THE SIMULATION AND TUNING OF THE GLOBAL MEMORY SUBSYSTEM OF A MULTIPROCESSOR

BY

THOMAS MARTIN CONTE

B.E.E., University of Delaware, 1986

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1988

Urbana, Illinois

ACKNOWLEDGEMENTS

I would like to thank my two advisors, Professors Edward Davidson and Wen-mei Hwu, for their time, insight, and faith in me, and especially my parents and family for their love and support. Also, I thank my good friends for encouraging me on and standing with me through difficult times.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Multistage interconnection networks (*MIN's*) have been proposed as a means for hardware sorting [1], data alignment in array (SIMD) machines [2], and for use in shared memory multiprocessor (MIMD) systems [3]. Projects to build large-scale shared memory multiprocessors are under way in industry and academia, among them the RP3 by IBM [4], and Cedar by the University of Illinois [5] [6]. In these MIMD systems, multistage interconnection networks replace the conceptually desirable, yet infeasible, crossbar switch as a means to connect memory modules to processors.

The performance of multistage interconnection networks has been studied extensively in the literature. Patel introduced the Delta network and commented on its expected bandwidth and cost-effectiveness in [3]. Kruskal and Snir used analytical queuing theory techniques to derive the performance of buffered and unbuffered banyan multistage interconnection networks under random traffic. For the buffered banyan network, they derived an equation for the performance of the first stage of the network and estimate the performance of the network from this first-stage approximation [7]. Work done by Dias and Jump [8] used Petri net techniques to analyze the performance of a buffered Delta network. They found that the length of buffers inside the network had a large impact on performance, with an optimal value of one or two buffers per stage.

In the design of the Cedar system, the results of these performance studies and separate simulation studies were used to make initial design considerations. The entire system was not simulated exactly in an effort to speed up the design cycle. During the fabrication of the system, it was decided that an effort should be started to predict the performance of the networks via a detailed simulation. The results of this simulation study would then be used to begin the design cycle for the next version of the Cedar system, while the current version of the system was being fabricated and assembled. This thesis presents an overview of the Cedar interconnection networks, and the techniques used in the simulation of the networks, and discusses the results of the simulation. When Cedar becomes operational, the behavior of the network will be investigated via measurement techniques, which will add to the knowledge gained in this study. However, not all behavior measurable in the simulation will be directly measurable in the real system. Hence, the study described herein has long-term usefulness to the designer along with the advantage of decreasing the design time of the next version of Cedar.

In Chapter 2, the design of the global memory subsystem is discussed, reviewing those features that impact the performance of the subsystem. Then, in Chapter 3, the algorithm used to simulate the subsystem is discussed. After Chapter 3, the reader should be confident that the performance of our simulator matches that of the subsystem.

The work presented in this thesis concentrates on the performance of a global memory subsystem composed of two unidirectional Omega networks, N memory units and N processors. Systems containing 8, 16, and 64 processors and memory units are investigated in Chapters 4 and 5. The simulator will be used to determine an expression for a lower bound on the global memory subsystem's performance To do this, the subsystem's performance under its worst-possible workload will be evaluated. In Chapter 5, the simulator will be used to evaluate the usefulness of intelligent access patterns, to help understand how the subsystem performs in general.

Several conclusions will be drawn:

- There is little difference between best-case and worst-case global memory subsystem performance for small-scale (8- and 16-processor) systems. There are noticeable differences in larger (64-processor) systems.

- For all system sizes, the memory units are the bottleneck to the performance of the subsystem for all traffic patterns discussed.

- Because prefetch operations cannot be guaranteed to start at the same time, algorithms for intelligently accessing data that exploit the characteristics of Omega networks may perform no better than the worst-case accessing scenario.

From these results, some guidelines can be proposed for building a global memory subsystem using Omega networks and an equal number of memory units and processors:

- The bandwidth of the memory units of the subsystem should be equal to the bandwidth of the networks

- Prefetch operations should be synchronizable

- The compiler should use heuristics derived from simulation to hide prefetch performance by scheduling prefetch operations far enough in advance of the operations that require the prefetch data.

CHAPTER 2

SYSTEM OVERVIEW

The simulation of any subsystem of a computer requires careful consideration of the components of the computer that service the subsystem and the components that make requests of the subsystem. the *global memory subsystem* is to be simulated and analyzed. This subsystem is composed of the memory units of the global memory and the networks that shuttle requests between the computational elements and the global memory. An architectural-level description of this subsystem is presented here. The simulator that was constructed to observe the behavior of the subsystem implements a slightly more detailed description than what follows in order to capture implementation issues of the subsystem that may affect its performance. This simulator was written in the C programming language and consists of approximately two thousand lines of code; it will be described in the following chapter.

In this chapter, an overview of the Cedar system is presented, followed by a description of the global memory subsystem in detail, and conclude with some examples of the operation of the subsystem.

2.1 Cedar System Overview

The Cedar multiprocessor is composed of clusters of K *computational elements (CEs)* (currently, $K = 8$), where each cluster is a modified Alliant FX/8 mini-supercomputer. The CEs in a cluster can communicate between themselves via a crossbar switch and a synchronization bus. All CEs in all clusters have peer-access to a global memory [6].

The global memory subsystem is composed of two unidirectional, $N \times N$ Omega networks and N *memory units (MUs)* [6]. The network that takes requests to the memory units is called the *to-network*, and the network that returns requests from the memory units is called the *from-network* (See Figure 2.1). The from-network and the to-network are identical in structure and built using $K \times K$ buffered crossbar switching elements (*SE*s). Handshaking control signals are used between SEs to implement flow control inside the network. For the current version of Cedar, the SEs are 8×8 wide.

CEs access global memory via a *global memory interface board (GIB).* A GIB can be instructed by a CE to prefetch a vector from global memory in addition to normal scalar variable accesses. Even though all accesses are actually emitted from a GIB, a CE will be thought of as performing a memory reference instead of a GIB performing the reference. Since there is a GIB associated with each CE, there will be no ambiguity.

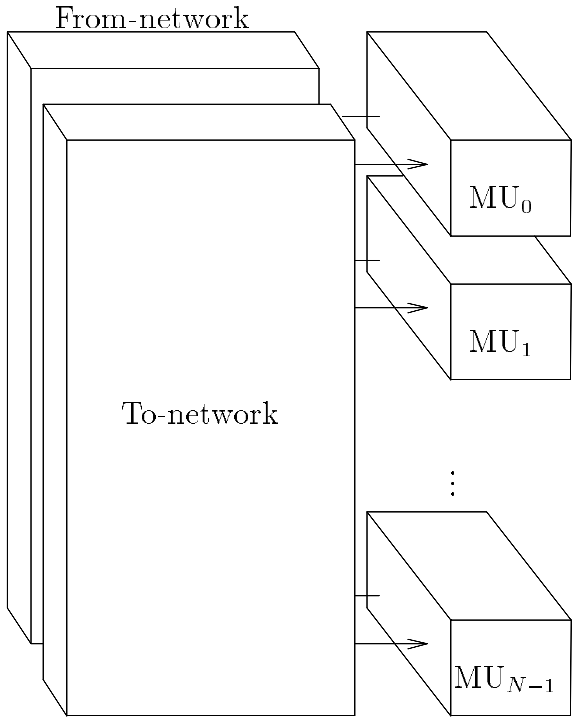


Figure 2.1: The Cedar global memory access subsystem.

The entire global memory subsystem is synchronous and controlled by a central clock. Time units delimited by this clock will be referred to as *clock tics.*

2.2 The Global Memory Subsystem

A detailed description of the global memory subsystem is now presented, first by describing the architecture of the Omega networks that comprise the to-network and the from-network. Then, the operation of the switching elements will be described and some notation will be introduced which will be useful in the examples section at the end of the chapter. Finally, at the close of this section operation of the memory units will be described.

2.2.1 The architecture of the networks

Each network is a generalized Omega network [2]. It has a total of $N = K^M$ inputs, and the SEs are arrayed into M columns of $B = N/K$ SEs. The stages, S_i, are numbered $0, \ldots, M$. In this notation, the computational elements (memory units) will be symbolically represented as stage S_0 (S_M) in the to-network (from-network) and the memory units (computational elements) as stage S_M (S_0) in the from-network (to-network).

An example of a Cedar configuration would have two stages of eight SEs ($N = 64, K = 8, M = 2$). The SEs are labeled $[S_i, j]$, where $j = 0$ is the uppermost SE and $j = B - 1$ is the lowermost SE, when stage S_i is viewed as a column of SEs.

Between each stage in a network is a $K{*}B$ shuffle connection defined by the function $\mathrm{Shuffle}_{K*B}$: Let IN_{S_i} be the label of an input line and OUT_{S_i} be the label of an output line of an SE in stage S_i , where the lines are labeled $0, \ldots, N-1$ from the top to the bottom of a column, then,

$$\mathrm{IN}_{S_i} = \mathrm{Shuffle}_{K*B}(\mathrm{OUT}_{S_{i-1}}) = (\mathrm{OUT}_{S_{i-1}} * B + \lfloor \frac{OUT_{S_{i-1}}}{K} \rfloor) \bmod N,$$

for all i, $0 < i < M - 1$ (see [2]). Additionally, there is a $K * B$ shuffle included before stage S_1 to allow a network to realize an identity permutation (CE_i accessing MU_i, $0 \leq i < N$) [3]. (For a general network, see Figure 2.2.) Routing through a network is distributed and performed by successively resolving the destination address at each stage. Let SOURCE and DEST be the source and destination addresses (respectively), expressed in base-K, for a request sent through a network. For example, for a

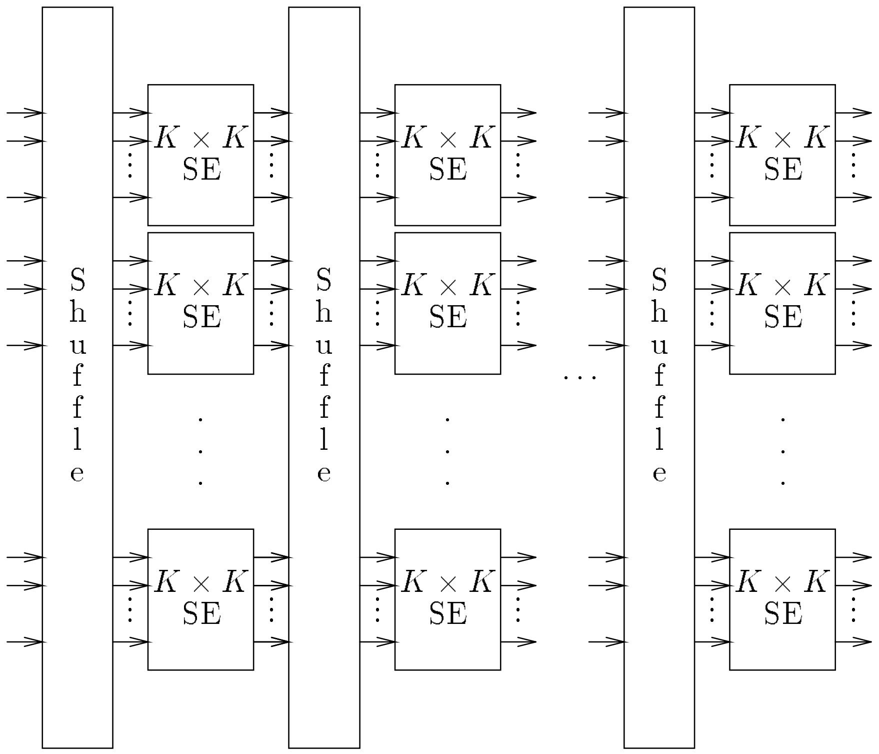


Figure 2.2: The architecture of a general Omega network.

request entering the to-network, SOURCE is the requesting CEs number, and DEST is the destination MUs number. An SE in stage S_i directs a message to the output port equal to the ith base-K digit of DEST. For notational convenience, the function $\text{Window}(\text{SOURCE}, \text{DEST}, S_i)$, is defined as the value of the network output that the request will be routed to in stage S_i, $0 < i \leq M$. Another definition of the Window function would be: concatenate SOURCE and DEST, discard the $(i-1)$st significant (base-K) digits, and then the value of Window is the M most significant digits of the remaining number. As a final comment, note that $\text{Window}(\text{SOURCE}, \text{DEST}, S_0) = \text{SOURCE}$, and that $\text{Window}(\text{SOURCE}, \text{DEST}, S_M) = \text{DEST}$.

2.2.2 Packet structure

The networks are packet-switched, where each packet is composed of a series of fixed-length words. Each line through a network is as wide as this word-size so that packet words can be transmitted in parallel. The first word in the packet is a routing and control header called the *access control word* (*ACW*). This word contains, among other information, the SOURCE and DEST addresses referred to above, and the offset into a memory module of a datum (i.e., the datum's memory address). The length of a packet is variable. If words follow the ACW, their meaning is not interpreted by the network hardware. Usually, if any words follow the ACW, they contain data and will be referred to as *data words.*

Three control signals govern the traffic on each network line. These control signals are: *BUSY,* used for flow control; *AD,* used to discriminate between access and data packets; and *HOLD,* used to implement variable length packets. The AD signal is asserted for the first word in each packet. Subsequent words are data words. The HOLD signal is set for each packet word except the last. Hence, these two signals implement variable length packets. The finite-length buffering of the network is regulated using the BUSY signal, and its detailed operation will be explained below.

There are three types of packets: read, write, and synchronization. Read packets are one word long in the *to-network* (i.e., an ACW only), and the acknowledgement to a read packet is two words long in the from-network (i.e., the ACW and the datum read). Write packets are two words long in the to-network (i.e., the ACW and the datum to be written) and one word long in the from-network (an ACW that serves as a write acknowledge). Synchronization packets can take various forms, depending on the operations performed. Since this research focuses on prefetching operations, synchronization packets will not be treated here.

Packet-word notation will be used in the following discussions of the switching elements and memory units. In this notation, an ACW is represented as one of "`R`," or "`W`," for read packets or write packets (respectively). The packet's source (i.e., the SOURCE value described above) is represented as a subscript, and the destination (i.e., DEST) as a superscript. (These sub- and superscripts will be omitted when they are not important to the discussion.) Additionally, data words are represented as "`D`." The case of the letters is significant: lower-case represents the last word of a packet, and upper-case is used for every other word of the packet. Hence, the case of the letters mirrors the setting of the HOLD signal. Since packet-word notation shall later be used in a network drawing where data flows from left to right, packets are written right to left, with the ACW on the right. For example, if CE_1 is sending to MU_8 one read and one write request, the requests would be written as shown in Table 2.1.

Table 2.1: An example of read and write requests.

	To-Network	From-Network
read	$\mathtt{r}_1^8$	$\mathtt{d}\ \mathtt{R}_8^1$
write	$\mathtt{d}\ \mathtt{W}_1^8$	$\mathtt{w}_8^1$

2.2.3 Switching element structure

As previously discussed, the switching elements perform all routing inside the networks. Inside each of these SEs, each of the K inputs connects to a first-in first-out input queue (*FIFO*) of length ℓ_{SE}. The outputs of the input queues are then connected to a crossbar switch. After the switch there is an output port (composed of a latch) before each of the K outputs. (See Figure 2.3.) Currently, the input queues are composed

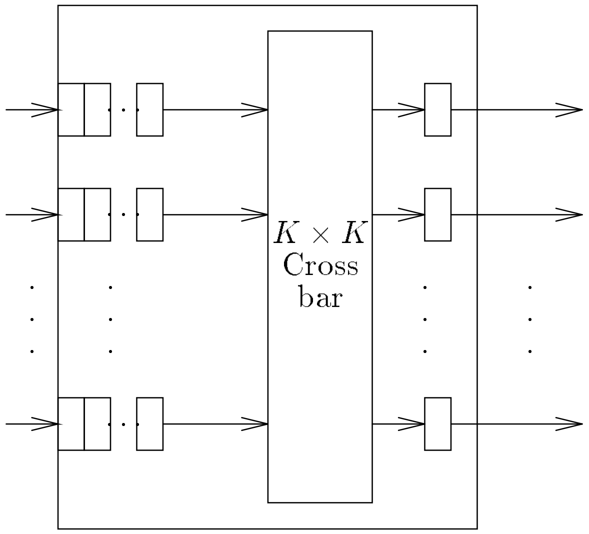


Figure 2.3: A Switching Element (*SE*).

of a latch followed by a register, and are effectively of length two ($\ell_{\text{SE}} = 2$).

When an ACW word reaches the head of an input queue, it is serviced and permission may be granted to transfer all of the words of the packet across the crossbar. There are two reasons why permission to transmit may not be granted. The first is a result of flow control and handshaking in the network (see Section 2.2.4). The second reason is contention, and it occurs when two or more (up to K) ACWs arrive at the head of their respective input queues at the same time and request the same output port. Such requests are said to be *blocked due to contention* for a specific output port of the crossbar. All such requests are placed in a *snapshot* for a given output port, and the output port is said to be *in contention*. A contention resolution policy is used to decide which of the blocked requests should be serviced first. The current policy is as follows:

1. The request at the head of the lowest numbered input queue of the snapshot is transmitted across the crossbar, then that request at the head of the next next lowest numbered input queue is transmitted, etc., until all packets in the snapshot have been transmitted.

2. If during the servicing of a snapshot another request arrives for the output port, it must wait until all requests in the snapshot have been transmitted before it can be transmitted.

3. Any request destined for an output port that is not in contention can pass to the specified output port without being delayed.

This policy provides some means of fairness while avoiding more complicated schemes and the complex hardware needed to implement them.

Switch-slice notation will be used to diagram the state of the switching elements in the network. It is useful to think of the switch in terms of *slices,* where each slice is the two input buffers and the output port being requested (by an ACW at the head of the input FIFO). Packet-word notation is used to represent the contents of the buffers.

$[1,0,0]$	-	W	[1]	W
$[1,0,0]$	-	d	(1)	d
$[1,0,0]$	-	-	()	-

Figure 2.4: A write request traveling through an SE.

When a buffer is empty, "-" is used. The crossbar is represented by "$[i]$," where i is the output port number requested. Square braces indicate that the current packet has been given permission to transmit to the output port, whereas "(i)," indicates that it is either in contention for the output port or finished transmission. The output port number, i, may be omitted when it is not important. The switch slice is labeled "$[S_m, b, i]$" where S_m is the stage number, b is the row number of the SE, and i is the input port number $(0 \leq i < K)$. The label "[]" indicates that the switch slice label is not important to the discussion at hand

As an example of the above notation, the progress of a write request traveling through the first input of an SE and requesting the second output of the SE is shown in Figure 2.4.

Note that an idle slice is represented by "[]$\boxed{\texttt{-}\quad\texttt{-}\quad\texttt{()}\quad\texttt{-}}$," in switch-slice notation.

2.2.4 Handshaking

Handshaking through the network is implemented using the BUSY signal. Recall that there is a latch on each of the output ports of an SE (Section 2.2.1). This latch is usually transparent as long as the buffers that the output leads to are not full (i.e., the input buffer of another SE, or the input ports of the memory units). However, when these buffers become full, the BUSY control signal associated with the network line (that connects the input buffer to the previous stage's output buffer) is asserted. It effectively closes the output latch in an SE. The output port is then said to be *busy*. Hence, the meaning of BUSY is simply, "hold further requests."

