permit misrouting not only within the network, but also when processor queues overflow. In other words, the chaos router in the literature performs some flow control and load balancing with regard to the network links; this study would include load imbalance over processors.

Ultimately, another fine-grained message-driven computer that runs message-driven applications well should be built. With the better understanding of the dynamics of message-driven computation that stems from this research, and with the increasingly prevalent integration of DRAM and logic, now is the time to design the next smart-memory message driven processor.

Appendix A

Adve and Vernon Wormhole Model

This appendix reviews the Adve and Vernon model (AV) for closed deterministic wormhole networks [AV91, AV94]. The appendix modifies this model for the J-Machine, with the goal being to show how the equilibrium mean queue size of the processors changes under different load conditions. The results of this model for the J-Machine are presented in Section 5.2.

The original AV model assumes a fair router; the J-Machine routers are unfair. This causes a substantial difference in the predictions from those observed on the machine and in simulations. Section 5.3 presents changes to the model for unfair routers.

Another substantial difference is that the AV model is for a Request/Reply workload. Section 5.4.1 makes modifications to the AV model – largely pruning AV's support for multiple classes used to model Request/Reply – to model the Snakes workload.

The AV model is very long. This appendix uses the notation and tabulation, occasionally verbatim, from Adve and Vernon's exposition of the model [AV94]. It is because this model is so long, and because of the importance of making clear my debt to their work, that this section is segregated in an appendix.

Still, there are original contributions here. These include: corrections to the original model and improvements to handle heavy workloads. The model was reimplemented to run with the specific tiered router topology of the J-Machine. The implementation replaced the AV model of remote memory with a model that has processors handling both requests and replies. These changes are described in the following text and in footnotes. This appendix also explains some of the assumptions behind the model and their relationship to the actual J-Machine hardware.

Although it is segregated into an appendix, understanding this model is a prerequisite to understanding the modifications to model the J-Machine's unfair routers and to model the Snakes workload in Chapter 5.

This appendix is structured as follows. Section A.1 gives an overview of the model. Section A.2 defines an AV channel. Section A.3 describes the AV Request/Reply workload. Section A.4 describes the system model which incorporates the network and processors. Section A.5 gives the AV network model: waiting time, contention, and blocking of wormhole messages. Section A.6 gives the processor model. Finally, Sections A.7 through A.9 review independence assumptions, modeling assumptions, and then concludes the Appendix.

A.1 Model Overview

The AV model is a closed multiclass queueing network incorporating a model of deterministically routed wormhole routing and blocking. The model is *big*. There are about 30 equations which constrain the 300 floating point metrics *at each channel*. A 3D 512-processor J-Machine has 9 channels per processor, so the network by itself is modeled by more than 15,000 implicit equations over 1.5 million variables. The model of the processor and node queue are even larger, tracking blocking probabilities individually over type, class, and flit, resulting in another 8 million parameters. The complete MVA model of the J-Machine requires about 30MB of memory and takes 20 hours to converge running on a 166 MHz Pentium processor.[1]

The model incorporates a blocking model similar to the one outlined in Section 5.1.5, but instead of basing the waiting time of a flit on the immediately downstream flit, it bases the blocking time on the residence time of the head flit of the wormhole message. The waiting time of the head flit can be determined because the messages are modeled as "rigid" entities; so the head flit of a message is always on a channel that is a fixed distance from any body flit of the message. This rigid message model is interesting, and is discussed in Section A.5.1.

The AV workload that is modeled is a "Request/Reply" workload. In a Request/Reply workload, each processor has a fixed maximum number of outstanding messages that are

[1] So, don't try this at home!

sent in succession to random remote destinations in the machine. After receiving service at the remote node, each message is sent back to its home processor.

Using their model, Adve and Vernon explored various architectural tradeoffs, such as the number of output channels from processors into the network and the effects of varying the amount of traffic locality and imbalance. Their primary metric was the processor efficiency and they examined torus, rather than mesh networks. One surprising result was that non-isomorphism introduced by using virtual channels to break deadlock in torus networks (an architectural solution invented by Dally and Seitz [DS87]) resulted in nonuniform distribution of processor efficiency over the processors in the machine, even for a balanced workload.

The goal of this work here is less to predict processor efficiency than it is to predict the expected processor queue size. Expected queue size is a parameter of the AV model but measurements of it were not reported in [AV94]. Furthermore, AV includes some architectural assumptions that differ from the J-Machine's. So, the research for this thesis required re-implementing their model, making the following changes:

- An implementation choice to abstract the network topology and routing from the model. It is not apparent from [AV94], but the AV solver code had hard-coded the topology into distinct solvers [Adv95]. In the version for this thesis, with the topology abstracted, it was easy to interface to all of the routing algorithms implemented in the J-Machine simulator and the J-Machine router's fall through channels, for example.

- The original AV model was structured more for shared-memory types of systems, where messages are used exclusively to fetch from and store data to remote memories; this meant that the system had remote memory servers that were distinct from the processors. The J-Machine is a message-passing machine – messages are handled only by processors. The implementation here has customized the AV model to the J-Machine, which required using a subroutine AMVA technique for the processors, where previously the model used a subroutine MVA technique.

- Modifications to the model for the J-machine's unfair routers. These modifications are described in Section 5.3.

- A version of the model that prunes multi-class support to model the Snakes workload. This change is described in Section 5.4.1.

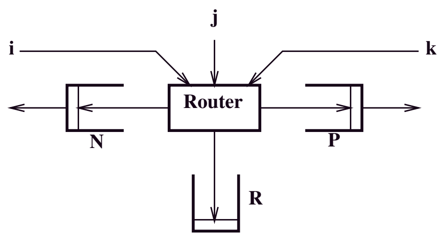


Figure A-1: One dimension of the the J-Machine router. The P channel goes to the processor one higher in that dimension, the N channel one lower, and the R channel falls into the next dimension. Channel P has possible inputs i and j (k is not permitted because messages cannot reverse direction), which come from other channels. Each queue corresponds to a channel in the AV model.

Term	Definition
$(c \pm k)_{sd}$	The virtual channel that is k steps after (or before) c on the path from s to d.
$\overline{c}$	The set of channels that share the same physical link as c.

Table A.1: AV Machine Architecture Notation

A.2 Channels

A "channel" in the AV model equations, designated c, corresponds to a queue, such as one of the three shown in the picture of one dimension of the J-Machine router in Figure A-1. The deterministic routing is represented in the notation given in Table A.1. In the model, a channel is assumed to be small, holding only one flit.[2]

Figure A-2 shows the complete set of channels for one Message Driven Processor within the J-Machine. The Priority 1 network is ignored. The figure shows the tiered router

[2]See Section A.5.1 for a discussion channel capacity and how it relates to assumptions about the rigidity of wormhole messages.

