WL-TR-93-1173

2

WRIGHT LABORATORY

ANALYSIS OF THE EFFECT OF ROUTING STRATEGIES FOR PARALLEL IMPLEMENTATIONS OF A SELECTED AVIONICS APPLICATION

JASON L. CARTER
STEPHEN L. HARY

DECEMBER 1993

INTERIM REPORT FOR 02/01/92-09/01/93

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

AVIONICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT PATTERSON AFB OH 45433-7409

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

JASON L. CARTER
Data & Signal Processing Section
Information Processing
Technology Branch
Systems Avionics Division

Jerry L Covert
JERRY L. COVERT, Chief
Information Processing
Technology Branch
Systems Avionics Division

Charl H Kruger
CHARLES H. KRUEGER, Chief
System Avionics Division
Avionics Directorate

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify WL/AAAT, WPAFB, OH 45433-7301 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE
Dec 93

3. REPORT TYPE AND DATES COVERED
Interim report for Feb 92 - Sept 93

4. TITLE AND SUBTITLE
Analysis of the Effect of Routing Strategies for Parallel Implementations of a Selected Avionics Application

5. FUNDING NUMBERS
PE: 62204F
PR: 2003
TA: 04
WU: 24

6. AUTHOR(S)
Jason L. Carter
Stephen L. Hary

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Avionics Directorate, Wright Laboratory
Air Force Materiel Command
Wright-Patterson AFB, OH 45433-7301
WL/AAAT-2 (Carter) 513-255-7709

8. PERFORMING ORGANIZATION REPORT NUMBER
WL-TR-93-1173

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Avionics Directorate, Wright Laboratory
Air Force Materiel Command
Wright-Patterson AFB, OH 45433-7301

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
WL-TR-93-1173

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The purpose of this study is to evaluate the Texas Instruments TMS320C40 (C40) digital signal processor for a selected avionics application. In this effort, the C40's on-chip communication ports are benchmarked to give indications of the amount of overhead involved in initiating a message transfer and the data throughput rates of the network. A network of C40s is compared to an iWarp parallel processor to compare the processing capabilities between the C40's store-and-forward router and the iWarp's wormhole router. Several network topologies that can easily be constructed from C40s are described, along with ways of partitioning an image processing task across each of them. Performance measurements of both the C40's and iWarp's performance on parallelized implementations of the image processing algorithm are detailed, along with estimated measurements for large-scale arrays of C40s. Finally, the C40's computing power is compared with an avionics hardware program being developed at Wright Laboratories, System Avionics Division, Information Processing Technology Branch.

14. SUBJECT TERMS
Parallel Processing, Digital Signal Processing, Routing Methods, Embedded Computer Systems.

15. NUMBER OF PAGES
66

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Table of Contents

List of Figures

List of Tables

Acknowledgments
TMS320C40, TMS320C4x, and TI are Trademarks of Texas Instruments, Inc.

OS/2 and IBM are Trademarks of International Business Machines Corporation.

Intel, i860, and iWarp are Trademarks of Intel Corporation.

1.0 Introduction

The Texas Instruments TMS320C40 (C40) Digital Signal Processor (DSP) is one of the first microprocessors available for embedded applications with extensive on-chip support for parallel processing. With six on-chip ports for direct point-to-point communication with other C40s and two independent busses, the C40 is ideally configured for parallel processing. The goal of the In-House Multiprocessor (IMP) Research Effort is to determine the usefulness of this parallel hardware in an avionics environment. This goal was not intended to be an exhaustive test of the processor's computing and communicating abilities in avionics programs but to be an evaluation of the capabilities and operating characteristics of the parallel hardware that the C40 provides in view of avionics processing requirements.

Section 2 describes the equipment used during the tests. Section 3 describes the tests that were made at the hardware level--generally small test programs that exercised one or two particular components of the C40's suite of parallel hardware. Section 4 details the tests that were made with an avionics application, which provided information on the capabilities of the processor while running a real application instead of stub programs and benchmarks. Section 5 presents future work and Section 6 presents the conclusions drawn as a result of this effort.

2.0 Testing Equipment

Several different hardware and software evaluation systems are available that are built around the C40. Most of them interface with VME bus-based workstations; the evaluation system available from TI, however, interfaces with ISA bus-based PCs. The TI evaluation system was chosen for that reason, since it made the setup and test of the processor simpler. For comparison with the C40, tests were also run on an iWarp parallel processor. The iWarp system used for testing is the iWarp machine located at Carnegie-Mellon University (CMU), Pittsburgh, PA. Section 2.1 discusses the hardware chosen for the evaluation, and Section 2.2 discusses the software used.

2.1 Hardware

TI's hardware for the evaluation system consists of a parallel processing development system (PPDS) and an ISA bus-based emulator (XDS510). Both of these devices will be described, along with a description of the internal hardware features of the C40.

The iWarp at CMU was accessed via the internet, and only the basic hardware configuration of the iWarp was used for these evaluations. The internal hardware features of the iWarp will be described in later sections.

2.1.1 The Parallel Processing Development System

The PPDS has four C40s on board, each connected by at least one communication port link to each of the other three processors. Two communication ports from each processor are brought to the edge of the board which allows connection to any of several types of devices. Each processor has 64 Kwords of Local Memory (LM) accessible through a local bus and all four processors share 128 Kwords of Global Memory (GM) accessible through a shared global bus. Figure 1 shows the connections on the PPDS.

2.1.2 The XDS510 Emulator

The PPDS is host-independent; in order to interface with it, a host-specific controller board is needed. The XDS510 is an emulator card made for the PC that gives the user access to the Joint Test Action Group (JTAG--IEEE Std. 1149.1) test port built onto the PPDS board. With the **emu4x** software discussed in Section 2.2.1, the user can control all four C40s independently through this JTAG port.

2.1.3 TMS320C40 Internal Architecture

The C40 has a number of internal devices that support parallel processing, including: six on-chip communication ports for direct point-to-point communication, a direct memory access (DMA) coprocessor for transferring data from place to place with minimal CPU intervention, two identical external busses, and a host of other features which will be discussed below.

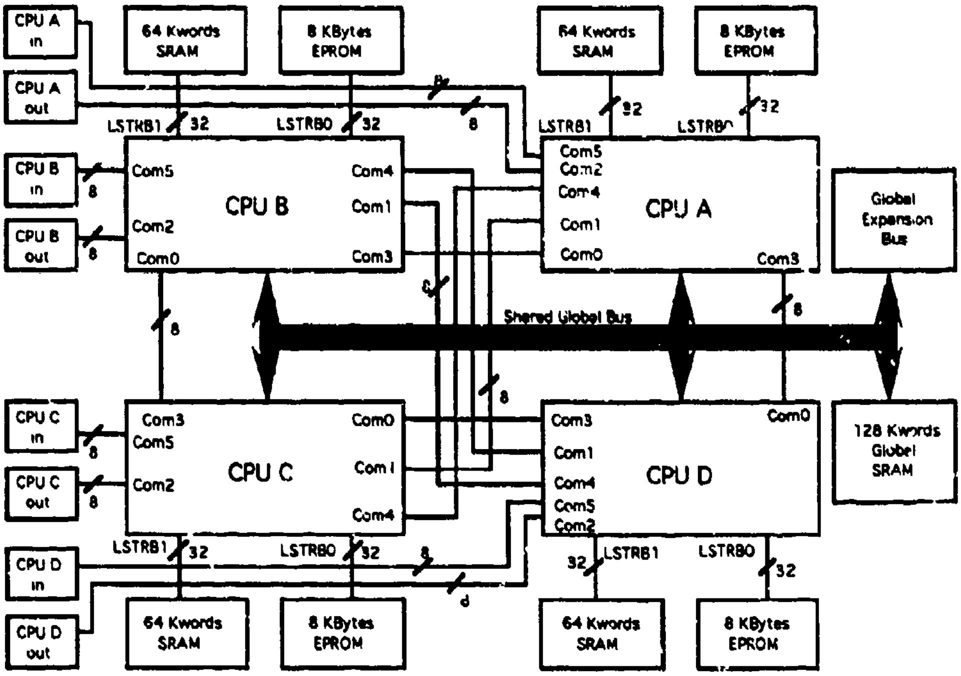


Figure 1. Communication Ports and Memory Architecture of the PPDS [1]

2.1.3.1 Communication Ports

The C40 comes equipped with six communication ports, each of which can be interfaced directly with another C40 communication port with no glue logic, as shown in Figure 2. Each port is capable of asynchronously transmitting up to 20 MBytes/sec and is capable of transmitting bidirectionally (half-duplex).

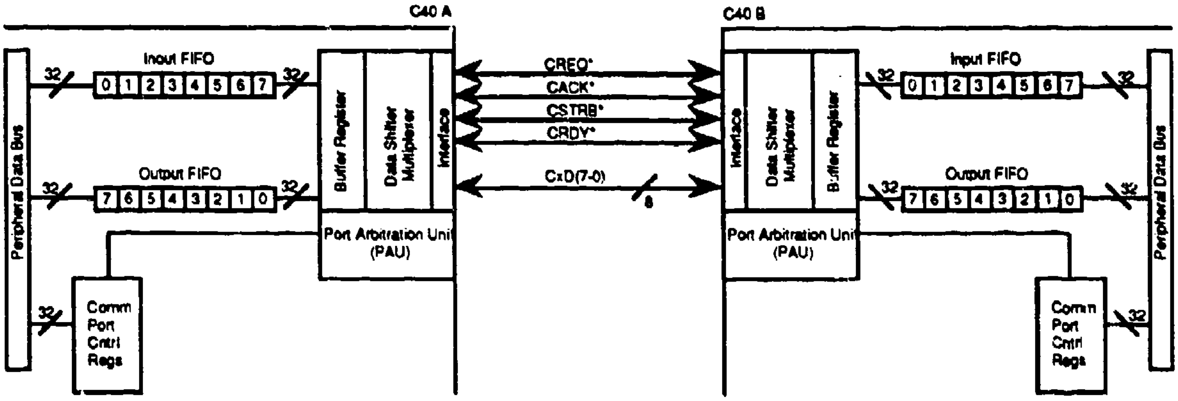


Figure 2. Details of Communication Port Hardware [2]

The port interface is composed of 12 wires: 8 for data and 4 for port control. The wires are labeled in Figure 2 according to the convention that if an asterisk (*) follows the signal name, the signal is considered an active-low signal. Thus CREQ*, CACK*, CSTRB*, and CRDY*, the signals used for port control, are all active-low signals. These four signals are used both to control the flow of data across the link and to control which port is the sending port and which port is the receiving port.

The process for controlling data flow across the link is as follows. Assume without loss of generality that CPU A has the sending port and CPU B has the receiving port. Also assume for brevity that CPU A's output FIFO is full and CPU A is ready to place the first byte of the first word to be sent on the data lines (CxD(7-0)). The data transfer sequence occurs as follows: 1) CPU A places the first byte of the data on the data lines at the first available rising edge of the internal clock; 2) CPU A brings CSTRB* low on the next rising edge of the internal clock; 3) CPU B recognizes the CSTRB* low signal and latches the data into its port; 4) CPU B brings CRDY* low to indicate that it has latched the data and that CPU A can place the next byte of the word on the data lines; 5) CPU A recognizes the CRDY* low signal and places the second byte on the data lines; 6) CPUs A and B repeat steps three through five for the remaining bytes. In this sequence, only steps one and two are synchronized to the internal clock, so the last three bytes of each word are transmitted asynchronously. This becomes significant in the benchmarking tests discussed in Section 3.1.1.1.

Each port can function as either an input or an output port, depending on how it is configured. The port's state as either a sender or a receiver is determined by ownership of a token, which is passed between the two ports as needed. (i.e. If only one port needs to send data, then no token transfers are performed, but if both ports need to send data, the token will be passed between them on a word-by-word basis.) Assume that CPU A currently has the token, and therefore is the sender. The sequence to transfer the token is as follows: 1) CPU B requests the token by bringing CREQ* low; 2) CPU A acknowledges the request by bringing CACK* low; 3) Both CPUs bring their signals high and change the status of the port direction bits in their respective port control registers.

The ports are accessed by the CPU or DMA via a memory-mapped address: to send a word, the word is written to a particular memory address that corresponds to the output port to be used; to receive a word, the word is read from another address that corresponds to the input port to be used. The output port address is a write-only address and the input port address is a read-only address. The output port corresponds to the tail of the output FIFO (described below) and the input port corresponds to the head of the input FIFO (also described below).

Each port has two independent 32-bit 8-position FIFOs for message storage--one for the output port and one for the input port. When FIFO position 0 of the output port holds a word to be sent, it is removed from the FIFO and placed in the buffer register. It is then moved to the data shifter/multiplexer to be split into four bytes before being sent one byte at a time across the link. When the first byte arrives at the receiver, the receiver buffers it and buffers each succeeding byte until all four bytes have arrived. Once the fourth byte arrives, they are combined into a single 32-bit word and sent to the input FIFO, where it can be read by the CPU or DMA. Since there are 8 positions in the sending FIFO and 8 positions in the receiving FIFO, the two FIFOs together can act as a 16-position FIFO (see Figure 2).

Systolic operations are a type of fine-grained approach to parallel processing, as will be discussed in Section 2.1.4.3. The approach used in systolic operations ıs to read data from a communication port, perform some operation on the data, and then write the result to memory. This saves time in that no extra reads from or writes to memory are required to perform a particular operation. The ability to place the data that arrives on the port directly into a CPU register is crucial to systolic operations. Since the input ports on the C40 are memory-mapped, this is a trivial operation for it. Systolic operations will be discussed in more detail in Sections 2.1.4.3 and 3.1.2.

The C40s have no innate hardware capability to automatically route data on a path from one C40 to another non-near neighbor, unless the routing was determined at compile-time (i.e., the programmer included code in the program to perform message routing). Thus, in general, the C40 is required to perform a type of message routing called store-and-forward (SAF) (However, an alternative method using software is proposed in Section 3.1.3). With SAF routing, the entire message being sent has to be stored at each intermediate C40 before it can be sent on to the next C40 along the path from source to destination. Thus it can be determined for SAF routing that the time to send a message from one C40 to the next can be expressed as:

$$t_{ms} = t_s + n{*}t_r + n{*}l{*}t_w + 2n{*}t_{r/w} + t_b, \qquad (1a)$$

(if t_w is sufficiently larger than $t_{r/w}$ so that all previously received words have been stored in memory by the time the last word arrives)

or

$$t_{ms} = t_s + n{*}t_r + n{*}l{*}t_w + n{*}t_{r/w} + n(t_w + t_{si} + t_r) + t_b, \qquad (1b)$$

(if t_w is approximately equal to $t_{r/w}$)

where

t_{ms} is the transfer time of a store-and-forward message (the time from startup on the source node to the time when the last word is stored in memory at the destination),

t_s is the setup time of the message at the source node,

n is the number of nodes after the source (including the destination),

t_r is the time a word spends in the router (after being retrieved from memory),

l is the length of the message (in words),

t_w is the time to transfer a word across a physical link,

$t_{r/w}$ is the time to read or write a word to memory (read and write times assumed identical for simplicity) (includes time to transfer from memory to router and vice versa),

t_b is the blocking time in the network, and

t_{si} is the setup time for the message at each intermediate node.

Figure 3 shows timing diagrams to demonstrate the derivation of these equations. (For all the timing diagrams used in this report, the time being measured is the time from when setup begins on the sending processor until the time when the last word is stored to memory on the receiving processor. Solid lines show time elapsed on each node, and dashed lines reference timelines above or below. Read/write terms in brackets represent items that are stored into memory before the next word arrives--thus only the last word to arrive has to be explicitly accounted for in the timing. Although the diagrams only show a small number of processors, the equations extrapolate the timings to include a set of n nodes.)

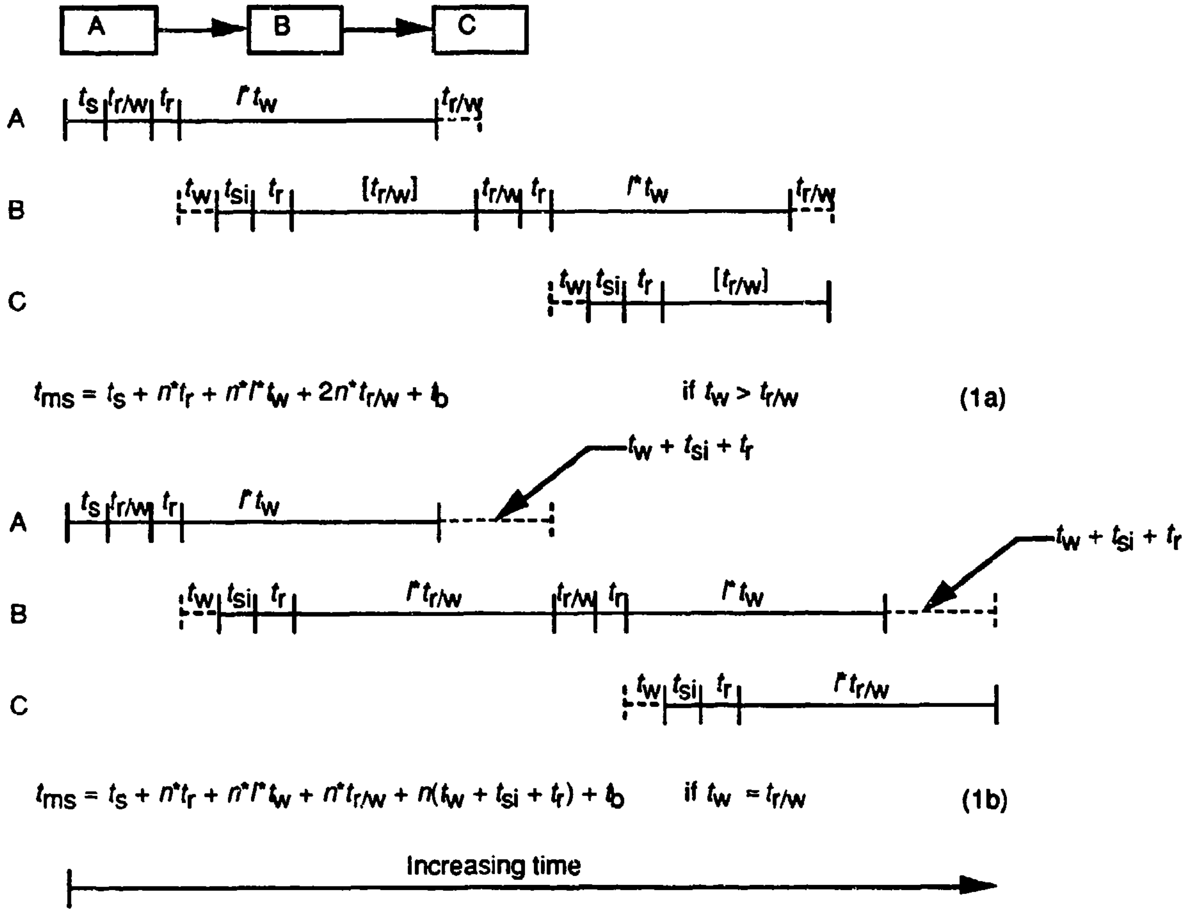


Figure 3. Timing Diagrams Used to Derive Equations 1

Store-and-forward routing will be discussed further in Section 2.1.4.1.

Since the ports can be directly connected together, and since the six ports are identical in operation, a large number of homogeneous parallel processing networks can be constructed, including rings, trees, hypercubes, two-dimensional meshes, and three-dimensional rectangular grids. Some of these topologies will be analyzed later in Section 4.2.

2.1.3.2 Direct Memory Access Coprocessor (DMA)

A six-channel DMA coprocessor is available for overlapped CPU and I/O operations (only one port can be used at a time, however, which will be discussed later). The DMA can transfer an arbitrary amount of information from any place in the memory map to any other place in the memory map, all without CPU intervention, except to set up the DMA and start it.

The DMA has nine registers per channel that are used to perform a data transfer. The registers are: 1) DMA Channel Control Register, 2) Source Address Register, 3) Source Address Index Register, 4) Transfer Counter Register, 5) Destination Address Register, 6) Destination Address Index Register, 7) Link Pointer Register, 8) Auxiliary Transfer Counter Register, and 9) Auxiliary Link Pointer Register. For any DMA action, at le·st the first six registers have to be set up by the CPU. The Link Pointer Register and the Auxiliary Registers are used only for specific functions of the DMA (the Link Pointer Register will be discussed further; the Auxiliary Registers are not crucial to the discussion at hand, therefore they will not be discussed further).