There are propagation delays in the transmission of a busy signal. This delay is represented as a two-slot queue between switch slices. An element of the queue can be empty, which is represented as an equals sign, "=", or, an element of the queue can contain an asserted busy signal, which is represented as a left arrow, "←". Figure 2.5 shows the busy signal between two SEs being asserted at at clock tic 0, and lowered at clock tic 1.

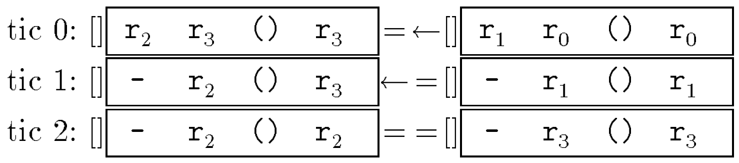

tic 0: [] $\mathtt{r}_2$ $\mathtt{r}_3$ () $\mathtt{r}_3$ = ← [] $\mathtt{r}_1$ $\mathtt{r}_0$ () $\mathtt{r}_0$
tic 1: [] - $\mathtt{r}_2$ () $\mathtt{r}_3$ ← = [] - $\mathtt{r}_1$ () $\mathtt{r}_1$
tic 2: [] - $\mathtt{r}_2$ () $\mathtt{r}_2$ = = [] - $\mathtt{r}_3$ () $\mathtt{r}_3$

Figure 2.5: Transmission of the busy signal.

This example is useful in illustrating some of the behavior of the SE buffering schemes. Note how the busy signal prevented the SE in the first stage from transmitting read request $\mathtt{r}_2$ across the crossbar. Because of the propagation delay, even though at clock tic 1 the input buffer of the SE in the second stage could accept the request $\mathtt{r}_3$, the request $\mathtt{r}_2$ was not serviced until the following clock tic. This latency is a major reason that the input queue of the SE in the second stage cannot be combined with the output latch of the SE in the first stage and treated conceptually as a single FIFO queue.

2.2.5 Memory unit structure

Each of the N memory units (MUs), has identical structure, with an input buffer, memory service area, and output buffer. The input buffer is composed of ℓ_{MU} registers, I_i, $0 \leq i < \ell_{\mathrm{MU}}$. Data enters from the to-network through I_0, and advances on the next clock to I_1, etc., until it reaches $I_{(\ell_{\mathrm{MU}}-1)}$. There is a special register called an *assembly register*, that follows the input buffer. The assembly register is composed of a pair of hardware registers, one to hold the header of a packet and one to hold the write-data portion of the packet for write operations. Once the packet is fully assembled in this input assembly register, it is copied into a service assembly register inside the service area. There the request waits a period of time called the *memory unit service delay,* δ*,* which is currently five clock tics. Requests wait in this service area while the actual memory operation is being performed. The resultant packet enters an assembly register in the output buffer of the memory. From there it is placed on the output FIFO, $O_i, 0 \leq i < \ell_{\mathrm{MU}}$. Unlike the input buffer, the output FIFO is not composed of registers and therefore performs like a true FIFO queue. $\ell_{\mathrm{MU}} = 2$ will be used, except where explicitly stated otherwise. (See Figure 2.6.)

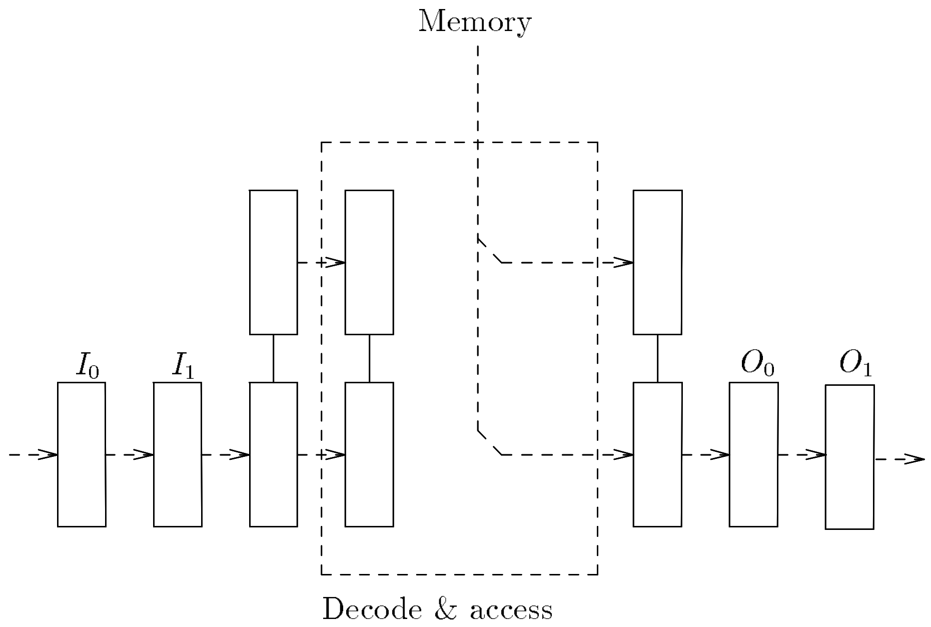


Figure 2.6: The pipeline of a memory unit, $\ell_{\mathrm{MU}} = 2$.

2.2.6 Prefetch operation

Prefetch instructions are issued by the Cedar FORTRAN compiler directly before the vector instructions that require them. Each CE inside a cluster performs prefetch operations independently by making requests to the GIB associated with the CE. The CE will stall until the operation is completed. Each GIB performs a prefetch operation independently from other GIBs, so that accesses are asynchronous across the processors using the subsystem. These details have impact on the performance of the subsystem, as will be shown in Chapter 4.

2.3 Examples

Some examples of the operation of the networks will now be presented. An example of the handshaking in the network has already been presented in Figure 2.5; here two more examples of the subsystem's operation are shown.

2.3.1 An example of a read from memory

In the first example, the to-network traffic for a read from MU$_0$ by CE$_3$ is shown for $N = 8$, $K = 2$, $M = 3$, and $B = 4$. These subsystem parameters correspond to an 8×8 Omega network for both networks, 8 CEs, and 8 MUs. The switch-slice notation along with the simulation clock are shown in Figure 2.7. Notice that it takes each packet word one clock tic to pass through a stage.

To-network: tic 0

$[0,2,1]\boxed{\texttt{-}\quad \mathtt{r}_3^0 \quad \texttt{0} \quad \mathtt{r}_3^0} == []\boxed{\texttt{-}\quad \texttt{-}\quad \texttt{()}\quad \texttt{-}} == []\boxed{\texttt{-}\quad \texttt{-}\quad \texttt{()}\quad \texttt{-}}$

To-network: tic 1

$[]\boxed{\texttt{-}\quad \texttt{-}\quad \texttt{()}\quad \texttt{-}} == [1,1,0]\boxed{\texttt{-}\quad \mathtt{r}_3^0 \quad \texttt{0} \quad \mathtt{r}_3^0} == []\boxed{\texttt{-}\quad \texttt{-}\quad \texttt{()}\quad \texttt{-}}$

To-network: tic 3

$[]\boxed{\texttt{-}\quad \texttt{-}\quad \texttt{()}\quad \texttt{-}} == []\boxed{\texttt{-}\quad \texttt{-}\quad \texttt{()}\quad \texttt{-}} == [2,0,1]\boxed{\texttt{-}\quad \mathtt{r}_3^0 \quad \texttt{0} \quad \mathtt{r}_3^0}$

Figure 2.7: The to-network traffic for a read from MU$_0$ by CE$_3$.

2.3.2 A contention example

An example using a subsystem of dimensions $N = 16, K = 4, M = 2,$ and $B = 4$ is now shown. In this example, CE$_0$, CE$_4$, CE$_8$, and CE$_{12}$, submit a read packet destined to MU$_0$. The traffic in the to-network is shown in Figure 2.8. Notice that the requests collide in the first stage of the network. Request $\mathtt{r}_0^0$ is allowed to transmit across the

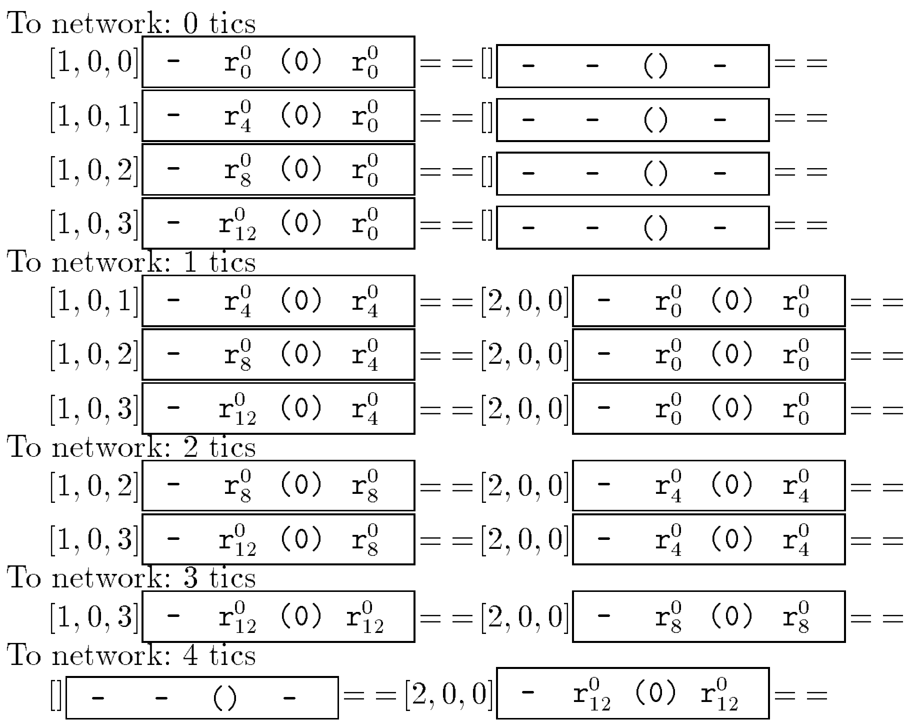

To network: 0 tics
[1,0,0] - r_0^0 (0) r_0^0 == [] - - () - ==
[1,0,1] - r_4^0 (0) r_0^0 == [] - - () - ==
[1,0,2] - r_8^0 (0) r_0^0 == [] - - () - ==
[1,0,3] - r_12^0 (0) r_0^0 == [] - - () - ==
To network: 1 tics
[1,0,1] - r_4^0 (0) r_4^0 == [2,0,0] - r_0^0 (0) r_0^0 ==
[1,0,2] - r_8^0 (0) r_4^0 == [2,0,0] - r_0^0 (0) r_0^0 ==
[1,0,3] - r_12^0 (0) r_4^0 == [2,0,0] - r_0^0 (0) r_0^0 ==
To network: 2 tics
[1,0,2] - r_8^0 (0) r_8^0 == [2,0,0] - r_4^0 (0) r_4^0 ==
[1,0,3] - r_12^0 (0) r_8^0 == [2,0,0] - r_4^0 (0) r_4^0 ==
To network: 3 tics
[1,0,3] - r_12^0 (0) r_12^0 == [2,0,0] - r_8^0 (0) r_8^0 ==
To network: 4 tics
[] - - () - == [2,0,0] - r_12^0 (0) r_12^0 ==

Figure 2.8: The to-network traffic for the second example.

crossbar first via the contention resolution policy explained in Section 2.2.3. During the next clock, r_4^0 is allowed to pass, etc., until all packets waiting have cleared the first stage, at clock tic 4.

The traffic in the from-network is shown in Figure 2.9. Notice that the output port reserved by the header of a packet remains reserved until the data portion of the packet has passed through the network. There is no contention in the from-network, since MU_0 issues requests sequentially and since each packet is destined for a different CE.

From network: 8 tics

$[1,0,0]\boxed{\texttt{-}\quad \mathtt{R}_0^0\quad \texttt{[0]}\quad \mathtt{R}_0^0} == []\boxed{\texttt{-}\quad \texttt{-}\quad \texttt{()}\quad \texttt{-}} ==$

From network: 9 tics

$[1,0,0]\boxed{\texttt{-}\quad \texttt{d}\quad \texttt{(0)}\quad \texttt{d}} == [2,0,0]\boxed{\texttt{-}\quad \mathtt{R}_0^0\quad \texttt{[0]}\quad \mathtt{R}_0^0} ==$

From network: 10 tics

$[]\boxed{\texttt{-}\quad \texttt{-}\quad \texttt{()}\quad \texttt{-}} == [2,0,0]\boxed{\texttt{-}\quad \texttt{d}\quad \texttt{(0)}\quad \texttt{d}} ==$

From network: 12 tics

$[1,0,0]\boxed{\texttt{-}\quad \mathtt{R}_0^4\quad \texttt{[1]}\quad \mathtt{R}_0^4} == []\boxed{\texttt{-}\quad \texttt{-}\quad \texttt{()}\quad \texttt{-}} ==$

From network: 13 tics

$[1,0,0]\boxed{\texttt{-}\quad \texttt{d}\quad \texttt{(1)}\quad \texttt{d}} == [2,1,0]\boxed{\texttt{-}\quad \mathtt{R}_0^4\quad \texttt{[0]}\quad \mathtt{R}_0^4} ==$

From network: 14 tics

$[]\boxed{\texttt{-}\quad \texttt{-}\quad \texttt{()}\quad \texttt{-}} == [2,1,0]\boxed{\texttt{-}\quad \texttt{d}\quad \texttt{(0)}\quad \texttt{d}} ==$

From network: 16 tics

$[1,0,0]\boxed{\texttt{-}\quad \mathtt{R}_0^8\quad \texttt{[2]}\quad \mathtt{R}_0^8} == []\boxed{\texttt{-}\quad \texttt{-}\quad \texttt{()}\quad \texttt{-}} ==$

From network: 17 tics

$[1,0,0]\boxed{\texttt{-}\quad \texttt{d}\quad \texttt{(2)}\quad \texttt{d}} == [2,2,0]\boxed{\texttt{-}\quad \mathtt{R}_0^8\quad \texttt{[0]}\quad \mathtt{R}_0^8} ==$

From network: 18 tics

$[2,2,0]\boxed{\texttt{-}\quad \texttt{d}\quad \texttt{(0)}\quad \texttt{d}} == []\boxed{\texttt{-}\quad \texttt{-}\quad \texttt{()}\quad \texttt{-}} ==$

From network: 20 tics

$[1,0,0]\boxed{\texttt{-}\quad \mathtt{R}_0^{12}\quad \texttt{[3]}\quad \mathtt{R}_0^{12}} == []\boxed{\texttt{-}\quad \texttt{-}\quad \texttt{()}\quad \texttt{-}} ==$

From network: 21 tics

$[1,0,0]\boxed{\texttt{-}\quad \texttt{d}\quad \texttt{(3)}\quad \texttt{d}} == [2,3,0]\boxed{\texttt{-}\quad \mathtt{R}_0^{12}\quad \texttt{[0]}\quad \mathtt{R}_0^{12}} ==$

From network: 22 tics

$[]\boxed{\texttt{-}\quad \texttt{-}\quad \texttt{()}\quad \texttt{-}} == [2,3,0]\boxed{\texttt{-}\quad \texttt{d}\quad \texttt{(0)}\quad \texttt{d}} ==$

Figure 2.9: The from-network traffic for the second example.

CHAPTER 3

THE SIMULATOR

How the simulator was constructed to match the performance of the global memory subsystem can now be described. This description reviews some of the simulator design constraints and should be useful to those who wish to write simulations of computer systems. Those who are not interested may wish to skip to the next chapter.

3.1 Simulator Overview

The simulator is composed of four modules. These modules are:

Pattern generator. This module generates a *scenario file* for the simulator which contains a list of packet words for input to the simulator. The output file is generated from a user-supplied subroutine written in C that simulates the CEs/GIBs.

Simulation main-loop. This module initializes the data structures, performs I/O on the scenario file and the statistics files, and executes the simulation components.

Network simulation. This module evaluates the state of the switching elements for each stage in a network. It is called twice by each iteration of the simulation main-loop, once for the to-network and once for the from-network.

Memory unit simulation. This module evaluates the state of the memory units in the global memory, taking the output of the to-network and producing a vector of words to be used as input to the from-network. Hence, it is called in-between invocations of the network simulation in the simulation main-loop.

Additionally, there is support for infinite-length buffers at the inputs of the to-network to buffer the words which cannot enter the subsystem because of full SE input queues. Statistics are collected via statistics gathering code dispersed throughout the simulator.

3.2 Theory of Operation

The theory of operation of the simulator is now described in depth by explaining the operation of each of the component modules.

3.2.1 The pattern generator

The pattern generator takes as input a user-supplied function written in C and produces as output a scenario file listing all the network words, in groups of N, to be submitted to the simulator. This user-supplied function, called `distrib`, generates a packet type and destination when it is supplied with the processor number and the simulation clock. The `distrib` function takes four arguments:

1. The processor number (i.e., the network input port number)
2. The simulation clock
3. The packet type that the `distrib` function returns (passed as a call-by-reference argument)
4. The destination of the packet (i.e., memory unit number) that the function returns (call-by-reference).

Finally, the pattern generator is supplied with a count of how many packets each processor will send. The pattern generator then calls the `distrib` function for each CE during each clock tic, until all packets have been generated. After the packet has been generated, it is stored in a linked list for each CE. After the lists have been built, the scenario file is generated from the lists' contents.

3.2.2 The simulation main-loop

The simulation main-loop takes as its input a scenario file from the pattern generator and executes the simulator based on the contents of this file. After the end of the simulation run, it produces a list of statistics collected during the simulation. Most simulations include a main-loop, the body of which corresponds to a clock tic. The main-loop for the simulator is shown in Figure 3.1.

Each block in the main-loop of Figure 3.1 represents a function that takes vectors of network words as input, where each one of these vectors is an N element one-dimensional array of type `Word`. This data type is a union of two structures; one is

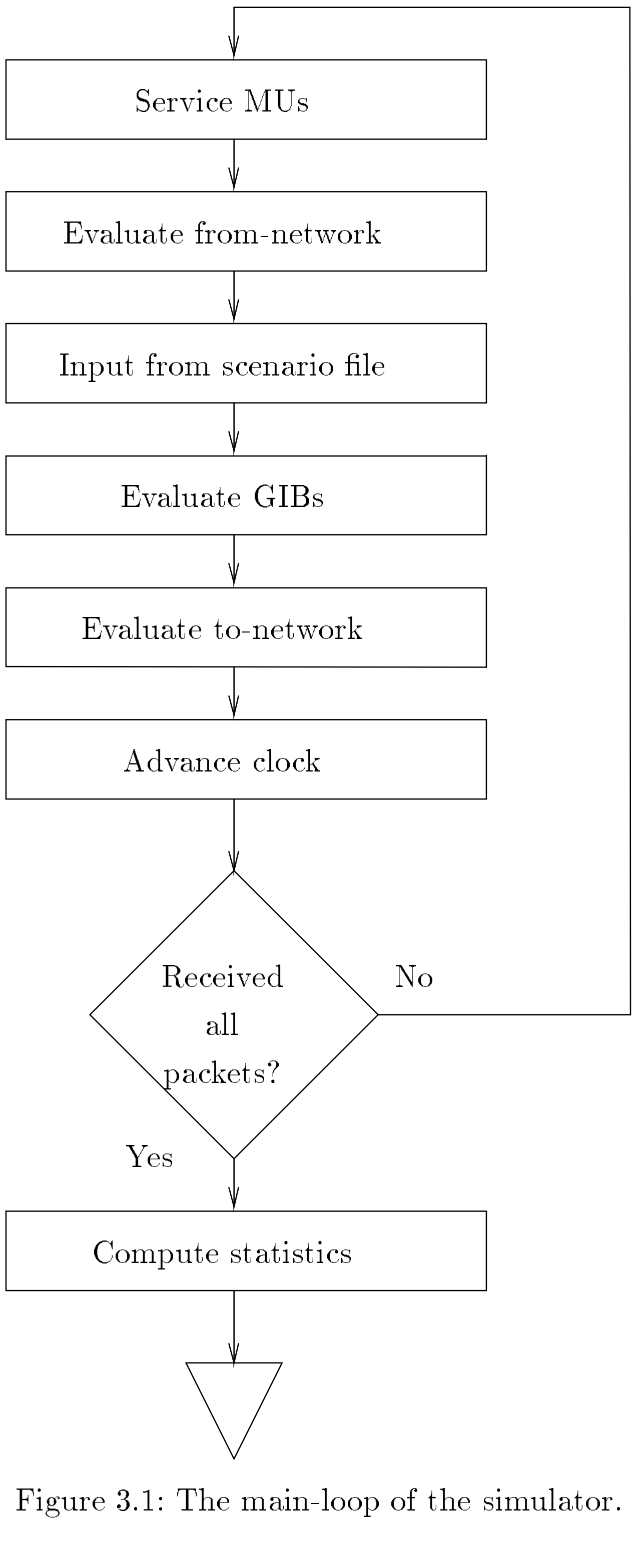


Figure 3.1: The main-loop of the simulator.

the access control word structure and one is the data structure. The data type was implemented in this way because a network word can be interpreted as either a data word or an access control word. (See Figure 3.2.)

```
typedef union acw_data {
    struct data_word {
        ⋮
    } data;
    struct access_word {
        ⋮
    } acw;
  } Word;
```

Figure 3.2: The definition of the data type `Word`.