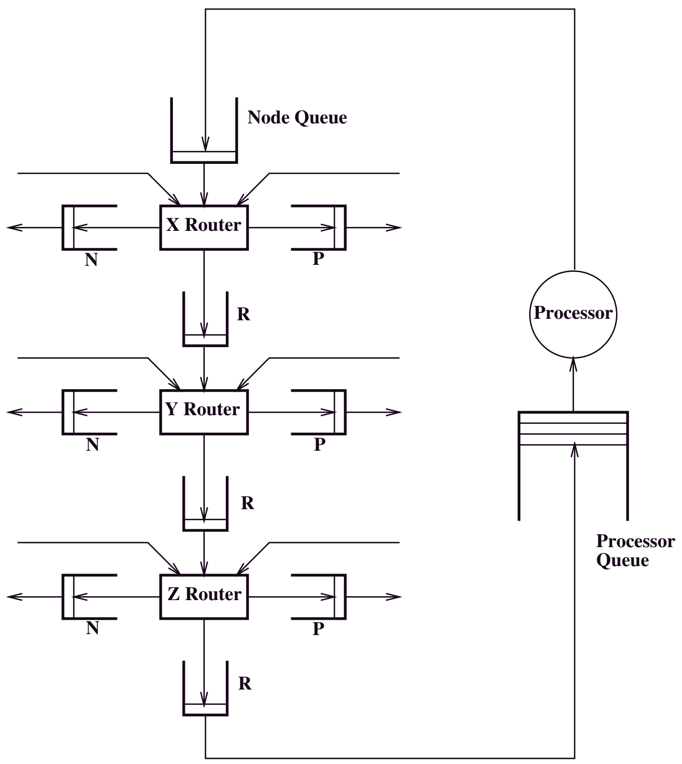


Figure A-2: Topology of a Message-Driven Processor in the J-Machine. The node queue and processor queue are modeled with a different set of equations from the rest of the model.

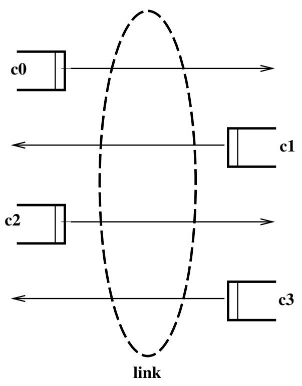


Figure A-3: The broken ellipse denotes a physical link shared by the four channels c_0, c_1, c_2, and c_3.

structure that implements the dimension ordered routing. AV instantiates the same basic set of equations over each channel. The node queue, which accepts messages from processors for the network, and the processor queue, which accepts messages for the processor from the network, are modeled by other equations, because those queues are assumed to be infinite.

Channels may share a physical link; in the J-Machine the wires between chips are shared between the positive and negative channels. In Figure A-3, four channels share a single physical link. In the AV notation (Table A.1), the other channels sharing a physical link with c are designated $\overline{c}$, so that in Figure A-3, $\overline{c_0} = \{c_1, c_2, c_3\}$.

A.3 Workload

The workload parameters for the AV model are given in Table A.2. The AV model is a multi-class MVA model; there are N classes, each associated with one of the processors in the machine. Each class has a fixed number of outstanding messages N_{out}, which follow a Request/Reply pattern. In this pattern, a given message starts at the home processor s for its class, moves into the communication network as a request to a destination d selected with probability F_{sd}, receives service at that remote destination with mean service time τ, and

Term	Definition
F_{sd}	Visit ratio for class s customer at processor d.
L_j	Length in flits of a message of type j
P_j	Probability that processor chooses to initiate a request of type j
N_{out}	Number of messages that each processor can have outstanding.
τ	Mean service time for a request at a processor.

Table A.2: AV Workload Parameters

then returns home to its class processor s via the communication network. When it reaches the home processor, it receives service (with mean service time τ) before repeating the cycle.

There are four types of messages, $\{\text{msg1}, \text{resp1}, \text{msg2}, \text{resp2}\}$. Two are "request" types and always flow from the home processor to a random destination. The other two are "response" or reply types and flow back from that random processor to the home processor. Message type is denoted by the subscript j. Each type can have different lengths; these are denoted $L_{\text{msg1}}, L_{\text{resp1}}, L_{\text{msg2}}, L_{\text{resp2}}$. When a source processor s generates a new message, it randomly chooses to generate msg1 request with probability P_{msg1} and a msg2 request with probability P_{msg2}. The response to a msg1 message is always a resp1 message, and so on.[3]

AV's workload specification differs from the multi-class MVA workload (presented in Section 5.1.3); this difference is mostly in notation. For example, in AV, the service rate for all processors is $1/\tau$, while the general multi-class model allows servers to have different rates $= \mu_i$. Also, the workload matrix in AV, $\{F_{sd}\}$, is a compact representation of a Request/Reply traffic pattern for which the closed multiclass queueing network routing matrix would be:

$$r_{ij,c} = \begin{cases} F_{ij} & (c = i) \wedge (j \neq i) \\ 1 & (c \neq i) \wedge (j = c) \\ 0 & (c = i) \wedge (j = i) \end{cases} \tag{A.1}$$

This equation shows that messages go to the remote destinations with probability F_{ij}, always

[3]Extending the model to add or remove pairs of message types is straightforward; the computation and memory requirements of the model are approximately linear in the number of types.

return to the home processor, and that the home processor never sends requests to itself.

A.4 System

The system model parameters are given in Table A.3. The primary parameters of the model are the mean round trip time (reciprocal of class throughput) for each class $R[i]$. This is the mean round trip time for a customer to be transmitted through the network as a request, to receive remote service, to be transmitted back home, and to receive home service. It is composed of three components – the home processor service time, the time spent in the network as request and reply, and the remote service time:

$$R[i] = r_{\text{proc}}[i] + r_{\text{network}}[i] + r_{\text{remote}}[i] \tag{A.2}$$

The mean time spent in the network is the weighted average of the time spent in the network over all remote destinations, and message type:

$$r_{\text{network}}[s] = \sum_{d \neq s} F_{sd}(P_{\text{msg1}}(r_{\text{msg1},sd} + r_{\text{resp1},ds}) + P_{\text{msg2}}(r_{\text{msg2},sd} + r_{\text{resp2},sd})) \tag{A.3}$$

The time spent in the network is the waiting time at the node queue, the time in the channels, and then the time for the body of the message to catch up after the head has reached the destination.

$$r_{j,sd} = w_{\text{node},j,sd} + \sum_{c} r_{j,c,sd}[1] + T_{\text{catchup},j,sd} \tag{A.4}$$

The sum is over all channels, starting at the node queue, and ending at the last channel before the processor queue.

A.5 Network

The AV model parameters for the channels within the network are in Table A.4. These parameters and the relationships among them are described in this section.

A.5.1 Rigid Messages

A key simplifying assumption in the AV model is that the messages are *rigid*, that is, that the head, body, and tail flits of the message all move in lock step through the channels in

Term	Definition
$R[i]$	Mean round-trip time for customer of class i.
$r_{\text{network}}[i], r_{\text{proc}}[i], r_{\text{remote}}[i]$	Mean residence time for a customer of class i in the network, at the processor, and at the remote server, respectively.
$r_{j,sd}$	Mean residence time in the network for a message of type j from s to d.

Table A.3: AV Model Parameters, System.

the network. If the head flit of a rigid message encounters some contention or blocking, all of the flits in the message are also immediately blocked (Figure A-5). The rigid message assumption implies instantaneous communication of blocking information backward along the flit.

Adve and Vernon do not describe their model as having rigid messages, but it is implicit in their model equations and in their description saying that their model has single-flit buffers. To say that messages are modeled as being rigid is more accurate because, without the instantaneous status communication, flits can make progress only if there are empty buffers, or holes, into which they can move.

The J-Machine messages are not rigid. In the J-Machine and in any other realistically designed network, this blocking status propagates backward along the channels holding message flits at the same rate as the head makes progress – one channel per cycle. Each flit independently makes the decision whether to advance, based on whether the downstream channel queue has space. If the flit buffers only held one flit, the message would need to stretch by a factor of two to create the empty buffers (holes) needed to make progress.