In order to start the DMA, the CPU must establish a data structure that holds the information that will be used to program the DMA. This information must include the Control Register configuration, the source address, the index value used to increment the source address, the destination address, the index value used to increment the destination address, and the number of words to be transferred (held in the Transfer Counter Register). If so desired, the other three registers may be defined also. To start DMA operation, the CPU writes the Control Register configuration into the Control Register, and the DMA begins its operation at that point.

The Link Pointer Register is used in the process of autoinitialization, which allows the DMA to follow a chain of pointers to data structures that contain the register values that tell the DMA what to do. In this way, the DMA can operate independently of the CPU, once started, and transfer an arbitrarily large amount of data without requiring any assistance whatsoever from the CPU, which allows the CPU to enjoy maximum sustained computing performance. To autoinitialize, the CPU must set up the chain of register data structures and connect them with link pointers, then tell the DMA where the first data structure is. This autoinitialization capability really shows its worth when data can be moved between locations on-chip (e.g., on-chip RAM, comm ports, etc.), since the DMA and CPU do not share a common internal bus. The two independent CPU and DMA busses allow both to operate in a cooperative rather than competitive manner. If the CPU and DMA both need data that is contained in an off-chip location, however, they will have to compete for bus access, but the user can define one of three priority schemes for arbitration: 1) CPU always has priority; 2) CPU and DMA share access to the bus equally; or 3) DMA always has priority.

The setup time for autoinitialized DMA operations can be significant for the CPU--often in the multiple hundreds of clock cycles. For simple data transfers without autoinitialization, the DMA setup time can be in the range of 150 clock cycles (see Section 3.1.1.2). Most of this setup time cannot be significantly reduced, since it is concerned with setting up the data structures to program the DMA. When autoinitialized transactions are initiated, a separate data structure must be set up for each transaction in the autoinitialization sequence, thus setup times can become very large for this type of transaction.

Each of the DMA channels can be assigned to a communication port to allow for the maximum transfer of data to and from the processor via the communication ports. The DMA channels are assigned one-to-one with the communication ports to keep things simple (i.e., DMA channel 0 is assigned to communication port 0, DMA channel 1 is assigned to communication port 1, etc.) In this way, all communication ports can interact with the DMA, if required. Again, the DMA does not require CPU intervention to interface with the communication ports, (once programmed) which allows the maximum amount of both computation and communication to occur concurrently.

Since the DMA can only use one channel at a time, there are two internal channel service priority schemes: fixed and rotating. Under the fixed priority scheme, DMA channel 0 is assigned the highest priority and DMA channel 5 is assigned the lowest priority. In this mode, the highest priority channel requesting service will be serviced first, regardless of how long the lower priority channels have been waiting. Under the rotating priority scheme, after reset DMA channel 0 has highest priority and DMA channel 5 has lowest priority. When a channel requests service, the highest priority channel will be serviced first. After it has completed its transaction, the highest priority channel will then be the channel immediately below the just-serviced channel (i.e., if DMA channel 2 has just been serviced, then DMA channel 3 is now the highest priority channel). The priority scheme rotates in this manner until no services are requested. All services are either one read access or one write access. Due to this characteristic, for the rotating priority scheme, if several channels are requesting both reads and writes, the highest priority channel will perform its read, then move to the bottom of the priority list while the next-highest priority channel performs its read. When all channels have finished reading, then the first channel may perform a write operation, and then move once again to the bottom of the priority list. Then the other channels may perform their write operations. This process will continue until all channels requesting service have completed their transactions.

2.1.3.3 Local and Global Busses

The C40 has two identical busses (except for notation), one for local memory access and one for global memory access. Each processor can access up to four Gwords of memory, the bottom half of which (addresses 0x00000000h to 0x7FFFFFFFh) map to on-chip locations and local memory, the top half of which (addresses 0x80000000h to 0xFFFFFFFFh) map to global memory (see Figure 4). Each processor can have its own local memory and can have its global bus connected to other processors' global busses, thereby allowing access to a global shared memory, as shown in Figure 1. Access to the global shared memory is controlled by a separate global bus controller.

Both busses have the capability of simulating atomic operations by using a locked-bus transaction. In these transactions, the bus is locked by an interlocked read and continues to be locked until an interlocked store occurs. In these cases, any number of operations can be performed on information held in global memory without the chance of intervention by other processors. Interlocked instructions are vital to the performance of shared-memory parallel processors, since they are used for synchronization and lockouts. When several processors have to share data or wait until a particular event occurs, often either a barrier synchronization primitive or a variant of a test-and-set primitive is used [3]. These types of primitives depend upon atomic operations so that race conditions do not develop and so that incorrect data are not accessed. For the C40s, atomic operations can be effectively implemented by locking the bus so that only one processor has access to the bus at a time.

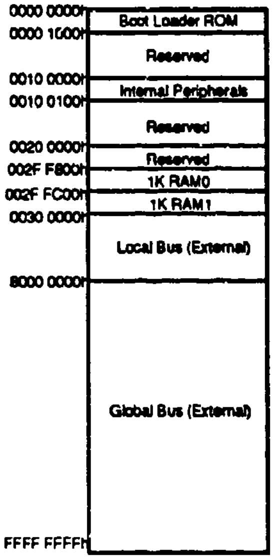


Figure 4. C40 Memory Map [2]

2.1.3.4 Miscellaneous Features of the TMS320C40

The C40 has a 512-byte on-chip instruction cache for quick access to program data. No data cache is provided, presumably to avoid cache coherency problems. Each C40 has two 1-Kword on-chip RAM spaces where programs, data, stacks, or other program essentials can be kept for quick access (see Figure 4). The C40 has three separate internal address/data bus sets: one set for instruction accesses, one set for data accesses, and one set for DMA operation. The C40 has a register-based CPU that is composed of the following components: 1) a floating-point/integer multiplier, 2) an ALU for floating-point and integer arithmetic and logical operations, 3) four internal busses (CPU1/CPU2 and REG1/REG2), 4) auxiliary register arithmetic units (ARAUs), and 5) a register file composed of 20 32-bit registers and 12 40-bit extended-precision registers which can be used for either integer or floating point calculations. The multiplier can perform single-cycle multiplications on either 32-bit integer or 40-bit floating-point operands. The ALU can also perform single-cycle operations on 32-bit integer, 32-bit logical, and 40-bit floating-point operands. The internal busses are capable of carrying two operands from (on-chip) memory and two operands from the register file concurrently for parallel operation of the functional units. The ARAUs generate up to two addresses in a single cycle, and they are used to support addressing with displacements, index registers, and circular and bit-reversed addressing. For more information about any of the features of the C40, see the *TMS320C4x User's Guide* [2].

2.1.4 iWarp Architecture

The iWarp architecture was developed at Carnegie Mellon University (CMU) and was built by Intel Corporation. The iWarp computer located at CMU consists of a 64-node systolic array. Each iWarp node contains an iWarp component and local memory (LM). The iWarp component consists of a communication agent and a computation agent that are independently controlled. The iWarp component is shown in Figure 5. Because each agent is independent, the processor does not always have to participate in the communication process, allowing communication to proceed without disturbing computation. The computation agent is a 32-bit processor with floating point and integer/logic units. The communication agent is connected to each of its four nearest neighbors by two unidirectional physical links that are eight bits wide. The communication agent and computation agent are connected through the Streaming/Spooling Unit (SSU). The gate is used for systolic processing operations (see Section 2.1.4.3) [4].

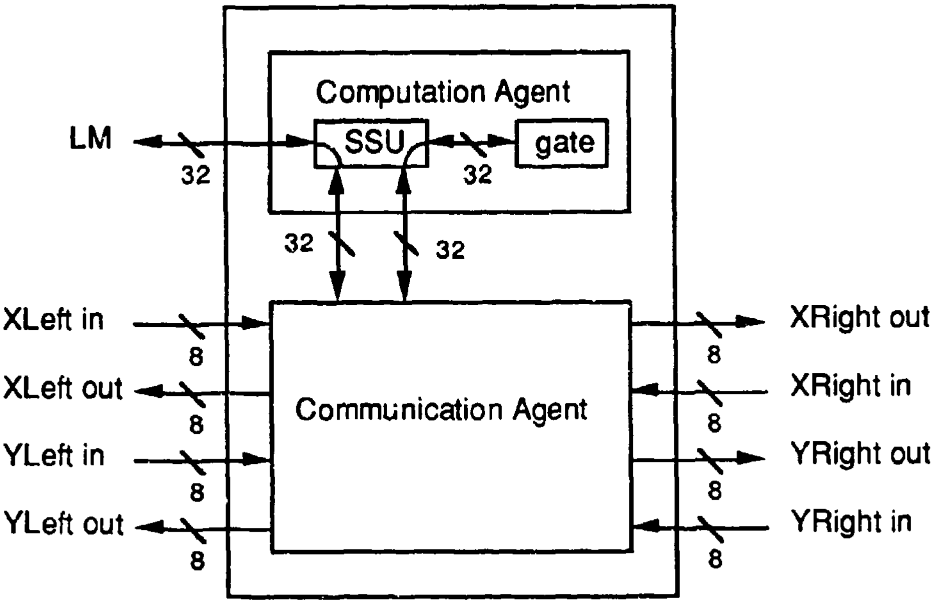


Figure 5. iWarp Component [4]

2.1.4.1 Wormhole Routing

The iWarp network is connected as a two-dimensional mesh with point-to-point links. The communication agent uses wormhole routing to transfer messages. Wormhole routing allows any node in the network to be connected to any other node even if they are not near neighbors. Figure 6 shows a connection between a source node (0,0) and a destination node (1,3) that are non-nearest neighbors and do not have a direct connection between them. In order to achieve this connection, the intermediate nodes (0,1), (1,1), and (1,2) need the capability to allow messages to flow through the node without being consumed. The iWarp communication agent has special hardware that reads header words from an incoming message and determines what to do with the message. For example, if a message is coming in the **XLeft in** link in Figure 5 then there are five possible destinations for this message: **XRight out, XLeft out, YLeft out, YRight out**, and the node itself. iWarp allows all of these connections to be made [5].

Once all the connections for a message have been set up from source to destination, the body of the message can be sent. Wormhole routers break messages into pieces called flits [6]. The size of a flit is implementation-dependent, but is often one word or less. The flits are sent one-by-one through the connections that have been established for the route. In this way, the set of connections forming the route can be thought of as a pipeline, where each link is a stage in the pipeline and each stage contains one flit. After a message has been set up and the pipeline from source to destination is full, words are transferred at a rate equal to the rate for a near neighbor connection regardless of the distance between source and destination.

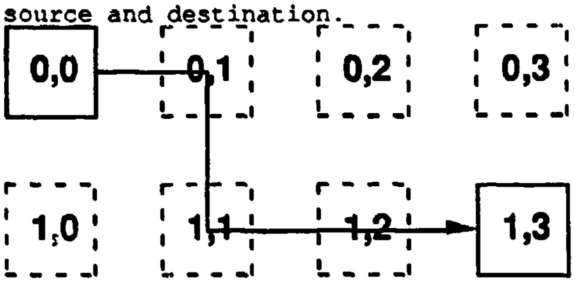


Figure 6. Non-Nearest Neighbor Connection

Wormhole routing is significantly different from older store-and-forward techniques. For SAF routing, the entire message needs to be buffered in memory on each node along the path from source to destination. There is no pipeline, therefore transfer time from source to destination is a function of the number of nodes the message needs to traverse, as shown in Equation 1.

In contrast, the transfer time between source and destination for a wormhole routed message is:

$$t_{mw} = t_s + t_r + l*t_w + (n-1)(t_w + t_{si} + t_r) + 2t_{r/w} + t_b, \qquad (2a)$$

(if t_w is sufficiently larger than $t_{r/w}$ so that all previously received words have been stored in memory by the time the last word arrives)

or

$$t_{mw} = t_s + t_{r/w} + t_r + l*t_w + n(t_w + t_{si} + t_r) + t_b, \qquad (2b)$$

(if t_w is approximately equal to $t_{r/w}$)

and the time to transfer an SAF message (from Section 2.1.3.1) is

$$t_{ms} = t_s + n*t_r + n*l*t_w + 2n*t_{r/w} + t_b, \qquad (1a)$$

$$t_{ms} = t_s + n*t_r + n*l*t_w + n*t_{r/w} + n(t_w + t_{si} + t_r) + t_b, \qquad (1b)$$

where

t_{mw} is the transfer time of a wormhole routed message (the time from startup on the source node to the time when the last word is stored in memory at the destination),

t_{ms} is the transfer time of a store-and-forward message (the time from startup on the source node to the time when the last word is stored in memory at the destination),

t_s is the setup time of the message at the source node,

$t_{r/w}$ is the time to read or write a word to memory (assumed identical for simplicity) (includes time to transfer from memory to router and vice versa),

t_r is the time a word spends in the router (after being retrieved from memory),

l is the length of the message (in words),

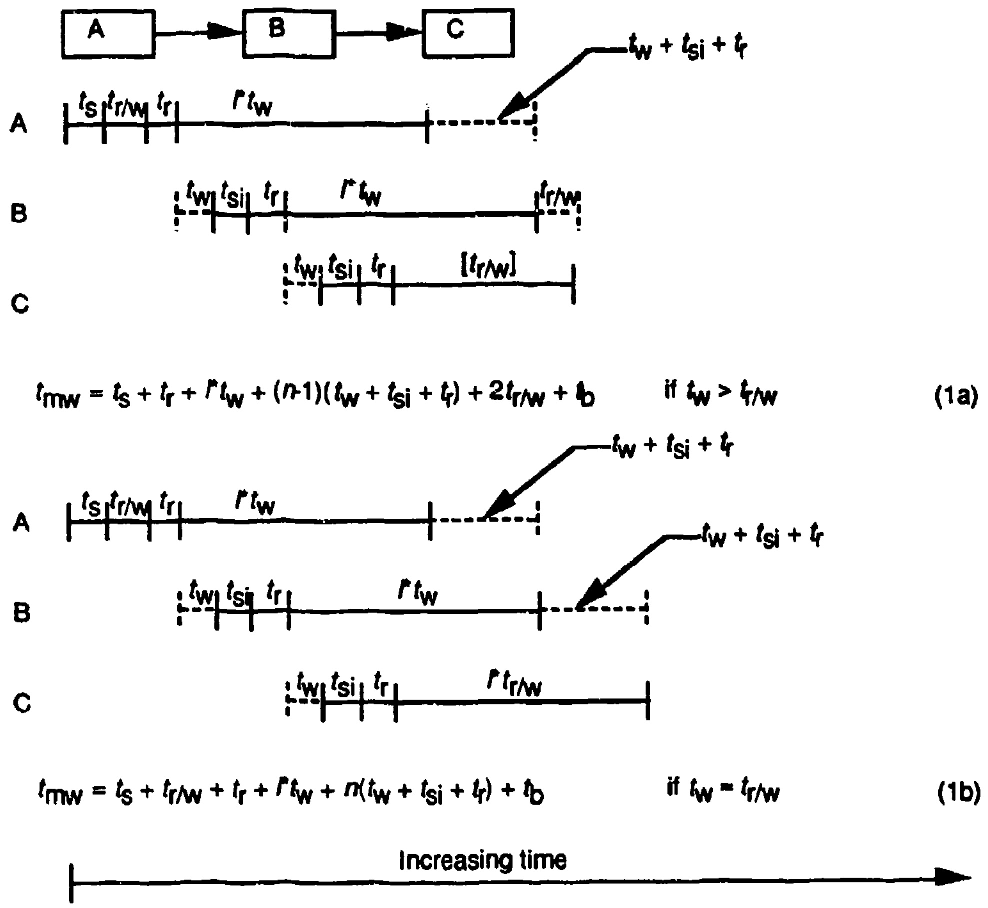
t_w is the time to transfer a word across a physical link,

n is the number of nodes after the source (including the destination),

t_{si} is the setup time for the message at each intermediate node, and

t_b is the blocking time in the network.

Figure 7 shows timing diagrams to demonstrate the derivation of these equations.



A
B
C
$t_w + t_{si} + t_r$
A
t_s $t_{r/w}$ t_r $l t_w$
B
t_w t_{si} t_r $l t_w$ $t_{r/w}$
C
t_w t_{si} t_r $[t_{r/w}]$
$t_w + t_{si} + t_r$
A
t_s $t_{r/w}$ t_r $l t_w$
$t_w + t_{si} + t_r$
B
t_w t_{si} t_r $l t_w$
C
t_w t_{si} t_r $l t_{r/w}$
Increasing time

$$t_{mw} = t_s + t_r + l t_w + (n\text{-}1)(t_w + t_{si} + t_r) + 2t_{r/w} + t_b \qquad \text{if } t_w > t_{r/w} \qquad (1a)$$

$$t_{mw} = t_s + t_{r/w} + t_r + l t_w + n(t_w + t_{si} + t_r) + t_b \qquad \text{if } t_w = t_{r/w} \qquad (1b)$$

Figure 7. Timing Diagrams Used to Derive Equations 2

The two routing methods should perform nearly equally for a nearest neighbor communication (n=1). Assuming sufficiently long messages and no blocking, the setup times should become negligible compared to the word transfer times and the transfer time for a wormhole routed message becomes dominated by the $l*t_w$ component while the transfer time for an SAF routed message becomes dominated by the $n*l*t_w$ component. Clearly, wormhole routing should outperform SAF routing when sending messages to non-nearest neighbors.

2.1.4.2 Virtual to Physical Mapping

One of the shortfalls of wormhole routing is that a single message needs control of all the physical links between its source and destination. This can cause other messages to block and may even cause deadlock in the system. Blocking can be reduced by introducing virtual channels. Each physical link can have multiple virtual channels associated with it. Each virtual channel requires its own queue, generally in the form of a FIFO buffer. As an example, Figure 8 shows links between two nodes with two virtual channels for each physical channel. Each message is assigned a virtual channel for each physical link it requires. The physical link is multiplexed between the virtual channels on an equal basis. Thus, virtual channels increase the degree of connectivity of a network without dramatically increasing hardware. For iWarp, 20 virtual channels are available in a pool for use by any link that needs them. The use of virtual channels gives woımhole routing an advantage over circuit switched routing. In circuit switched routing, the entire path from source to destination is secured before any data are sent. Once data begins transferring there is no need to buffer data at intermediate nodes, so transfer is very fast. However, circuit switching requires that a message have complete control of all links it uses which increases the probability that other messages will block.

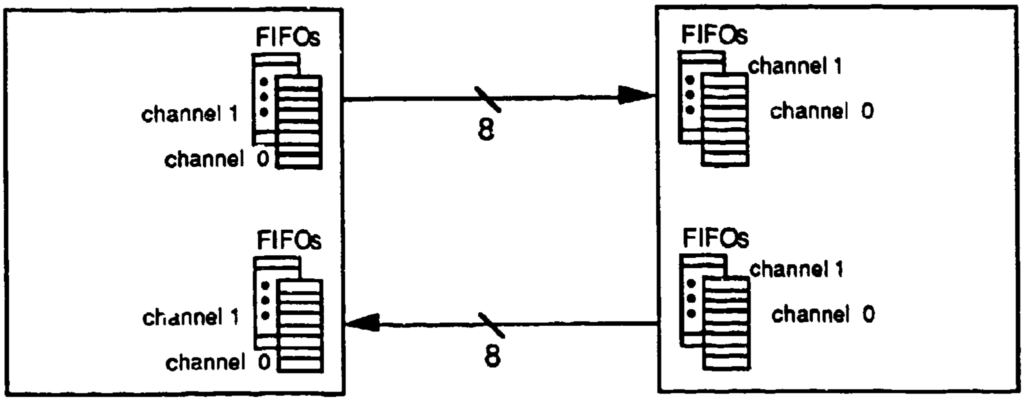


Figure 8. Virtual to Physical Mapping

2.1.4.3 Systolic Processing

Another important part of the iWarp architecture is its support for systolic processing. Systolic processing is a very fine-grained approach to parallel processing, where, rather than just transferring an incoming message directly to memory, systolic processing allows computations to be performed on the incoming data before the data are stored to memory. In this way, systolic operations can reduce the number of costly reads and writes to local memory.

iWarp supports two modes of communication, streaming and spooling. Spooling is very much like traditional message passing, in that the communication agent sends incoming data directly to the local memory of its node. This can be thought of as a DMA-like operation. The CPU sets up the communication agent to send data to or receive data from local memory and then the communication agent performs the transfer.

In streaming, the communication agent sends incoming messages directly to a stream gate in the computation agent. The stream gate is read like a CPU register. There are two input stream gates and two output stream gates on each iWarp CPU. The input gates are read-only and the output gates are write-only. Streaming can be very efficient because data do not have to be buffered in local memory before they are accessed. However, streaming requires a very tightly-coupled algorithm. If an incoming data element is not present on the stream gate when the CPU requires it, then the CPU spin-waits until the data are present. Also, if the CPU does not read the data soon after it appears on an input stream gate, the input FIFO on the receiving node and the output FIFO on the sending node could fill up, causing the message transfer to be halted for that message (i.e., if message A fills up its FIFOs, then message A will block, but message A's blocking does not inhibit the progress of message B, unless there are no virtual channels available for message B's use).