The first function called in Figure 3.1 services the memory units based on the output of the to-network. The `network_advance` function is called next. It evaluates the state of a network; this first time it is called is to evaluate the state of the from-network based on the output of the memory units. After this, a vector of network words is taken from the scenario file, representing the input from the CEs/GIBs. This is applied to a function which simulates the global interface boards. This function buffers pending requests that cannot enter the network due to full network buffers (i.e., received BUSY signals, see Section 2.2.4); it also initiates the collection of statistics for the words as they enter the to-network. Finally, the function `network_advance` is called again, this time to evaluate the status of the to-network based on the outputs of the GIBs. The output of this second call to `network_advance` is used at the beginning of the next iteration of the main-loop as the inputs to the memory units. Note how this ordering of event evaluation allows requests to enter the to-network during the first clock tic, i.e., when `clock` = 0.

After the number of packets specified in the scenario file has gone through the entire network, the program exits from the main-loop and statistics gathered during the simulation are output to several cumulative files.

3.2.3 The network simulation

The network is simulated by the function `network_advance`. Before explaining how this function implements the network described in the previous chapter, the data structure `se_state` must be introduced. `Se_state` is a four-dimensional array used to hold the state information of the two networks (the definition is shown in Figure 3.3). The

```
SE se_state[2][M+1][B][K];
```

Figure 3.3: The definition of `se_state`.

first index of `se_state` specifies the network in question (0 = from-network, 1 = to-network), the second index specifies the stage of a network, the third specifies a particular SE in the stage, and the final index specifies an input port of the SE.

The definition of the data type `SE` is shown in Figure 3.4. The type `SE` is implemented as a structure of integers, short integers, and several items of type `Word`, representing the various registers and latches of switch-slice notation. The structure elements `ilatch`, and `ff` implement the latch and register (or flip-flop) of an input buffer of an SE. The short integer `ff_busy` is used as a flag and set when `ff` contains a valid data item, essentially used to implement the semantics of a hardware register. The element

```
typedef struct se_state_st
    Word ilatch;
    Word ff;
    Word olatch;
    int olatch_clock;
    short busy;
    short oldbusy;
    short ff_busy;
 SE;
```

Figure 3.4: The definition of the data type `SE`.

`olatch` is used to implement the output latch of an output port of an SE. The integer `olatch_clock` is used to simulate the behavior of a latch, holding the last time a word was stored in `olatch`. The elements `busy` and `oldbusy` are used to hold the busy signal and simulate the signal propagation delay. There are other elements of the data structure used for bookkeeping and statistics gathering that are not shown in the figure.

The essential body of the `network-advance` function is shown in Figure 3.5. The

```
for (stage = M; stage > 0; --stage)
    for (se = 0; se < B; se++) {
    for (inp = 0; inp < K; inp++) {
        ninp = se*B+ninp;
        se_state[net][stage-1][shufse(ninp)][shufinp(ninp)].busy =
          se_slice(...);
    }
```

Figure 3.5: The body of `network_advance`.

functions `shufse(ninp)` and `shufinp(ninp)` implement the Shuffle function, returning the destination SE index and SE input port number index, respectively. The function `se_slice` (arguments not shown) evaluates the state of the switch slice and returns the busy signal to be transmitted to the previous stage. Note how the network state is evaluated from the last stage through to the first stage. This is opposite to the direction that data flows in the system. Evaluating the network in this direction simplifies the problem of keeping the state information consistent with the simulation clock.

The semantics of the switch slice introduced in the previous chapter are implemented using the `se_slice` function. Figure 3.6 shows a diagram of this function. A network word is taken from the output of the previous stage of the network and placed in the `ilatch`, where it waits to be shifted to the head of the queue. This shifting is performed in the second block from the top of Figure 3.6. If after the shifting, the word in the top of the queue (i.e., in the variable "`ff`") is an access control word, it is used to determine which output port to transmit the packet to. If the port is being used by another packet, the word becomes part of a snapshot and must wait its turn to transmit across the crossbar. If `ff` is not an ACW, it is transmitted through the crossbar to the output port specified by an ACW that occupied the `ff` some time before this word. No matter what type of word `ff` is, if the BUSY signal is asserted, this transmission is not performed and the word waits until the the next clock to try again. (See Section 2.2.3.)

After the evaluation of each switch slice for each SE in each stage of the network, `network_advance` returns a vector composed of the words that are in each `olatch` of each SE of the last stage of the network. If the network being evaluated is the to-network, this vector is used as input to the memory unit simulation.

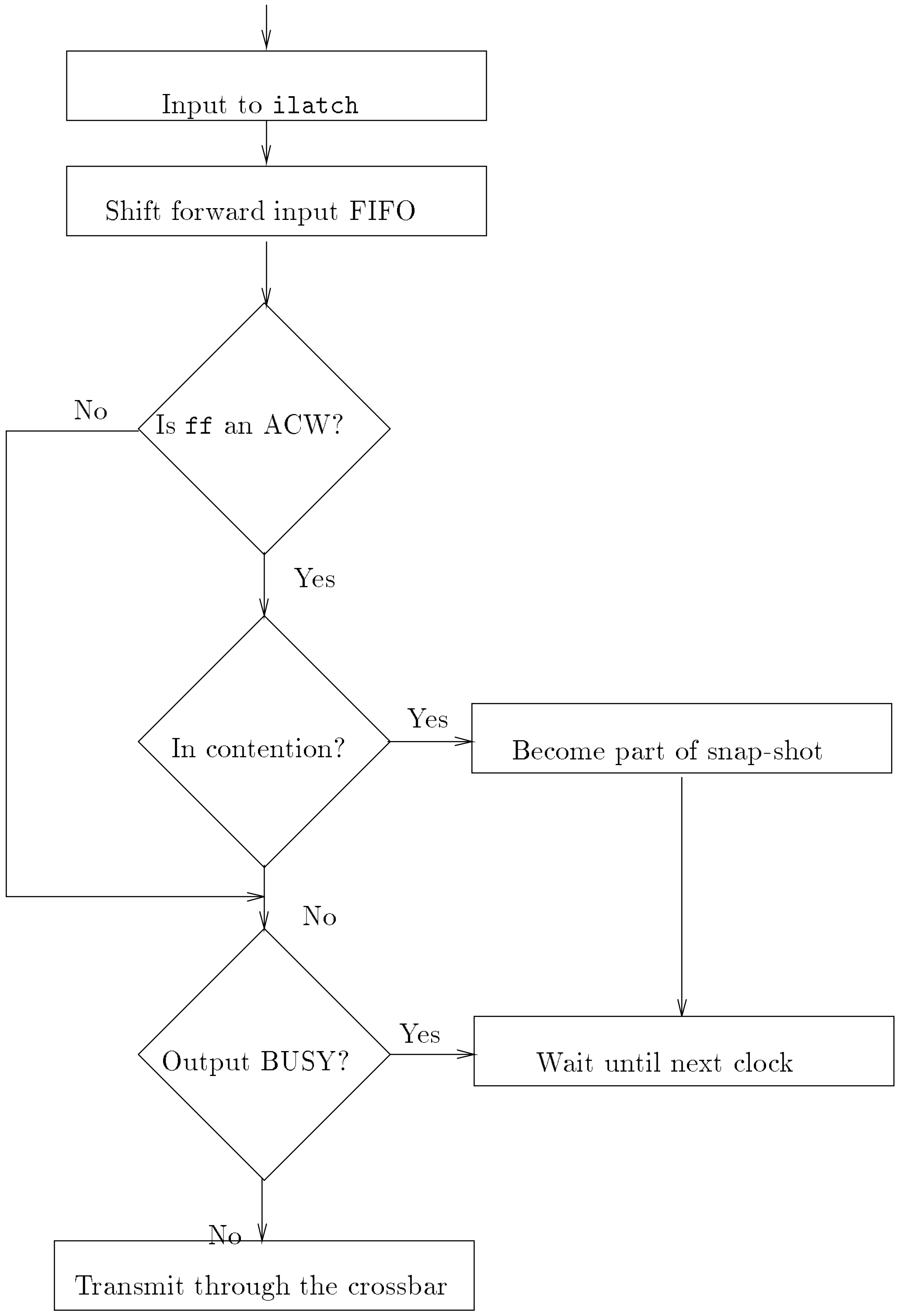


Figure 3.6: A diagram of `se_slice`.

3.2.4 The memory unit simulation

How the memory units are simulated is now described. Figure 3.7, Figure 3.8, and Figure 3.9 diagram the operation of the memory unit simulation. Beginning with Figure 3.7, the first task the simulation performs is to load the input assembly register (AR) with the word from the head of the memory unit's input queue. After this has been done, the input queue can be shifted forward. Then, a word can be taken from the output of the to-network and placed at the tail of the input queue. If the queue is full, however, the word remains on the output of the to-network and the BUSY signal is asserted to flag the full-queue condition.

Attention now shifts to the state of the service area of the memory unit. If there is a request currently being serviced, the delay time (initially set to δ) is decremented and the request continues to wait. In Figure 3.8, if the request in the input assembly register is not serviced and the service area is empty, the delay time is set to δ and servicing begins for that request. If the servicing is finished, the output packet is shipped out to the output assembly register, if it is not already occupied. Otherwise, the request in the service area will wait a time greater than δ, until the output assembly register is free. Finally, in Figure 3.9, the output queue is advanced and the head of the queue is made available to the from-network. The from-network may then take the data word at the head of the queue or leave it, depending on the availability of space in the input queues of the SEs in the first stage of the network.

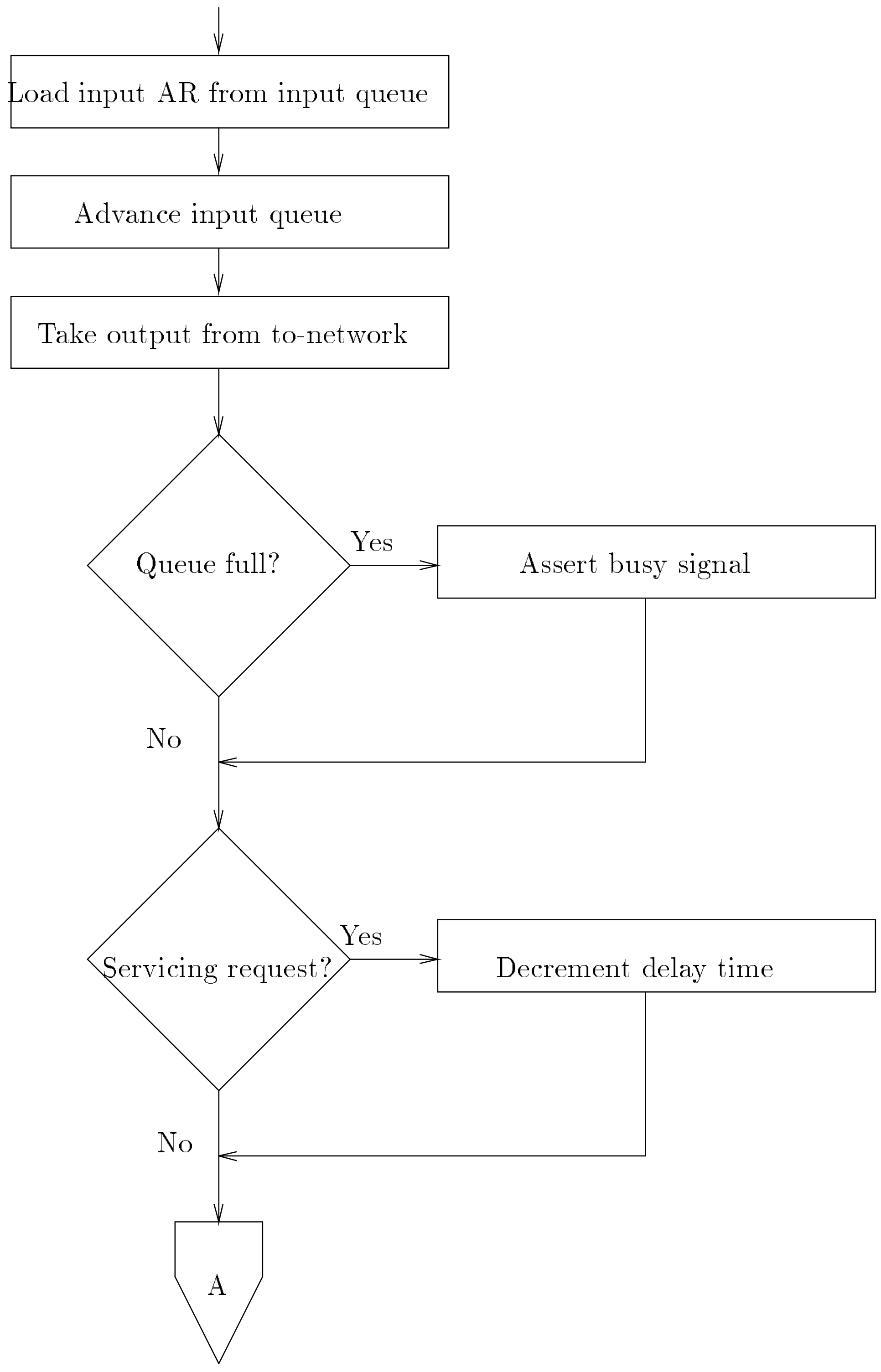


Figure 3.7: A diagram of the memory unit simulation, part 1.

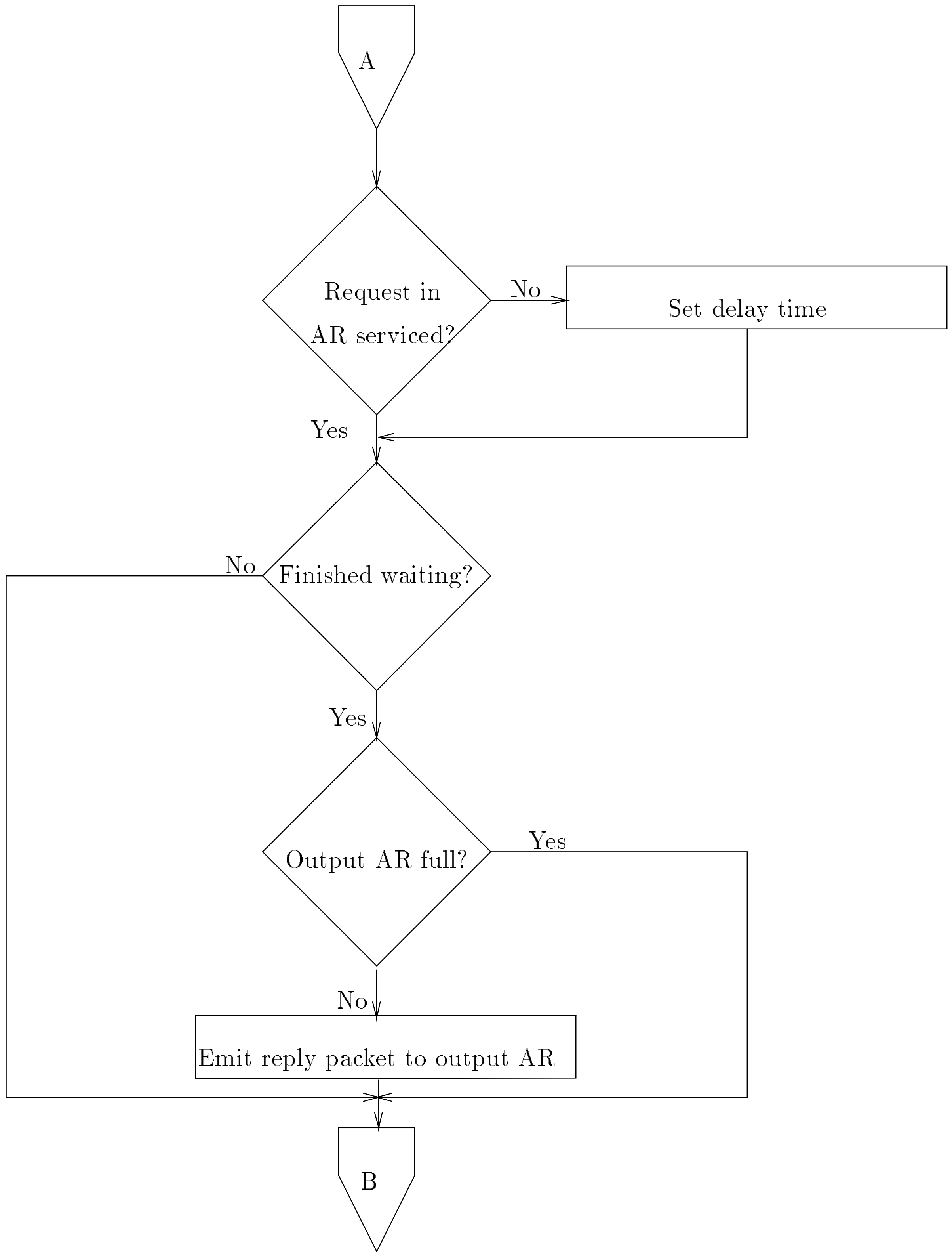


Figure 3.8: A diagram of the memory unit simulation, part 2.

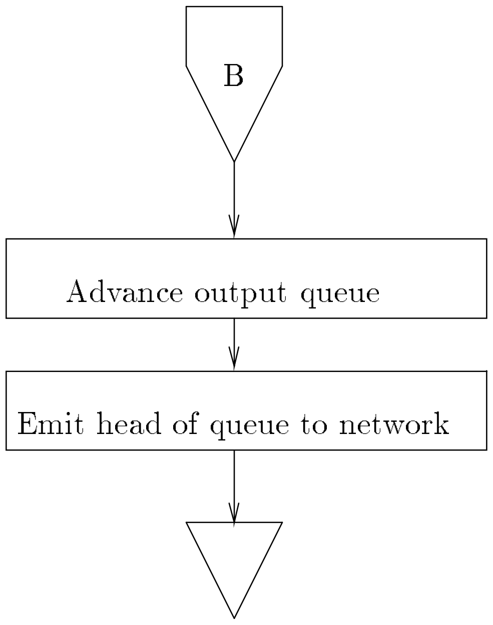


Figure 3.9: A diagram of the memory unit simulation, part 3.