Because of the need for instantaneous status communication, rigid messages are unrealizable in practice. However, the rigid modeling assumption is good because it is difficult to model non-rigid messages (see Section 5.1.5). In the course of validating the AV model against a non-rigid two-flit buffer simulation, it became apparent that, while the individual channel residencies can differ greatly, the system results are qualitatively correct; even though the

Term	Definition
$b_{\mathrm{node},j,s\|q}[k]$, $q_{\mathrm{node},j,s\|q}$	For a message of class q arriving to the link going from processor to switch(the "node queue") at server s: respectively, the probability that the link is busy serving the kth flit of a request of type j, and the mean number of waiting requests of type j.
$r_{\mathrm{node},j,s}[k]$	Mean residence time for the kth flit of a request of type j, when it is the head flit on the link from processor to switch at server s.
$D_{c,i\|j}(s)$	Set $\{d\|$ Messages from s to d, or responses from d to s if j is a reply message, visit c via input port $i\}$.
$r_{j,c,I}[k]$, $u_{j,c,I}[k]$	Respectively, the mean residence time of the kth flit of messages of type j on channel c, which arrive to c via input port I, and the mean utilization of channel c by such flits.
$r_{j,c,sd}[k]$	Mean residence time on channel c of the kth flit of a message of type j from s to d.
$T_{\mathrm{catchup},j,sd}$	The time that it takes the tail of the message of type j from processor s to reach the destination d once the head flit has reached that destination.
$u_{\mathrm{link},c}$	Utilization of the physical link by channel c.
$u_{\mathrm{link},\bar{c}}$	Utilization of the physical link by all channels sharing it with c.
$u_{\mathrm{out},j,(c-1)_i}[k]$	The utilization of the channel feeding c from input port i by flit k of type j messages which will travel to channel c.
$w_{\mathrm{link},c}$	The waiting time for the physical link for a flit on channel c.
$w_{\mathrm{node},s\|q}$	Mean waiting time for the link from processor to switch at server s, for the header flit of a message of class q.
$w_{\mathrm{node},j,sd}$	Mean waiting time for the link from processor to switch at server s, for the header flit of a message of type j going to destination d.
$w_{c\|I}$	Mean waiting time for channel c, for a message arriving to c from input port I.
$q_{\mathrm{proc}}(i,\vec{K})$, $r_{\mathrm{proc}}(i,\vec{K})$	Steady state mean queue length, and residence time of processor i when the population vector (the number of customers in each class) is $\vec{K}$

Table A.4: AV Model Parameters, Channel

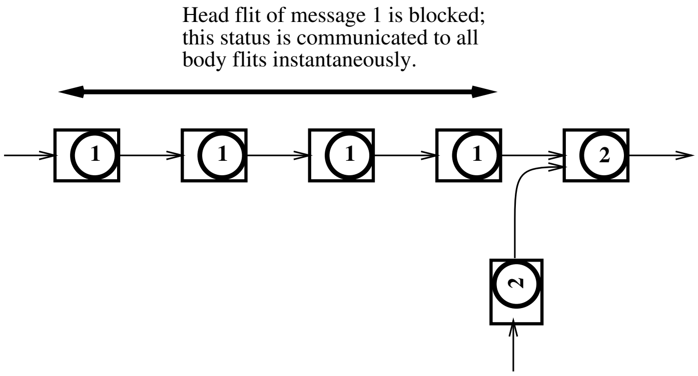


Figure A-4: A rigid message encounters blocking. Message 1 is moving to the right, with each flit residing in a different buffer. The head flit of the message is blocked by message 2. Because the message is rigid, at the moment that message 1's head flit is blocked, all of message 1's body flits also immediately block. This implies instantaneous communication of the blocking status back along the message. See Equation (A.5). The J-Machine is *not* like this; but the assumption makes the AV model work.

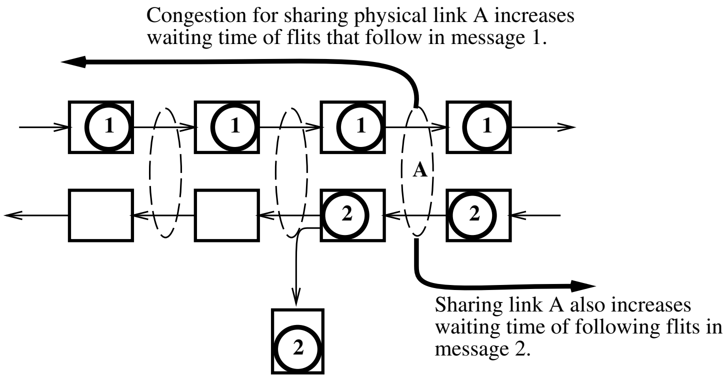


Figure A-5: A rigid Message encounters congestion. Message 1 shares link A with message 2; this results in a loss in throughput and increase in waiting time. Equation (A.5) propagates this blockage information backward along the message. This assumes fair sharing of the link. The J-Machine is *not* like this; but the assumption makes the AV model work.

rigid model is unrealistic, it works well.

The AV rigid channel model is as follows:[4]

$$r_{j,c,sd}[k] = \begin{cases} 1 + w_{(c+k)_{sd}|I_{s,(c+k)_{sd}}} + \max_{c' \in \{c,(c+1)_{sd},\ldots,(c+k-1)_{sd}\}}(w_{\text{link},c'}) & d \text{ is } k \text{ or more steps from } c, \\ 1 + \max_{c' \in \{c,(c+1)_{sd},\ldots,(d-1)_{sd}\}}(w_{\text{link},c'}) & \text{otherwise} \end{cases} \tag{A.5}$$

The notation $I_{s,c}$ is the input channel to c when coming from source s.

This equation says that the amount of time that flit k of a type j message spends at channel c when going from source s to destination d is one cycle for the transmission time, plus the waiting time that the head flit (which is k flits ahead along the network) is going to experience, plus the maximum delay for link sharing along the network from the flit to the head. If the head flit has reached the destination, then the delay is due only to link sharing to the destination.

A.5.2 Catchup Time

Once the head flit reaches the destination, the time for the flits behind the tail of the message to reach the destination is $T_{\text{catchup},j,sd}$.[5]

$$T_{\text{catchup},j,sd} = (L_j - 1) \times \max_{c'}(w_{\text{link},c'}) \tag{A.6}$$

where $c' \in \{(d-L_j)_{sd}, (d-L_j+1)_{sd}, \ldots, (d-1)_{sd}\}$, i.e., along all channels from tail to the head when the head is at the destination.

A.5.3 Channel Waiting Time

The value $w_{c|I}$ is the waiting time, from the moment a flit arrives on channel I, for channel c. It is calculated as follows:

$$w_{c|I} \;=\; \sum_{i \neq I} \sum_{j} \sum_{k=1}^{L_j} u_{j,c,i}[k] \left(\frac{1}{2} r_{j,c,i}[k] + \sum_{l=k+1}^{L_j} r_{j,c,i}[l] \right)$$

[4]This equation differs from the AV version in that it is written differently, and the cost for sharing the physical channel is calculated as the maximum over several flits. Also, I use $w_{\text{link},c}$ instead of $u_{\text{link},c}$.

[5]This differs from AV because the highest cost of sharing physical links of the busiest channel is imposed over the entire message.

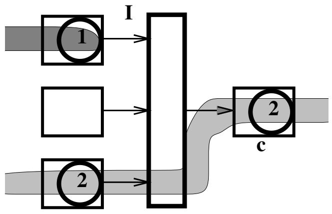


Figure A-6: First part of waiting time. Message 1 arrives to switch into c, from input port I but is blocked by message 2, which is already in progress.

$$+\sum_j \left(\frac{u_{j,c,I}[L_j]}{1 - \sum_{j'} \sum_{l=1}^{L_{j'}-1} u_{j',c,I}[l]} \times \frac{r_{j,c,I}[L_j]}{2} \right)$$
$$+\sum_{i \neq I} \sum_j \left(u_{\mathrm{out},j,(c-1)_i}[1] \sum_{l=1}^{L_j} r_{j,c,i}[l] \right). \tag{A.7}$$

AV designate this, properly, as one of the key contributions of their model.[6] There are three components, corresponding to the three lines in Equation (A.7).