The two stream gates can be thought of as "consumption" and "injection" channels. A consumption channel is an interface between the router and the CPU, and an injection channel is an interface between the CPU and the router. Each stream gate can be configured as either a consumption or an injection channel. The iWarp therefcre has two consumption/injection channels that can be appropriately mapped to any of the eight physical I/O channels (four input and four output).

The iWarp communication agent and the C40 communication ports have many similarities. The DMA associated with the communication ports on the C40 can be thought of as the spooling unit of the iWarp communication agent. Both the iWarp and the C40 can also read incoming messages directly into CPU registers. Although the C40s are not advertised as systolic processors, they are certainly capable of performing systolic operations.

2.1.5 Configurable Hardware Algorithm Mappable Preprocesor (CHAMP)

Wright Laboratory's Data and Signal Processing Section (WL/AAAT-2), in conjunction with Lockheed-Sanders, is building a reconfigurable special-purpose processor that can be used for a variety of purposes. The original design criteria call for CHAMP to execute one of the Infrared Missile Warning (IRMW) algorithm suites (a representative suite is described in Section 4.1) at a frame rate of between 200 and 1000 frames per second. Since an array of C40s can be used to perform image processing tasks, and all of the IRMW algorithm suites are composed of image processing tasks, it is of interest to compare the CHAMP's performance to the C40's. Since the CHAMP prototype is still in development at the time of this writing, and since the architectural design of CHAMP differs markedly from the architectures of the C40 and iWarp, many of the analyses conducted below will not be applicable to the CHAMP system, but CHAMP will be discussed where appropriate.

CHAMP owes its reconfigurability to the fact that it will be composed of Field Programmable Gate Arrays (FPGAs), which have two virtual layers: 1) a gate array layer, and 2) a static-RAM-like layer that is used to configure the gate array layer. The devices are programmed by loading configuration information onto the RAM-like layer, which sets up the Configurable Logic Blocks (CLBs) and interconnects of the gate array layer according to the design the user specifies. In this way, a device can be built that performs many of the same functions that an Application Specific Integrated Circuit (ASIC) can perform, but at a lower cost, since FPGAs are commercial off-the-shelf (COTS) devices and are reprogrammable.

Since the devices are reprogrammable, and since the RAM-like layer takes silicon area on the device, FPGAs have both lower speed and lower gate density than ASICs, but for many functions, the difference in cost and the advantage of reprogrammability can easily outweigh the extra board space required and the comparatively lower speed achievable by the FPGAs. For more information about FPGAs, see *The Programmable Gate Array Data Book* [7].

2.2 Software

All the software used for testing the C40s was supplied by Texas Instruments, with the exception of the operating system for the PC that hosted the XDS510 Emulator. TI's PC software can run under either DOS or IBM's OS/2. With DOS, only one emulator can be run on the host PC, and thus only one processor on the PPDS can be examined at a time. However, OS/2's multitasking capability allows multiple emulators to be run concurrently, thus all four processors on the PPDS can be examined in parallel. For this reason, OS/2 was chosen as the operating system for the host PC. The software used for testing the iWarp is the software made available on the iWarp at CMU [5].

2.2.1 EMU4X Emulator

The main interface between the programmer and the PPDS is the **emu4x** emulator software (release 2.20 was used for the experiments performed here), along with the parallel debug manager (**pdm**). The emulator software runs in an OS/2 window and uses its own window-based interface which can be set by the user to any desired configuration. With this windowing interface, the user can examine C code, assembly code, the register file, a set of memory locations (anywhere in the memory map that is accessible), the program call stack, and a command window where the user enters input. Up to three windows to memory can be open concurrently, as well as up to three windows for displaying the status of watch variables.

The **pdm** is a program separate from the **emu4x** emulator that can interact with one or multiple emulator sessions. In this way, the user can direct several processors to perform specific tasks concurrently, without having to reissue each individual instruction to each emulator session. The number of processors affected and the order in which the commands are sent to the processors is fully configurable by the user. For more information, consult the *TMS320C4x C Source Debugger User's Guide* [8] and the *Parallel Debug Manager Addendum* [9].

2.2.2 TMS320C3X/40 C Compiler, Assembler, and Linker

The C compiler used was a fully ANSI-standard C compiler that translates C code into C3x or C40 assembly code, depending upon the switches used. The compiler has three levels of optimization, plus a no-optimization level in which no optimizations are performed on the code and extra debugging information is included in the symbol table. In order to allow the **emu4x** debugger access to the symbol table so that variables can be accurately represented, all programs run through the emulator must be compiled with no optimizations, and thus all the programs used in this research effort have been compiled that way. For more information, consult the *TMS320 Floating-Point DSP Optimizing C Compiler User's Guide* [10].

The assembler translates TMS320C3x or C40 assembly code into machine language object files, ready for the linker. The assembler's object files are in a format called Common Object File Format (COFF). COFF files are designed to be broken into blocks, which gives the programmer a great amount of flexibility when deciding how to write code for the TMS320 devices. This block-structured approach allows for simple integration of C source files with assembly source files. This allows the programmer the ability to write the bulk of the program in a high-level language like C and only write the hardware-specific functions or time-critical functions in assembly language for optimal performance. For more details on the assembler and the COFF format, refer to the *TMS320 Floating-Point DSP Assembly Language Tools User's Guide* [11].

The linker takes specified COFF object files and joins them together into a single executable file. The assembler and the linker work together in that they define and place sections of code. A section is a part of the executable code that can be placed (as a unit) practically anywhere in the memory map that is accessible as RAM, subject to memory space limitations. Some of the sections are: *.text*, which contains the program code; *.data*, which contains initialized data; *.bss*, which reserves space for uninitialized variables; and *.cinit*, which stores absolute addresses, constants, and immediate-mode operands that are greater than 16 bits. The user also has the ability to define new sections for any particular use.

The linker accepts a large number of arguments, so it is often best to use a linker command file to specify the arguments. In this file all the necessary input file names, output file names, switches, and usable memory locations can be specified, along with information relating to where the user would like the sections placed. For more information about sections, the linker, or linker command files, see the *TMS320 Floating-Point DSP Assembly Language Tools User's Guide* [11]. The release of the C compiler, assembler, and linker used was version 4.50.

2.2.3 iWarp Software

All the software tools used to test the iWarp were developed by CMU and are supported by Intel.

2.2.3.1 iWarp C Compiler

The iWarp C compiler is an ANSI C compiler with some extensions that support direct access of the systolic gates and assembly language inlining. The compiler allows for five levels of optimization. For all the iWarp tests no optimization was used in order to present a fair comparison with the C40.

2.2.3.2 iWarp Program Communication Service (PCS)

The PCS is a collection of tools and library routines for writing iWarp programs. An iWarp program consists of an array program and cell program(s). The array program defines the way in which nodes are connected. A cell program is the program that actually runs on an iWarp node. The array and cell program(s) are written in C. For more information on the PCS, see the *External Product Specification for the iWarp Programmed Communication Services (PCS) Array Combiner* [5].

3.0 Performance Measurements of Hardware Components

Network performance can be measured using two criteria, bandwidth and latency. Bandwidth is defined as the number of bits per second that a communication system can transfer. The maximum attainable bandwidth is limited by physics (type of wires used, physical distance traveled, etc.). The effective bandwidth will be limited by other system issues, including clock speed, contention, and the like. The C40 links have a maximum attainable bandwidth of 20 MBytes/sec per link and are bidirectional (i.e., the link can transmit 20 MBytes/sec if no token swapping is requested). The iWarp links have a maximum attainable bandwidth of 40 MBytes/sec per link and are unidirectional (i.e., the two links in one direction (e.g., output to west and input from west) can transmit 80 MBytes/sec if both are used).

Latency is defined as the delay between the request for information and the time the information is supplied to the requester. Latency can include but is not limited to: overhead for handshaking protocols, overhead due to software setup routines, and contention in the network.

Section 3.1 discusses bandwidth and latency issues for the C40 network, Section 3.2 discusses bandwidth and latency issues for the iWarp network, and Section 3.3 presents a summary of the findings for both networks.

3.1 C40 Network

Benchmarks were devised to test the capabilities of the C40 network. The benchmark described in Section 3.1.1 tested direct and DMA message passing between two near neighbor C40s for messages of varying length (1, 8, 16, 64, 128, 256, 512, 1024, and 4096 words). Message passing routines provided with the TMS320C40 C compiler were used for this benchmark. The benchmark described in Section 3.1.2 tests the C40's capabilities for systolic processing.

3.1.1 Direct vs. DMA Data Transfers

The C40s allow for two kinds of message transfers, direct and DMA. For direct transfers, the CPU has control of the communication ports and passes data through them. For DMA transfers, the CPU sets up the DMA and then the DMA has control of the communication port and performs the data transfer.

3.1.1.1 Direct Data Transfers

Equation 3 is a modification of Equation 1a (Equation 2a will also reduce to Equation 3) to represent the direct message passing benchmark. There is no blocking in the network for this benchmark so the t_b term drops out. There are no intermediate nodes, so n equals 1, and thus the transfer time for a direct message is:

$$t_{mdirect} = t_s + t_r + l*t_w + 2t_{r/w}, \qquad (3)$$

where

$t_{mdirect}$ is the time to send a direct message,

t_s is the time to set up a direct message,

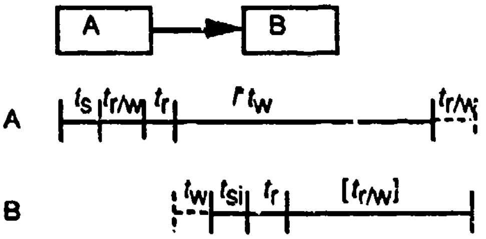
t_r is the time a word spends in the router (after being retrieved from memory),

l is the length of the message,

t_w is the per word transfer time, and

$t_{r/w}$ is the time to read or write a word to memory (assumed identical for simplicity) (includes time to transfer from memory to router and vice versa).

Figure 9 shows timing diagrams to demonstrate the derivation of these equations.



$$t_{mdirect} = t_s + 2t_{r/w} + l\,t_w + t_r \qquad (3)$$

Figure 9. Timing Diagrams Used to Derive Equation 3

The C40 communication port links have a maximum attainable bandwidth of 20 MByte/sec when the rated internal clock speed of 40 ns is used. The C40s on the PPDS have an internal clock speed of 62.5 ns. The discrepancy between the rated clock speed and the PPDS clock speed will affect the attainable bandwidth of the PPDS C40 communication ports. The C40 communication ports are only synchronized to the internal clock for token transfers and for the transfer of the first byte of each word transferred during a message, as explained in Section 2.1.3.1. After the first byte of each word is sent, the communication ports transfer the remaining three bytes in an asynchronous fashion [2].

For this benchmark, the time it takes to send one continuous message from one node to another was measured. As a result, there is at most one token transfer. For long messages the token transfer can be ignored since it happens at most once. According to the *TMS320C4x User's Guide*, the "type two" synchronization delay between the last byte of word *n* of a message and the first byte of word *n*+1 of the same message is a minimum of 1.5 machine clocks and a maximum of 2.5 machine clocks. The transfer times for best case and worst case delay are:

$$t_w = t_{byte}*3 + 1.5*t_{clk} \qquad (4)$$

and

$$t_w = t_{byte}*3 + 2.5*t_{clk}, \qquad (5)$$

where
t_w is the per-word transfer time,
t_{byte} is the time to asynchronously transfer one byte, and
t_{clk} is the internal clock speed of the C40.

The *TMS320C4x User's Guide* states the bandwidth to be 5 Mword/sec (t_w = 200 ns/word) for a 40 ns t_{clk}. Making these substitutions into Equations 4 and 5 and solving for t_{byte} yields:

$$33.3 \text{ ns} < t_{byte} < 46.66 \text{ ns}. \qquad (6)$$

t_{byte} is not only a function of the switching speed of the gates on each C40, but also a function of the physical distance between adjacent nodes. The longer the communication port data and control wires are the greater t_{byte} will be.

Assuming from Equation 6 that the average t_{byte} is 40 ns then the expected best and worst case word transfer times for the PPDS clock speed (62.5 ns) are:

$$t_{wbest} = 40 \text{ ns}*3 \text{ bytes} + 1.5*62.5 \text{ ns} = 217.5 \text{ ns} \qquad (7)$$

and

$$t_{wworst} = 40 \text{ ns}*3 \text{ bytes} + 2.5*62.5 \text{ ns} = 276.25 \text{ ns}. \qquad (8)$$

From Equation 7 the best case bandwidth ($1/t_{wbest}$) is 4.598 Mword/sec and from Equation 8 the worst case bandwidth ($1/t_{wworst}$) is 3.62 Mwords/sec. Taking the average of the best and worst case word transfer times we get a "best guess" average word transfer time of t_w = 246.875 ns which translates into 4.05 Mwords/sec for the PPDS board. Returning to the benchmark of the direct message passing function, the time to transfer a message is given in Equation 3. The value of t_w derived from Equations 7 and 8 is only an estimate; a better measure would be more useful. Table 1 shows the time it took to send word arrays of various sizes (1, 8, 16, 64, 128, 256, 512, 1024, and 4096 words) between two adjacent C40s using direct message passing. All of the message transfer equations discussed above assume that the sending and receiving nodes can access their data faster than the communication ports can transfer it (i.e., the sending node can read a word out of memory faster than the communication link can transfer it, and the receiving node can store a word to memory faster than the communication link can transfer it). Under full-speed conditions, this would be a reasonable assumption; however, it was discovered that for the PPDS system used in this test, the C40 communication ports can actually transfer data faster than the CPU can access LM. Obviously, if faster LM was used this would not be the case.

Table 1. Message Transfer Times for Near Neighbor C40s

Type of message	Using LM (cycles)	Using RAM1 (cycles)	Using LM (cycles/word)	Using RAM1 (cycles/word)
1 word send	62	60	62.000	60.000
1 word receive	60	58	60.000	58.000
8 word send	82	74	10.250	9.250
8 word receive	92	72	11.500	9.000
16 word send	212	188	13.250	11.750
16 word receive	128	88	8.000	5.500
64 word send	400	314	6.250	4.906
64 word receive	344	254	5.375	3.969
128 word send	666	544	5.203	4.250
128 word receive	632	478	4.938	3.734
256 word send	1242	992	4.852	3.875
256 word receive	1208	926	4.719	3.617
512 word send	2394	1888	4.676	3.688
512 word receive	2360	1822	4.609	3.559
1024 word send	4698		4.588	
1024 word receive	4664		4.555	
4096 word send	18522		4.522	
4096 word receive	18488		4.514	

Because the speed of the memory on the nodes affects transfer time, messages sent to and from both LM and on-chip RAM (RAM1) were benchmarked. RAM1 is very fast and can be read from and written to in one machine cycle (no wait states). Reading or writing to LM on the PPDS takes three cycles. Because of this, the messages sent to RAM1 are a better test of the communication port bandwidth and latency.

For long messages using RAM1 on the PPDS (128, 256, and 512 words), the per-word message transfer time (t_w) approaches 3.5 cycles (218.75 ns). This per-word message transfer time was calculated by measuring the time to set up the port and to send the entire message, then dividing that measured time by the number of words sent. The longer a message is, the more the setup times t_{ssend} and t_{srec} become a negligible component of the per-word transfer time. Thus, when an asymptotic limit for the per-word transfer time is determined, it can be safely assumed that words are transferred at that per-word rate for any length message. Given the per-word message transfer rate and the total time taken to set up and send a message, the setup time can be calculated. Rewriting Equation 3 to solve for t_s gives:

$$t_s = t_{mdirect} - 1*t_w - 2t_{r/w} - t_r. \qquad (3)$$

Solving for t_s using values for $t_{mdirect}$ from the second column of Table 1 gives an approximate setup time for sending a message of t_{ssend} = 96 cycles, and an approximate setup time for receiving a message of t_{srec} = 30 cycles.

The benchmark was written to call the receive function and to begin timing it after the first data word appears on the input FIFO. This helps to explain the large difference between t_{ssend} and t_{srec}. It is likely that the input FIFO on the receiving processor is already full by the time the receive routine finishes setting up. This implies that the first several words can be read from the port and stored to on-chip RAM at the rate of two cycles per word. This will make the setup time t_{srec}

appear smaller than it ought to be; nevertheless, t_{srec} is constant for long messages. Similarly, it is likely that the send function will fill up the output FIFO on the sending processor and the input FIFO on the receiving processor before the receiving processor begins to read in data. This will cause the send function to spin-wait, making t_{ssend} appear larger than it ought to be; again, similar to t_{srec}, t_{ssend} is constant for long messages.

For short messages (one to eight words), the sending processor can fill up its own output FIFO and finish processing the sending routine instructions before the communication port sends the first word across its link. This occurs because the CPU instructions in the sending routine simply read data from on-chip RAM, store it in a register, then write it to the output FIFO, which only takes two cycles per word. As a result, it only takes 14 more cycles to send an eight-word message than it does to send a one-word message. Similarly, by the time the receive message is set up all of the incoming words are already on the input FIFO and it only takes two cycles to write each word to RAM1 (one to load the incoming word into a register and one to store it to memory).

As soon as a message exceeds 16 words (8 words each for the sending and receiving FIFO), the sending routine has to wait for the receiving node to start consuming words in order to continue transmitting. Also, the message functions provided with the C40 C compiler actually send l+1 words for a message of length l, because the first word sent across a link is the length of the message in words. After this extra word is sent, the body of the message is sent. The time to send this extra word is included in the calculation of t_{ssend} and the time to receive this extra word is included in the calculation of t_{srec}, because the extra word is considered part of the message setup routines.

For long messages the PPDS board achieved a t_w of 218.7 ns which gives a bandwidth of about 4.57 Mwords/sec. This is closer to the best-case bandwidth calculated in Equation 7 than it is to the worst-case bandwidth calculated in Equation 8. This is not surprising, since the C40s are rated for 5 Mwords/sec at the 40 ns clock. On the PPDS board the communication ports of the four C40s are physically very close to one another and Printed Circuit Board (PCB) wires are used for the communication port links, therefore the asynchronous portion of a word transfer should occur as fast as the C40 chip can drive it, since there should be no propagation delays due to the links (such as if the links had to travel through connectors from one PCB to another). Also it was assumed that the probability of incurring either a best-case or a worst-case "type two" delay was equal. If the best case delay occurs more often, then the t_{byte} shown in Equation 6 would be less than the 40 ns average assumed above.

3.1.1.2 DMA Data Transfers

The setup times for a DMA send and a DMA receive were measured as t_{sdsend} = 146 cycles and t_{sdrec} = 154 cycles. These times are independent of message length; however, it should be noted that the message transfer time is not independent of message length, as shown in Section 3.1.1.1 above. The DMA transfer time for a message of a specific length was identical to the direct transfe: time for sending the same message, so the only time difference between direct and DMA transfers is their respective setup times. It is assumed for these DMA benchmarks, similar to the benchmarks discussed in Section 3.1.1.1, that there is enough overlap of computation and communication that when the receiving CPU needs data that was sent by another CPU, data will already be in the receiver's input FIFO.

It should be noted that if the DMA is used to transfer data, it is the responsibility of the programmer to ensure that the correct data are transferred. For example, the sending processor could set up the DMA to transfer a long word array and then, after the DMA has begun sending data, have the CPU change some values in that array before those values are sent. In this case the receiver would get some data that was partially stale. The C40 C compiler comes with functions to check the status of a DMA transfer (i.e., if the transfer has been completed or not). These functions should be used to check the status of any DMA transfers before data are updated.

3.1.2 C40s as Systolic Processors

As discussed in Section 2.1.3.1, C40s could be used for systolic processing. It was shown previously that it takes 3.5 cycles to transfer a word for long messages. As discussed in Section 3.1.1, the read and write operation from the communication port to memory could take as few as two cycles (if on-chip RAM is used). If it takes 3.5 cycles for each word to appear on a communication port's input FIFO and it only takes two cycles to read and store a word, then for long messages, the input routine will have to wait an average of 1.5 cycles per incoming word. These are wasted cycles that could be used for some other computation.