This completes the discussion of the simulator. In the next chapter the results of experiments with the simulator will be described.

CHAPTER 4

WORST-CASE ANALYSIS

In this chapter, the worst-case performance bottlenecks of the global memory subsystem is identified using simulation results. The worst-case is of interest since it can be interpreted as the guaranteed performance of the subsystem. The investigation begins by defining the performance measures, defining the test subsystem configurations, and deriving the worst-case performance scenario. An expression for our main performance metric in the worst-case is then constructed, which could possibly be used as a lower bound on prefetch performance by a compiler optimization pass. Finally, the worst-case scenario is used to identify the performance bottleneck in the subsystem and investigate means of tuning the subsystem to improve the worst-case performance.

4.1 Definitions

Some definitions are now presented that will be used to describe the results in this chapter and the next chapter.

4.1.1 Performance measures

If it is assumed at a given time unit that CE_i, $0 \leq i < N$, fetches data from MU_{α_i}, $0 \leq \alpha_i < N$, this situation may be written as a permutation, $p = (\alpha_0\ \alpha_1\ \alpha_2 \ldots \alpha_{N-1})$, of N integers, $\alpha_i, 0 \leq \alpha_i < N$. Define f_c, the *fraction of contention,* as an index of the contention a permutation would cause in the to-network. If $N_{\mathrm{int}}(\alpha_i, p)$ is defined to be the number of times the integer α_i occurs in the permutation p, then:

$$f_c = \frac{1}{N} \sum_{i=0}^{N-1} N_{\mathrm{int}}(\alpha_i, p).$$

Note that f_c is not a measure of the contention in an Omega network, only a statement about the input to the network. This is due to the buffering in the network (i.e., the storing of packets by the input queues of the SEs and the queuing structure of the MU's).

The entire prefetch of vector data by all N CEs is referred to as a *prefetch operation*, and the amount of time it takes from when the first CE makes its request to when the last request finishes traversing the from-network is referred to as the *prefetch delay, T*. The main performance metric that will be used is *inverse bandwidth,* BW^{-1}, defined as the prefetch delay divided by the length of the vector (L). Hence, the formula $\mathrm{BW}^{-1}(L) = T/L$, can be used to convert between values of inverse bandwidth and prefetch delay. The inverse bandwidth is in units of (clocktics) $\times$ (vector element)$^{-1}$. In general, a lower value of inverse bandwidth is preferred. The value of inverse bandwidth can also be interpreted as the delay of the slowest stage in the subsystem, when the subsystem is viewed as a pipeline, for $L \geq N$. Hence, inverse bandwidth can be used to identify the bottleneck of the subsystem's performance.

The average time spent by all packets in a stage, S_i, of either the to-network or the from-network will be called the *latency* for that stage, $t(S_i)$. An input port of an SE can be thought to be in one of four general states: the MOVE state, when a request can move freely to the requested output port; the BUSY state, when a request cannot move due to a received BUSY signal at the output port; the CONT state, when a request cannot move due to a contention condition for the requested output port; and, the BC state, when a request cannot move due to both a received BUSY signal and contention for the requested output port. The fraction of time a typical request spends in each of these states will be measured. These fractions are labeled, F_{MOVE}, F_{BUSY}, F_{CONT}, and F_{BC}, respectively. Collectively, these fractions are called the *latency statistics*, since they are useful in determining the cause of unusual packet latencies for each stage.

4.1.2 Test subsystems

Three subsystem dimensions will be chosen for examination to try to keep the conclusions general. The first subsystem contains eight CEs, eight MUs, and the two networks are constructed using 2×2 SEs. The second subsystem contains 16 CEs/MUs, and the networks are constructed using 4×4 SEs. Finally, a large subsystem composed of 64 CEs/MUs, with the networks constructed using 8×8 SEs will be examined. Table 4.1 summarizes the characteristics of the subsystems chosen.

Table 4.1: The test subsystem dimensions

Subsystem title $(N \times N)$	*Number of* CE's, MU's, *network inputs* (N)	SE*inputs* (K)	*Stages* (M)	SE's *per stage* (B)
8×8	8	2	3	4
16×16	16	4	2	4
64×64	64	8	2	8

4.1.3 Worst-case scenario

The *L-length vector simultaneous prefetch* is defined as the case when all the GIBs simultaneously fetch a vector, each vector of length L. When the vector they are fetching happens to reside in the same sequence of memory modules, this scenario will be referred to as a prefetch of the *same vector*, or simply SV. Figure 4.1 shows the permutations for a simultaneous prefetch of the same vector for the 8×8 subsystem, where the vector begins in MU_0, and its length is $L = 10$. This is obviously a worst-case for vector prefetch performance, since each permutation in Figure 4.1 has a unity fraction of contention. There is one case even worse than this, when all the elements of a vector reside in the same memory unit. However, this implies the vector elements were separated N-locations apart in when they were stored in physical memory, which is an avoidable situation. Therefore, the SV scenario shall be used for the worst-case. Since the contention appears in the to-network, the analysis will be simplified by neglecting the performance of the from-network, where possible.

$$
\begin{aligned}
p_0 &= (0\ 0\ 0\ 0\ 0\ 0\ 0\ 0),\ f_c = 1\\
p_1 &= (1\ 1\ 1\ 1\ 1\ 1\ 1\ 1),\ f_c = 1\\
p_2 &= (2\ 2\ 2\ 2\ 2\ 2\ 2\ 2),\ f_c = 1\\
p_3 &= (3\ 3\ 3\ 3\ 3\ 3\ 3\ 3),\ f_c = 1\\
p_4 &= (4\ 4\ 4\ 4\ 4\ 4\ 4\ 4),\ f_c = 1\\
p_5 &= (5\ 5\ 5\ 5\ 5\ 5\ 5\ 5),\ f_c = 1\\
p_6 &= (6\ 6\ 6\ 6\ 6\ 6\ 6\ 6),\ f_c = 1\\
p_7 &= (7\ 7\ 7\ 7\ 7\ 7\ 7\ 7),\ f_c = 1\\
p_8 &= (0\ 0\ 0\ 0\ 0\ 0\ 0\ 0),\ f_c = 1\\
p_9 &= (1\ 1\ 1\ 1\ 1\ 1\ 1\ 1),\ f_c = 1
\end{aligned}
$$

Figure 4.1: The permutations and their fractions of contention for simultaneous prefetch, $L = 10$, the 8×8, subsystem.

4.1.4 Configurations

The effect of altering several subsystem parameters will be observed. These parameters are the memory unit service delay, δ, the length of the input queues of the SEs, ℓ_{SE}, and the length of the input buffers and output FIFOs of the MUs, ℓ_{MU}. Table 4.2 lists the different combinations of these parameters associated with a given parameter configuration name.

Table 4.2: Subsystem configurations

Configuration Name	*Memory unit service delay (δ)*	*SE input queue length (ℓ_{SE})*	*MU buffer/ FIFO length (ℓ_{MU})*
Normal	5 tics	2	2
Fmem	1 tic	2	∞
MD2	2 tics	2	2
MD10	10 tics	2	2
SE3I	5 tics	3	2
SE8I	5 tics	8	2
SE8IFmem	1 tic	8	∞

Once the effect of the memory units on the overall performance of the subsystem has been investigated, this will be removed by replacing the memory units with single-cycle, or *fast* memory units. Figure 4.2 shows a diagram of the structure of these fast memory

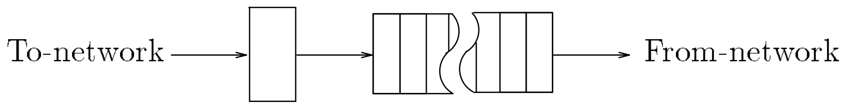


Figure 4.2: The structure of a fast memory unit.

units. Requests are processed by a server as they exit the to-network, in a single cycle. The resultant reply packets are placed on an infinite-length queue to isolate the server from the handshaking signals received from the from-network. Subsystems built with the fast memory units will be referred to as the "Fmem" configuration.

4.1.5 Definition of experiments

Each experiment will be described using a three-tuple, the first item of which is the subsystem used, the second is the configuration used, and the third is the scenario. For example, "(8×8, MD2, SV)," is an experiment using the simultaneous prefetch of the same vector scenario, using the 8×8 subsystem, with the memory delay, $\delta = 2$ (i.e., the MD2 configuration).

It is re-emphasized that the vector elements are stored sequentially across memory modules. Simulation has shown that the memory unit where a vector begins is not important to the characteristic performance of the SV scenario, so the effects of varying the starting location of a vector will be ignored and it will be assumed that vectors begin in MU_0.

Each experiment will involve many runs of the simulator, for increasing vector lengths, until the values of the performance measures of the subsystem are of predictable behavior. Hence, vector lengths up to $L = 80$, for the 8×8 subsystem, up to $L = 100$, for the 16×16 subsystem, and up to $L = 200$, for the 64×64 subsystem will typically be investigated.

4.2 Analysis of Memory Units

Presented here is some analysis of the performance of the memory units that will be useful in the interpretation of the simulation results presented later in this chapter. Two questions about the memory units will be answered: what is the inverse bandwidth for various values of δ, and what is the delay for a request traveling through the memory units. The inverse bandwidth for a memory unit is its slowest stage. Normally, then, $\mathrm{BW}_{\mathrm{MU}}^{-1} = \delta$. However, it was mentioned above that the MD2 configuration, when $\delta = 2$, will be investigated. In this situation, however, the loading of the input assembly register, processing in the service area, and unloading to the output assembly register of the reply packet take four clock tics. Hence, in general, the inverse bandwidth for the memory units is $\mathrm{BW}_{\mathrm{MU}}^{-1} = \delta, \delta > 2$, and $\mathrm{BW}_{\mathrm{MU}}^{-1} = 4, \delta \leq 2$.

In Section 2.2.5 of Chapter 2, the structure of the memory unit was discussed. From this structure, a general expression for the latency of a full memory unit, t_{MU}, can be derived:

$$t_{\mathrm{MU}} = \begin{cases} (\ell_{\mathrm{MU}} + 2) * \delta - 1, & \text{if } \delta \geq 5; \\ \delta + 4\ell_{\mathrm{MU}} + 3, & \text{otherwise.} \end{cases}$$

4.3 Experimental Results

The results of some experiments performed on the three subsystems are now discussed. A model of the performance of the subsystem in the worst-case using the simulator is derived. Experimental results are then used to investigate the effectiveness of improvements to the memory units and switching elements.

4.3.1 Worst-case performance

Figure 4.3 shows the average latency for packets in the to-network for the experiment (8×8, Normal, SV). Figure 4.4 shows the latency statistics for this experiment. In these figures, stage one's behavior is shown with a solid line, stage two's behavior with a dashed line, and stage three's behavior with a dotted line.

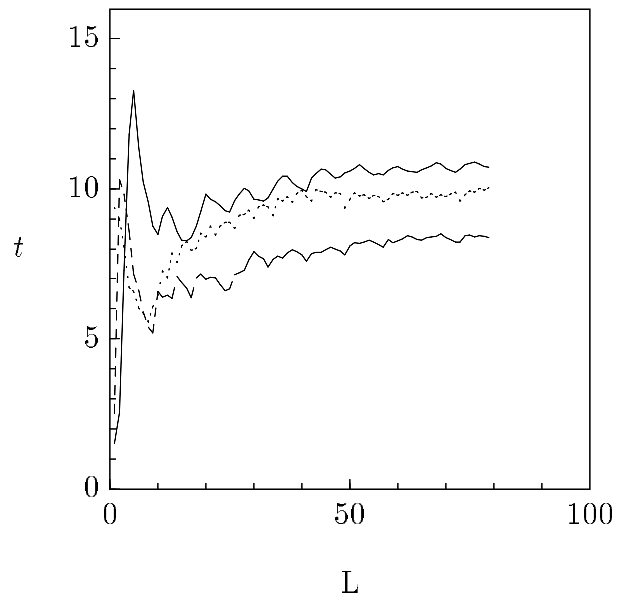


Figure 4.3: Latency for to-network, (8×8, Normal, SV).

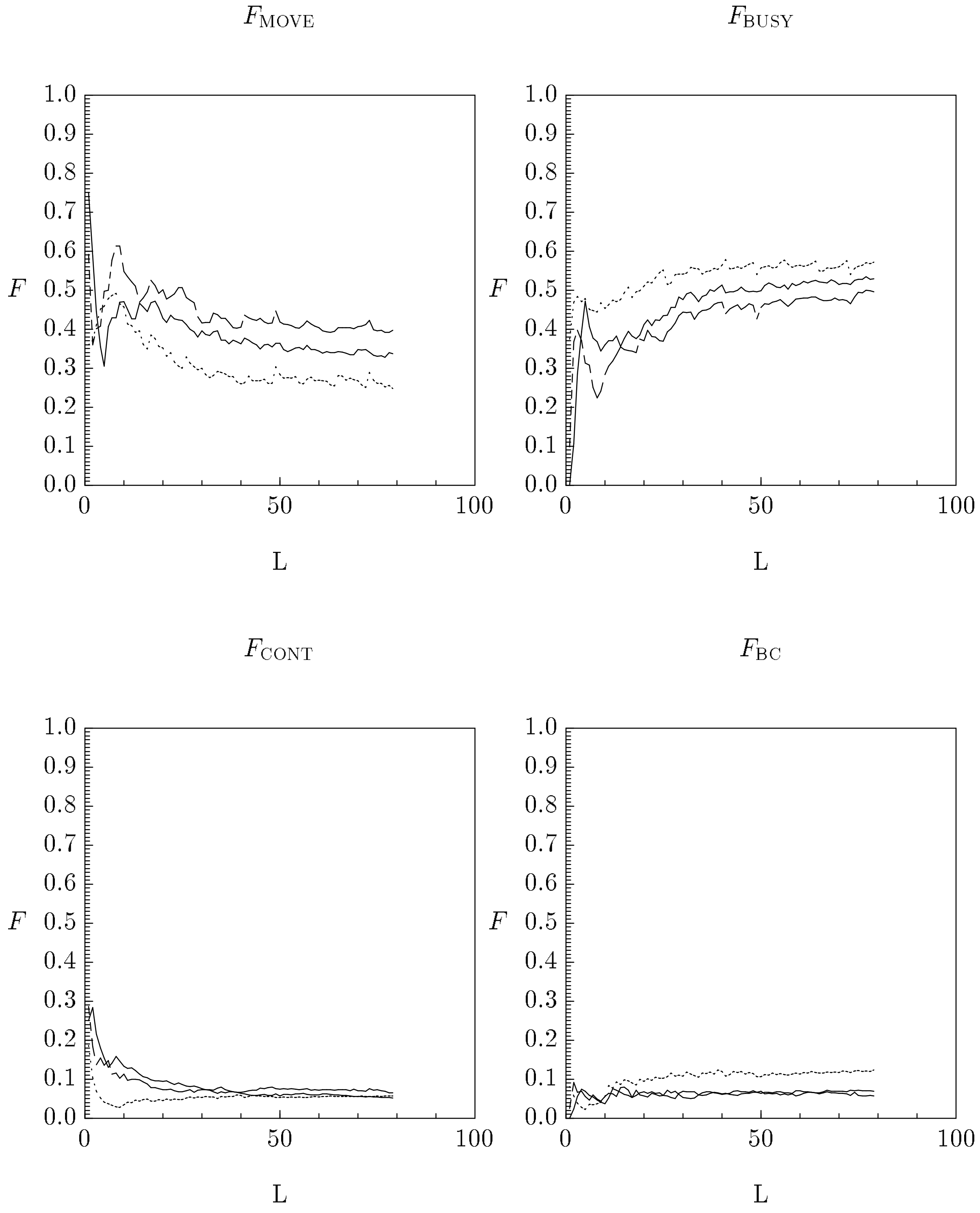


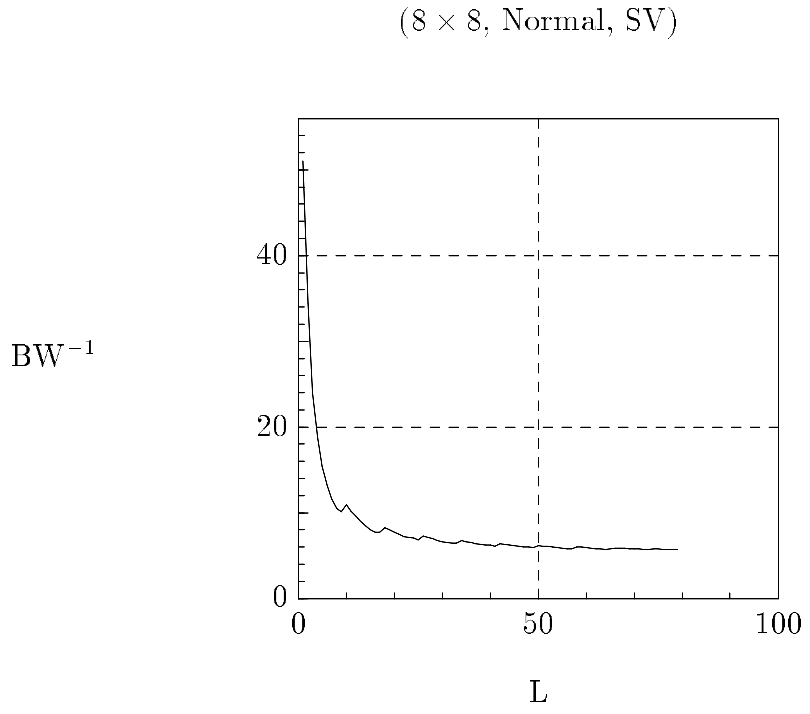
Figure 4.4: The latency statistics for the to-network for the experiment (8×8, Normal, SV).

The latency for the stages of the to-network (Figure 4.3) reveals that stage two is the fastest stage. The latency statistics (Figure 4.4) explains why: stage two is the most mobile stage (See F_{MOVE}) and is busy the most (See F_{BUSY}). Another interesting feature is the fraction of time a stage is in contention alone (i.e., F_{CONT}) is approximately the same as the fraction of time a stage is in busy-contention (F_{BC}), for long vectors, and F_{CONT} and F_{BC} do not vary significantly between stages. Also, the fraction of time a request cannot move due to busy alone (F_{BUSY}) is approximately four times that due to contention or busy-contention. Therefore, contention is not the major cause of the delays in the stages. Instead, the cause is the BUSY signal, which is caused by the memory units. Attention now turns to inverse bandwidth to find the slowest stage of the pipeline.