1. The first component is the waiting time if the head flit arrives on I when some other message from another input port is occupying c (Figure A-6). $u_{j,c,i}[k]$, the utilization of channel c by flit k of type j messages from input port i, is used as the probability of encountering flit k of a type j message from input port i. This weights the waiting time for such an encounter, which is the time that the rest of the message occupies the channel.

2. The second component is the waiting time if the head flit arrives on input port I just behind another message from port I (Figure A-7). The waiting time is the residency of that last flit. It is not possible for a head flit on I to arrive while anything other than a tail flit from I occupies c or channel c is empty. Assuming that the arrival of a head flit on I is independent of whether channel c has a tail flit from I or has nothing from I, the probability of encountering a tail flit is the ratio of time that the channel

[6]This is also, incidentally, the critical equation to properly model the J-Machine's unfair router.

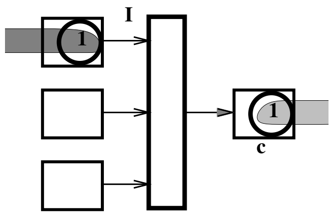


Figure A-7: Second part of waiting time. Message 1 arrives to switch into c, from input port I but is blocked by the tail of a previous message which arrived from the same input port I.

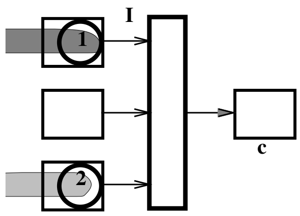


Figure A-8: Third part of waiting time. Message 1 arrives to switch into c from input I, but the head of another message 2 has already arrived at a channel which also feeds the switch and which wants to go to c. Message 1 must wait for message 2.

is occupied by a tail flit ($u_{j,c,I}[L_j]$) to the probability that the channel is empty of a tail flit ($1 - \sum_{j'} \sum_{l=1}^{L_{j'}-1} u_{j',c,I}[l]$). It is necessary to adjust the probability in this way because the waiting time is conditioned on an arrival of a head flit at input port I; this cannot occur if there is anything present other than a tail flit on c which came from I.

3. The third component is the waiting time for flits that arrive simultaneously with this flit (Figure A-8). The probability of simultaneous arrival of a message for c on port i is $u_{\text{out},j,(c-1)_i}[1]$. This conditions the waiting time for the entire message.

This waiting time model applies for a router that arbitrates fairly among the input channels to c. You can see this in the third term of Equation (A.7), where the waiting time does not consider the possibility that another message might arrive behind the one that is being waited for. Unfortunately, the routers in the J-Machine are unfair. This causes the model to differ substantially from the results on the J-Machine. Later, Section 5.3 gives changes to this equation to model the unfair routers.

A.5.4 Channel Residency

The residence time for flits on the channel is the weighted average of the source/destination residencies.[7]

$$r_{j,c,i}[k] = \begin{cases} \dfrac{\sum_{s=1}^{N} \sum_{d \in D_{c,i|j(s)}} F_{sd} r_{j,c,sd}[k]/R[s]}{\sum_{s=1}^{N} \sum_{d \in D_{c,i|j(s)}} F_{sd}/R[s]} & j \in \{\text{msg1}, \text{msg2}\} \\ \dfrac{\sum_{s=1}^{N} \sum_{d \in D_{c,i|j(s)}} F_{ds} r_{j,c,sd}[k]/R[d]}{\sum_{s=1}^{N} \sum_{d \in D_{c,i|j(s)}} F_{ds}/R[d]} & j \in \{\text{resp1}, \text{resp2}\} \end{cases} \tag{A.8}$$

The weight used to compute an average residency is the visit ratio and inverse of the round trip time. If a particular processor has a larger round trip time, then its appearance on the channel would diminish proportionally. To capture this, the residence time due to that round trip will be weighted less by the equation.

A.5.5 Channel Utilization

The utilization of the channel by flit is the cumulative residency weighted by visit ratio and inverse round trip time:[8]

$$u_{j,c,i}[k] = \begin{cases} \sum_{s=1}^{N} \sum_{d \in D_{c,i|j(s)}} \frac{N_{\text{out}} F_{sd} P_j r_{j,c,sd}[k]}{R[s]} & j \in \{\text{msg1}, \text{msg2}\} \\ \sum_{s=1}^{N} \sum_{d \in D_{c,i|j(s)}} \frac{N_{\text{out}} F_{ds} P_j r_{j,c,sd}[k]}{R[d]} & j \in \{\text{resp1}, \text{resp2}\} \end{cases} \tag{A.9}$$

This tracks the utilization independently for each input port i to c.

[7] The equation in AV is incorrect, because it accounts improperly for reply messages. It is corrected here by breaking it out by type. They also show a term P_j which cancels in the numerator and denominator. Finally, I have added the round trip time to the weight, which seems to improve the accuracy when there is a large difference in round trip time.

[8] AV also have incorrectly given this equation in their paper; again I correct it here by accounting for the reply messages.

The model also needs the utilization of a channel distinguished by each output port, employed by Equation (A.7):

$$u_{\mathrm{out},j,(c-1)_i}[k] = \begin{cases} \sum_{s=1}^{N} \sum_{d \in D_{c,i|j(s)}} \frac{N_{\mathrm{out}} F_{sd} P_j r_{j,(c-1)_i,sd}[k]}{R[s]} & j \in \{\mathrm{msg1}, \mathrm{msg2}\} \\ \sum_{s=1}^{N} \sum_{d \in D_{c,i|j(s)}} \frac{N_{\mathrm{out}} F_{ds} P_j r_{j,(c-1)_i,sd}[k]}{R[d]} & j \in \{\mathrm{resp1}, \mathrm{resp2}\} \end{cases} \quad \text{(A.10)}$$

This is the utilization by flit k of type j messages of the channel one hop upstream along input port i from c that is destined to visit c. It is used to compute channel waiting times in Equation (A.7) and is an important part of the unfair router model.

For every channel c, the model tracks its utilization as a function of the input port $u_{j,c,i}[k]$ and also as a function of the output port $u_{\mathrm{out},j,(c'-1)_{i'}}$ for each channel c' which receives its input from c via input port i'.

A.5.6 Physical Link

The utilization of a physical link by the channel is the rate at which the channel moves flits:

$$u_{\mathrm{link},c} = \sum_j \sum_{s=1}^{N} \sum_{d \in \cup_i D_{c,i|j}(s)} \frac{N_{\mathrm{out}} F_{sd} P_j L_j}{R[s]} \quad \text{(A.11)}$$

The total utilization of the physical link by other channels is the sum of their individual link utilizations:

$$u_{\mathrm{link},\overline{c}} = \sum_{c' \in \overline{c}} u_{\mathrm{link},c'}. \quad \text{(A.12)}$$

The waiting time for a flit waiting for a physical channel that is shared by other virtual channels is a function of the utilization of the physical channels by those other channels. $u_{\mathrm{link},\overline{c}}$ is the probability that the physical channel is busy with some other channel than c. If the channel is busy, then the flit must wait another cycle. Assuming that the wait is independent from one cycle to the next, then the waiting time for the other channels is equal to the sum of the infinite geometric series with ratio $u_{\mathrm{link},\overline{c}}$:

$$w_{\mathrm{link},c} = \frac{u_{\mathrm{link},\overline{c}}}{1 - u_{\mathrm{link},\overline{c}}}. \quad \text{(A.13)}$$

This estimate[9] has the desirable property that if the other channels have a 50% utilization, then $w_{\mathrm{link},c} = 1$. This waiting time for other virtual channels on the same virtual channel

[9] [AV91] used $u_{\mathrm{link},\overline{c}}$ directly as $w_{\mathrm{link},c}$. This is a good estimate of Equation (A.13) if $u_{\mathrm{link},\overline{c}}$ is small. But at the heavy load points in this thesis, it is necessary to strengthen the modeling of inter-channel interference as function of load, because $w_{\mathrm{link},c}$ and $u_{\mathrm{link},\overline{c}}$ diverge.

was used by AV to model the virtual channels used to break deadlocks in the torus network. The J-Machine (being a mesh topology) does not include deadlock-breaking virtual channels. The use of $w_{\mathrm{link},\bar{c}}$ here accounts for the multiplexing of different directions onto the same wire pins.