Systolic message routines were created to test the C40's capabilities for this type of processing. Only incoming message routines were developed for this benchmark; similar functions could be developed for outgoing messages. There are two C functions included with the TMS320C40 C compiler for reading in word-length messages, **in_msg** and **receive_msg**. The **receive_msg** function sets up the DMA to asynchronously write data from a communication channel to memory. Use of the DMA implies that the CPU gives up control of the data transfer; systolic operations require that the CPU has control of the data transfer. It follows, therefore, that using the DMA for systolic operations is not feasible. The **in_msg** function reads an incoming array word by word from the input FIFO buffer into a CPU register and writes each word into an array in memory. The **in_msg** function is the best candidate function for implementing systolic operations; therefore, **in_msg** was modified to add them. Four new systolic message reading functions were created, a scalar add (**in_mesa**), a scalar multiply (**in_mesm**), a vector add (**in_mesva**), and a vector multiply (**in_mesvm**). These functions are described below:

in_msg(int **ch_no**, void ***mess**, int **step**);
Reads data from communication port **ch_no**, to a word array that is pointed to by ***mess**, with a step size of **step**.

in_mesa(int **ch_no**, void ***mess**, int **step**, int **sc_add**);
Reads data from communication port **ch_no**, adds a scalar value (**sc_add**) to each data element and writes the result to a word array that is pointed to by ***mess**, with a step size of **step**.

in_mesm(int **ch_no**, void ***mess**, int **step**, int **sc_mult**);
Reads data from communication port **ch_no**, multiplies each data element by a scalar value (**sc_mult**) and writes the

result to a word array that is pointed to by ***mess**, with a step size of **step**.

in_mesva(int **ch_no**, void ***mess**, int **step**, int ***vec**, int **vstep**);
Reads data from communication port **ch_no**, adds elements of an array pointed to by ***vec**, with a step size of **vstep**, to each incoming data element and writes the result to a word array that is pointed to by ***mess**, with a step size of **step**.

in_mesvm(int **ch_no**, void ***mess**, int **step**, int ***vec**, int **vstep**);
Reads data from communication port **ch_no**, multiplies elements of an array pointed to by ***vec**, with a step size of **vstep**, with each incoming data element and writes the result to a word array that is pointed to by ***mess**, with a step size of **step**.

These functions were used to transfer arrays of length 1, 16, 128, 256, 1024, and 4096 words to LM and RAM1.

Table 2 shows the number of cycles it took to perform each function for different message lengths and different destinations (LM or RAM1). Table 3 shows the number of cycles/word it took to perform the functions in Table 2. These benchmarks began timing after data first appeared on the input FIFO. Therefore, they include the time required to set up and call the function.

Table 2. Systolic Operation Times for C40s

Type of message	Receive message	Receive with scalar add	Receive with scalar mult	Receive with vector add	Receive with vector mult
1 word to LM	60	64	64	74	74
1 word to RAM1	58	62	62	70	70
16 words to LM	128	154	154	194	194
16 words to RAM1	88	106	106	130	130
128 words to LM	632	826	826	1090	1090
128 words to RAM1	478	482	482	578	578
256 words to LM	1208	1594	1594	2114	2114
256 words to RAM1	926	930	930	1090	1090
1024 words to LM	4664	6202	6202	8258	8258
4096 words to LM	18488	24634	24634	32834	32834

Table 3. Per-Word Systolic Operation Times for C40s

Type of message	Receive message	Receive with scalar add	Receive with scalar mult	Receive with vector add	Receive with vector mult
1 word to LM	60.000	64.000	64.000	74.000	74.000
1 word to RAM1	58.000	62.000	62.000	70.000	70.000
16 words to LM	8.000	9.625	9.625	12.125	12.125
16 words to RAM1	5.500	6.625	6.625	8.125	8.125
128 words to LM	4.938	6.453	6.453	8.516	8.516
128 words to RAM1	3.734	3.766	3.766	4.516	4.516
256 words to LM	4.719	6.227	6.227	8.258	8.258
256 words to RAM1	3.617	3.633	3.633	4.258	4.258
1024 words to LM	4.555	6.057	6.057	8.064	8.064
4096 words to LM	4.514	6.014	6.014	8.016	8.016

When writing to RAM1, the scalar add and multiply can be performed with no penalty. The vector add and multiply take an extra 0.5 cycles when writing to RAM1. This is because it takes 2 extra cycles to perform the vector operations, one to get the operand out of RAM1 and one to perform the operation, and there are only 1.5 cycles for systolic operation. When storing to LM, scalar operations add 1.5 cycles. This occurs because it takes a total of 5 cycles to perform these operations (four get the data out of the input FIFO and store it to LM plus one cycle to perform the scalar operation). Vector operations to LM add 3.5 cycles. This is because it takes a total of 7 cycles to perform these operations (four get the data out of the input FIFO and store it to LM, plus three cycles to get the operand out of memory and perform the operation).

For systems running at the rated clock speed (40 ns), a message transfer would take 5 cycles/word rather than 3.5 cycles/word. For this type of system, systolic operations would be even more useful, because there would be more unutilized cycles with which to perform systolic operations. Also, internal clock cycle times for microprocessors are continuing to drop. The C40s rated cycle time of 40 ns is not excessively fast by today's standards. It seems more likely that the on-chip clock speed of the C40 will increase than it is that the communication port transfer speed will increase. In light of this, it is possible that the number of CPU cycles needed to transfer a word will be greater than five in the future, allowing even more cycles for use in systolic operations.

3.1.3 C40 Observations

The C40s have a high bandwidth (160 Mbits/sec for a 40 ns clock), and relatively low latency for sending point-to-point messages (roughly 100 cycles for message function setup). The low latency for point-to-point messages is partially a result of the fact that each communication port has its own memory-mapped registers and DMA channel, and thus resources do not need to be dynamically allocated to perform communications.

One major thing lacking in the design of the C40 is support for flow-though messages. In order to send a message to a non-nearest neighbor node, the message would have to be stored and forwarded at each intermediate node. This increases latency dramatically. One possible solution is to have the intermediate processor perform the store and forward in a systolic fashion. That is, the intermediate node(s) could read data from an incoming FIFO into a register and write the data to the appropriate outgoing FIFO without buffering the data in memory, thus reducing latency to near zero. This would require a significant amount of software design to generate the proper functions, and would have the drawback of using up the intermediate nodes' CPU resources for communication instead of computation. This type of routing software is mentioned in Section 5.1 as future work.

3.2 iWarp Network

Two benchmarks similar to the C40 benchmarks were also devised for the iWarp Network. The benchmark described in Section 3.2.1 tested the streaming and spooling message functions for sending messages of varying length (1, 8, 16, 64, 128, 256, 512, 1024, and 4096 words) between near-neighbor nodes. The second iWarp benchmark described in Section 3.2.2 was created to test the latency of the wormhole router at each node. The benchmark routine sends messages of varying length (1, 16, 128, 512, 1024, and 4096 words) between nodes that were 1 hop (near neighbors), 2 hops, 3 hops, 4 hops, 5 hops, and 6 hops from each other.

3.2.1 Streaming vs. Spooling

The first benchmark was designed to compare streaming and spooling. Messages of varying length (1, 8, 16, 64, 128, 256, 512, 1024, and 4096 words) were sent between two near neighbor nodes using both streaming and spooling. Each iWarp node CPU has two stream gates that can read in data directly from the communication unit. These gates need to be bound to the proper network connection in order for data to be transferred to the proper place. The number of network connections any node can have is not limited. Therefore, each node could be logically connected to every other node in the network; however, because there are only two stream gates, only two of these connections can be bound at any one time. The network connections to be used during program execution must be defined at compile time. Any of these predefined network connections can be bound during execution to either stream gate in a given iWarp node, allowing the desired communication to take place.

The time required to send a message using streaming is a modification of Equation 2a, given in Equation 9. For this example there is no blocking, so the t_b term drops out, and the message is being sent to a near neighbor (n=1), so the $(n-1)(t_w + t_{si} + t_r)$ term drops out. The setup time has been partitioned into the time to bind a stream gate (t_{bind}), the time to unbind a stream gate (t_{unbind}) and the time to set up the stream message (t_{sst}).

The time required to send a message using spooling is given in Equation 10. As in the streaming case described above, there is no blocking and the messages are being transferred between near neighbors. Because spooling is performed in the background by the communication agent, the time to set up a spool message is independent of the message length l. There is additional setup time for spooling (t_{ssp}), however. Not only do the stream gates need to be bound (t_{bind}) and unbound (t_{unbind}), but also the spool gates need to be bound (t_{bsp}) and unbound (t_{ubsp}). The transfer times for a stream message and a spool message, then, are:

$$t_{stream} = t_{bind} + t_{unbind} + t_{sst} + 2t_{r/w} + t_r + l*t_w \qquad (9)$$

and

$$t_{spool} = t_{bind} + t_{unbind} + t_{bsp} + t_{ubsp} + t_{ssp} + 2t_{r/w} + t_r + l*t_w, \qquad (10)$$

where

t_{stream} is the time required to send a stream message,
t_{spool} is the time required to send a spool message,
t_{bind} is the time required to bind a stream gate and open a message,
t_{unbind} is the time required to close a message and unbind a stream gate,
t_{sst} is the time required to set up a stream message,
l is the length of the message,
t_w is the time required to send a word across a physical link,
t_{bsp} is the time required to bind a spool gate,
t_{ubsp} is the time required to unbind a spool gate, and
t_{ssp} is the time required to set up a spooling message.

Table 4 shows the number of cycles required to send and receive messages of varying length using streaming. From Table 4, it takes about 300 cycles to bind a stream gate and about 240 cycles to unbind a stream gate. Table 4 also shows the per-word transfer time for each message length. Two per-word times are given: one that includes the overhead time to bind the gates, and one that only includes the time to set up the messages. This is because once a network connection has been bound to gates on the CPU any number of messages can be sent along that connection.

Table 4. Times to Send Streaming Messages on iWarp

Type of message	Bind and open gates	Release gates	Transfer time	Transfer time per word	Transfer time per word inc. setup
Send 1 word	302	240	149	149.0000	691.0000
Receive 1 word	328	238	138	138.0000	704.0000
Send 8 words	302	240	134	16.7500	84.5000
Receive 8 words	328	238	126	15.7500	86.5000
Send 16 words	302	240	146	9.1250	43.0000
Receive 16 words	328	238	142	8.8750	44.2500
Send 64 words	302	240	243	3.7969	12.2656
Receive 64 words	328	238	238	3.7188	12.5625
Send 128 words	302	240	370	2.8906	7.1250
Receive 128 words	328	238	370	2.8906	7.3125
Send 256 words	302	240	648	2.5313	4.6484
Receive 256 words	328	238	650	2.5391	4.7500
Send 512 words	302	240	1154	2.2539	3.3125
Receive 512 words	328	238	1174	2.2930	3.3984
Send 1024 words	302	240	2166	2.1152	2.6445
Receive 1024 words	328	238	2162	2.1113	2.2658
Send 4096 words	302	240	8310	2.0288	2.1611
Receive 4096 words	328	238	8306	2.0278	2.1660

Each iWarp link has a bandwidth of 40 MBytes/sec, and each iWarp CPU runs at 20 MHz. Therefore, a link can transfer a four-byte word in two cycles (t_w = 2). The time required to set up a send message using streaming can be calculated using Equation 11 below. This equation is simply the total time it takes to send a message minus the amount of time it takes to send the data across the links ($2*l$ because t_w = 2). The total time required to send several messages is shown in Table 4. Since the total time and length of each of these messages is known, the setup time can be calculated. Equation 12 is a similar equation to calculate the time to set up a receive message using streaming. For long message lengths, the time to set up a send message using streaming was about 120 cycles. The time to set up a receive message using streaming was about 115 cycles for long messages. The setup times are, therefore:

$$t_{ssend} = t_{send} - 2*l \qquad (11)$$

and

$$t_{srec} = t_{rec} - 2*l, \qquad (12)$$

where

t_{ssend} is the time required to send a message using streaming,
t_{srec} is the time required to receive a message using streaming,
t_{send} is the total time to send a message,
l is the length of the message, and
t_{rec} is the total time to receive a message.

Table 5 compares the setup times for messages that use streaming and spooling (the setup times for the stream messages were calculated using Equations 11 and 12; all other times were measured). For spooling, it takes about 350 cycles to bind a spool gate and about 430 cycles to unbind a gate. It also takes longer to set up a spool message than it does to set up a stream message. If gate binding times are factored in, it takes about 1000 more cycles to set up a spool message than it does to set up a stream message. It is worth noting that although the setup time is independent of the length of the message for spooling, the actual amount of time it takes before the data reaches its destination is a function of the message length. It is assumed for these benchmarks that if streaming is used, there is enough overlap of computation and communication that when the receiving CPU needs the data that was sent, the transfer will be complete. For the spooling benchmark program a long loop was inserted between the time each message was sent and received on each node to ensure that there was enough time to complete the background transfer.

Table 5. Comparison of Streaming and Spooling Setup Times on iWarp

Type of message	Bind and open gates	Bind spool gates	Release spool gates	Release gates	Setup time	Total overhead
Send using streaming	302			240	120	662
Send using spooling	320	347	434	240	355	1696
Receive using streaming	328			238	115	681
Receive using spooling	328	338	432	238	315	1651

From Table 5 it is clear that it takes considerably more time to set up a transfer using spooling than it does to set up a transfer using streaming. Therefore, in order to effectively use streaming, longer messages need to be used in order to overcome the setup times. When sending messages of less than 128 words it is always better to use streaming because it takes as long to set up a spool transfer of 128 words as it does to actually transfer 128 words using streaming. When the gate binding overhead is factored in, it is faster to use streaming for all messages that are 512 words or smaller. This is, of course, assuming no blocking in the network and that the sending and receiving nodes are reasonably synchronized. If there are many messages in the network that can cause blocking or the sending and receiving nodes are not synchronized, then either the sending or receiving node could be forced to wait when transferring data using streaming. This could potentially be a great waste of CPU resources.

3.2.2 Multi-Hop Messages

The second iWarp benchmark was created to test the latency of the wormhole router at each node. A benchmark routine was created to send data of varying length (1, 8, 16, 64, 128, 256, 512, 1024, and 4096 words) between nodes that were 1 hop (near neighbors), 2 hops, 3 hops, 4 hops, 5 hops, and 6 hops from each other. Streaming was used for all of these benchmarks. The iWarp uses wormhole routing, therefore it should follow the data transfer model given in Equation 2a. For this benchmark, there is no blocking and the t_w term is known to be two cycles. Making these substitutions into Equation 2a yields:

$$t_m = t_s + t_r + 2l + (n-1)(2 + t_{si} + t_r) + 2t_{r/w}. \qquad (13)$$

Table 6 shows the message transfer time for the sending and receiving node. Table 7 shows the per-word average transfer time for each message. All of the times in Table 7 only include setup and transfer time; the overhead associated with binding a connection is not included. Therefore, the t_s term in Equation 13 only includes the time to set up a message and not the binding overhead. As expected, the message times were almost completely independent of distance traveled (since this benchmark had no network blocking). One can conclude from this that the t_{si} term in Equation 13 is negligible for the iWarp router. Another conclusion that can be made is that the advertised iWarp bandwidth of 40 MBytes/sec per link is correct, because for long messages the per-word transfer time approaches two cycles per word.

Table 6. Results of Multi-Hop Latency Benchmark on iWarp

Type of message	1 hop	2 hops	3 hops	4 hops	5 hops	6 hops
Send open connection	408	405	406	406	406	406
Receive open connection	609	603	597	598	594	591
Send 1 word	144	141	141	141	140	138
Receive 1 word	136	141	139	139	140	141
Send 16 words	201	158	158	158	158	158
Receive 16 words	144	145	143	143	143	142
Send 128 words	512	561	554	555	544	542
Receive 128 words	368	370	371	371	373	371
Send 512 words	1152	1160	1166	1166	1166	1166
Receive 512 words	1144	1149	1153	1158	1150	1151
Send 1024 words	2168	2189	2191	2190	2200	2196
Receive 1024 words	2160	2186	2194	2188	2196	2191
Send 4096 words	8336	8426	8428	8455	8441	8446
Receive 4096 words	8344	8426	8430	8452	8445	8448
Send close connection	232	232	232	232	232	232
Receive close connection	248	246	246	245	247	247

Table 7. Per-word Transfer Time for the Multi-Hop Latency Benchmark on iWarp

Type of message	1 hop	2 hops	3 hops	4 hops	5 hops	6 hops
Send 1 word	144.000	141.000	141.000	141.000	140.000	138.000
Receive 1 word	136.000	141.000	139.000	139.000	140.000	141.000
Send 16 words	12.563	9.875	9.875	9.875	9.875	9.875
Receive 16 words	9.000	9.063	8.938	8.938	8.938	8.875
Send 128 words	4.000	4.383	4.328	4.336	4.250	4.234
Receive 128 words	2.875	2.891	2.898	2.898	2.914	2.898
Send 512 words	2.250	2.266	2.277	2.277	2.277	2.277
Receive 512 words	2.234	2.244	2.252	2.262	2.246	2.248
Send 1024 words	2.117	2.138	2.140	2.139	2.148	2.145
Receive 1024 words	2.109	2.135	2.143	2.137	2.145	2.140
Send 4096 words	2.035	2.057	2.058	2.064	2.061	2.062
Receive 4096 words	2.037	2.057	2.058	2.063	2.062	2.063

3.2.3 iWarp Observations

The iWarp communication links have a very high bandwidth (320 Mbits per second), and the wormhole router has very low latency for pass-through messages. However, the large overhead associated with binding the CPU's stream gates to network connections increases latency and makes the two stream gates a potential bottleneck. For algorithms that require each node to access multiple network connections, the overhead associated with switching from one connection to another will have a serious impact on performance. One possible solution to this problem is to recode the software binding and unbinding routines in a more efficient manner. For all of these benchmarks the routines provided by CMU were used, and it is not known how efficient they are and thus how much room for improvement there is. Another possible solution is to build more stream gates on the iWarp CPU (perhaps eight, one per external link). This of course would require extra hardware that cannot be added to the current iWarp, but could be incorporated into the next generation iWarp. Studies have shown that the addition of multiple consumption channels to wormhole routed networks can improve performance [12].

3.3 Comparison of C40 and iWarp network

Both the C40 and the iWarp networks have a high bandwidth and low latency for point-to-point messages. Table 8 compares the C40 and iWarp transfer times for point-to-point messages. These are the same results shown in Table 1 and Table 4 with the times converted to microseconds. The times are shown in microseconds because the tested iWarp and C40 had different cycle times.

Table 8. Comparison of iWarp and C40 Point-to-Point Transfer Times in Microseconds

Type of message	C40 using RAM1	C40 using LM	iWarp transfer time	iWarp transfer time including bind time
Send 1 word	3.75	3.88	7.45	34.55
Receive 1 word	3.63	3.75	6.90	35.20
Send 8 words	4.63	5.13	6.70	33.80
Receive 8 words	4.50	5.75	6.30	34.60
Send 16 words	11.75	13.25	7.30	34.40
Receive 16 words	5.50	8.00	7.10	35.40
Send 64 words	19.63	25.00	12.15	39.25
Receive 64 words	15.88	21.50	11.90	40.20
Send 128 words	34.00	41.63	18.50	45.60
Receive 128 words	29.88	39.50	18.50	46.80
Send 256 words	62.00	77.63	32.40	59.50
Receive 256 words	57.88	75.50	32.50	60.80
Send 512 words	118.00	149.63	57.70	84.80
Receive 512 words	113.88	147.50	58.70	86.00
Send 1024 words		293.63	108.30	135.40
Receive 1024 words		291.50	108.10	136.40
Send 4096 words		1157.63	415.50	442.60
Receive 4096 words		1155.50	415.30	443.60

The iWarp unidirectional links have a bandwidth of 40 MBytes/sec compared to the C40 bidirectional link's bandwidth of 20 MBytes/sec. As expected, for longer messages (greater than 128 words) the setup times become more negligible, and the iWarp links can transfer the same amount of data as the C40 links in about half the time. For short messages (less than 16 words), however, the setup time is not a negligible component of overall transfer time.

Since the time needed to set up an iWarp message is longer than the time required to set up a C40 message, the C40s can transfer short messages faster than the iWarp even though the C40 links have half the bandwidth of the iWarp links. If the gate binding time is included in the iWarp setup time, then the iWarp setup time is no longer negligible, and the iWarp can only transfer very long messages (greater than 256 words) faster than the C40s. It can then be concluded that even though the iWarp network has a greater bandwidth than the C40 network, for near-neighbor messages the C40s have a lower latency.

As shown in Section 3.2.2, the iWarp has very low (negligible) latency for multi-hop messages. This is due to the extra hardware on the iWarp to support wormhole routing. As explained in Section 2.1.3.1 the C40s do not have any special hardware to support multi-hop messages, thus store-and-forward transfer methods must be used (unless the alternate method proposed in Section 3.1.3 is used). The wormhole routing hardware on the iWarp makes multi-hop message latency much lower on the iWarp network than on the C40 network.