Figure 4.5 shows the inverse bandwidth for the experiments (8×8, Normal, SV), and (64×64, Normal, SV). Note that the inverse bandwidth in the worst-case in general is very high for small vector lengths, L. For larger N however, the inverse bandwidth then falls, seeking an asymptote. This curve can be fitted reasonably well using a hyperbolic formula for inverse bandwidth,

$$\mathrm{BW}^{-1}(L) = \frac{\mathrm{BW}^{-1}(1) - \mathrm{BW}^{-1}(\infty)}{L} + \mathrm{BW}^{-1}(\infty),$$

where $\mathrm{BW}^{-1}(1)$ is the inverse bandwidth for a single-element vector prefetch, and $\mathrm{BW}^{-1}(\infty)$ is the inverse bandwidth for a very long vector prefetch. $\mathrm{BW}^{-1}(1) = 51.0$, and 329, for the ($8 \times 8$, Normal, SV), and ($64 \times 64$, Normal, SV) experiments, respectively. The asymptotic inverse bandwidth, $\mathrm{BW}^{-1}(\infty) \approx 5.0$, for the ($8 \times 8$, Normal, SV)



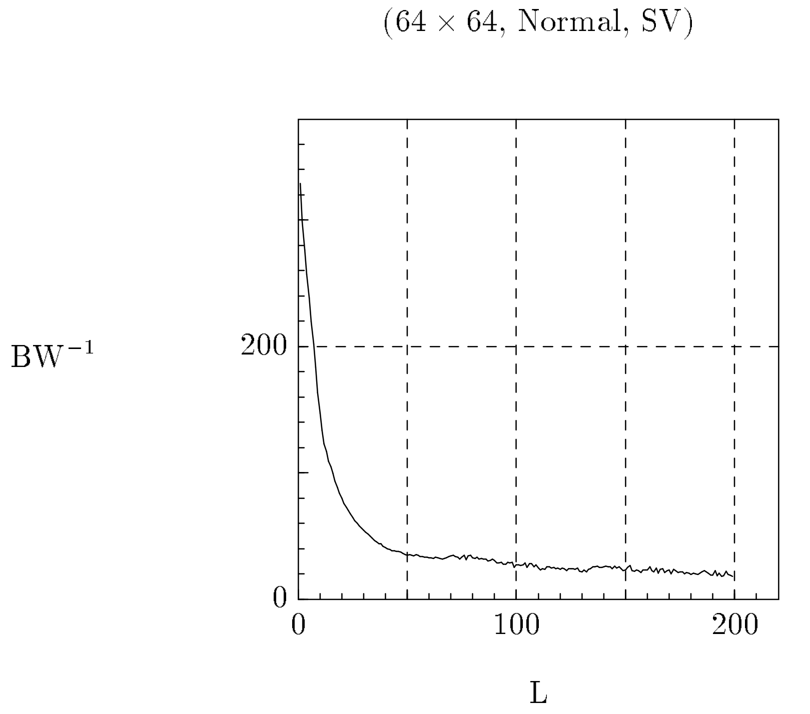


Figure 4.5: Inverse bandwidth for experiments (8 $\times$ 8, Normal, SV) [*top graph*], and (64 $\times$ 64, Normal, SV) [*bottom graph*].

experiment. This is the same as the value for $\mathrm{BW}_{\mathrm{MU}}^{-1}$, further indicating that the reasons for the values of F_{BUSY} in Figure 4.4 were the memory units. To verify this, we performed the experiments (8×8, MD10, SV). The results are shown in Figure 4.6. Observe that $\mathrm{BW}^{-1}(\infty) \approx 10.0$, justifying the statement, $\mathrm{BW}^{-1}(\infty) = \mathrm{BW}_{\mathrm{MU}}^{-1}$, for the

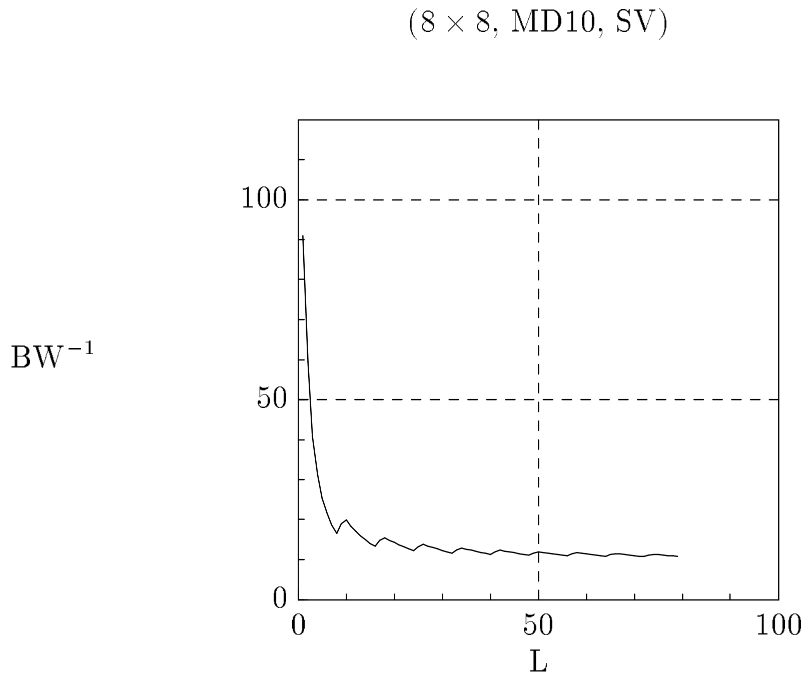


Figure 4.6: Inverse bandwidth for experiment (8×8, MD10, SV).

8×8 subsystem. Equivalent experiments performed for the 16×16 subsystem indicate $\mathrm{BW}^{-1}(\infty) = \mathrm{BW}_{\mathrm{MU}}^{-1}$, for this subsystem also.

For the (64×64, Normal, SV) experiment, $\mathrm{BW}^{-1} = 20.1$ (See Figure 4.5). The latency for the to-network for this experiment for the range of prefetch vector lengths, $180 \leq L < 200$ is shown in Figure 4.7. The latency statistics are shown in Figure 4.8. The latency for the first stage is much higher than that for the second stage. Also, the fraction of time a packet is mobile (F_{MOVE}) is approximately 20% less for the

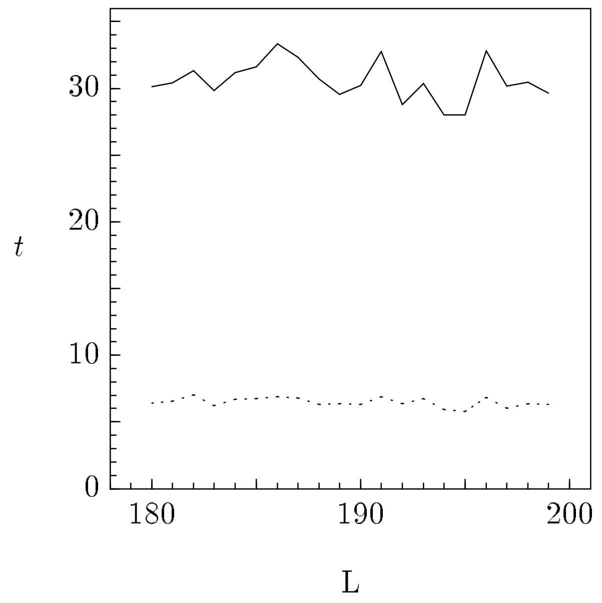


Figure 4.7: Latency for to-network for the experiment (64×64, Normal, SV).

first stage than the same fraction for the second stage. The fraction of time spent in contention for the first stage is approximately 0.20. Also, the fraction of time spent busy is approximately equal for both stages, indicating that the BUSY signal was not the reason for the difference between the latencies of the two stages. Therefore, the reason for the high latency of the first stage is largely due to contention. The value of $\mathrm{BW}^{-1}(\infty)$ equals the inverse bandwidth measured for the first stage of the to-network. It can therefore be concluded that the bottleneck for this experiment is the contention in the first stage of the to-network.

The bottlenecks for the subsystem for long vector lengths have been found. Attention now turns to understanding the subsystem's behavior for short vector lengths. To do this, the observed values of $\mathrm{BW}^{-1}(1)$ will be explained using a *completion time*

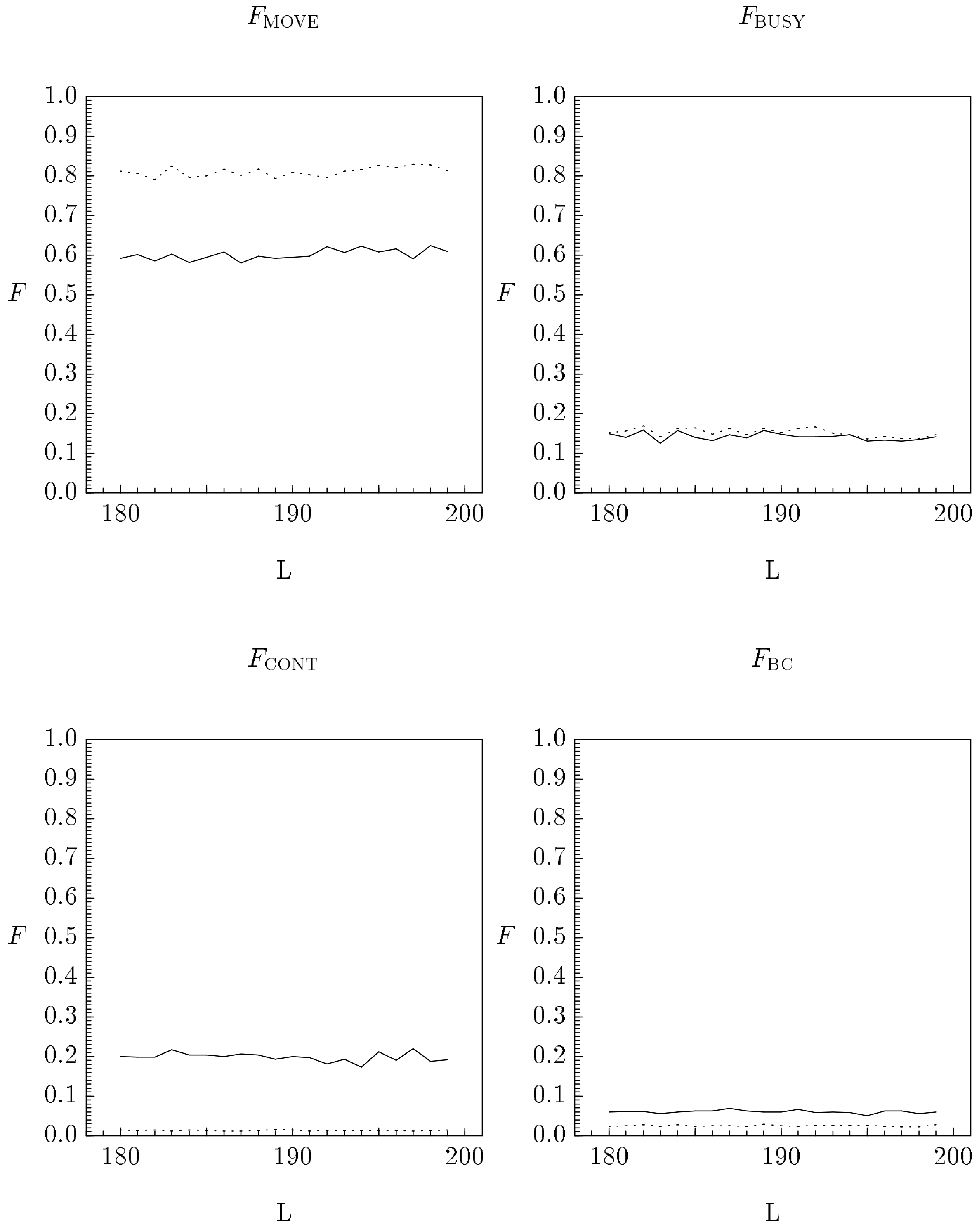


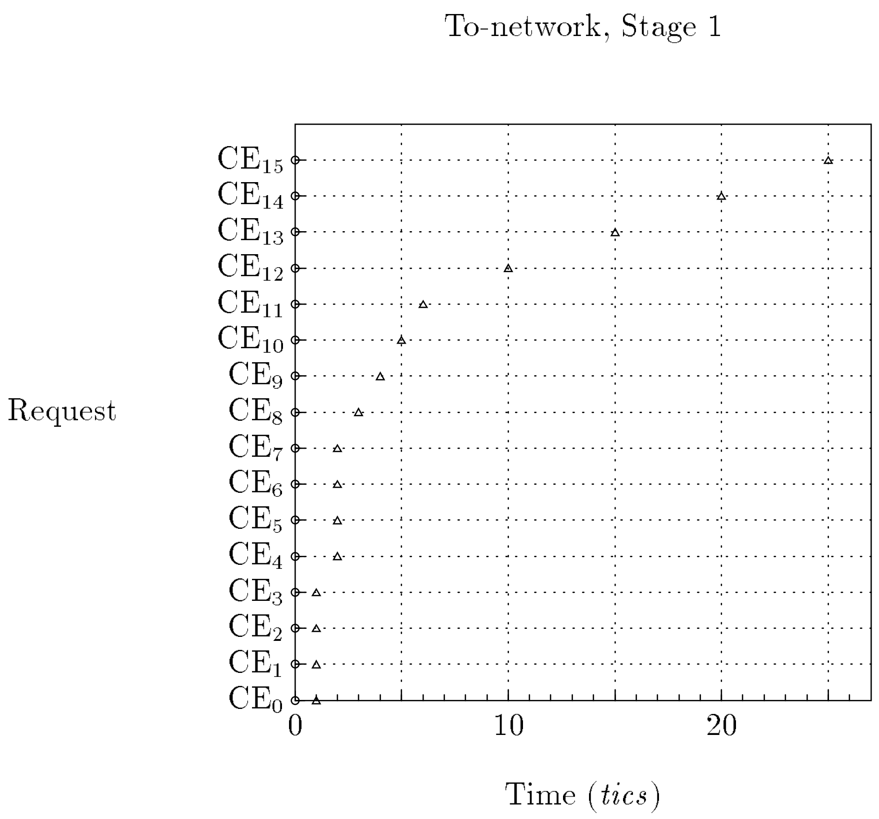
Figure 4.8: The latency statistics for the to-network for the experiment (64×64, Normal, SV), $180 \leq L < 200$.

histogram for each stage of the to-network for the vector prefetch $L = 1$. The rationale is as follows: the last read packet to exit the subsystem determines the total prefetch delay, T. For $L = 1$, $\mathrm{BW}^{-1}(1) = T$. Therefore, if a general equation for the delay of the last request can be written, a general equation for $\mathrm{BW}^{-1}(1)$ is also obtained.

Figure 4.9 shows the completion time histograms for the (16×16, Normal, SV), $L = 1$. Here, the CE that emitted each read request is plotted against the time the request spent in each stage. The time a request enters a stage is shown with a small circle, and the time a request leaves the stage is shown with a small triangle. The prefetch operation is illustrated in Figure 4.10, which shows the path from each CE to MU_0.

There are several interesting features of Figure 4.9. First, beginning with Stage 1 (the upper graph in the figure), four requests complete at clock tic 1 and again at clock tic 2, (i.e., requests 0, 1, 2, 3, and then requests 4, 5, 6, and 7). These requests are from each of the four SEs in the first stage. After they have cleared the stage, they have filled the input buffers of the second stage. Now requests 8, 9, 10, and 11 finish sequentially, due to the contention in the second stage for the single output port that is attached to the requested MU_0. It would expected that after this time, the behavior of the first stage will mirror the behavior of the second stage, which is indeed the case.

Attention now turns to the second graph in Figure 4.9, which shows the completion times of the second stage. Notice here that three requests enter without delay between them (requests 0, 1, and 2). Then there is a delay of four before the next request can



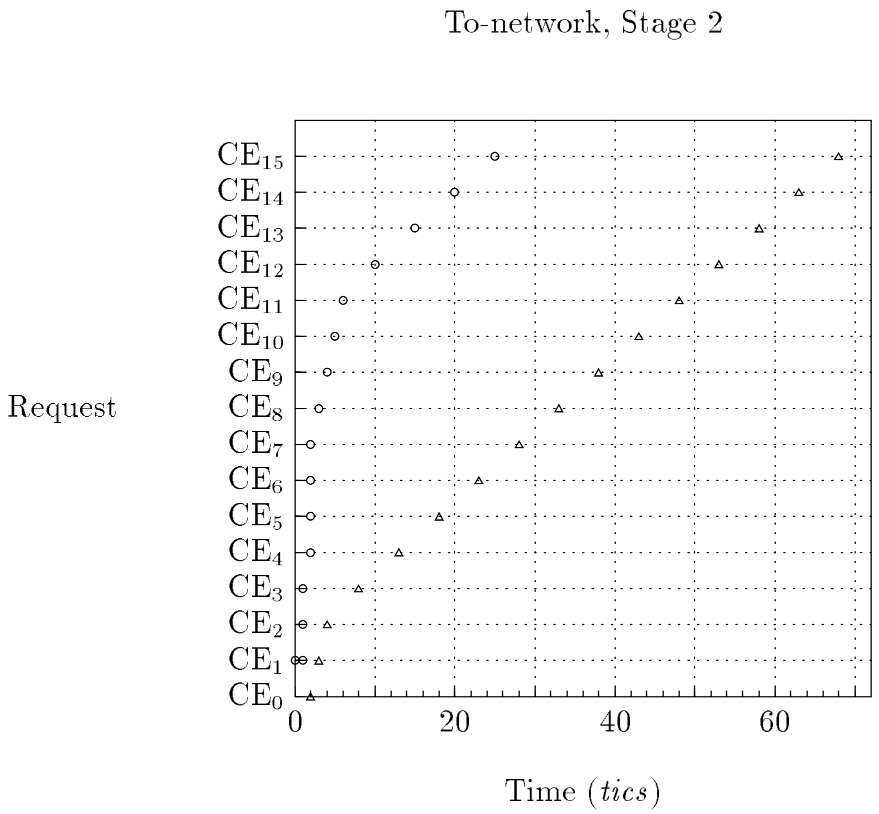


Figure 4.9: Completion time histograms for (16 × 16, Normal, SV), $L = 1$.

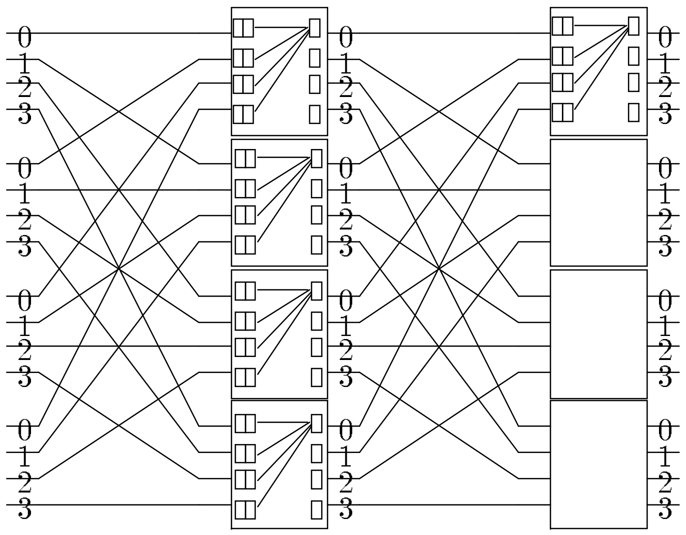


Figure 4.10: The paths from each CE to MU_0 for the 16×16 subsystem.