A.5.7 Node Queue

The queue between the processor and the network channels is named the "node" queue (see Figure A-2). On the MDP this queue is only 8 flits long, and if it fills, this processor experiences a SEND fault. However, the AV model assumes that this queue is infinite, and so does the implementation here.[10]

The waiting time at the node for processor s for a class q arriving message must account for queued messages and the waiting time to enter the network. This is done by the following equation:

$$w_{\mathrm{node},s|q} = \sum_j \left\{ \begin{array}{l} \sum_{k=1}^{L_j} \left\{ b_{\mathrm{node},j,s|q}[k] \left(\frac{r_{\mathrm{node},j,s}[k]}{2} + \sum_{l=k+1}^{L_j} r_{\mathrm{node},j,s}[l] \right) \right\} \\ \quad + q_{\mathrm{node},j,s|q} \sum_{l=1}^{L_j} r_{\mathrm{node},j,s}[l] \end{array} \right\} \tag{A.14}$$

The first part of the equation, in the innermost set of braces, is the waiting time for any message that is in progress when the message arrives. The second part accounts for the wait for any queued messages.

The waiting time at the node contributes to the network time in Equation (A.4), using the following small definition:

$$w_{\mathrm{node},j,sd} = \left\{ \begin{array}{ll} w_{\mathrm{node},s|s} & j \in \mathrm{msg1}, \mathrm{msg2} \\ w_{\mathrm{node},s|d} & j \in \mathrm{resp1}, \mathrm{resp2} \end{array} \right. \tag{A.15}$$

[10] I would like to change this, and made some simple attempts which did not work. My current intuition is that the ability for the iterative AMVA algorithm to solve the complete system model relies on this queue being infinite, as a sort of "shock absorber" to the stiff blocking and waiting time for the wormhole network. One could argue that assuming that this queue is finite could be a substantial weakness in this J-Machine model. In response, I would argue that blocking feedback from the network which leads to accumulations in this queue is instead reflected as accumulation in the processor queue (again, see Figure A-2) because the node queue is fed only from a single choice, the processor, which is fed only by the processor queue. In Sections 5.2 and 5.3, my measurements from the MVA model assume that its predictions of node queue size are equivalent to predictions of processor queue size.

The mean number of messages in the node queue is computed using Little's Law:[11] [12]

$$q_{\mathrm{node},j,s|q} = \begin{cases} \sum_{d\neq s} \frac{(N_{\mathrm{out}}-(s=q))F_{sd}P_j w_{\mathrm{node},s|s}}{R[s]} & j \in \{\mathrm{msg1}, \mathrm{msg2}\} \\ \sum_{s'\neq s} \frac{(N_{\mathrm{out}}-(s'=q))F_{s's}P_j w_{\mathrm{node},s|s'}}{R[s']} & j \in \{\mathrm{resp1}, \mathrm{resp2}\} \end{cases} \tag{A.16}$$

This equation uses the Schweitzer approximation, subtracting one from the total number of customers in the loop when the class of the arriving message is equal to the class being observed.

The probability that on arrival of a class q message, a node s is busy with flit k of a type j message is the weighted residency over the loop time.[13]

$$b_{\mathrm{node},j,s|q}[k] = \begin{cases} \sum_{d\neq s} \frac{(N_{\mathrm{out}}-(s=q))F_{sd}P_j r_{\mathrm{node},j,sd}[k]}{R[s]} & j \in \{\mathrm{msg1}, \mathrm{msg2}\} \\ \sum_{s'\neq s} \frac{(N_{\mathrm{out}}-(s'=q))F_{s's}P_j r_{\mathrm{node},j,ss'}[k]}{R[s']} & j \in \{\mathrm{resp1}, \mathrm{resp2}\} \end{cases} \tag{A.17}$$

The Schweitzer approximation accounts for when the class of the arrival node matches the home node.

The residence time at the node for each flit is the weighted residence time. This is similar to the channel residence time, Equation (A.8).[14]

$$r_{\mathrm{node},j,s}[k] = \begin{cases} \frac{\sum_{d=1}^{N} F_{sd}P_j r_{j,\mathrm{node},sd}[k]/R[s]}{\sum_{d=1}^{N} F_{sd}P_j/R[s]} & j \in \{\mathrm{msg1}, \mathrm{msg2}\} \\ \frac{\sum_{s'=1}^{N} F_{s's}P_j r_{j,\mathrm{node},s's}[k]/R[s']}{\sum_{s'=1}^{N} F_{s's}P_j/R[s']} & j \in \{\mathrm{msg1}, \mathrm{msg2}\} \end{cases} \tag{A.18}$$

A.6 Processor

The processors are modeled with a subroutine AMVA from the system AMVA. That is, for each long main iteration AMVA iteration to improve network parameters, a subroutine AMVA is performed *to completion* to compute processor parameters (Figure A-9). This is the structure used by AV in their solver, though they used full MVA for the processors. Complete MVA is impractical when modeling the J-Machine because the MDP does not isolate "request" and "reply" queues, but instead handles all traffic by the single processor.

[11]The first term here is the same as in AV, but note that there is no dependence on d other than F_{sd}, whose sum over d is 1. The second term in AV is incorrect.

[12]As in previous chapters, the value of a relation such as $(a = b)$ in an expression is 1 if the relation is true and 0 if it is false.

[13]AV completely omitted the dependence on $r_{\mathrm{node},g,sd}[k]$.

[14]As in Equation (A.8), I have included the inverse round trip time in the weight.

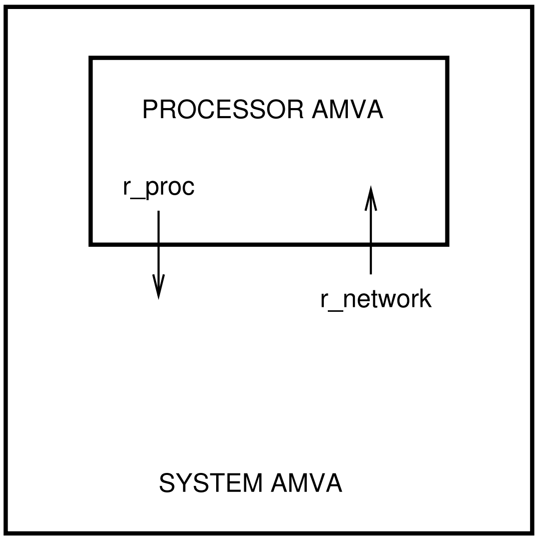


Figure A-9: Information flow within AV model. The system MVA includes a subroutine MVA for the processors alone. The network residencies are communicated in to the processor model, which iterates to recompute new processor residencies. A complete solution to the processor model is computed for each iteration of the system model.

The following equations describe the processor model. Note their resemblance to the generic multiclass AMVA equations (5.10) through (5.13). The processor AMVA subroutine model differs from the generic AMVA in only the manner in which the current status of the high level AMVA is coupled: it includes the wormhole network latency $r_{\text{network},i}[\vec{K}]$ in Equation (A.23). Otherwise, the equations are simple rearrangements of the basic AMVA equations. The rearrangements are needed so that the many routing matrices $r_{ij,c}$ can be collapsed into a single Request/Reply load matrix F_{sd}.