The iWarp network has more dedicated hardware to support message passing than the C40 network; therefore, the iWarp network can handle more types of messages more efficiently than the C40s. However, the C40 message passing hardware is quite good for point-to-point communication. Also, as will be seen in Section 4.6, the C40 CPU is more powerful than the iWarp CPU. Even running at a slower internal clock speed (16 MHz vs. 20 MHz) the C40 was able to complete a benchmark algorithm faster than the iWarp.

4.0 Performance Measurements of Avionics Capability

In keeping with current opinion of the ineffectiveness of benchmarks for evaluating computer systems, it was decided to test the C40s by running an application program in both uniprocessor and parallel implementations [13]. Section 4.1 describes this application program, Section 4.2 describes several candidate topologies that were considered for a parallel implementation of the program, Section 4.3 describes the mapping strategies that were employed to map the program to the previously identified topologies, Section 4.4 details the performance measurements that were made on the PPDS, Section 4.5 details the performance of the program on iWarp, Section 4.6 compares the performance of the C40 and iWarp, Section 4.7 gives the CHAMP specifications, and Section 4.8 compares the C40 and CHAMP.

4.1 Description of Algorithm Suite Four

Figure 10 shows the data flow for algorithm suite four (AS4). AS4 is one of the Infrared Missile Warning (IRMW) algorithm suites evaluated by WL/AAAT-2 during the time period of April 1992-February 1993 [14]. This algorithm accepts data from two infrared bands, which represent sampled image data. Band Two (B_2) is the primary band (the band in which burning CO_2--missile exhaust--appears brightest) and Band One (B_1) is the secondary band (for reference use in the spectral filter). First, a median subtraction filter (MSF) is run on each frame of each band. The output of the two MSFs is then input into a correlation spectral filter (CSF), which in turn feeds the background normalizer and thresholder (BNT). The output of the BNT is a list of pixel locations the algorithm has identified as potentially being missiles.

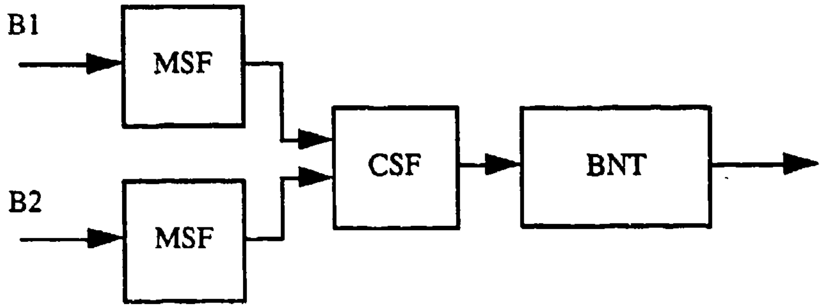


Figure 10. Data Flow for Algorithm Suite 4 [14]

4.1.1 Median Subtraction Filter

The MSF rank orders the eight nearest neighbors of the input pixel $P(x,y)$ by sorting them from smallest to largest. The ordered pixels are then labeled x_0, x_1, $x_2,\ldots,x_7$. For any band of data, the value of the output pixel from the MSF, $B(x,y)$, is defined as:

$$B(x,y) = P(x,y) - M(x,y), \qquad (14)$$

where $M(x,y)$ is defined as:

$$M(x,y) = (x_3+x_4)/2.$$

The output pixel value is the median value of the eight nearest neighbors of the center pixel subtracted from the value of the center pixel.

4.1.2 Correlation Spectral Filter

This spectral filter algorithm uses a 5 x 5 pixel window around each pixel in multiple bands to generate a correlation factor, $\alpha(x,y)$, which is an estimate of the correlation between the background data in the primary and secondary bands. After $\alpha(x,y)$ has been generated, the value of the center pixel in the secondary band, $B_1(x,y)$, is multiplied by $\alpha(x,y)$ and the result is subtracted from the center pixel in the primary band, $B_2(x,y)$. If the center pixel, $S(x,y)$, contains a potential target, the result will be a pixel value much larger than zero. Conversely, if the center pixel $S(x,y)$ of the 5 x 5 window is part of the clutter, the output value should be near zero. Thus, the value of the center pixel, $S(x,y)$, after filtering is:

$$S(x,y) = B_2(x,y) - \alpha(x,y)*B_1(x,y), \tag{15}$$

where

$$\alpha(x,y) = \frac{<B_1B_2>}{<B_1^2>},$$

$$<B_1B_2> = \sum_{m=x-2}^{x+2} \sum_{n=y-2}^{y+2} (B_1(m,n))(B_2(m,n)),$$

$$<B_1^2> = \sum_{m=x-2}^{x+2} \sum_{n=y-2}^{y+2} (B_1(m,n))^2,$$

and the sums do not include the terms where $m=x$ and $n=y$.

4.1.3 Background Normalizer and Thresholder

The BNT compares the center pixel of a 7x7 window to the average background energy, $\mu_b(x,y)$, of the remaining 48 pixels around the center. The resulting contrast metric, $C(x,y)$, is defined as:

$$C(x,y) = S(x,y) \;/\; \mu_b(x,y), \tag{16}$$

where $S(x,y)$ is the value of the center pixel and $\mu_b(x,y)$ is defined as:

$$\mu_b = \frac{1}{48} \sum_{m=x-3}^{x+3} \sum_{n=y-3}^{y+3} |S(m,n)|.$$

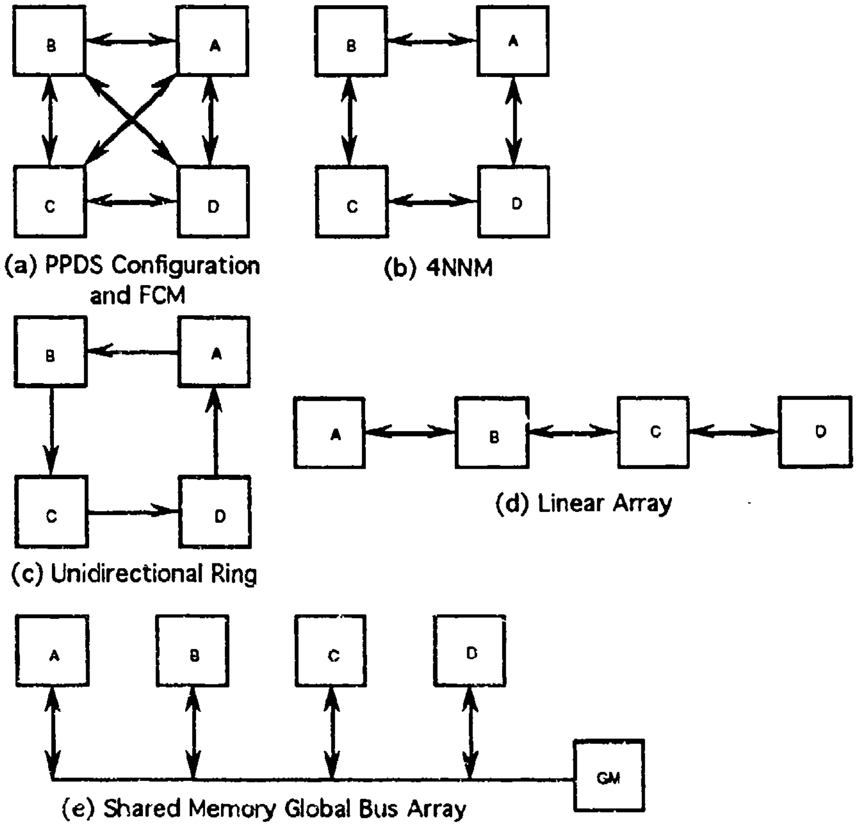
Note that the term where $m=x$ and $n=y$ is not included in the background mean calculation. The contrast metric for each pixel is then compared to a threshold provided by a tracking algorithm or by the user. Pixels whose contrast metric exceeds the threshold are declared exceedances and their intensity values $S(x,y)$ and their average background energy are output to a tracking process (not included in this algorithm).

4.1.4 Handling of Edge Pixels

If a pixel in an image falls sufficiently close to one of the edges of the image, it becomes impossible to center a filtering window on that pixel (i.e., assuming a square image of N pixels on a side, none of the pixels in column 0, column N, row 0, and row N of an image can be filtered using the 3x3 filtering window used in the MSF routine, or any other window used in AS4). Thus, a determination must be made about how to treat these so-called edge pixels. For the purposes of this study, it was decided to include only pixels known to be valid for further filtering (i.e., the entire image can be used for the MSF, the entire image minus column 0, column *N*, row 0, and row *N* can be used for the CSF, etc.). In this way the size of the image was reduced from an *N* x *N* image before the MSF to an *N*-2 x *N*-2 image after the MSF, and was similarly reduced for the CSF and BNT stages. Thus, after the entire algorithm has completed, the image dimensions were *N*-6 x *N*-6. The amount of reduction in the size of the image was constant across all processing platforms, so it will not affect the results of the tests made in this effort.

4.2 Description of Topologies

Since the PPDS has four C40s configured as a fully-connected mesh (FCM), several topologies become immediately apparent for implementing AS4, namely a four nearest neighbor mesh (4NNM), a ring, a linear array, and a shared memory global bus array. These topologies are shown in Figure 11, where the layout on the PPDS is shown in Figure 11(a), and the C40s are labeled accordingly on the other topologies.



(a) PPDS Configuration and FCM

(b) 4NNM

(c) Unidirectional Ring

(d) Linear Array

(e) Shared Memory Global Bus Array

Figure 11. Topologies Realizable from PPDS Configuration

4.2.1 Fully-Connected Mesh

For a two-dimensional data set (i.e., an image), most filtering operations require information from the surrounding pixels in all directions. In the uniprocessor case, there is never any problem with these filtering operations, since the entire data set is available to the processor. However, for a parallel case, only certain portions of the data set will be available to any specific processor, which means that it will have to receive data from its neighbors. Due to the windowing operations of AS4, data from the neighbors to the north, south, east, and west are needed, along with data from the neighbors to the northeast, northwest, southeast, and southwest. Thus, an FCM seems the optimal connection strategy. Given the configuration of the PPDS, it is simple to implement a four-chip FCM (see Figures 1 and 11(a)). In this topology, the data from one processor can be sent to all three of its neighbors in one hop, where a hop is the number of distinct communication port links the data traverse before reaching their destination.

4.2.2 Four Nearest Neighbor Mesh

The four nearest neighbor mesh is very like the FCM, except for the missing northeast, northwest, southeast, and southwest diagonal connections (see Figure 11(a) and (b)). Due to the similarity between the two, it should be apparent that they would operate along fundamentally similar lines, except that sending from one processor to another processor that is diagonally next to it will now take two hops instead of one (one horizontal and one vertical).

On certain machines, like the iWarp, multiple-hop routing can be handled quite efficiently by the wormhole routing technique described in Section 2.1.4.1. Because the C40 uses SAF routing (see Section 2.1.3.1), direct intervention by the CPU will be required in order to route messages to their correct destinations (assuming a complex or non-deterministic communication scheme that precludes use of the DMA's autoinitialization capability or systolic message routing discussed in Section 3.1.3), so it becomes vitally important to keep the number of hops required to send a message to as small a number as possible.

4.2.3 Ring-Based Network

Figure 11(c) shows a unidirectional ring, and Figure 12 shows a bidirectional ring. Ring networks are perhaps the simplest non-bus-based network topologies. Each node is connected to its left and right neighbor, therefore each node has two nearest neighbors. The two end nodes can have wraparound links as shown in Figure 12 to complete the ring, or they could have no wraparound in which case the network would reduce to the linear array shown in Figure 11(d). Figure 12 shows two unidirectional links, one sending to the right and one sending to the left, connecting each pair of nearest neighbor nodes. These could be substituted with a single bidirectional link, but bidirectional links can reduce throughput if they are constructed as half-duplex links.

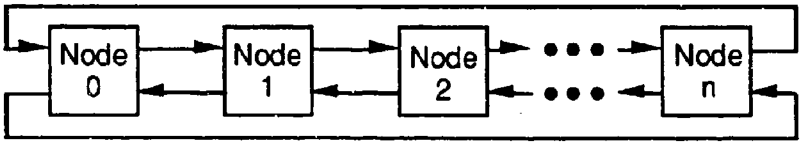


Figure 12. One-Dimensional Ring with Wraparound

4.2.4 Global Shared Bus

The most straightforward and least hardware-intensive approach to parallel processing is to connect several processors together by sharing a common global bus, as shown in Figure 11(e). This provides equal access to data stored in global memory to all processors. Coupled with a local memory for each processor, the global memory can be used for communication and synchronization.

A specific algorithm can be partitioned in practically any manner with the global shared bus topology, but instead of using the point-to-point communication links, all shared data are written to and rι ad from the global memory. GM writes should be atomic to ensure that if a given processor attempts to read a data structure currently bei; g written to by another processor, there will be no chance of the reading processor accessing incomplete or partially stale data. Also, if the processors on the bus have data caches, it becomes imperative to avoid stale data in the caches. Some processors, like the C40, avoid the problem by not caching data. Others, like the Intel i860, use a version of MESI (Modified, Exclusive, Shared, Invalid) protocols for ensuring cache coherency [15].

4.2.5 Other Topologies

Given the limited number of processors on the PPDS, only a small subset of the total number of possible interconnection networks can be formed, since many of the topologies reduce to others when the number of processors is small. For instance, a two-dimensional hypercube is indistinguishable from a four-processor four-nearest-neighbor mesh. However, if one were to assume an unlimited amount of processors were available, several other connection strategies become apparent.

4.2.5.1 Hexagonal Mesh

Using all six of the C40's communication ports, a hexagonal mesh can be formed, like the one shown in Figure 13. This tcpology has the advantage of being able to reach all neighboring processors in all directions in one hop.

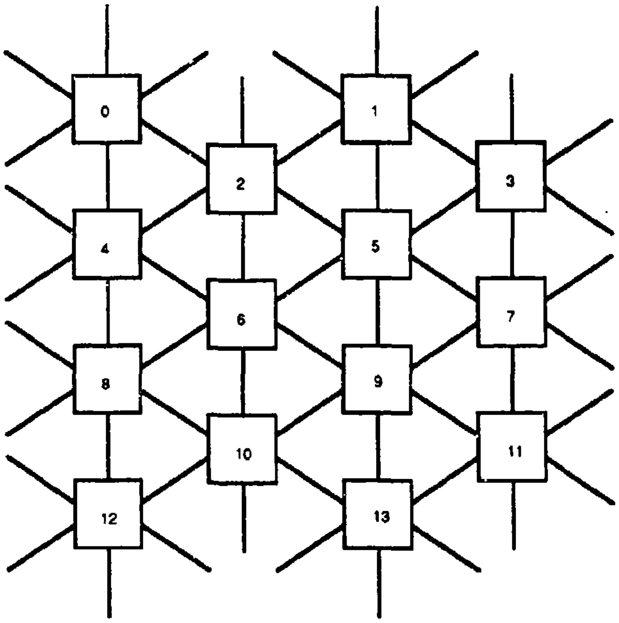


Figure 13. Hexagonal Mesh

4.2.5.2 Hypercube

An array of C40s can be arranged in a hypercube topology (Figure 14 shows a three-dimensional hypercube), for low-dimensional hypercubes (dimension ≤ 6). With three- and four-dimensional hypercubes, an interesting variation can be applied, which will be discussed in Sections 4.3.4 and 4.3.5.

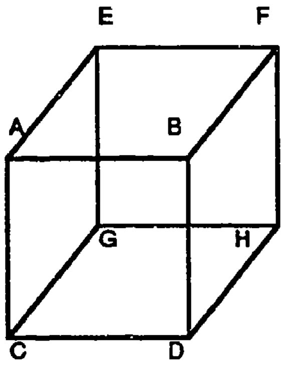


Figure 14. Three-Dimensional Hypercube

4.2.5.3 Pipelined Topology

Since AS4 (discussed in Section 4.1 above) follows a pipelined configuration, a pipelined topology like the one shown in Figure 15 can be explored. More will be discussed on this topology in Section 4.3.6.

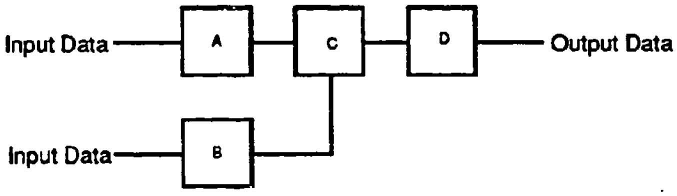


Figure 15. Pipelined Topology

4.2.5.4 CHAMP Topology

CHAMP uses a hybrid topology, incorporating both the bidirectional ring and fully-connected topologies (using a crossbar) as shown in Figure 16. This mixing of topologies gives several advantages, namely that all processors have the expediency of two nearest neighbor point-to-point connections, but also have the ability to send information to any given processor through the crossbar.

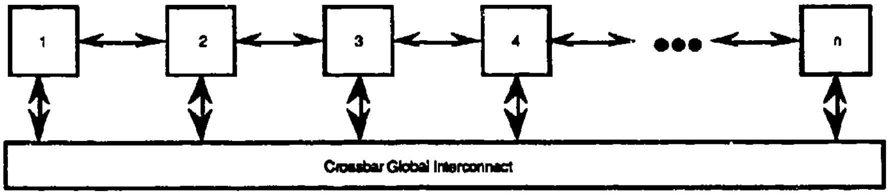


Figure 16. CHAMP Topology

4.3 Mapping AS4 Onto the Various Topologies

In general, for computation-bound algorithms (i.e., algorithms that require more computation time than communication time), the higher the communication/computation ratio, the worse the algorithm's performance. Thus, for these types of algorithms, it becomes imperative to reduce the amount of communication as a whole and optimize the required communication in order to ensure the fastest performance.

Investigations into the behavior of AS4 show it to be a computation-bound algorithm, but as will be shown in Sections 4.4.4 and 4.5.1, the communication/computation ratio is very low (on the order of 0.017:1). Thus, for this particular algorithm, the communication patterns of AS4 do not require an extensive effort to reduce the amount of communication. However, as discussed in Section 4.2.2, since the C40 must route data using the SAF method, it is beneficial to minimize the amount of non-nearest neighbor communication required to perform this algorithm in parallel.

Therefore, all mappings described in this section were made with the goal of reducing the number of hops required to send data from one processor to another. In most topologies, all data can be transferred in one hop; in others, in no more than two hops, although in certain topologies some modifications were made to these multiple-hop topologies to achieve the goal of single-hop routing. For the purposes of this study, it will be assumed that all images processed by AS4 are square with an edge size of N pixels, for a total of N^2 pixels per image.

4.3.1 Four Nearest Neighbor Mesh

This is perhaps the most straightforward of the mappings. If the 4NNM follows a square pattern (i.e., the same number of processors on all edges), then for an array of processors with edge size n it becomes a problem of simply dividing the image into n^2 pieces of N^2/n^2 pixels, as shown in Figure 17, where the regions of the image covered by a particular processor are labeled "PE *i*" (PE stands for processing element). In this topology, all horizontal and vertical data transfers can be done in one hop, and all diagonal data transfers will take two hops.

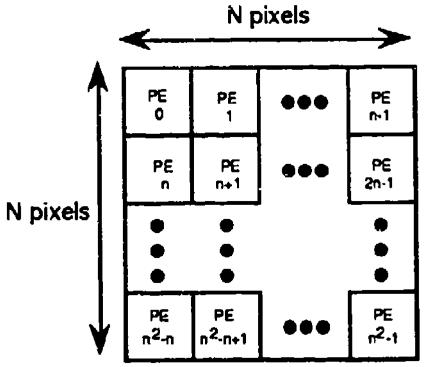


Figure 17. Image Partitioning Scheme for 4NNM and 8NNM

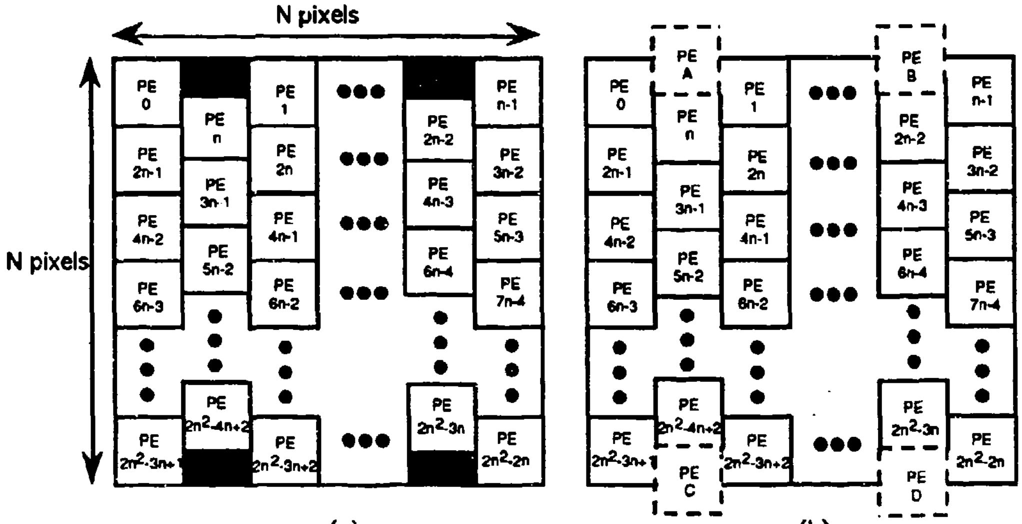
4.3.2 Fully-Connected Mesh

In cases where n is relatively large (i.e., $n \geq 9$), if the FCM connection pattern shown in Figure 11(a) is extrapolated (e.g., two horizontal links, two vertical links, and four diagonal links) it becomes an eight nearest neighbor mesh (8NNM). In the same manner as the 4NNM, mapping AS4 to an 8NNM will entail dividing the image into n^2 pieces of N^2/n^2 pixels, as shown in Figure 17. In this topology, all horizontal, vertical, and diagonal transfers can be done in one hop. Since the C40 has only six communication ports, an 8NNM cannot be formed with more than four processors, unless a custom hardware design is implemented to allow greater connectivity in the network.