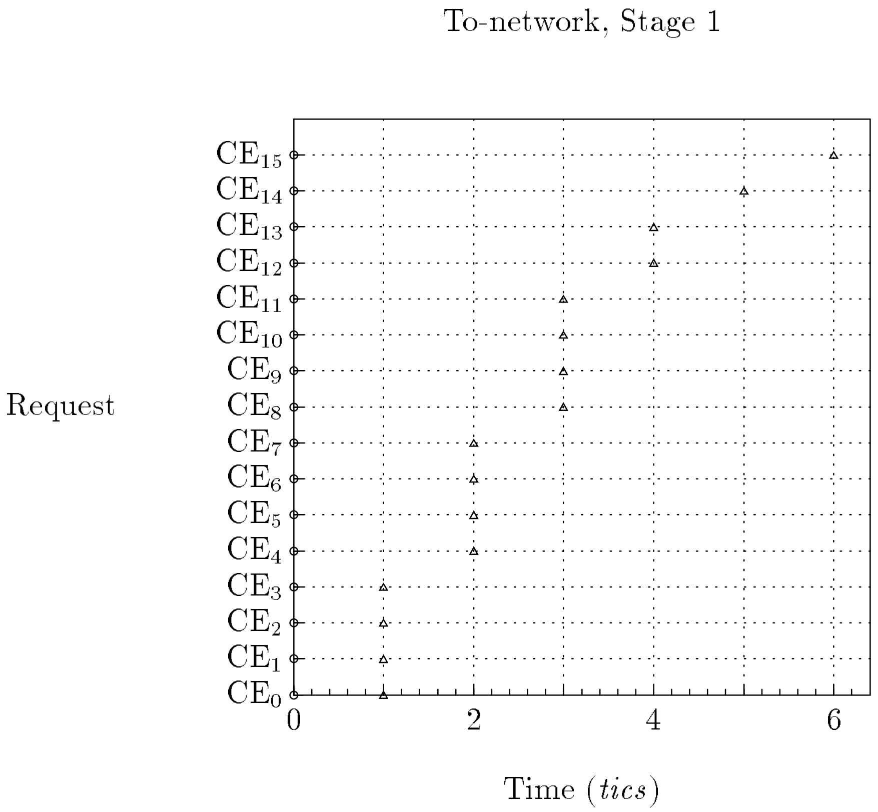
enter the memory unit (request 3). After this delay, the requests enter the memory unit once every five clock tics, which is the inverse bandwidth of the memory unit, $\mathrm{BW}_{\mathrm{MU}}^{-1}$. Hence, the time it takes the last request to complete the second stage can be written as, $M - 1 + (\ell_{\mathrm{MU}} + 1) + 4 + (16 - \ell_{\mathrm{MU}} - 2) \times \mathrm{BW}_{\mathrm{MU}}^{-1}$. The first term in this expression is the time it takes for the first request to pass the first few stages of the network. The second term is the time it takes to fill the memory unit buffers. The third term, 4, has to do with the characteristics of the memory unit's register transfer operation (see Section 4.2). Finally, the last term is the number of remaining requests times the inverse bandwidth of the memory unit. The latency for a full memory unit with $\delta = 5$, is 18. Also, it takes two clock tics for a read request to pass through each stage of the from-network. This implies the total prefetch delay is,

$$\mathrm{BW}^{-1}(1) = T = 2M + (\ell_{\mathrm{MU}} + 1) + 4 + (N - \ell_{\mathrm{MU}} - 2) \times \mathrm{BW}_{\mathrm{MU}}^{-1} + t_{\mathrm{MU}} - 1,$$

which when evaluated produces values identical to those observed for $\mathrm{BW}^{-1}(1)$.

Some of the observations made above are now investigated further to confirm this equation for $\mathrm{BW}^{-1}(1)$. First, it will be determined if the SE buffer length plays a role in the behavior of the last stage in the to-network. The buffers in the switching elements will be increased by one and the behavior of the subsystem will be observed. Figure 4.11 shows the completion time histograms for the 16×16 subsystem, where the SE3I configuration has been used. Here, observe that the completion times of the first stage have indeed changed; however, the influence of the memory unit's performance on the second stage masks this improvement. Indeed, the completion times for the second stage are exactly the same as for the Normal configuration. Without simulation, it can be clearly seen that increasing the memory unit buffer length would decrease the effect of the memory unit's inverse bandwidth on the subsystem performance in the $L = 1$ case.

Now it will be determined how the memory delay affects the equation for $\mathrm{BW}^{-1}(1)$. Shown in Figure 4.12 are the completion-time histograms for (16×16, MD10, SV), $L = 1$. The only differences found between these histograms and those shown in Figure 4.9 are that the requests are now separated by ten clock tics, instead of five, once the input buffer inside the memory unit becomes full. This indicates the use of $\mathrm{BW}_{\mathrm{MU}}^{-1}$ was correct. To make sure, the experiment (16×16, MD2, SV), $L = 1$ was performed (see Figure 4.13). This graph also confirms the use of $\mathrm{BW}_{\mathrm{MU}}^{-1}$ in the expression for $\mathrm{BW}^{-1}(1)$.



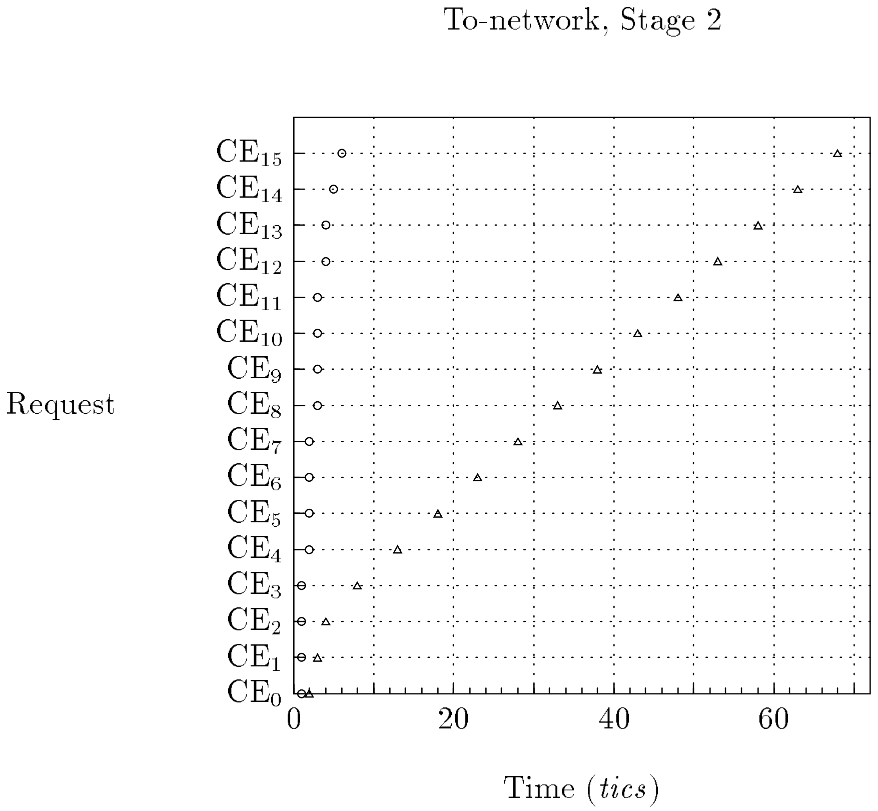
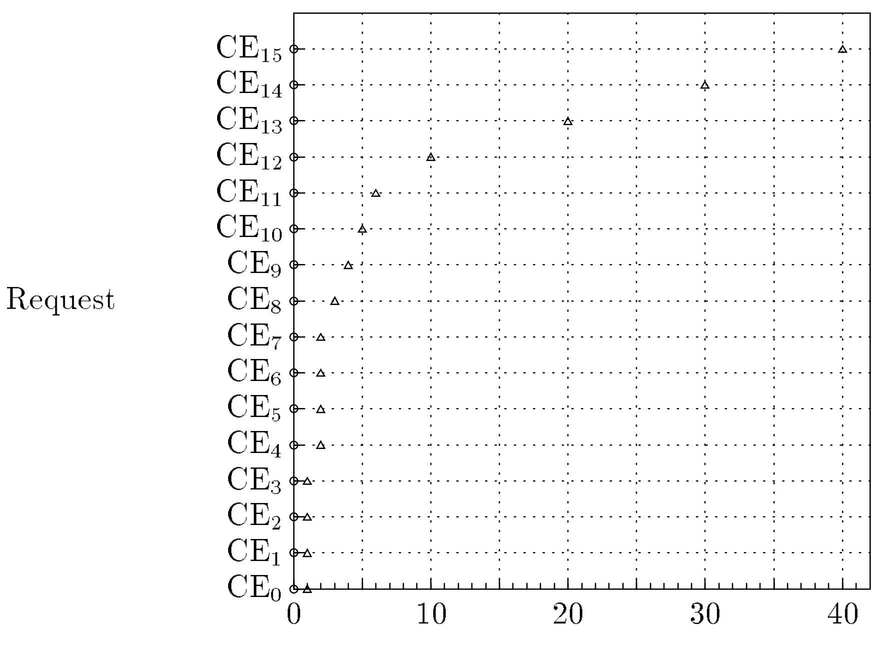


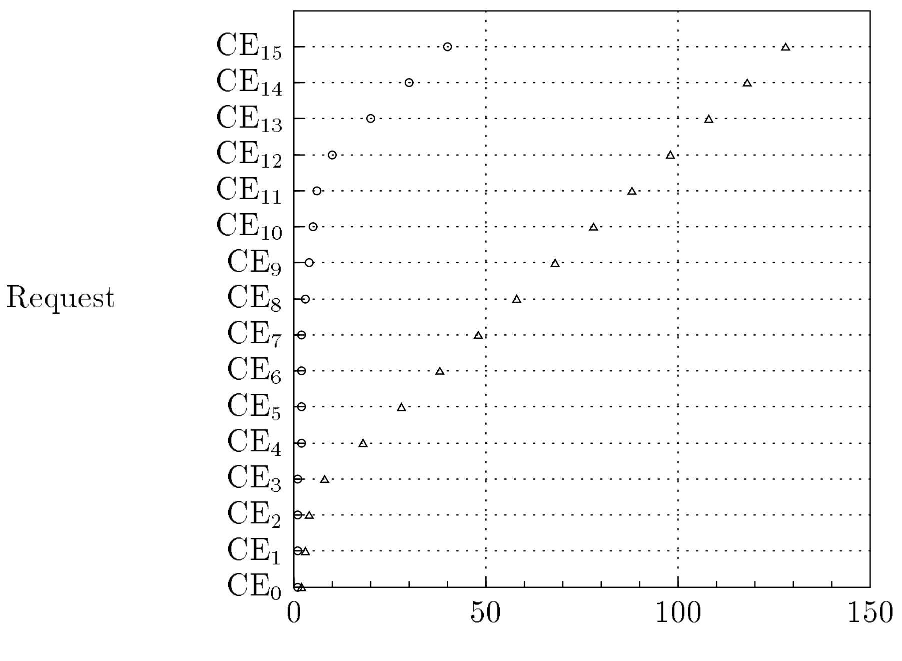
Figure 4.11: The completion time histograms for (16 $\times$ 16, SE3I, SV), $L = 1$.

To-network, Stage 1



Time (*tics*)

To-network, Stage 2



Time (*tics*)

Figure 4.12: The completion-time histograms for (16×16, MD10, SV), $L = 1$.

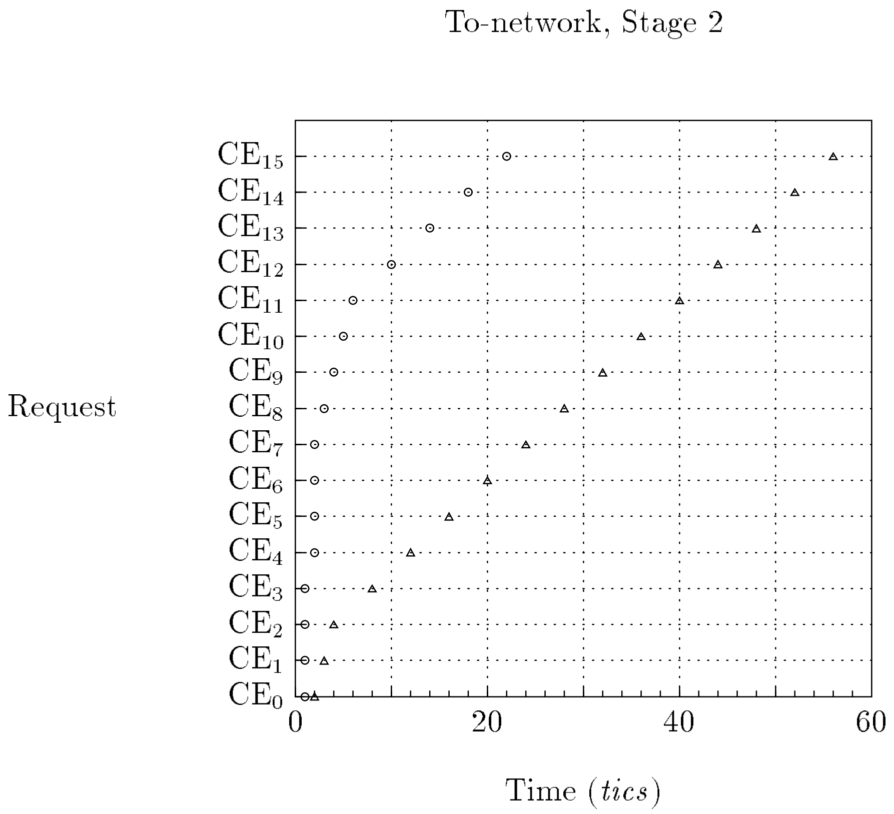


Figure 4.13: The completion-time histogram for stage 2 of the to-network, (16 $\times$ 16, MD2, SV), $L = 1$.

The total equation for inverse bandwidth as a function of the subsystem parameters is:

$$\mathrm{BW}^{-1}(L) =$$

$$\frac{1}{L}\Big(2M + (\ell_{\mathrm{MU}} + 1) + 4 + (N - \ell_{\mathrm{MU}} - 2) \times \mathrm{BW}_{\mathrm{MU}}^{-1} + t_{\mathrm{MU}} - 1 - \mathrm{BW}^{-1}(\mathrm{MU})\Big)$$

$$+\mathrm{BW}^{-1}(\infty),$$

where, $\mathrm{BW}^{-1}(\infty) = \mathrm{BW}_{\mathrm{MU}}^{-1}$ for the 8×8 and 16×16 subsystems, and $\mathrm{BW}^{-1}(\infty)$ is due to the contention in the first stage of the to-network for the 64×64 subsystem.

Finally, on the difference between this equation and the observed inverse bandwidth are now discussed. Figure 4.14 shows a comparison between the actual inverse bandwidth for the (8×8, Normal, SV) experiment [*solid line*] and the hyperbolic model [*dotted line*]. This difference is graphed in Figure 4.15. (The envelope of this curve is hyperbolic, although the hyperbolic that delimits the peaks of the curve is not the same shape as the hyperbolic that delimits the valleys.) The valleys occur at multiples of eight. The peaks occur at multiples of eight, offset by two (e.g., 10, 18, 26, etc.). This implies that vector prefetch operations that are multiples of the subsystem's dimension perform slightly better than other vector prefetches. For the (8×8, Normal, SV) experiment, the RMS error is 0.105. The two points of interest, the initial and final values, are exact.

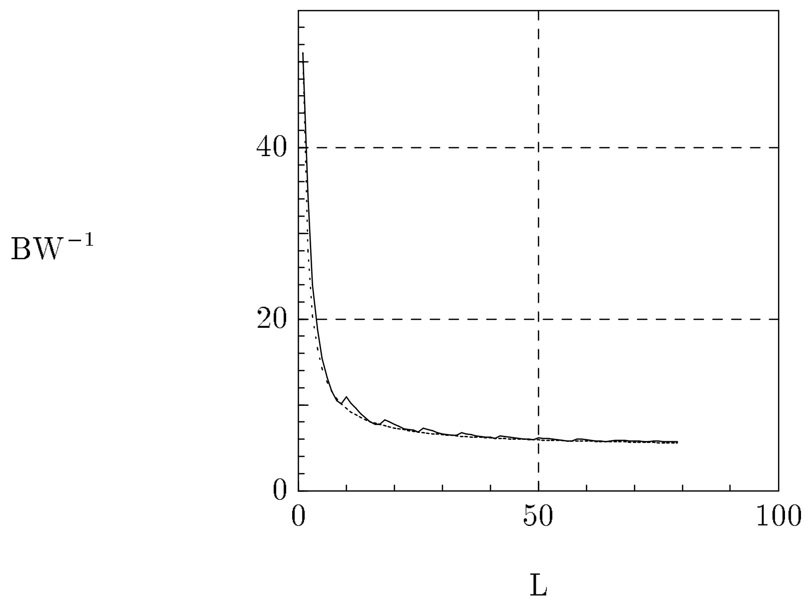


Figure 4.14: Inverse bandwidth for (8 × 8, Normal, SV) [*solid line*] and its hyperbolic model [*dotted line*].

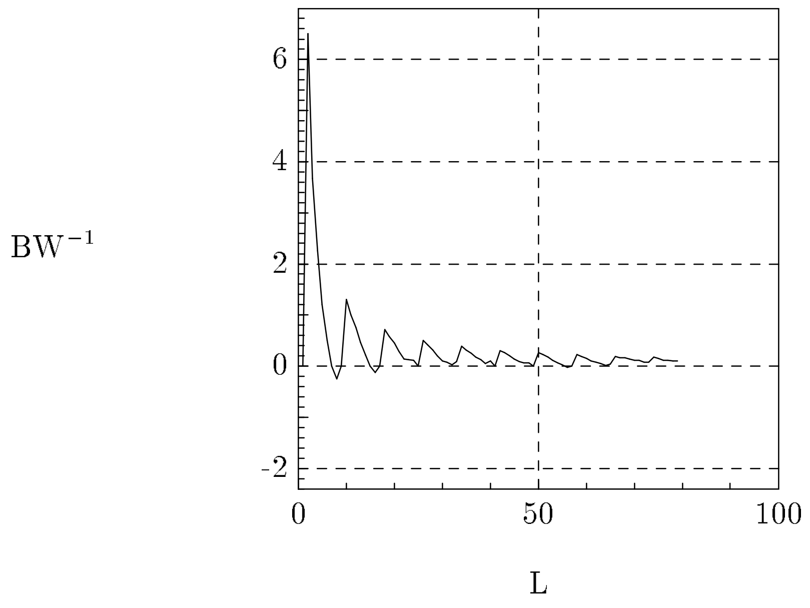


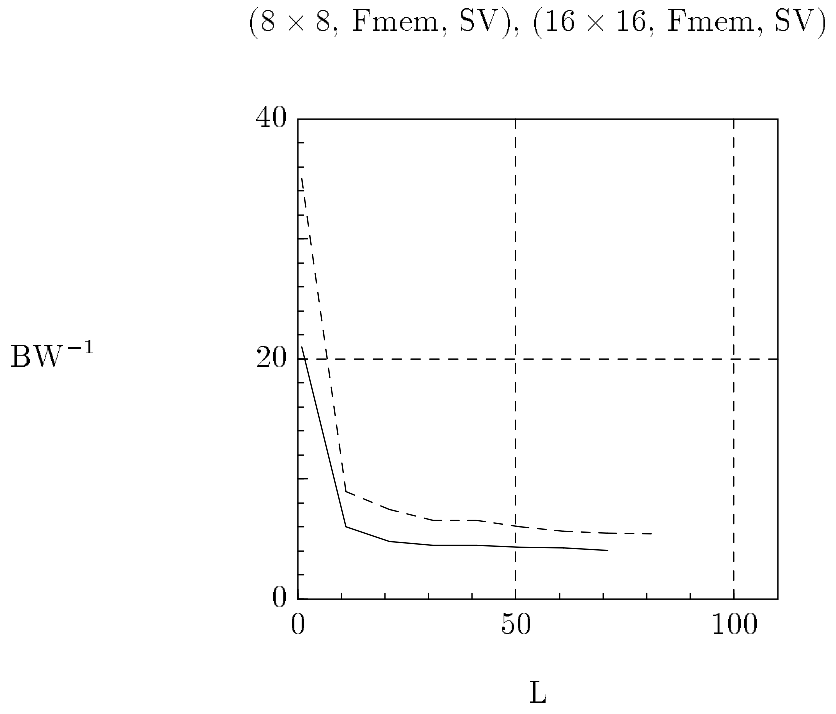
Figure 4.15: The difference between the hyperbolic model and the actual inverse bandwidth for (8 × 8, Normal, SV).