The complete queue size at processor i is the sum of the queue sizes by class:

$$q_{\text{proc},i}(\vec{K}) = \sum_{c=1}^{N} q_{\text{proc},i,c}(\vec{K}). \tag{A.19}$$

The time that a customer spends on the processor is the service time for the customers that are present, plus the service time for itself:

$$r_{\text{proc},i,c}(\vec{K}) = \tau \left(q_{\text{proc},i}(\vec{K} - \vec{e_c}) + 1 \right). \tag{A.20}$$

The queue size by class is computed using the Schweitzer approximation and Arrival Theorem:

$$q_{\text{proc},i,c}(\vec{K} - \vec{e_{c'}}) = \begin{cases} q_{\text{proc},i,c}(\vec{K}) & \text{if } c \neq c' \\ \left(\frac{k_c - 1}{k_c}\right) q_{\text{proc},i,c}(\vec{K}) & \text{if } c = c' \end{cases} \tag{A.21}$$

The time spent on the remote processor is used by the high level network AMVA so it is designated separately:

$$r_{\text{remote},i}(\vec{K}) = \sum_{j \neq i} F_{ij} r_{\text{proc},j,i}(\vec{K}). \tag{A.22}$$

This throughput at processor i is the throughput of the home class (the N_{out} messages that return every other message to processor i), not the throughput of all messages through the home processor:

$$T_{\text{proc},i}(\vec{K}) = \frac{k_c}{r_{\text{proc},i,i}(\vec{K}) + r_{\text{remote},i}(\vec{K}) + r_{\text{network}}[i]}. \tag{A.23}$$

The queue sizes by class are computed using Little's Law:

$$q_{\text{proc},i,c}(\vec{K}) = \begin{cases} T_{\text{proc},i}(\vec{K}) F_{ci} r_{\text{proc},i,c}(\vec{K}) & \text{if } i \neq c \\ T_{\text{proc},i}(\vec{K}) r_{\text{proc},i,c}(\vec{K}) & \text{if } i = c \end{cases} \tag{A.24}$$

The above equations are evaluated at the population vector:

$$\vec{K} = (N_{\text{out}}, N_{\text{out}}, \ldots, N_{\text{out}}). \tag{A.25}$$

The values from the processor AMVA are copied back into the wormhole model in Equation (A.2) via the values $r_{\text{proc}}[i]$ and $r_{\text{remote}}[i]$ defined as: [15] [16]

$$r_{\text{proc}}[i] = r_{\text{proc},i,i}(\vec{K}) \tag{A.26}$$

$$r_{\text{remote}}[i] = r_{\text{remote},i}(\vec{K}) \tag{A.27}$$

A.7 Independence Assumptions

AV's equations rely on many assumptions of independence between events.

For an example of a place in the model where independence is assumed, consider Equation (A.7) which is used to compute $w_{c|I}$. This value is the waiting time for a message destined for channel c from the time that it arrives on channel c's input channel I. This long equation makes several assumptions about independence; one of the assumptions is in its use of the value $u_{j,c,i}[k]$. This value is the utilization of channel c by type, input and flit, and for a randomly selected instant of arrival to the channel. Another way of looking at $u_{j,c,i[k]}$ is that it is equal to the probability that the channel will be occupied by flit k of a type j message from input i. Equation (A.7) uses $u_{j,c,i[k]}$ as an estimate of the probability that, when a message arrives on channel I, channel c will be occupied by flit k of a message of type j from input channel i. The assumption is that the utilization with random sampling instants is the same as the utilization with sampling on arrival instances. Is this assumption good? Consider the case when there is only one message in the entire system. In that case, the utilization that an arrival on input I would see would be zero – the message could never arrive and see itself. But the way that the model computed $u_{j,c,i}[k]$ (as the weighted ratio of the residence time $r_{j,c,i}[k]$ to the round trip time for the messages) the value would be nonzero. The assumption fails with one message. It gets better with more messages, as long as arrival events are independent.

[15] AV reduce the residence time at the processor by considering any departures that might take place while the tail of the message flows into the processor. I do not do this.

[16] AV found that the Schweitzer approximation was inaccurate. They do exact MVA, but I can't because the processor queues get traffic from all other processors.

The basic MVA method compensates, using the Arrival Theorem, for this difference in the utilization statistics between when the channel is sampled randomly and when it is sampled at arrival instants. But this type of compensation is not used in Equation (A.7). Why? Two reasons come to mind:

- One would need to use an approximation like the Schweitzer approximation; this is not done for $w_{c|I}$ but it *is* done for other values within the model. For example, Equation (A.16) conditions $q_{\text{node},j,s|q}$ on the arriving class, and stores a separate value for each arriving class q. This is a large overhead, requiring values for N classes to be stored for N processors. In contrast, though, conditioning $u_{j,c,i}[k]$ by class would require storing *many* more values because there are several channels per processor, several inputs per channel, and many flits k.
- Even if the model did compensate for self-sampling, the amount of compensation would be slight because the probability of encountering another message of the same class within the network would be small, except near the home processor. This is another reason why the Schweitzer approximation is used at the node queue and not in the channels: the probability of encountering the same class on the reference processor (the home node) is high.

In summary, it would be hard to be more accurate, and the increased accuracy still might not be that significant.

There are other subtle aspect to this assumption of independence; systematic sampling at message arrivals might cause the mean $u_{j,c,i}[k]$ to differ from the value experienced at sample instants. For example, there is no guarantee that the network wouldn't become synchronized in some way; this might lead an arrival on input port I to always be concurrent with message presence from i on c. This synchronized arrival is difficult to model because it's not clear what this systematic process might actually *be*.

A.8 Modeling Issues

There are a number of differences between the AV model and the J-Machine.

1. **Single flit buffers rather than the J-Machine's 2-flit buffers.** The channel queues within the J-Machine routers can hold two flits; the AV model is a "single-flit" channel model, which is implicit in Equation (A.5)'s assumption of rigid messages. As mentioned in the description of Equation (A.5), the two-flit buffer is a better match to the rigid messages than is a single-flit channel. However, when there is congestion, the extent (distance between head and tail) of the messages will compress by 2x into the two-flit buffers upstream, eliminating a bit of blocking that the tail flits might have caused.

2. **Spreading in the J-Machine.** Once the head flit passes a congestion point, the head flit will move ahead faster than subsequent flits in the same message can follow. The empty channels between the head flit and the congestion point block other messages from using those channels, though the physical links may get used. Within AV, the rigid message Equation (A.5) assumes that the head flit travels at a rate determined by the maximum congestion along the message; this eliminates the "racing" effect.

3. **Processor queues/node queues.** The J-Machine has a large processor queue, and a small (8-flit) "node" queue. See Figure A-2. The AV model assumes both an infinite processor queue and an infinite node queue. For the J-Machine, network backpressure quickly fills the node queue, which blocks the processor and leads to accumulation of traffic in the processor queue. For the AV model, node queue accumulations do not couple into processor accumulations. This thesis assumes that this difference is insignificant, and that the accumulations in the node queue and the processor queue are roughly equivalent. This is important because the plots of the AV models predicted queue size are the *node* queue size.

4. **SEND faults.** When the J-Machine experiences network backpressure, the processor takes a SEND fault.[17] The SEND fault takes a long time, and the effect on performance, can be extreme, because with long messages, the SEND fault can occur in the middle of the message, leading to a big "hole" (see item 2, Spreading, above) in the message which blocks en-route network channels for the many cycles that the processor is busy handling the fault. This thesis ignores the SEND faults; attempts at modeling them proved unsuccessful.