An FCM, in contrast to an 8NNM, can be constructed with up to seven C40s (or perhaps more with specialized interface hardware), but the communication patterns of AS4 (see Section 4.3 above) do not warrant mapping it onto this type of topology, since the high degree of connectivity given by an FCM network is not required for this algorithm.

4.3.3 Hexagonal Mesh

Since the C40 has six communication ports and a 4NNM only uses four of them, an attempt was made to map AS4 into a topology that would make better use of the existing C40 hardware. One such topology is the hexagonal mesh shown in Figure 18. In this mesh, each processor can send data to all of its neighbors in one hop, thus eliminating some of the communication bottleneck incurred in mapping AS4 into a 4NNM. By dividing the image to be processed into nonoverlapping square pieces of size N^2/n^2, similar to that shown in Figure 18(a), only a small portion of the image, represented by the darkened areas in Figure 18(a), remains unprocessed. By using extra processors, labeled with letters in Figure 18(b), the entire image can be covered.



(a) (b)

Figure 18. Image Partitioning and Extra Processor Overlap

Each of the extra processors will have half the number of pixels ($N^2/2n^2$) of the other processors. This implies that some of the computational capacity of the extra processors will be idle during part of the algorithm, so a more detailed tradeoff study would have to be made before it could be determined whether using the extra processors to improve communication speed outweighs the extra cost involved and the idle time incurred with the extra processors. Such a study is beyond the scope of this report, and is mentioned as future work in Section 5.2.

For an array of processors with edge size n, the number of processors required for full coverage of the image with a hexagonal mesh can be calculated as follows: if n is even, then $n/2$ long columns and $n/2$ short columns of processors will be required (see Figure 18, where a long column is a column of tiles that has no unprocessed regions, and a short column is a column of tiles that has two unprocessed regions, one on the top of the column and one on the bottom). Each long column will have *n* processors. Each short column will have *n-1* processors. The number of extra processors required will be *n* (one each on the top and bottom ends of the short columns). Therefore, for *n* even, the total number of processors required, n_{tot}, is:

$$\begin{aligned} n_{tot} &= n(\frac{n}{2}) + (n-1)(\frac{n}{2}) + n \\ &= \frac{n^2}{2} + \frac{n^2-n}{2} + n \\ &= n^2 + \frac{n}{2}. \end{aligned} \tag{17}$$

Similarly, for *n* odd, the number of long columns required will be $(n+1)/2$, the number of short columns will be $(n-1)/2$, and the number of extra processors required will be $n-1$. Therefore, for *n* odd, the total number of processors required, n_{tot}, is:

$$\begin{aligned} n_{tot} &= n(\frac{n+1}{2}) + (n-1)(\frac{n-1}{2}) + n - 1 \\ &= \frac{n^2+n}{2} + \frac{n^2-2n+1}{2} + n - 1 \\ &= n^2 + \frac{n-1}{2}. \end{aligned} \tag{18}$$

Since the number of processors required for a 4NNM or an 8NNM is n^2, then it can easily be seen that the number of extra processors required to implement a hexagonal mesh is approximately $n/2$, depending on whether *n* is even or odd. For small values of *n*, it may prove worth the increase in cost to use the extra processors, since higher bandwidth communication rates can be sustained. However, for large values of *n*, the increased expense of including the extra processors may outweigh the benefits of the increased communication bandwidth.

4.3.4 Three-Dimensional Hypercube

Eight processors are required for a three-dimensional hypercube. Figure 19 shows one possible mapping of an image onto the hypercube. With this mapping, any processor can send to any neighboring processor (on the image partition shown in Figure 19(a)) in no more than two hops. This is true because there is no need, for example, for PE A to send data to PE F. However, if diagonal connections are made on the front and back faces of the hypercube, as shown in Figure 19(c), then each processor can send data to its neighbors in only one hop.

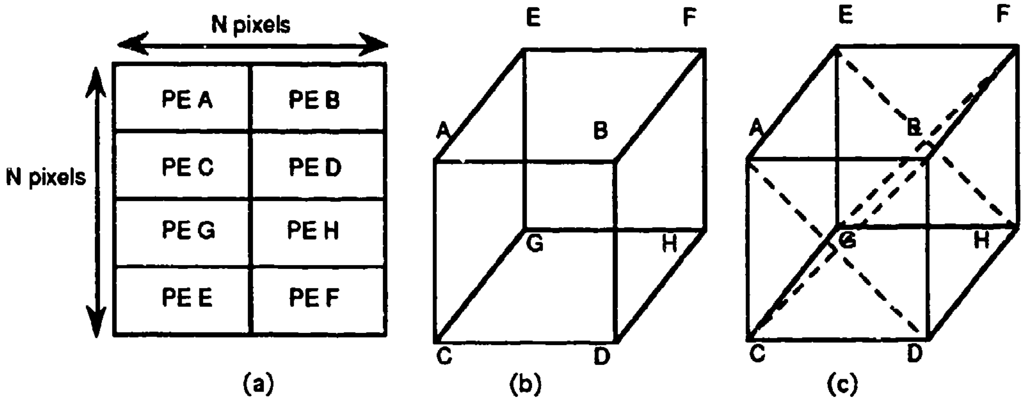


Figure 19. Mapping a Square Image onto a 3D Hypercube

While the topology shown in Figure 19(c) achieves the objective of single-hop communication, it still has two communication ports per processor remaining unused. However, like the 4NNM, 8NNM, and FCM topologies, it does make better use of computational resources than the hexagonal mesh, since no extra processors are required to completely cover an image. The biggest detriment of the hypercube architecture is that it does not scale well. For hypercube dimensions of greater than four, it becomes impossible for the C40 to form diagonal links like those used in Figure 19(c) due to the limited number of communication ports available, leading to multiple-hop communication. Hypercube dimensions of greater than six cannot be attained with the C40, since only six ports are available, unless special interface hardware is constructed for that purpose.

4.3.5 Four-Dimensional Hypercube

Similar to the three-dimensional hypercube, the four-dimensional hypercube can easily attain a worst case communication scheme of two hops (see Figure 20(a) and (b)). Again, similar to the three-dimensional case, adding extra connections between processors on the front, bottom, and back faces of the three-dimensional subcubes and moving many of the links in the fourth dimension as shown in Figure 20(c) allows the objective of single-hop communication for all processors.

Since several links in the fourth dimension were removed (as shown in Figure 20(c)), this topology can no longer strictly be called a four-dimensional hypercube, but since that is what it resembles most, it shall be called a modified four-dimensional hypercube. Several variations of this type of modified four-dimensional hypercube can be constructed, all of which will allow single-hop communication, but will not be shown here for reasons of space.

4.3.6 Pipelined Topology

Since AS4 follows a pipelined algorithm (i.e., BNT follows the CSF which follows MSF--see Section 4.1) a logical topology to implement would be a pipelined approach similar to those shown in Figure 21. In one implementation, each stage (MSF, CSF, BNT) would be one processor, and the entire image would be passed from processor to processor. This would most likely be a poor implementation, since the amount of communication between processors could easily overwhelm the amount of computation performed in each stage.

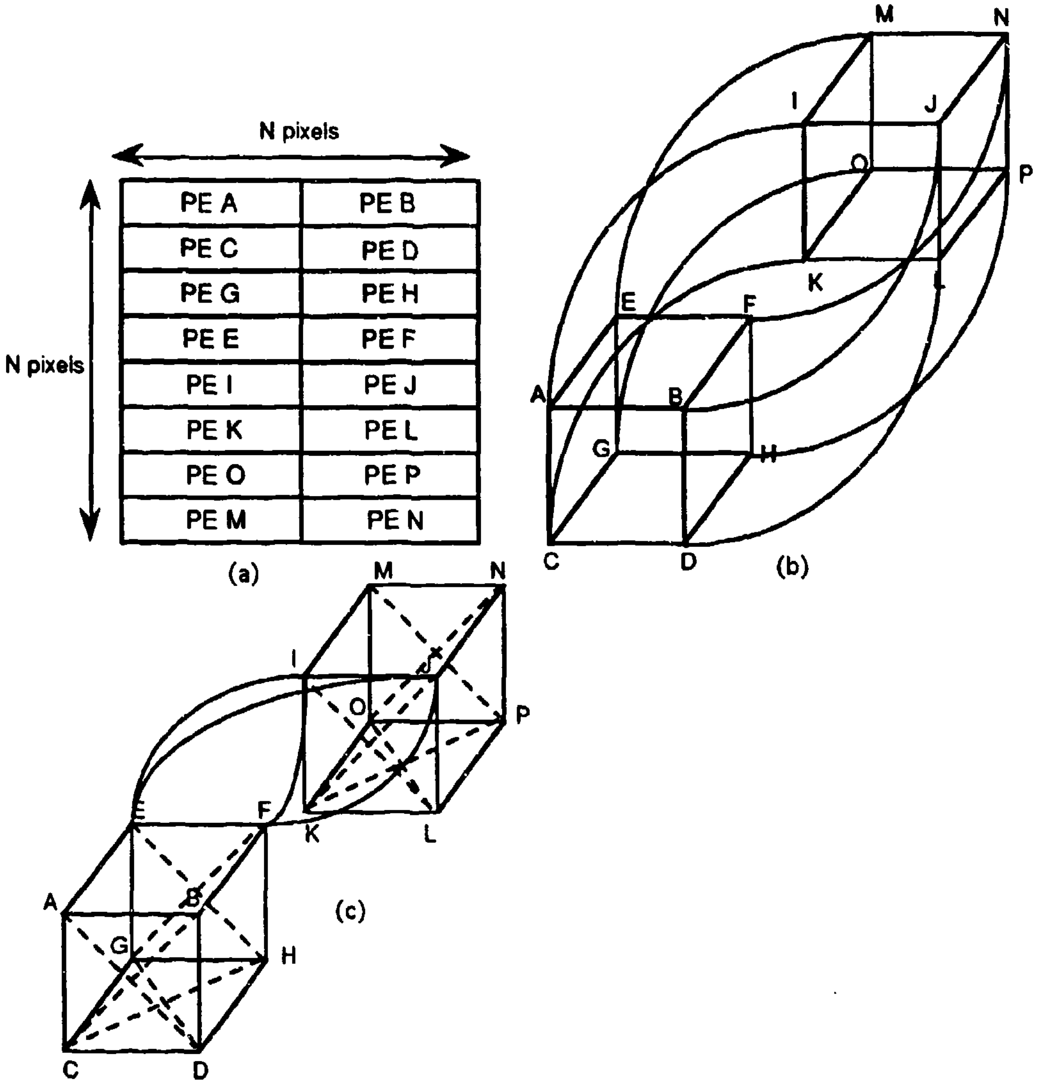


Figure 20. Mapping a Square Image onto a 4D Hypercube

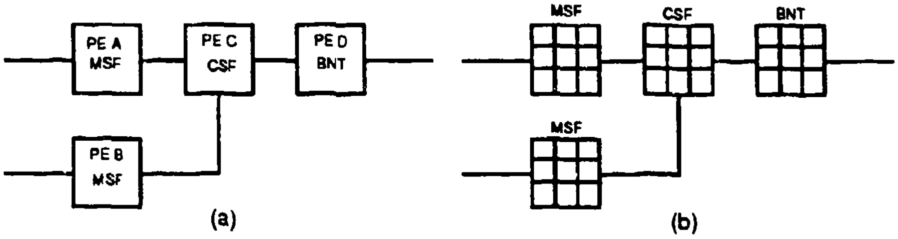


Figure 21. Mapping Images Onto a Pipelined Topology

The solution to this problem would be to use multiple processors in each stage--use a 4NNM in each of the MSF, the CSF, and the BNT stages. However, due to the nature of the algorithm, some additional switching hardware would be required for each processor to implement the CSF stage, since it would require seven communication links--four for the 4NNM, two for inputs, and one for output.

4.3.7 Ring Network

Mapping AS4 onto a ring network is a straightforward operation. One approach is to divide the image into stripes (rectangular sections of *N* x *N*/*n* pixels), as shown in Figure 22. One particular feature of this approach is that the number of pixels that must be sent from processor to processor remains constant regardless of the number of processors used (assuming that each processor has at least the amount of data that has to be communicated to its neighboring processors). The advantage of this feature is that predicting the scalability of this approach becomes a trivial problem, since the amount of communication required by each processor remains constant. The disadvantage of this feature is that as more processors are added, the communication/computation ratio increases.

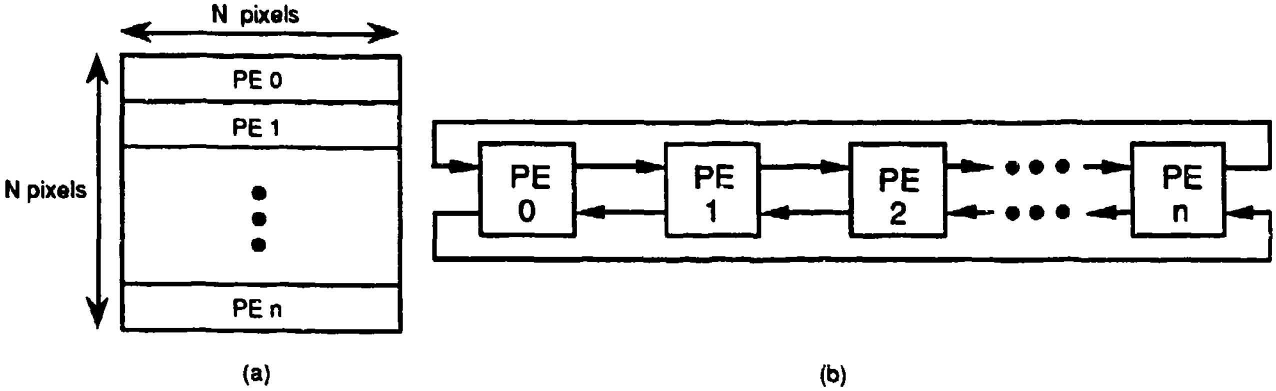


Figure 22. Mapping AS4 onto a Ring Network

4.3.8 Global Shared Memory

Mapping AS4 onto a global shared memory topology can be accomplished in several different ways. Just about any partitioning scheme can be used, with the required communication occurring through the global shared memory, as mentioned in Section 4.2.4.

Possibly the most straightforward (but worst performing) approach would be to store the entire image in global memory and let the individual processors contend for the bus to access the data. This would be a poor approach, because to ensure that any given processor could complete its write of processed data back to global memory without getting interrupted, a locked bus transaction would have to take place, as described in Section 2.1.3.3, which would severely limit the ability of other processors to continue computation without having to spin wait on the bus.

A better approach would be to store each processor's data in local memory and have small arrays reserved in global memory for the exchange of data. In this scheme, each processor could in turn access the global bus, lock it, then write only the data from its image that need to be shared into the previously reserved arrays. After completing the write, it would unlock the bus and spin-wait on a barrier until all processors had completed their writes to the shared arrays. After all processors had finished writing, then each processor could pass the barrier, access the shared data, load the data into its local memory, and proceed with processing.

While this approach seems relatively straightforward, the inherent delays in spin-waiting will severely impact the performance of this type of system, especially as the number of processors sharing the bus increases. Since all data updates must be performed atomically, the amount of time the bus is free for reads will decrease dramatically as the number of processors grows. For this type of data sharing, even the addition of data caches for the C40s would not significantly improve performance, since the computing delays are chiefly caused by the locked-bus transactions and the required barrier synchronizations.

4.3.9 Mapping AS4 onto CHAMP

Mapping AS4 onto CHAMP is a more complex problem than those mentioned previously, due to the architecture of the FPGAs used in constructing CHAMP. Because of the FPGA's structure, it is possible to incorporate more than one particular function (e.g., adder, multiplier, etc.) on any particular FPGA. Thus, to map an algorithm onto CHAMP, the design should be done in such a manner that it leads to optimal utilization of the available number of FPGAs on a given board.

Lockheed Sanders partitioned AS4 across the CHAMP architecture in a pipelined fashion, shown in Figure 23. In this pipelined architecture, data are processed in clusters of FPGAs (the crossbar is composed of FPGAs as well), and it is routed between FPGAs via either point-to-point links or the crossbar. In this manner, then, CHAMP resembles a special-purpose design optimized for this particular algorithm, rather than a parallel array of DSPs or general-purpose processors (e.g., C40s or iWarp). More specific details of the partitioning will not be discussed here, since it is beyond the scope of this discussion, and since the final design has not been completed at the time of this writing.

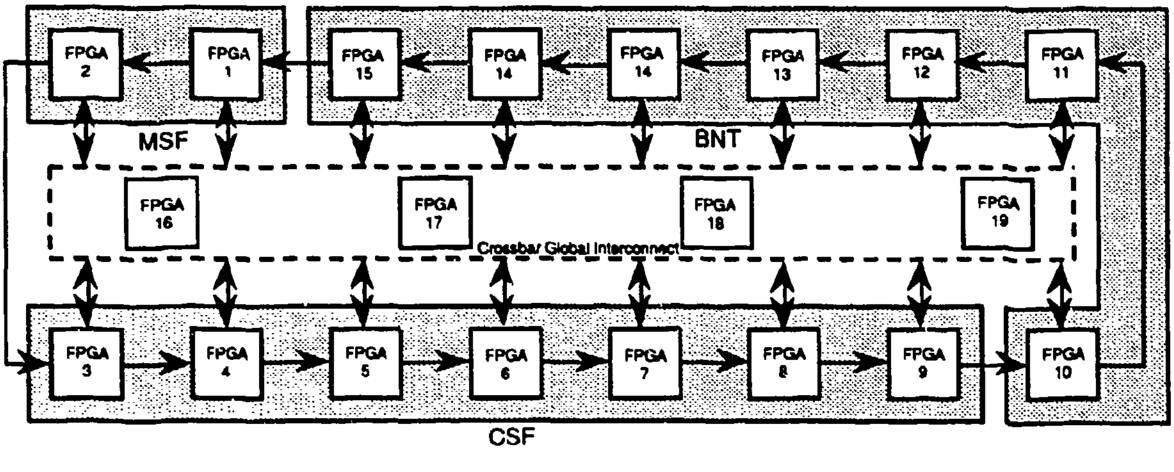


Figure 23. Mapping AS4 onto CHAMP

4.4 Performance Measurements of AS4 on the PPDS

Several network topologies have been presented in the last few sections. These topologies range from a simple multidrop linear bus to a fully interconnected network. One of the presumptions of this study was that bus-based architectures will not be able to handle future avionics processing needs because there is a limit to the number of processors that can be effectively used in a linear bus topology, as discussed in

Section 4.3.8. A fully-connected network would seem to be an ideal solution, then; however, the amount of hardware required to fully connect a large set of processors would be expensive and would not scale well. Implementing a global shared memory version of AS4 would be useful, however, in order to compare it to multidrop linear bus architectures in current use. Such a comparison is discussed in Section 5.3 as future work.

Having ruled out the extremes of multidrop linear busses and fully-connected meshes as legitimate possibilities for future processor interconnection topologies, the other topologies need to be examined. These topologies include rings, toroidal meshes, hypercubes, and pipelined architectures. The ring, 4NNM, and hypercube networks fall into the category of k-ary n-cubes, where k is the number of nodes along each dimension and n is the number of dimensions. A significant amount of research has gone into the design of optimal k-ary n-cube networks [6] [16]. There are several design tradeoffs involved in the design of a network, which include, but are not limited to: the dimensionality of the network, the type of router to be used (wormhole, store-and-forward, circuit-switched, etc.), the number of virtual channels to map to each physical link if wormhole routing is used, and the number of I/O channels.