4.4 Subsystem Tuning

Now that the performance of the subsystem is understood in greater detail, attention now turns to investigating methods for improving the worst-case performance.

It was showed above that for the (8×8, Normal, SV), and (16×16, Normal, SV) configurations, the asymptotic inverse bandwidth, $\mathrm{BW}^{-1}(\infty) = \mathrm{BW}_{\mathrm{MU}}^{-1}$. The performance of the memory units cannot be controlled. However, pipelining and interleaving of memory unit pipelines within one MU are plausible methods for reducing $\mathrm{BW}_{\mathrm{MU}}^{-1}$ to unity. Therefore other bottlenecks that may exist in the subsystem after the memory unit bottleneck has been removed will be investigated. This will be done by assuming that the memory units behave similarly to the fast memory units described in Section 4.1.4, above. Specifically, these memory units have the ideal $\mathrm{BW}_{\mathrm{MU}}^{-1} = 1$, and have sufficient buffering on their outputs to decouple the BUSY signal propagation of the from-network from that of the to-network.

The inverse bandwidths for the three subsystems in the Fmem configuration will now be discussed. Figure 4.16 shows the inverse bandwidth for the three experiments, (8×8, Fmem, SV), (16×16, Fmem, SV), and (64×64, Fmem, SV). Notice that $\mathrm{BW}^{-1}(\infty) = 4.0$, for the ($8 \times 8$, Fmem, SV) experiment, $\mathrm{BW}^{-1}(\infty) = 5.4$ for the (16×16, Fmem, SV) experiment, and $\mathrm{BW}^{-1}(\infty) = 8.4$ for the (64×64, Fmem,SV) subsystem. This indicates that reducing the memory unit inverse bandwidth to unity has a noticeable effect on the asymptotic inverse bandwidth. Although the bottleneck previously was not the memory unit in the 64×64 subsystem, reducing the memory



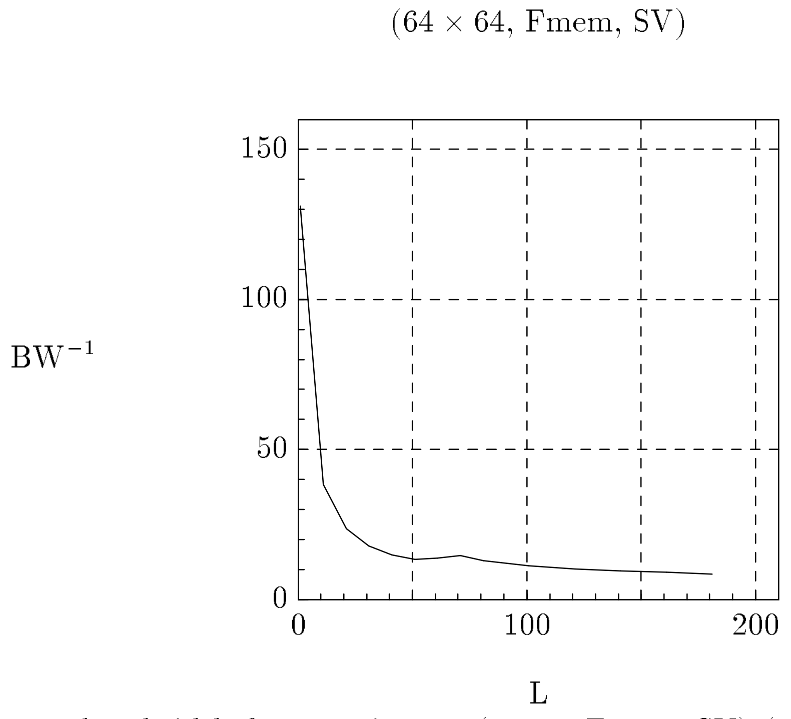


Figure 4.16: Inverse bandwidth for experiments (8×8, Fmem, SV) (*top graph, solid line*), (16×16, Fmem, SV) (*top graph, dashed line*), and (64×64, Fmem, SV) (*bottom graph*).

unit's inverse bandwidth decreased the second stage's latency and therefore decreased the overall prefetch delay.

The latencies of the to-network for the stages of the three subsystems in the Fmem configuration are now investigated. Figure 4.17, Figure 4.18, and Figure 4.19, show the latencies for the experiment (8×8, Fmem, SV), (16×16, Fmem, SV), and (64×64, Fmem, SV), respectively. Here, as before, the latency of the first stage is shown using

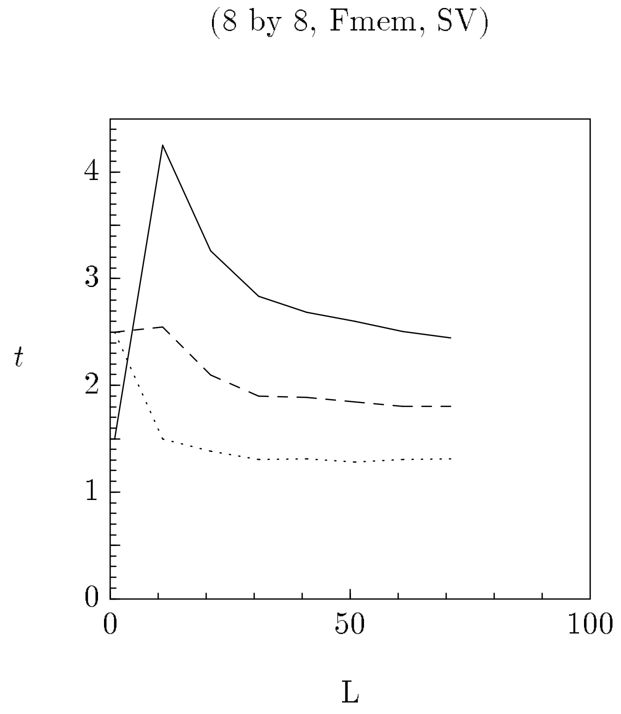


Figure 4.17: Latencies for (8×8, Fmem, SV).

a solid line, the latency of the second stage as a dashed line, and where applicable, the latency of the third stage is shown as a dotted line. There are several things to notice: first, the shape of the latency curve is very similar for the first stage across all three experiments. Second, the latency for the second stage is similar for the 16×16 and 64×64 subsystems. Finally, the magnitude of the latencies seems to be proportional

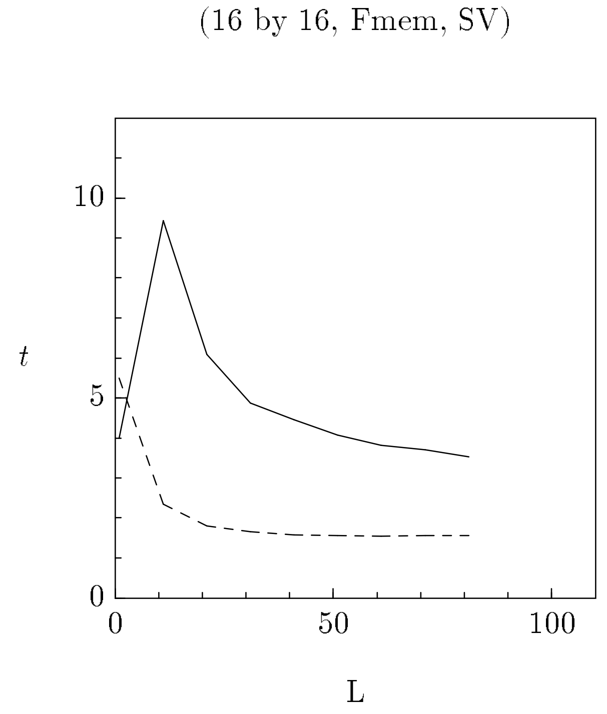


Figure 4.18: Latencies for (16 × 16, Fmem, SV).

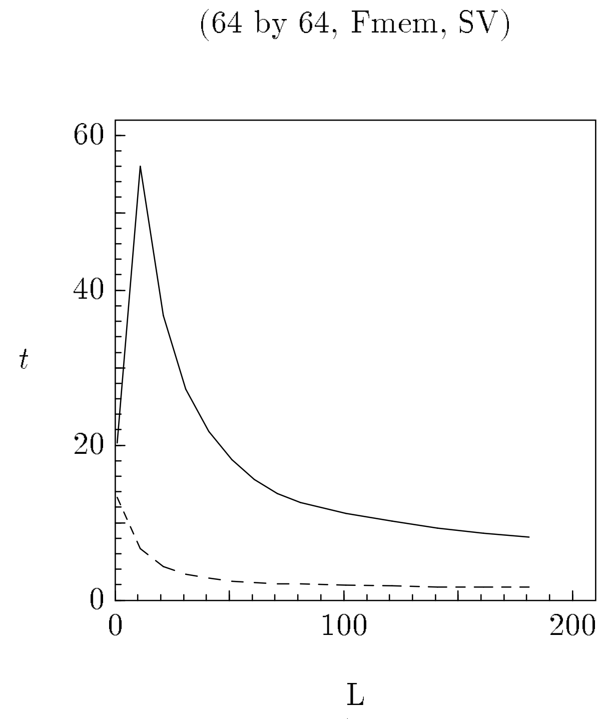


Figure 4.19: Latencies for (64 × 64, Fmem, SV).

to N, the subsystem size. In all cases, the stage with the longest latency is the first stage. Since the memory units always accept, we know that contention is the cause of the latencies. The input ports of the SEs in the first stage are in the BUSY state more often than the other stages of the network. In the (16×16, Fmem, SV) and (64×64, Fmem, SV) experiments, $F_{\text{BUSY}} = 0.12$ for the first stage, and the second stage's input ports are never in the BUSY state (i.e., the fast memory units always accept requests). In the (8×8, Fmem, SV) subsystems, $F_{\text{BUSY}} = 0.18$ for the first stage, $F_{\text{BUSY}} = 0.08$ for the second stage, and $F_{\text{BUSY}} = 0.0$ for the third stage. The BUSY signal is only being asserted due to the difference in arrival rate of packets at an input port in the second (or third) stage, which is one request per clock tic, and the service rate of the contention resolution policy of the SEs in the second (third) stage, which is nominally one request per K clock tics. Therefore, increasing the length of the SEs input buffers should not improve performance due to this mismatched arrival and service rates since even very large queues would not be useful. To test this claim, the SE input buffer lengths of the 16×16 subsystem was increased to $\ell_{\text{SE}} = 8$ (configuration "SE8IFmem") and the inverse bandwidth is shown in Figure 4.20, compared to the inverse bandwidth for the nominal $\ell_{\text{SE}} = 2$ for the Fmem configuration. The performance of the SE8IFmem configuration is roughly twice that of the Fmem configuration for the asymptotic case, indicating buffering actually worsens the subsystem performance.

The arrival rate of requests inside the network cannot be controlled; however, the issue rate of requests at the GIB/CE end of the subsystem can be. Hence, in the next

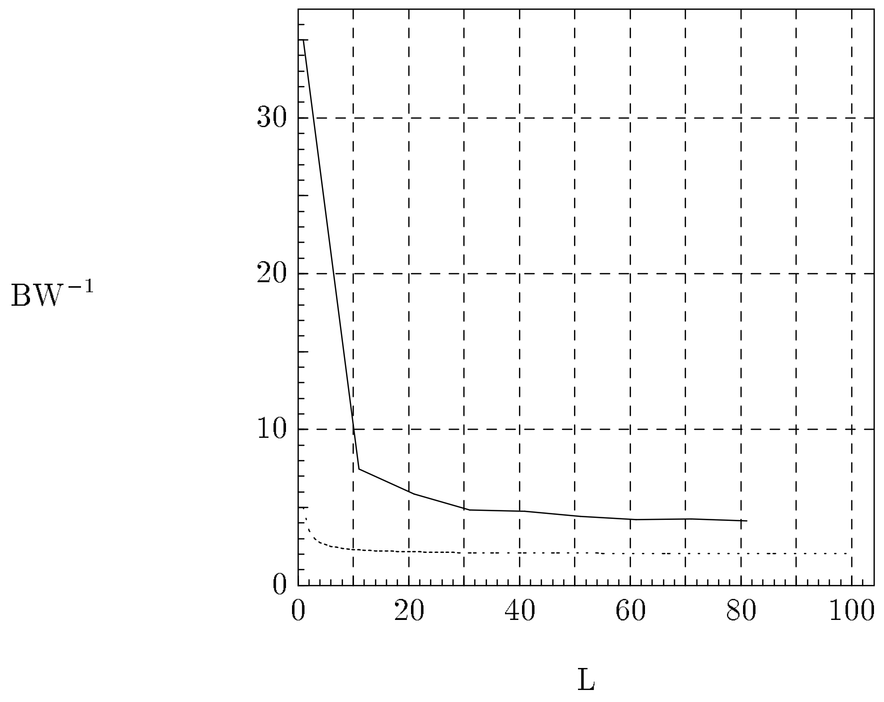


Figure 4.20: The inverse bandwidth for the experiment (16 × 16, SE8IFmem, SV) [*solid line*], and the experiment (16 × 16, Fmem, SV) [*dotted line*].

experiment the issue rate of requests from the GIBs/CEs was reduced to one request every K clock tics and the results were observed. Figure 4.21 shows the stage latencies of this experiment for the 16×16 subsystem, which is typical of the results obtained for all three subsystems. Though the latency of stage one for vector lengths shorter

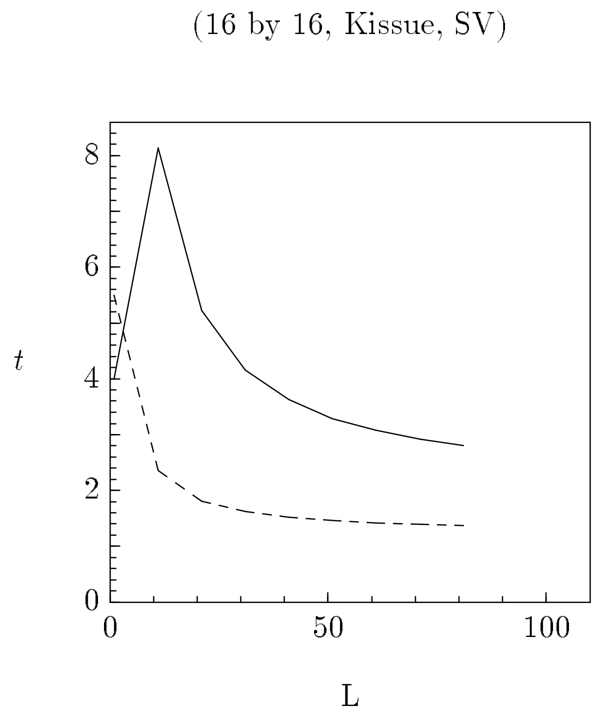


Figure 4.21: Latencies for (16×16, Fmem, SV), with reduced issue rate.

than $L = 50$ is now 14% less than before, the latencies for longer vectors are the same. Hence, this approach also fails to reduce the severity of the latencies in the first stage.

Hence, none of the obvious ways of improving Omega network performance in the worst case have significant effects on the contention inside the to-network. Buffering only exacerbates the problem and changing issue rate of the GIB has little effect.

In conclusion, then, the largest worst-case performance increase of the subsystem can be obtained by improving the memory units so that the inverse bandwidth, $\mathrm{BW}_{\mathrm{MU}}^{-1}$, approaches unity. It is suggested that the networks remain essentially the same. In the next chapter these conclusions will be confirmed for more realistic scenarios.

CHAPTER 5

BEST-CASE AND REALISTIC SCENARIO PERFORMANCE

In the previous chapter, the performance of the subsystem in the worst-case was analyzed. Attention now turns to the performance of the subsystem under more realistic scenarios.

5.1 Best-Case Scenarios

It is useful to have a best-case simultaneous prefetch scenario to evaluate the performance of the subsystem. In [2], Lawrie describes several permutations that an Omega network can pass without contention. The following theorem adapted from [2] will be used,

Theorem: [Lawrie] An Omega network passes without contention an access by CE_i *to* $\mathrm{MU}_{(i+k) \bmod N}$, *for all* $0 \leq i < N$ *and all integer* k.

For a proof, see [2].

When the vector for each CE$_i$ starts in MU$_i$, this will referred to this as the *identity vector* scenario, or *ID*, since it causes a series of identity permutations [2]. Figure 5.1 shows the permutations for the 8×8 subsystem, where $L = 10$. Note that all the permutations have zero f_c, indicating this scenario should not cause any contention in the to-network. This is therefore picked to be the best-case scenario.

$$\begin{array}{l}
p_0 = (0\ 1\ 2\ 3\ 4\ 5\ 6\ 7),\ f_c = 0 \\
p_1 = (7\ 0\ 1\ 2\ 3\ 4\ 5\ 6),\ f_c = 0 \\
p_2 = (6\ 7\ 0\ 1\ 2\ 3\ 4\ 5),\ f_c = 0 \\
p_3 = (5\ 6\ 7\ 0\ 1\ 2\ 3\ 4),\ f_c = 0 \\
p_4 = (4\ 5\ 6\ 7\ 0\ 1\ 2\ 3),\ f_c = 0 \\
p_5 = (3\ 4\ 5\ 6\ 7\ 0\ 1\ 2),\ f_c = 0 \\
p_6 = (2\ 3\ 4\ 5\ 6\ 7\ 0\ 1),\ f_c = 0 \\
p_7 = (1\ 2\ 3\ 4\ 5\ 6\ 7\ 0),\ f_c = 0 \\
p_8 = (0\ 1\ 2\ 3\ 4\ 5\ 6\ 7),\ f_c = 0 \\
p_9 = (7\ 0\ 1\ 2\ 3\ 4\ 5\ 6),\ f_c = 0
\end{array}$$

Figure 5.1: The permutations and their fractions of contention for identity vector prefetches, $L = 10$, the 8×8 subsystem.

Althought the ID scenario is the best-case for a simultaneous vector prefetch, it is not a prefetch of the same vector by all CEs. Since this was the purpose of the worst-case scenario of the last chapter, scenarios that perform the same task but do so more efficiently will now be introduced.

By changing the order by which CEs emit requests, and assuming the time at which CEs make requests is controllable, an SV scenario prefetch operation can be modified to obtain the same zero fraction of contention of the ID scenario. This idea is displayed in the following algorithm, assuming L is a multiple of N and $L > N$, *Algorithm I:*

CE$_i$ *issues a request to* MU$_i$. *The next request made by* CE$_i$ *is to* MU$_{(i+1)\text{mod}N}$, *etc., until the entire vector has been retrieved.*

Without loss of generality and for readability it has been assumed that the vector being fetched starts in MU$_0$. This will be referred to as the *Algorithm I* scenario, or *A1*. This scenario is illustrated in Figure 5.2 for the 8×8, $L = 16$, where we have written the permutations in vertical columns and time of issue increases from left to right. One can see that each column is an identity permutation and there are no requests for the same memory unit at the same time. Hence, $f_c = 0$ for all permutations.