[17] See Section 2.1.3 in the Architecture chapter for a description of the SEND fault

A.9 Closing

This Appendix has presented a modified implementation of the Adve and Vernon mean value model of closed wormhole networks. For the results of the numerical solutions of this model at the fine-grained, heavy-load points, and for the modifications to the model for the fair router, return to Section 5.2.

References

Bibliography

[AC94] Kazuhiro Aoyama and Andrew A. Chien. The cost of adaptivity and virtual lanes in a wormhole router. *Journal of VLSI Design*, 1994.

[Adv95] Vikram S. Adve. Private communication - solver source code, August 1995.

[Aky88] Ian F. Akyildiz. On the exact and approximate throughput analysis of closed queueing networks with blocking. *IEEE Transactions on Software Engineering*, 14(1):62–70, January 1988.

[AV91] Vikram S. Adve and Mary K. Vernon. Performance analysis of multiprocessor mesh interconnection networks with wormhole routing. Technical Report 1001, Computer Sciences Department, University of Wisconsin-Madison, February 1991.

[AV94] Vikram S. Adve and Mary K. Vernon. Performance analysis of mesh interconnection networks with deterministic routing. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):225–246, March 1994.

[BCL+95] Eric A. Brewer, Frederick T. Chong, Lok T. Liu, Shamik D. Sharma, and John D. Kubiatowicz. Remote queues: Exposing message queues for optimization and atomicity. In *Symposium on Parallel Algorithms and Architecture (SPAA)*, 1995.

[BCMP75] Forest Basket, K. Mani Chandi, Richard R. Muntz, and Fernando G. Palacios. Open, closed, and mixed networks of queues with different classes. *Journal of the ACM*, 22(2):248–260, April 1975.

[BG87] Dimitri Bertsekas and Robert Gallager. *Data Networks*. Prentice Hal, 1987.

[BGT95] Dimitris Bertsimas, David Gamarnik, and John N. Tsitsiklis. Stability conditions for multiclass fluid queueing networks. Sloan School of Management and Operations Research Center, December 1995.

[BJK+95] Robert D. Blumofe, Chistopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.

[BK94] Eric A. Brewer and Bradley C. Kuzmaul. How to get good performance from the CM-5 data network. In *Proceedings of the 8th international parallel processing symposium (IPPS 94)*, pages 858–867, April 1994.

[BL93] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing (STOC)*, May 1993.

[Buz72] Jeffrey P. Buzen. *Queueing Network Models of Multiprogramming.* PhD thesis, Harvard University, Cambridge, MA, August 1972.

[Buz73] Jeffrey P. Buzen. Computational algorithms for closed queueing networks with exponential servers. *Communications of the ACM*, 16(9):527–531, 1973.

[CA96] Frederick T. Chong and Anant Agarwal. Shared memory versus message passing for iterative solution of sparse, irregular problems. Technical Report MIT/LCS/TR-697, Laboratory for Computer Science Massachusetts Institute of Technology, Cambridge, Massachusetts, October 1996.

[CD88a] Andrew Chien and William J. Dally. CST: An object-oriented concurrent language. In *Object-Based Concurrent Programming Workshop*, pages 28–31. OOPSLA '88, September 1988. Conference held at San Diego, CA. SIGPLAN Notices, February 1989.

[CD88b] Andrew Chien and William J. Dally. Object-oriented concurrent programming in CST. In *Proceedings of the Third Conference on Hypercube Computers*, pages 434–439. SIAM, January 1988. Conference held at Pasadena, CA; VLSI memo 88-450.

[Cha87] Linda Chao. Architectural features of a message-driven processor. SB Thesis, May 1987.

[CKP93] Andrew A. Chien, Vijay Karamcheti, and John Plevyak. The concert system - compiler and runtime support for efficient, fine-grained concurrent object-oriented programs. Department of Computer Science, 1034 W. Springfield Avenue, Urbana, IL 61801, June 1993.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[CN82] K. Mani Chandy and Doug Neuse. Linearizer: A heuristic algorithm for queueing network models of computer systems. *Communications of the ACM*, 25(2):126–134, February 1982.

[Dal87] William J. Dally. *A VLSI Architecture for Concurrent Data Structures*. Kluwer Academic Publishers, Norwood, Massachusetts, 1987.

[Dal90a] William J. Dally. The J-Machine system. In Patrick Winston with Sarah A. Shellard, editor, *Artificial Intelligence at MIT: Expanding Frontiers*, chapter 21, pages 536–569. MIT Press, 1990.

[Dal90b] William J. Dally. Network and processor architecture for message-driven computers. In Suaya and Birtwhistle, editors, *VLSI and Parallel Computation*. Morgan Kaufmann, 1990.

[Dal90c] William J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 39(6), June 1990.

[Dal90d] William J. Dally. Virtual-channel flow control. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 60–68, May 1990.

[Dal92] William J. Dally. The J-Machine: System support for actors. In Carl Hewitt and Gul Agha, editors, *Towards Open Information Science*. MIT Press, Cambridge, MA, 1992. VLSI memo 88-491.

[Dal93] William J. Dally. Personal Communication, 1993.

[DCF+89] William J. Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, John Keen, Michael Larivee, Richard Lethin, Peter Nuth, Scott Wills, Paul Carrick, and Greg Fyler. The J-Machine: A fine-grain concurrent computer. In G.X. Ritter, editor, *Proceedings of the IFIP Congress*, pages 1147–1153. North-Holland, August 1989.

[DDH+94] William J. Dally, Larry R. Dennison, David Harris, Kinhong Kan, and Thucydides Xanthopoulos. The reliable router: A reliable and high-performance communication substrate for parallel computers. In *Proceedings of Parallel Computers Routing and Communication Workshop*, 1994.

[DFK+92] William J. Dally, J.A. Stuart Fiske, John S. Keen, Richard A. Lethin, Michael D. Noakes, Peter R. Nuth, Roy E. Davison, and Gregory A. Fyler. The Message-Driven Processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, 12(2):23–39, April 1992.

[DK85] William J. Dally and James T. Kajiya. An object oriented architecture. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 154–161, Boston, MA, June 1985.

[DKN93] William J. Dally, John S. Keen, and Michael D. Noakes. The J-machine multicomputer: Architecture and evaluation. In *Compcon Spring 93*, pages 183–188. IEEE Computer Society Press, February 1993.

[DS87] William J. Dally and Charles L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–53, May 1987.

[dSeSLM84] E. de Souza e Silva, S. S. Lavenberg, and R. R. Muntz. A perspective on iterative methods for the approximate analysis of closed queueing networks. *Mathematical Computer Performance and Reliability*, 1984.

[DW94] J. G. Dai and G. Weiss. Stability and instability of fluid models for certain re-entrant lines. Georgia Institute of Technology, February 1994.

[Fat92] Jerko Fatovic. A ray tracer for the J-Machine. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1992.

[FKD+95] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine Multicomputer. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 146–156, Ann Arbor, MI, December 1995. ACM.

[Gal94] Robert G. Gallager. Discrete stochastic processes. Notes for M.I.T. Subject 6.262, January 1994.

[GJ91] Michael R. Garey and David S. Johnson. *Computers and Intractibility, A Guide to the Theory of NP-Completeness*. W.H Freeman and Company, 1991.

[GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley Publishing Company, second edition, 1994.

[Gur94] Yevgeny Gurevich. An assembler and linker system for the M-machine software project, April 1994. Bachelor's Thesis.