It has been shown that VLSI communication networks are wire-limited [6]. That is, the cost of a network is not a function of the number of switches required, but rather a function of the wiring density required to construct the network. This is especially true in avionics, since the strict space requirements associated with putting electronics on an aircraft make it very costly to run wires between modules and line replaceable units.

Analyses have shown that for a constant wire bisection, low dimensional networks (e.g., tori) outperform high-dimensional networks (e.g., hypercubes) [6]. Therefore, based upon these results, this study focused on low-dimensional k-ary n-cube networks and networks that have structures similar to low-dimensional k-ary n-cube networks. In particular, one-dimensional rings and a four-processor eight nearest neighbor-like (8NN-like) mesh (see Figure 11(a)) were examined.

AS4 was partitioned in two different ways for placement on the PPDS, as shown in Figure 24. The areas between the dashed and solid lines in Figure 24 represent the areas of the image that had to be communicated between processors to allow each processor to properly execute the filtering operations of AS4. The partitioning used in Figure 24(a) was mapped onto a four-processor 8NN-like mesh (shown in Figure 11(a)) so that the diagonal connections could be used. This decreased the complexity of communication required to perform the filtering operations of AS4, but increased the complexity of coding, since each processor had to have a slightly different version of the program.

The partitioning used in Figure 24(b) was mapped onto a bidirectional ring (see Figure 12). In this mapping, the amount of communication between processors constant, which allows all processors to have identical code.

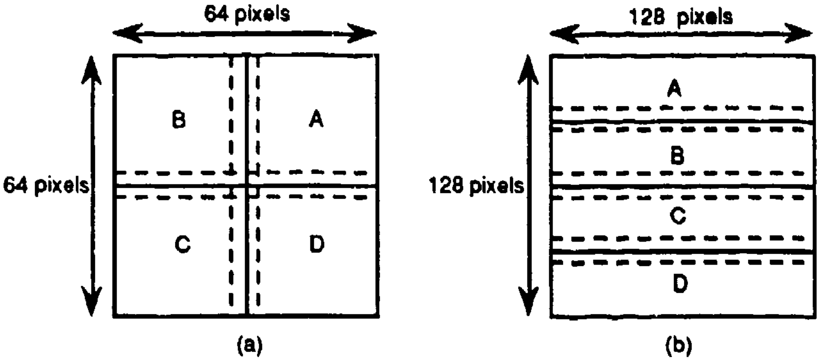


Figure 24. Partitioning Schemes Used on PPDS

The two image sizes shown in Figure 24 arose from operational concerns at the beginning of the benchmarking tests. The amount of LM on the PPDS is very limited, so it was advantageous to use a smaller image size to ensure correct operation of the program when the testing began. Once the program was verified correct, and when the scalability studies began in earnest, discussed in Section 4.4.4, it was decided to move to a larger image size in order to keep comparisons with CHAMP on a more equal footing.

The program code written to implement AS4 and perform the timing tests is included in the Appendix. The specific details of implementation are not crucial to the discussion at hand; the reader should refer to the Appendix to see the details of the optimizations described below.

4.4.1 Original Version of AS4

In the implementations of the original version of AS4, due to the memory model used by the C40, all arrays were dynamically allocated with a **calloc()** statement, which reserves space in memory for an array of a predefined length. Arrays defined in this manner are often more easily addressed as one-dimensional arrays, even if they are intended to hold multi-dimensional data. Thus, all of the array references in this implementation were made in an easily readable format that made it obvious the row and column of the element being accessed (i.e., row_number*number_of_elements_per_row + column_number). Unfortunately, this required many additions and multiplies to evaluate the correct array position, which made the executable code perform more slowly than was strictly necessary.

A uniprocessor version and an 8NN-like parallel version of this algorithm were implemented. The uniprocessor version stored the entire data set and program in local memory, keeping only the processor stack, the constant initialization section (*.cinit*), and the uninitialized variables section (*.bss*) in on-chip RAM.

The 8NN-like version also stored each processor's data in local memory, with the stack, *.cinit*, and *.bss* in on-chip RAM. The 8NN-like version had all the processors working together in a Multiple Instruction Multiple Data (MIMD) mode, in which the communication mechanism was used to synchronize the processors. In this version, each processor had the same algorithm, with different boundary conditions on the image (due to the location of edge pixels) and different communication patterns.

The processors performed the following steps in sequence to transfer data: 1) processors A and D sent to processors B and C, respectively; 2) processors B and C sent to processors A and D, respectively; 3) processors A and B sent to processors D and C, respectively; 4) processors D and C sent to processors A and B, respectively; 5) processors A and B sent to processors C and D, respectively; and 6) processors C and D sent to processors A and B, respectively (see Figures 1 and 11(a) for the layout of the C40s on the PPDS). In this way, all data transfers could be completed in such a way that no time was wasted in competing for control of the communication port until an entire message had been sent from one processor to another. The speedup gained by parallelizing in an 8NN-like topology is shown in Table 9, in the rows labeled "Uniproc. OV" and "Multiproc. OV." The different values for communication time, computation time, and speedup in the multiprocessor versions are due to the fact that each processor had a slightly different version of the program used to implement AS4.

Table 9. Results of AS4 Benchmark for Three Versions of AS4 (1 Frame of 64 x 64 Pixel Data)

AS4 version	Node	Comm. time (Kcycles)	Comp. time (Kcycles)	Total time (Kcycles)	Speedup
Uniproc. OV	A	0	10534.0	10534.0	1.00
Uniproc. OV1	A	0	8960.0	8960.0	1.18
Uniproc. OV2	A	0	3506.0	3506.0	2.56
Multiproc. OV	A	70.3	2687.4	2757.7	3.82
	B	524.7	2586.6	3111.3	3.39
	C	336.7	2681.0	3017.7	3.49
	D	152.9	2785.8	2938.7	3.58
Multiproc. OV1	A	60.4	2308.8	2369.2	3.78
	B	274.6	2221.4	2496.0	3.59
	C	168.9	2302.6	2471.5	3.63
	D	68.1	2393.2	2461.3	3.64
Multiproc. OV2	A	40.0	861.8	901.8	3.89
	B	229.8	830.5	1060.3	3.31
	C	146.4	858.8	1005.2	3.49
	D	68.2	891.2	959.4	3.65

4.4.2 Optimized Version One of AS4

The optimized version one of AS4 modified the array references so that they would be easier for the compiler to evaluate and would save time during execution. It also made some changes to index variables in loops such that if a certain array reference was calculated (via the row/column approach described in Section 4.4.1 above) more than once per loop, the calculation was removed from each individual array reference and placed in a location at the top of the loop. The result of the calculation was stored in an index variable, and that index variable was used to resolve array references. In most cases, these optimizations offered about an 18 percent speedup (see Table 9 in rows labeled "Uniproc. OV" and "Uniproc. OV1.")

Like the original version of AS4, a uniprocessor version and an 8NN-like version were implemented. Both versions had the same optimizations in the filtering functions (MSF, CSF, and BNT) and the parallel versions used the array reference simplification optimizations in the message passing routines also. The 8NN-like version was partitioned in the same way the 8NN-like original version was, and the corresponding speedup is shown in Table 9, in rows labeled "Uniproc. OV1" and "Multiproc. OV1" (for calculating multiprocessor speedup, the uniprocessor optimized version one is considered to have a speedup of 1.0; the multiprocessor speedups are calculated relative to that).

4.4.3 Optimized Version Two of AS4

The optimized version two of AS4 kept the array reference modifications made in optimized version one of AS4 described in Section 4.4.2 above and reduced the amount of redundant computations performed during filtering. In the original version and in the optimized version one of AS4, the code for the CSF and BNT filters (algorithms described in Sections 4.1.2 and 4.1.3 above) unnecessarily recalculated certain computations each time the filtering window moved. The optimized version two added small arrays to store the results of these calculations so that they could be used in more than one window. This brought a speedup on the order of five times for the CSF and BNT routines (not shown in Table 9), and a speedup on the order of 2.5 times for the entire program, shown in Table 9 in rows labeled "Uniproc. OV1" and "Uniproc OV2" (for calculating uniprocessor speedup, the uniprocessor optimized version one is considered to have a speedup of 1.0; the speedup for the uniprocessor optimized version two is calculated relative to that).

As in the original version and the optimized version one, a uniprocessor version and an 8NN-like version were implemented. Again, both versions retained the optimizations made in optimized version one and added the optimizations of optimized version two. Since the enhancements of optimized version two were limited to the CSF and BNT stages, there was no change in the message passing routines as there was in optimized version one. The speedups obtained from optimized version two are shown in Table 9 in rows labeled "Uniproc. OV2" and "Multiproc. OV2" (for calculating multiprocessor speedups, the uniprocessor optimized version two is considered to have a speedup of 1.0; the speedups for the multiprocessor versions are calculated relative to that).

Overall, speedups for the multiprocessor versions, relative to their respective uniprocessor versions, averaged around 3.6. This average will almost certainly never be obtained; even though some processors finish before others, due to AS4's pipelined structure all processors must complete their computations before the next image can be processed. Therefore, the expected speedup is limited by the slowest processor. Thus, the expected speedups are: for the original version, a speedup of 3.39; for optimized version one, a speedup of 3.59, and for optimized version two, a speedup of 3.31. It is interesting to note that both the average speedup and the expected speedup for optimized version one are higher than either the original version or optimized version two. This agrees with intuition--since all array references were sped up, the program would be capable of finishing its task quicker. However, for optimized version two, the expected speedup is lower than the expected speedup for both the original version and optimized version one. This seems at first to be counterintuitive--however, due to the optimizations made, the amount of computation time decreased dramatically, while the amount of communication time did not change as drastically (see Table 9). Thus, the communication-to-computation ratio has risen, which impacts algorithm speedup (as discussed in Section 4.3).

4.4.4 Ring-Based Implementations of Optimized Version Two of AS4

The ring implementation of AS4 was run on the PPDS system for one- and four-node networks. A 128 x 128 pixel image was used, and 100 frames of identical data were processed, due to the memory limitations of the

PPDS. For the ring implementation, each node has an identical program, and the interprocessor communication remains constant regardless of size of the ring. For these reasons, larger ring networks (8-node, 16-node, etc.) can be simulated accurately even though the PPDS only has four processors. For example, an eight-node ring can be simulated by placing one eighth of the image on each of the four processors (each node gets a 16 x 128 pixel stripe). Each node still needs to communicate up to three border rows with each of its two near neighbors. For this example, the four nodes combined only process a 64 x 128 pixel array, but each processor does the same amount of work as a processor in an eight-node system processing a full 128 x 128 pixel image. Eight- and 16-node rings were simulated, along with one- and four-node implementations. Table 10 shows the communication and computation times in Kcycles for each processor in the different array sizes as well as the total time in Kcycles and seconds. The speed-up of the multiprocessor versions is also shown compared to the uniprocessor version.

Table 10. Results of AS4 Benchmark on C40 Ring Networks Using Direct Message Passing (Times to Process 100 Frames of 128 x 128 Pixel Data)

Array		Comm. time	Comp. time	Total time	Total time	
size	Node	(Kcycles)	(Kcycles)	(Kcycles)	(sec)	Speedup
1	A	0	1452153	1452153	90.76	1.00
4	A	1593	376835	378428	23.65	3.82
	B	1593	376835	378428	23.65	
	C	1593	376835	378428	23.65	
	D	1593	376835	378428	23.65	
8	A	1588	188845	190433	11.90	7.59
	B	1588	188845	190433	11.90	
	C	1588	188845	190433	11.90	
	D	1588	188845	190433	11.90	
16	A	1589	94850	96439	6.03	14.89
	B	1588	94850	96438	6.03	
	C	1588	94850	96438	6.03	
	D	1588	94850	96438	6.03	

The network benchmarks described in Sections 3.1.1 and 3.2.1 showed that a C40 network takes much less overhead to set up a DMA transfer than the iWarp takes to set up a spooling transfer. For this reason it may be useful to use the DMA to overlap communication and computation for this algorithm. A ring implementation that overlaps communication and computation was created and run on a four-processor ring, and simulated on an eight-node and 16-node ring. Table 11 shows the results of this implementation compared to the uniprocessor version. From Table 11, overlapping communication and computation only made a slight improvement for the overall algorithm (about 0.2%) for each size ring. The communication time was greatly reduced (from 1590 Kcycles down to 970 Kcycles) for each node; however, this was nearly offset by a slight increase in computation time due to recoding to allow for DMA transfers. The communication-to-computation ratio for this problem is too low for overlapping communication and computation to be very effective.

Table 11. Results of AS4 Benchmark on C40 Ring Networks Using Overlapped Communication and Computation (Times to Process 100 Frames of 128 x 128 Pixel Data)

Array size	Node	Comm. time (Kcycles)	Comp. time (Kcycles)	Total time (Kcycles)	Total time (sec)	Speedup
1	A	0	1452153	1452153	90.76	1.00
4	A	972	376907	377879	23.62	3.82
	B	972	376907	377879	23.62	
	C	972	376907	377879	23.62	
	D	972	376907	377879	23.62	
8	A	972	189148	190120	11.88	7.59
	B	972	189148	190120	11.88	
	C	972	189148	190120	11.88	
	D	972	189148	190120	11.88	
16	A	971	95269	96240	6.02	14.89
	B	971	95269	96240	6.02	
	C	971	95269	96240	6.02	
	D	971	95269	96240	6.02	

As discussed in Sections 4.4.1-4.4.3, many software optimizations were used to reduce the computation time of the algorithm. Still, the algorithm remains computation-bound. Table 12 shows the amount of time each algorithm component took for a uniprocessor implementation of optimized version two of AS4 (the percentages are the same for any size image). From Table 12 it is clear that the algorithm spends most of its time processing the median filter for the two bands. In an attempt to reduce the time spent computing the median filter, a hand-coded assembly version of the median filter was created. This hand-coded version used some simple optimizations (such as assigning variables to registers) that most optimizers would do. The results of the uniprocessor version run with the optimized median filter are shown in Table 13. The assembly-optimized median filter ran about 25% faster than the C-coded filter. It is likely that if the spectral and BNT filters were optimized that they would show a similar improvement. The assembly optimizations were coded by a hardware engineer; had they been coded by a software engineer or a finely tuned optimizer, it is likely that they would show an even greater improvement.

Table 12. Computation Times for Each Algorithm Component in AS4

Type of filter	Time (Kcycles)	Percent of total time
Median filter	1041790	71.73
Spectral filter	231713	15.95
BNT	178844	12.31

Table 13. Computation Times for Each Algorithm Component in AS4 with the Optimized Median Filter

Type of filter	Time (Kcycles)	Percent of total time
Median filter	783021	65.60
Spectral filter	231713	19.41
BNT	178844	14.98

The algorithms were benchmarked on a 32 MHz development board. It is likely that a fielded system would run at the rated 50 MHz speed. Equation 19 is an approximation of how the algorithms would run with a 50 MHz clock. The communication time would actually increase (in terms of cycles) because it would take 5 cycles/word to transfer data rather than the 4.5 cycles/word it takes to transfer words to LM on the PPDS shown in Table 1 and Table 3. Equation 20 is similar to Equation 19 except that it calculates the effect of using an optimizer with an efficiency of *eff* on this coding of AS4. The total times to process AS4 are:

$$t_{tot} = (t_{comm}*5/4.5 + t_{comp})/25{,}000 \qquad (19)$$

and

$$t_{tot} = (t_{comm}*5/4.5 + t_{comp}*eff)/25{,}000, \qquad (20)$$

where

t_{tot} is the estimated time for a 50 MHz system (25 MHz internal clock) (in seconds),

t_{comm} is the communication time on the PPDS (in Kcycles),

t_{comp} is the computation time on the PPDS (in Kcycles), and

eff is the average efficiency of an optimizer.

Table 14 shows projected times in seconds for processing 100 frames of data at a 50 MHz clock speed. The first column shows the measured results on the PPDS system. The second column estimates the time for the 50 MHz clock using Equation 19. The third column assumes an optimizer was used that produces a conservative 25% improvement on the AS4 code. Equation 20 with an *eff* of 0.75 was used to calculate the optimizer time.

Table 14. Projected Processing Times in Seconds for 100 Frames of 128 x 128 Pixel Data

Number of nodes	AS4 at 32 MHz	AS4 at 50 MHz	AS4 at 50 MHz and 25% optimizer
1	90.76	58.09	43.56
4	23.65	15.14	11.38
8	11.90	7.62	5.74
16	6.03	3.92	2.96

4.5 Performance Measurements of AS4 on iWarp

Since the goal of implementing AS4 on iWarp was to give a comparison to the implementation of AS4 on the C40s both in speedup and scalability, it was decided to implement only the ring implementation of AS4. This implementation of AS4 was run on the iWarp at CMU on 1-, 4-, 8-, and 16-node networks. A 128 x 128 pixel array was used, and 100 frames of identical data were processed, in order to be consistent with the C40 benchmarks described in Section 4.4.4 above. Table 15 shows the communication and computation times in Kcycles for each processor in the different array sizes as well as the total time in Kcycles and seconds. The speed-up of the multiprocessor versions is also shown compared to the uniprocessor version.

Table 15. Results of AS4 Benchmark on iWarp Ring Networks (100 Frames of 128 x 128 Pixel Data)

Array size	Node no.	Comm. time (Kcycles)	Comp. time (Kcycles)	Total time (Kcycles)	Total time (sec)	Speedup
1	1	0	3032279	3032279	151.61	1.00
4	1	1695	792664	794359	39.72	3.82
	2	1695	792664	794359	39.72	
	3	1695	792664	794359	39.72	
	4	1695	792664	794359	39.72	
8	1	1692	397662	399354	19.97	7.59
	2	1692	397663	399355	19.97	
	3	1692	397663	399355	19.97	
	4	1693	397663	399356	19.97	
	5	1692	397663	399355	19.97	
	6	1693	397663	399356	19.97	
	7	1693	397663	399356	19.97	
	8	1693	397662	399355	19.97	
16	1	3350	200259	203609	10.18	14.89
	2	3350	200257	203607	10.18	
	3	3351	200258	203609	10.18	
	4	3349	200258	203607	10.18	
	5	3349	200259	203608	10.18	
	6	3350	200258	203608	10.18	
	7	3349	200258	203607	10.18	
	8	3350	200258	203608	10.18	
	9	3349	200259	203608	10.18	
	10	3350	200259	203609	10.18	
	11	3349	200258	203607	10.18	
	12	3350	200259	203609	10.18	
	13	3349	200259	203608	10.18	
	14	3349	200258	203607	10.18	
	15	3350	200259	203609	10.18	
	16	3350	200259	203609	10.18	

Streaming was used for message passing in this benchmark. For this implementation, each node needs to make two connections with each of its two nearest neighbors. This is a total of four connections, and with only two stream gates, this means that the gates will have to be bound and unbound to change which communication channel is being used, which adds extra overhead. As discussed in Section 3.2.1, if binding times are included, messages have to be very long for spooling to be advantageous. The messages sent in AS4 are not long enough to make spooling efficient.

The algorithm scales well, with nearly linear speed-up. This can be attributed to the low communication to computation ratio of this algorithm. Even with 16 processors, each node spent about 98.4% of its time computing and only 1.6% of its time communicating. For 16 nodes the iWarp processed 100 frames in 10.18 seconds (about 10 frames/sec). Again, this is with the optimizer off and all the code written in C (no assembly hand tuning).

4.6 C40/iWarp Comparison

Since AS4 is a computation-bound algorithm, the effects of the iWarp's faster communication links are likely to have a small impact on the speed of processing an image. Therefore, the advantage in speed will go to the processor with the more powerful ALU. In Table 16, the C40 is the unqualified winner in terms of processing speed. It should be noted that even though these measurements were made with a 32 MHz external clock for the C40s, the internal clock speed is only 16 MHz. The iWarp's internal and external clocks run at the same speed, 20 MHz. Thus, even though the C40's internal cycle time is slower than the internal cycle time of the iWarp, the C40 still turns in better performance measurements than iWarp. If the C40 was evaluated at its rated external clock speed of 50 MHz, the difference would be even greater.