CE$_0$	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
CE$_1$	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0
CE$_2$	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1
CE$_3$	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2
CE$_4$	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3
CE$_5$	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4
CE$_6$	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5
CE$_7$	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6
$f_c =$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 5.2: Algorithm I scenario, the 8×8 subsystem, $L = 16$.

However, L may not always be an integral multiple of N. One possible solution is to extend the Algorithm I scenario in the following manner:

Step 1 CE$_i$ *issues a request to* MU$_i$, *The next request made by* CE$_i$ *is to* MU$_{(i+1)\text{mod}N}$, *etc. This pattern is continued until* CE$_i$ *makes a request of* MU$_{(L-1)\text{mod}N}$.

Step 2 CE$_i$ *then issues a request to* MU$_0$*, then* MU$_1$*, etc., until the entire vector has been retrieved.*

. This is illustrated in Figure 5.3 for the 8×8 subsystem, with $L = 15$. Notice that

CE$_0$	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6
CE$_1$	1	2	3	4	5	6	7	0	1	2	3	4	5	6	0
CE$_2$	2	3	4	5	6	7	0	1	2	3	4	5	6	0	1
CE$_3$	3	4	5	6	7	0	1	2	3	4	5	6	0	1	2
CE$_4$	4	5	6	7	0	1	2	3	4	5	6	0	1	2	3
CE$_5$	5	6	7	0	1	2	3	4	5	6	0	1	2	3	4
CE$_6$	6	7	0	1	2	3	4	5	6	0	1	2	3	4	5
CE$_7$	7	0	1	2	3	4	5	6	0	1	2	3	4	5	6
$f_c =$	0	0	0	0	0	0	0	0	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$

Figure 5.3: Algorithm I scenario extended, the 8×8 subsystem, $L = 15$, for each CE.

the property of each time issuing an identity permutation has been lost, and hence contention does occur. A possible way to avoid this contention is to always request vectors in multiples of N, filling up the extra unused elements with idle periods so that an identity permutation always results. This leads to what will be called the *Algorithm II* scenario, or *A2*:

Algorithm II:

If L is a multiple of N*, use Algorithm I. Else, do the following for all* $0 \leq i < N$*:*

Step 1. CE$_i$ *begins by issuing a request to* MU$_i$*; this pattern is continued until* CE$_i$ *makes a request to* MU$_{(L-1)\text{mod}N}$

Step 2. CE$_i$ *does not issue a request for* $N - (L \bmod N)$ *time units.*

Step 3. CE_i *begins issuing requests to* MU_0*; this pattern is continued until* CE_i *makes a request to* MU_{i-1}*, at which time the entire vector has been fetched by all processors.*

The Algorithm II scenario is illustrated in Figure 5.4 for $L = 15$, where “x” signifies an idle CE. Notice that identity permutations always result.

CE_0	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	x
CE_1	1	2	3	4	5	6	7	0	1	2	3	4	5	6	x	0
CE_2	2	3	4	5	6	7	0	1	2	3	4	5	6	x	0	1
CE_3	3	4	5	6	7	0	1	2	3	4	5	6	x	0	1	2
CE_4	4	5	6	7	0	1	2	3	4	5	6	x	0	1	2	3
CE_5	5	6	7	0	1	2	3	4	5	6	x	0	1	2	3	4
CE_6	6	7	0	1	2	3	4	5	6	x	0	1	2	3	4	5
CE_7	7	0	1	2	3	4	5	6	x	0	1	2	3	4	5	6
$f_c =$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 5.4: Algorithm II scenario, the 8×8 subsystem, $L = 15$.

However, for a value of L that is not reasonably close to a multiple of N, each CE is idle for a noticeable amount of time in the Algorithm II scenario. Witness in Figure 5.5 what occurs for $L = 9$. Notice that there is no way to compact the scheme without contention occurring. However, the cost of contention might not be as severe as the cost of avoiding contention in this situation. The time a CE spends idle might be better spent issuing requests while its earlier requests wait to get out of contention in the network. For this reason, both the A1 and A2 scenarios will be used for the performance evaluations.

CE_0	0	1	2	3	4	5	6	7	0	x	x	x	x	x	x	x
CE_1	1	2	3	4	5	6	7	0	x	x	x	x	x	x	x	0
CE_2	2	3	4	5	6	7	0	x	x	x	x	x	x	x	0	1
CE_3	3	4	5	6	7	0	x	x	x	x	x	x	x	0	1	2
CE_4	4	5	6	7	0	x	x	x	x	x	x	x	0	1	2	3
CE_5	5	6	7	0	x	x	x	x	x	x	x	0	1	2	3	4
CE_6	6	7	0	x	x	x	x	x	x	x	0	1	2	3	4	5
CE_7	7	0	x	x	x	x	x	x	x	0	1	2	3	4	5	6
$f_c =$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 5.5: Algorithm II scenario, the 8×8 subsystem, $L = 9$.

5.1.1 Prefetch skewing

What if the assumption that the time at which CEs make requests is controllable (i.e., the CEs issue requests synchronously) is relaxed? In other words, all GIBs may not start the prefetch operation at the same time. Although this assumption might seem unrealistic, the reader is reminded that an MIMD machine does not have a synchronization operation to *insure* that all CEs execute an instruction at the same time. Indeed, the general philosophy of an MIMD machine, in contrast to a SIMD machine, implies a degree of inherent asynchrony. Barrier synchronization does seem to be a candidate for such an operation. However, though the conventional barrier synchronization operation is implemented in Cedar, this operation guarantees only that all CEs have passed into a critical region of code and not that they have done so at the same time.

With the synchronous issue assumption relaxed, the time at which requests enter a network is not predictable. However, once a GIB starts making requests, we must assume it will continue to make requests once every time unit. Hence, this new assumption can be interpreted as making the starting time for each vector prefetch variable. This will be referred to as *prefetch skewing.*

What would be the worst possible amount of contention assuming prefetch skewing of the Algorithm I scenario and assuming L was a multiple of N? If CE$_i$ started making requests of the network i clock units late, then one would expect some permutations with unity fraction of contention. This is illustrated for $L = 16$ in Figure 5.6. This particular skewing will be referred to as *worst-case skewing,* and this scenario will be referred to as the *worst-case static-skewed Algorithm I* scenario, or WCSA1. Notice

CE$_0$	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7							
CE$_1$	x	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0						
CE$_2$	x	x	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1					
CE$_3$	x	x	x	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2				
CE$_4$	x	x	x	x	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3			
CE$_5$	x	x	x	x	x	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4		
CE$_6$	x	x	x	x	x	x	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	
CE$_7$	x	x	x	x	x	x	x	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6
$f_c =$	0	$\frac{1}{4}$	$\frac{3}{8}$	$\frac{1}{2}$	$\frac{5}{8}$	$\frac{3}{4}$	$\frac{7}{8}$	1	1	1	1	1	1	1	1	1	$\frac{7}{8}$	$\frac{3}{4}$	$\frac{5}{8}$	$\frac{1}{2}$	$\frac{3}{8}$	$\frac{1}{2}$	0

Figure 5.6: Algorithm I scenario with worst-case skewing, the 8×8 subsystem, $L = 16$.

that this region of unity fraction of contention will grow as the vector length grows. Therefore, a worst-case skewing of the A1 scenario will degenerate into the SV scenario in the limit.

In summary, three scenarios have been presented that will be use to evaluate the performance of the subsystem. The ID scenario is the best-possible prefetch scenario.

Somewhat more realistic are the A1 and A2 scenarios. Finally, the WCSA1 scenario was presented to demonstrate the need for synchronization.

5.2 Performance Results

In this section, the data from simulation is presented and commented on. First, the reseults of experiments for all three subsystems using the ID scenario will be presented; then some other interesting results of the other scenarios will be presented. The Normal configuration and the Fmem configuration will be used to observe the effects of memory delay.

5.2.1 Performance in the best-case

The stage latencies for the experiments (8×8, Fmem, ID), (16×16, Fmem, ID), and (64×64, Fmem, ID) were constant at 1 clock tic for all stages of the to-network, and all vector lengths, indicating there was no contention whatsoever. The inverse bandwidth for these three experiments was constant at $\mathrm{BW}^{-1} = 2$, since it takes two clock tics for the two-word reply packets to move across each stage of the from-network.

For the Normal configuration, the inverse bandwidth for all three subsystems is constant at $\mathrm{BW}^{-1} = 5$, due to the memory units. Because of this, the stage latencies are not unity, as shown in Figure 5.7 for the experiment (16×16, Normal, ID). Therefore, the performance of the ID scenario is not ideal with the current memory units. For this reason, rest of the experiments will use the Fmem configuration.

(16 by 16, Normal, SV)

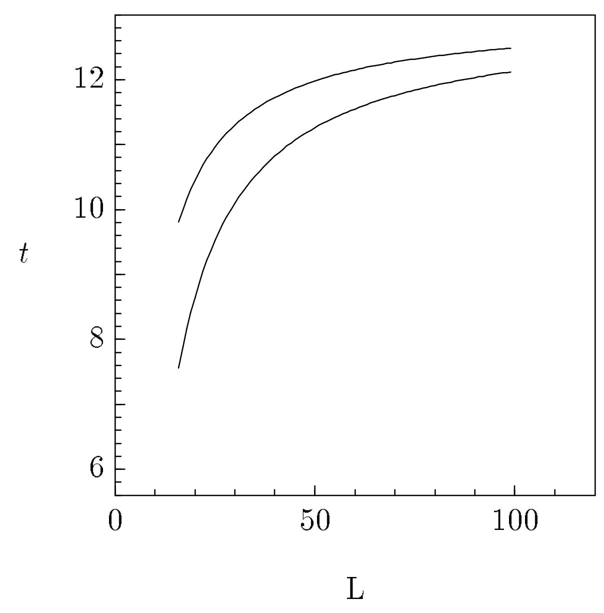


Figure 5.7: Latency for to-network for the experiment (16×16, Normal, SV).

5.2.2 The effectiveness of scenarios

The results of the simulator can be used to determine which of the two scenarios, A1 or A2, performs better. The inverse bandwidth for the two experiments, (16×16, Fmem, A1) and (16×16, Fmem, A2), are shown in Figure 5.8. Additionally, we have shown (16×16, Fmem, SV), as a reference. The A2 scenario performs approximately 0.5 clock tics better than the A1 scenario. This means the time spent delaying a request before it enters the network (i.e., the A2 scenario) produces better results than not delaying requests and allowing some contention to occur (i.e., the A1 scenario).

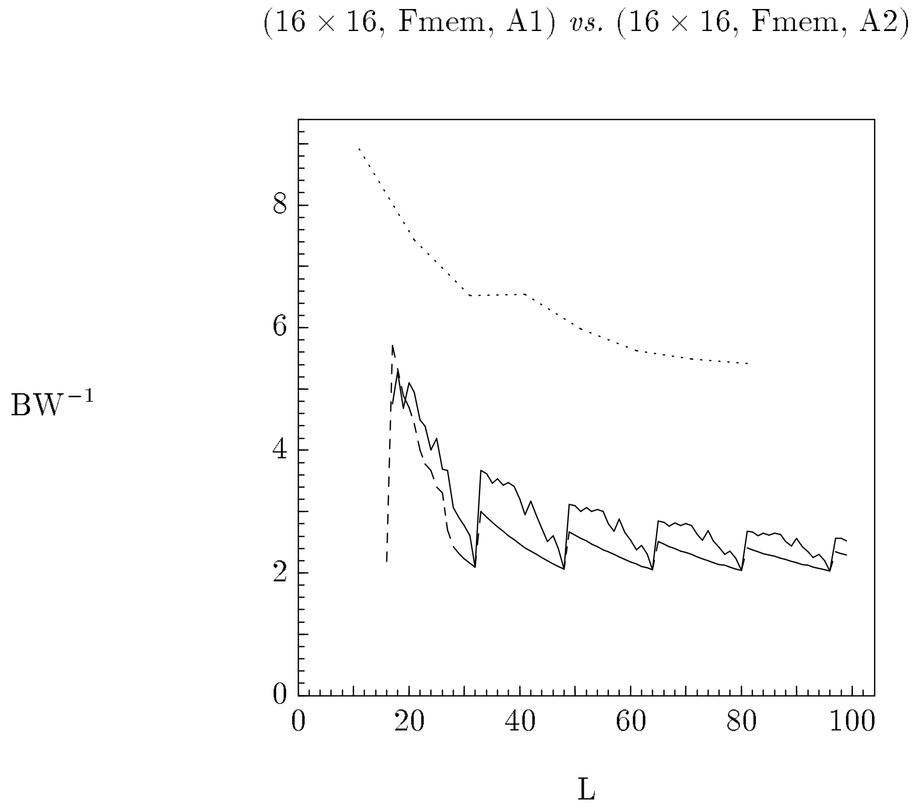


Figure 5.8: Inverse bandwidth for experiments (16 × 16, Fmem, A1) [*solid line*], and (16 × 16, Fmem, A2) [*dashed line*], with (16 × 16, Fmem, SV) [*dotted line*] shown as a reference.

The performance of the worst-case scenario indicated that contention was a very costly phenomenon in larger subsystems. Avoiding contention is very important. However, the A1 and A2 scenarios assume a synchronization operation that is not available in the subsystem.

5.3 Conclusions

Some interesting scenarios for simultaneously accessing the same vector have been presented. The previous chapter proposed that the memory units be redesigned to match the best-case bandwidth of the network. In this chapter, skewing due to the lack of synchronization was also commented on. In SIMD machines, permutations that enter the Omega network can be controlled. In MIMD machines, however, they cannot be. It is proposed that hardware be added to selectively synchronize the initiation of prefetch operations so that the characteristics of Omega networks can be used to improve the performance of prefetch operations, as in [2].

CHAPTER 6

CONCLUSIONS

6.1 Summary and Conclusions

This thesis reviewed the design of a global memory subsystem, described the algorithm used to simulate it, and then simulated its performance in several scenarios of a simultaneous vector prefetch. The performance in the worst case and best case of a simultaneous prefetch was investigated. Also investigated was the performance of the subsystem using algorithms designed to effectively prefetch information.

Several conclusions can be drawn from the results:

- Reviewing the results for the Normal configuration reveals that the 8×8, and 16×16 subsystems did not show much difference in performance between the best case and the worst case for very long vector lengths. The 64×64, however, did show a difference in performance. Therefore it can be concluded that for small subsystems with the current memory units, the traffic through the network does not affect performance as much as for larger systems. When the subsystem is made large ($N = 64$), contention takes over as the primary performance bottleneck. In all cases, performance is best for long vector lengths.

- The memory units were the direct cause (8×8 and 16×16 subsystems) or indirect cause (64×64 subsystem) of the upper bound on performance of the subsystem. The simulator was then used to derive an expression for this upper bound as a function of subsystem design parameters, where possible. The performance in the best-case showed that the memory units' performance determined the subsystems' performance.

- The effectiveness of using algorithms for improving the performance of a simultaneous prefetch was investigated. It was discovered that for small subsystems these algorithms performed only slightly better than the worst-case simultaneous prefetch, due mainly to the memory units' performance. When the memory units were replaced with higher performance units, the algorithms performed better. However, it was shown that the algorithms can degenerate into the worst-case scenario due to a lack of synchronization.

6.2 Designing a Global Memory Subsystem

From the experiments performed, some guidelines for building a global memory subsystem using Omega networks and an equal number of memory units and processors can be proposed:

- The memory units of the subsystem should have an inverse bandwidth that matches the best-case inverse bandwidth of the network stages.
- There should be provisions for synchronization of a prefetch operation so that the characteristics of an Omega network can be exploited to improve individual prefetch performance. This synchronization need not eliminate skewing, rather it need only make the skewing predictable and exposed to the compiler.
- The compiler should use heuristics derived from simulation, such as the performance of the subsystem in the worst case, to hide prefetch performance by scheduling prefetch operations far enough in advance of the operations that require the prefetch data. This scheduling of prefetch operations is especially important for larger systems.

6.3 Future Work

There are several possible extensions to this thesis. One extension would be to expand the concept of scenario-based performance analysis. There are a finite number of possible vector-prefetch scenarios a given subsystem can encounter. Though this number is large, it can be reduced greatly by observing that many patterns have similar performance due to the symmetry inherent in the Omega networks. With enough insight, the performance of the subsystem could be fully characterized in terms of indices such as the permutation of memory units, the individual vector lengths being prefetched, and the amount of traffic already in the network. Once this information is collected, a full expression for the performance of the subsystem in terms of these indices could be written. Finally, this expression could be used to create a new global interface that intelligently controls its issue rate to improve inverse bandwidth and trace delay. Alternately, prefetch scheduling could be placed under the control of the compiler without modifying the global interface.

Another possible extension would be to evaluate the subsystem's performance based on actual traces of executing applications. When Cedar becomes operational, such traces will be available. Work based on traces of multiprocessor applications is currently being done by Steve Turner.

In conclusion, it was shown that the use of Omega networks for interconnection between the memory units and processors of the Cedar multiprocessor was a wise choice. These networks allow parallel accesses for some input patterns that a bus architecture would not allow. There are just four control lines associated with each network line, much less than usually required in bus systems. However, to use the networks efficiently, the memory units and global interfaces attached to them should be carefully tuned.

Ways of modifying the hardware to improve the performance of the subsystem have discussed. It is suggested that the compiler use reasonable heuristics to schedule the traffic in a memory system, much the way compilers are now being used to schedule instructions inside processors themselves. For this reason, a heuristic was provided in the form of an upper bound on the subsystem performance and its usefulness to the compiler for improving prefetch performance was discussed.

REFERENCES

[1] K. C. Batcher, “Sorting networks and their applications,” in *Proc. AFIPS 1968 Spring Joint Comput. Conf.*, pp. 307–314, 1968.

[2] D. H. Lawrie, “Access and alignment of data in an array processor,” *IEEE Trans. Comput.*, vol. c-24, pp. 1145–1155, Dec. 1975.

[3] J. H. Patel, “Processor-memory interconnections for multiprocessors,” in *Proc. 6th Ann. Symp. on Comput. Arch.*, New York, N. Y., pp. 168–177, Apr. 1979.

[4] G. F. Pfister, W. C. Brantley, D. A. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, and J. Weiss, “The IBM Research Parallel Processor Prototype (RP3): introduction and architecture,” in *Proc. Int'l Conf. on Parallel Processing*, St. Charles, Il., pp. 764–771, Aug. 1985.

[5] D. J. Kuck, E. S. Davidson, D. H. Lawrie, and A. H. Sameh, “Parallel supercomputing today and the Cedar approach,” *Science*, vol. 231, pp. 967–974, Feb. 1986.

[6] P. Yew, “The architecture of the Cedar parallel supercomputer,” Tech. Rep. 609, University of Illinois Center for Supercomputing Research and Development, Urbana-Champaign, Illinois, Aug. 1986.

[7] C. P. Kruskal and M. Snir, “The performance of multistage interconnection networks for multiprocessors,” *IEEE Trans. Comput.*, vol. c-32, pp. 1091–1098, Dec. 1983.

[8] D. M. Dias and J. R. Jump, “Analysis and simulation of buffered Delta networks,” *IEEE Trans. Comput.*, vol. c-30, pp. 273–282, Apr. 1981.