[HB93] Mor Harchol and Paul E. Black. Queueing theory analysis of greedy routing on square arrays. Technical Report UCB/CSD 93/756, Computer Science Division, University of California, Berkeley, California, 1993.

[HBW95] Mor Harchol-Balter and David Wolfe. Bounding delays in packet-routing networks. In *Proceedings of the 27th ACM Symposium on Theory of Computing*, May 1995.

[HCD89] Waldemar Horwat, Andrew Chien, and William J. Dally. Experience with CST:programming and implementation. In *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation*, pages 101–109, 1989.

[HN93] J. Michael Harrison and Vien Nguyen. Brownian models of multiclass queueing networks: Current status and open problems. *Queueing Systems*, 13:5–40, 1993.

[Hor89] Waldemar Horwat. Concurrent smalltalk on the message-driven processor. Master's thesis, MIT, May 1989.

[HP93] Peter G. Harrison and Naresh M. Patel. *Performance Modelling of Communication Networks and Computer Architecture.* Addison Wesley, 1993.

[HTD] Waldemar Horwat, Brian Totty, and William J. Dally. Cosmos: An operating system for a fine-grain concurrent computer. submitted for publication.

[JGG+95] E. G. Coffman Jr., E. N. Gilbert, A. G. Greenberg, F. T. Leighton, Philippe Robert, and A. L. Stolyar. Queues served by a rotating ring. Unpublished Manuscript, May 1995.

[JK88] Van Jacobson and Mike Karels. Congestion avoidance and control. In *SIGCOMM*, pages 314–328, 1988.

[Kan93] Shaun Yoshie Kaneshiro. Branch and bound search on the J-Machine. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1993.

[KC94] Vijay Karamcheti and Andrew A. Chien. Software overhead in messaging layers: Where does the time go? In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI).* ACM, 1994.

[KD93] Demetres D. Kouvatsos and Spiros G. Denazis. Entropy maximized queueing networks with blocking and multiple job classes. *Performance Evaluation*, 17:189–205, 1993.

[KJA+93] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Ben-Hong Lim. Integrating message-passing and shared-memory: Early experiences. In *PPOPP*, 1993.

[KLC94] Jae H. Kim, Ziqiang Liu, and Andrew A. Chien. Compressionless routing: A framework for adaptive and fault-tolerant routing. In *21st International Symposium on Computer Architecture*, Urbana, IL, 1994. Department of Computer Science, University of Illinois at Urbana-Champaign.

[Kle64] Leonard Kleinrock. *Communication nets; stochastic message flow and delay.* McGraw-Hill, 1964.

[Kle75] Leonard Kleinrock. *Queueing Systems, Volume 1: Theory.* Wiley, 1975.

[KS91] S. Konstantinidou and L. Snyder. Chaos router: architecture and performance. In *18th Annual Symposium on Computer Architecture*, pages 212–221, 1991.

[KS93] R. E. Kessler and J. L. Schwarzmeier. CRAY T3D: A New Dimension for Cray Research. In *COMPCON*, 1993.

[KSS+95] Yuetsu Kodama, Hirohumi Sakane, Mitsuhisa Sato, Hayato Yamana, Shuichi Sakai, and Yoshinori Yamaguchi. The EM-X parallel computer: Architecture and basic performance. In *Proceedings of the 22nd International Symposium On Computer Architecture (ISCA)*, pages 14–23, Santa Margherita Ligure Italy, 1995.

[Kub95] John Kubiatowicz. Personal Communication, 1995.

[L+92] Charles E. Leiserson et al. The network architecture of the connection machine CM-5. In *Symposium on Parallel Architectures and Algorithms*, pages 272–285, San Diego, California, June 1992. ACM.

[Lei85] Charles E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, October 1985.

[Lei97] F. Thomson Leighton. Personal Communication, January 1997.

[Len89] Arjen K. Lenstra. Long integer package. Computer Program, 1989. Bell Labs, also part of the rsa129 project source.

[Liu68] C. L. Liu. *Introduction to Combinatorial Mathematics.* MacGraw-Hill Book Company, 1968.

[LLJ+92] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH prototype: Implementation and performance. In *Proceedings of 19th Annual International Symposium on Computer Architecture*, pages 92–103. IEEE, 1992.

[Mas94] Daniel Maskit. A message-driven programming system for fine-grain multicomputers. Master's thesis, Computer Science Department, California Institute of Technology, Pasadena, California, February 1994.

[MM84] J. McKenna and Debasis Mitra. Asymptotic expansions and integral representations of moments of queue lengths in closed Markovian networks. *Journal of the ACM*, 31(2):346–360, April 1984.

[MM94] Debasis Mitra and Isi Mitrani. Efficient window flow control for high speed data networks with small buffers. *Annals of Operations Research*, 49:1–24, 1994.

[MT93] Daniel Maskit and Stephen Taylor. Experiences in programming the J-machine. Technical Report Caltech-CS-TR-93-11, California Institute of Technology, April 1993.

[Mun72] Richard R. Muntz. Poisson departure processes and queueing networks. Technical Report RC-4145, IBM Research, December 1972.

[MZT93] Daniel Maskit, Yair Zadik, and Stephen Taylor. System tools for the J-machine. Technical Report Caltech-CS-TR-93-12, California Institute of Technology, April 1993.

[ND90] Michael Noakes and William J. Dally. System design of the J-Machine. In William J. Dally, editor, *Sixth MIT Conference of Advanced Research in VLSI*, pages 179–194, Cambridge, MA 02139, 1990. The MIT Press.

[ND92] Peter R. Nuth and William J. Dally. The J-machine network. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors*, pages 420–423. IEEE, October 1992.

[Noa91] Michael Noakes. MDP programmer's manual. Concurrent VLSI Architecture Memo 40, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, October 1991.

[Nut] Peter Nuth. Personal Communications.

[NWD93] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The J-Machine multicomputer: An architectural evaluation. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 224–235, San Diego, California, May 1993. IEEE.

[Onv90] Raif O. Onvural. Survey of closed queuing networks with blocking. *Computing Surveys*, 22:83–121, June 1990.

[Per94] Harry G. Perros. *Queueing Networks with Blocking.* Oxford University Press, New York, 1994.

[Rei65] F. Reif. *Fundamentals of statistical and thermal physics.* McGraw-Hill, New York, 1965.

[RL80] M. Reiser and S.S Lavenberg. Mean-value analysis of closed multichain queuing networks. *Journal of the Association for Computing Machinery*, 27(2):313–322, April 1980.

[S+93] Ellen Spertus et al. Evaluation of mechanisms for fine-grained parallel programs in the J-machine and the cm-5. In *Proceedings of the International Symposium on Computer Architecture*, pages 302–313, May 1993.

[SD] Ellen Spertus and William J. Dally. Trading off control and data locality in fine-grained computing. Submitted to Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, 1994. (Not published).

[SD95] Ellen Spertus and William J. Dally. Evaluating the locality benefits of active messages. In *Proceedings of the Conference on Principles and Practice of Parallel Programming (PPOPP)*, 1995.

[ST94] Steve Scott and Greg Thorson. Optimized routing in the Cray T3D. *Lecture notes in computer science*, (953), 1994. Parallel Computer Router and Communication.

[vECGS92] Thorsten von Eicken, David Culler, Seth Goldstein, and Klaus Schauser. Active messages: A mechanism for integrated communication and computation.

In *Proceedings of 19th Annual International Symposium on Computer Architecture*, pages 256–266. IEEE, 1992.

[Whi97] Jacob White. Personal Communication, 1997.

[WHJ+95] Deborah A. Wallach, Wilson C. Hsieh, Kirk Johnson, M. Frans Kaashoek, and William E. Weihl. Optimistic active messages: A mechanism for scheduling communication and computation. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1995.