Table 16. Processing Times in Seconds for AS4 on the C40 and iWarp

Array size	C40 at 32 MHz	iWarp at 20 MHz
1	90.76	151.61
4	23.65	39.72
8	11.90	19.97
16	6.03	10.18

For this implementation of AS4, then, the C40s can deliver a higher number of frames processed per unit time than the iWarp. It is likely, given the overhead imposed by gate binding on the iWarp, that for almost any implementation of AS4, the C40 will outperform the iWarp. Since the communication patterns of AS4 are not conducive to the use of wormhole routing on a large scale, and since data must be passed between more than two neighboring processors in most of the mappings discussed in Section 4.3, the gate binding overhead incurred by the iWarp when it changes destinations is likely to seriously impact its performance. In contrast, the C40 can communicate with up to six near neighbor processors with much lower overhead, as mentioned in Section 3.3. As the number of processors in an array grows, the number of image pixels in a specific IRMW algorithm that must be shared between processors decreases, making the C40's advantage even more dramatic.

4.7 Performance of AS4 on CHAMP

Since CHAMP has not been fully constructed at the time of this writing, and since the mapping of AS4 onto CHAMP was abandoned in favor of a more computationally complex algorithm, no performance measurements will be available for direct comparison between CHAMP, the C40, and iWarp. However, the baseline specifications for CHAMP in terms of the number of FPGAs required to process AS4 and the number of images to be processed per unit time will be used for comparison between the three types of processors. An implementation of the more computationally complex algorithm mentioned above on both the C40s and iWarp will be considered in the future, and is mentioned as future work in Section 5.4.

4.7.1 CHAMP Baseline Specification

The primary purpose of the IRMW algorithms is to process infrared imagery in such a way that hostile aircraft and missiles can be identified while an airplane containing CHAMP is in flight. Since a given aircraft can be expected to fly at different speeds and altitudes during the course of its travel, it is advantageous to attempt to remove as much of the effects of aircraft motion as possible from the images being processed. One way of removing the effects of aircraft motion is to capture images at a high frame rate (i.e., much faster than standard video rates of 30 frames per second). Thus, CHAMP is to be capable of processing anywhere between 200 and 1000 frames (128 x 128 pixels per frame) of infrared image data per second.

CHAMP would use 20 FPGAs to process AS4, not including the FPGAs used for interfacing CHAMP to a host and the FPGAs used to supply data to and remove data from CHAMP while in operation. These FPGAs were not included since they are not strictly involved in the computation itself, merely in support functions. For all comparisons to follow, then, the figures of merit will be the number of frames processed per second and the number of processors required for processing images at that rate.

4.7.2 C40 Scalability With AS4

Large ring networks of C40s can be hypothesized, using the equations and data presented in Section 4.4.4. As mentioned previously, the communication time per node will remain constant regardless of the network size, and the computation time can be estimated. With the optimizations described in Section 4.4.3, each node processes the first row of its partition and then uses previously processed information to process the remaining rows. Therefore, the first row will take a little longer to process than the remaining rows. Equation 21 is an estimate of the computation time per node for this algorithm. For these calculations, 128 x 128 pixel images were used; therefore, all rows are 128 pixels wide and each node gets 128/*n* rows (where *n* is the number of nodes in the ring). The total time to process a given image, therefore, is:

$$t_{tot} = t_{comm} + t_{r1} + (r-1)*t_r \tag{21}$$

where

t_{tot} is the total computation time for a node,

t_{comm} is the communication time for a node,

t_{r1} is the time needed to process the first row of data,

r is the number of rows assigned to a node, and

t_r is the time needed to process the remaining rows of data.

The per-node computation times for a 4-node, 8-node, and 16-node ring have been calculated in Table 10. Using these results and Equation 21, t_{r1} and t_r can be calculated. The times calculated for the 4-node, 8-node, and 16-node rings were plugged into Equation 21 and the resulting equations are:

$$\begin{aligned} 1593 + t_{r1} + t_r*31 &= 378428, \\ 1588 + t_{r1} + t_r*15 &= 190433, \text{ and} \\ 1589 + t_{r1} + t_r*7 &= 96439. \end{aligned} \tag{22}$$

By solving Equations 22, t_{r1} was found to be approximately 12604 Kcycles and t_r was found to be approximately 11749 Kcycles, where t_{comm} was approximately 1590 Kcycles. Using these three values and Equation 21, the computation time for virtually any size ring can be estimated. Table 16 presents the time in seconds for processing 100 frames for a ring of a given size (the data for ring sizes between 1 and 16 nodes are from Table 14) and the estimated number of frames per second that can be processed with a ring of the same size for both a 50 MHz clock and a 50 MHz clock with the previously assumed 25% optimization.

Table 17. Projected Processing Times in Seconds and Frame Rates in Frames/Second for 100 Frames of Data

Number of nodes	AS4 at 50 MHz	AS4 at 50 MHz with 25% opt.	Frames/sec at 50 MHz	Frames/sec at 50 MHz with 25% opt.
1	58.09	43.56	1.72	2.30
4	15.14	11.38	6.61	8.79
8	7.62	5.74	13.12	17.42
16	3.92	2.96	25.51	33.78
32	1.98	1.51	50.50	66.23
64	1.04	0.80	96.15	125.00

Based on the projections shown in Table 16, an array of 64 C40s could conceivably process as many as 125 frames/sec of data. Using any more processors would not be feasible, since the number of rows of data per processor cannot be less than one (i.e., if the ring has more than 128 nodes) and even the usefulness of having only one row of data per processor is highly debatable. This frame rate of 125 frames/sec is only slightly more than half of CHAMP's lower bound of 200 frames/sec and far from its upper bound of 1000 frames/sec. Thus, it can be concluded that with a ring-based partitioning scheme and a 128 x 128 pixel image, 125 frames/sec can be considered the absolute maximum frame rate.

Thus, in order for an array of C40s to achieve a frame rate equal to that of CHAMP, a mapping strategy different from the ring implementation must be used. Equations 21 and 22 cannot be used with the data presented in Table 9 (the data from the 8NN-like mesh) in the same way they were used with the data from Table 10, in part because the communication times vary so widely from processor to processor, and in part because the amount of communication performed per processor changes as the number of processors changes. In any case, an 8NN-like mesh of C40s would not be the best topology for comparison, since extra hardware would be required to implement an 8NNM (as discussed in Section 4.3.2), and the purpose of this study was to evaluate the on-chip parallel hardware of the C40. Thus, another alternative must be found. The topology most likely to be scalable to the degree required is the 4NNM. Since no studies were performed in this effort on the 4NNM, a discussion of that topology's properties will be completed at a later date, and that discussion is mentioned as future work in Section 5.5. Regardless of the results of that study, it is certain that at least 64 C40s (and quite possibly more) will be required to achieve the frame rate of CHAMP.

4.8 C40/CHAMP Comparisons

Since iWarp was never intended to be used in an embedded environment, comparisons with CHAMP will be limited to the C40. Lockheed Sanders have designed CHAMP to fit on one double-sided 6U VME card, and to have a sustained processing capability in excess of one billion operations per second (BOPS). A few vendors are marketing 6U VME card implementations of an array of C40s, typically with either one, two, or four C40s per card. An average of performance statistics across those boards with four processors will be used for comparison with CHAMP.

As mentioned in the previous section, neither the data from Table 9 nor the data from Table 17 will be completely accurate in terms of predicting the performance of an array of C40s, but some rough estimates can be made. Table 18 shows a comparison of some of the characteristics of both the CHAMP and the C40 boards.

Table 18. Comparison of CHAMP and C40 VME board characteristics

Characteristics	CHAMP	C40
Supply voltage (volts) (nominal)	5	5
Power requirement (watts) (typical)	35	15
Operating temperature (°C)	0 - 55	0 - 85
Storage temperature (°C)	-40 - 85	-55 - 150
Number of processing elements per board	16	4
Approximate cost of board (1993 dollars)	40,000	17,000

4.8.1 Hardware Considerations

CHAMP was designed to be capable of processing AS4 at a frame rate of at least 200 frames per second (128 x 128 pixel frame size) with only one board. Therefore, the comparisons made in this section will focus upon how many C40 VME boards will be required to match the performance of CHAMP, and how many watts of power will be required to operate at that level.

The data from Table 9 are for frames of 64 x 64 pixels, which is one-quarter of the size of a frame that has 128 x 128 pixels. The initial computations will be made based upon this 64 x 64 pixel frame size, but computations for the full 128 x 128 pixel frame size will be estimated later.

The per-pixel operation count of AS4 is 205 operations per pixel. For a four-processor C40 VME card operating at 25 MHz (internal clock speed), the frame rate for 64 x 64 pixel frames is (from Table 9):

$$25 \text{ MHz} / 1060300 \text{ cycles/frame} = 23.58 \text{ frames/sec} \qquad (23)$$

However, the data presented in Table 9 were recorded from programs that were not compiled with an optimizer. Using the 25% performance improvement gained by optimization assumed in Section 4.4.4, a new frame rate can be estimated:

$$23.58 \text{ fm/sec} * 1.25 = 29.48 \text{ fm/sec.} \qquad (24)$$

With this frame rate, a MOPS rating can be calculated:

$$205 \text{ ops/pix} * 64^2 \text{ pix/fm} * 29.48 \text{ fm/sec} = 24.8 \text{ MOPS,} \qquad (25)$$

and the number of MOPS per C40 is then:

$$24.8 \text{ MOPS} / 4 = 6.20 \text{ MOPS/C40.} \qquad (26)$$

Holding the number of MOPS per C40 constant, an estimate of the frame rate can be calculated for various array sizes. It will be assumed for these calculations that the image is partitioned in a regular fashion (i.e., all the tiles of the image are the same size) for the sake of programming simplicity. For brevity, only the first estimate whose frame rate exceeds 200 frames/sec will be shown. The estimated frame rate is, therefore:

$$6.20 \text{ MOPS} / (205 \text{ ops/pix} * (16*8) \text{ pix/fm}) = 236.28 \text{ fm/sec}, \quad (27)$$

where all the tiles are 16 x 8 pixels in extent. This size of tile requires 32 processors to completely cover the image. Thus, in order to meet the minimum frame rate of CHAMP, a minimum of eight C40 boards will be required, requiring 120 watts of power, 85 watts more than the single CHAMP board. There will not be an appreciable amount of additional overhead incurred by scaling the array size up, since with the commercially available boards it is possible to construct a 4NNM without any additional hardware (aside from the wires required to connect the communication ports together).

In order to obtain some rough estimates for the number of cards required for frames of 128 x 128 pixels, it can easily be verified (proof is left to the reader) that the number of MOPS/C40 will not change, and thus the number of cards required to process 128 x 128 pixel frames will be at least four times the number used to process 64 x 64 pixel frames. Thus, in order to meet the minimum frame rate of CHAMP, a minimum of 32 C40 boards will be required. Table 19 shows that based only upon hardware cost, CHAMP is the unqualified winner in terms of cost/performance tradeoffs.

Table 19. Comparison of processing required to process AS4 at 200 frames per second

Metric	64 x 64 CHAMP	64 x 64 C40	128 x 128 CHAMP	128 x 128 C40
Number of cards	1	8	1	32
Power required (watts)	35	120	35	480
Total number processing elements	20	32	20	128
Total cost of boards	$40,000	$136,000	$40,000	$544,000

At the time of this writing, simulations of the CHAMP system reveal that it will be capable of sustaining a frame rate of greater than 1000 128 x 128 pixel frames per second. In order for an array of C40s to achieve this rate, it would require at least five times the number of CPUs listed in the final column of Table 19, with approximately five times that cost.

4.8.2 Software Considerations

C40s and CHAMP are programmed in very different ways: the C40 is capable of being programmed in a high-level language (C and Ada are the most common), while CHAMP must be reconfigured via a hardware design process.

Programming a C40 is just as straightforward an operation as programming any other CPU of similar complexity, with the additional difficulty of having to include the multiprocessor communication (via the communication ports) in the program. For nearest neighbor communication, this operation is trivial (using the message passing routines provided with the compiler), but for multi-hop communication, the task can become more difficult, depending upon how the programmer chooses to implement the communication algorithm and how far the message must travel before it reaches its destination.

However, to program CHAMP, a designer must understand significantly more detail about the CHAMP hardware itself and hardware design methods in general. This is in marked contrast to the C40, which only requires that the programmer understand the register-level design of the processor. The CHAMP design process is shown in an overview format in Figure 25.

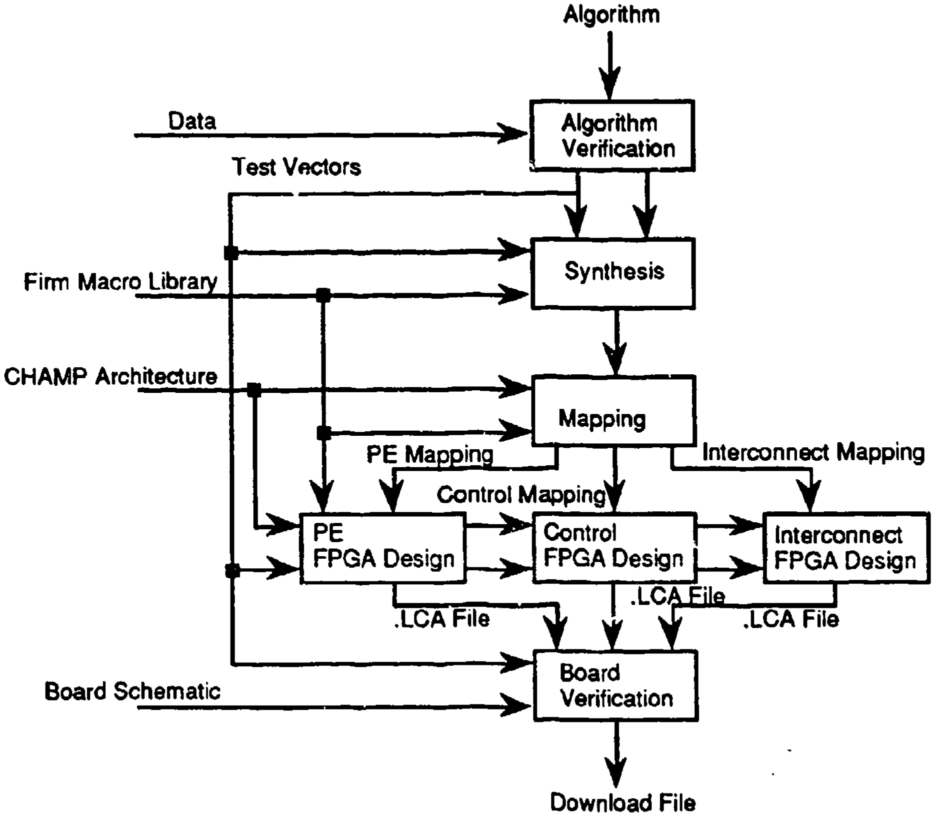


Figure 25. CHAMP Programming Process Overview

In a later phase of the CHAMP project, this design process will be integrated into a more automated procedure such that the amount of information required about the CHAMP board and hardware design in general will be reduced, which implies that the time to design a new configuration will be considerably less. Nonetheless, it will be some time before the CHAMP design process becomes automated and efficient to the point where it can compare favorably with the complexity of a C or Ada compiler for the C40. Due to the fact that the CHAMP design is still under development, it is difficult to quantify the exact amount of time involved in reprogramming CHAMP for an alternate algorithm. It is safe to say, however, that it would involve a considerable amount of time (up to several months), depending upon the skills of the people involved in its programming, the amount of the design actually being changed, and the number of macros that must be defined for the synthesis step.

5.0 Future Work

Several areas that merit further investigation have been previously identified in this document and are discussed here. Section 5.1 discusses software implementations of flow-through routing, Section 5.2 discusses a tradeoff study for the hexagonal mesh, Section 5.3 discusses the need for a global shared memory implementation of AS4, Section 5.4 describes timing both the C40 and iWarp with a more computationally complex version of AS4, and Section 5.5 details implementing AS4 in a 4NNM topology.

5.1 Flow-Through Routing

As mentioned previously, AS4 does not require extensive use of multi-hop routing, so the usefulness of a flow-through routing mechanism for implementing AS4 is limited. However, many algorithms suited to implementation in parallel will require far more use of multi-hop routing (e.g., one- and two-dimensional fast Fourier transforms, recursive doubling algorithms, histogramming algorithms, etc.), and the usefulness of flow-through routing will quickly become apparent.

In order to implement a flow-through routing scheme, such as the one mentioned in Section 3.1.3, each processor must have knowledge not only of its position in the mesh and the positions of its neighbors, but each processor must also have a deterministic routing mechanism, such that when a message arrives at a given processor, the processor will be able to decide which link the message should be sent out upon in order to minimize the length of the path taken by the message between source and destination. Several schemes exist which will allow this type of routing, but further research needs to be done to determine whether these routing schemes can avoid deadlock, blocking, or other undesirable results.

While the implementation of flow-through routing on the C40 would be beneficial in its own right, it would be even more useful if it were made part of a parallel operating system designed to operate on a network of C40s. Such an operating system would allow the user to write code at a higher level and let the operating system handle message routing between non-nearest neighbor processors. Ideally, such an operating system would work hand-in-hand with a compiler capable of extracting parallelism from a serial program, further reducing the workload of the programmer. Unfortunately, an operating system like this would severely impact the raw performance of the parallel machine, due to the amount of processing overhead that such an operating system would surely require.

5.2 Hexagonal Mesh Study

In Section 4.3.3, it was observed that a square image cannot be completely covered by a hexagonal mesh, leaving several areas of the image unprocessed. One approach to solving the problem is to add extra processors to the mesh, allowing each processor in the mesh to have an identical number of pixels to process (except for the extra processors). Studies need to be performed to determine how much of an increase in processing speed the extra processors add, compared to distributing the leftover pixels among the original processors. This data will then have to be coordinated with cost data to determine where a break-even point exists (i.e. where the benefit of having extra processors is outweighed by the increase in cost).

5.3 Global Memory Version of AS4 and RAS4

Even though the global shared memory architecture was determined to be a poor implementation for AS4 in Section 4.4, it would be useful to have data for such an implementation on hand for comparisons with existing avionics multidrop linear busses, such as MIL-STD-1553B, PI-Bus, etc. Code was written to implement a global shared memory version of AS4, but due to difficulties that proved intractable in the time allotted for this effort, no useful results were obtained. Future tuning of the programs may allow timing measurements to be taken in the future, however.

5.4 Revised Algorithm Suite Four (RAS4)

After the initial studies of IRMW algorithms conducted by WL/AAAT-2 were complete, it was clear that AS4 was one of the better performing algorithm suites tested [14]. Nevertheless, it was decided that modifications were required in order to improve AS4's ability to detect hostile missiles. Essentially, the modifications consisted of adding extra processing stages to the pipelined structure of AS4; each extra stage made the following stage's processing more accurate. For instance, in one of the extra stages (prerejection), all pixels that exceeded a normalized value of background intensity were removed from the image. This allows the CSF and BNT stages to compute their background statistical measurements for non-missile pixels without the interference of missile pixels that happen to fall within the window centered around the non-missile pixel.

These extra stages increased the amount of computing power required to process images substantially. It was found that the CHAMP architecture designed for AS4 would be capable of supporting RAS4 without the need for extra FPGAs, so AS4 was replaced by RAS4 in the CHAMP development. RAS4 also has high data storage requirements, and because of this, it was decided to put off coding RAS4 on the C40s until it was determined if the LM on the PPDS would be capable of storing all the required data. Even if the amount of memory is too limited on the PPDS, the actual number of operations required to perform RAS4 can be simulated by reusing image data (i.e., instead of adding four distinct numbers together, adding one number to itself three times will produce the same amount of computation, although the result will be different.) This technique can be applied here because there are no data conditional computations performed until the thresholding stage.

5.5 Implementing AS4 on a 4NNM of C40s

As mentioned in Section 4.7, the implementations of AS4 studied to date in this effort have proven inadequate to match CHAMP in terms of processing speed. Thus, it will be useful to implement AS4 in a 4NNM configuration so that large networks of C40s can be postulated and that their projected performance levels can be compared with CHAMP's baseline specifications. Early studies have determined ways to improve the amount of communication required (i.e., how to get diagonal transfers without sending a separate message that must be routed via a two-hop approach), which will make the coding more efficient and easier to troubleshoot.

It will have to be determined by implementation, however, whether the communication times will vary as widely as they did for the data presented in Table 9. If not, then it is conceivable that projections can be made about scaling the number of processors, and thus reasonably accurate predictions can be made about how many C40s will be required to process images at the same speed as CHAMP.

6.0 Conclusions

Texas Instruments' TMS320C40 digital signal processing CPU breaks new ground in the signal and data processing arenas by including on-chip multiprocessing support for embedded microprocessors. This has allowed not only RISC-like high-speed computing (from the high performance CPU core) but the ability to partition a task across a set of processors in such a way that the problem can be solved more efficiently (and thus more quickly).

It was shown in this report that the setup overhead for the C40 communication ports is small (often in the range of only 100 cycles), especially when compared to other microprocessors with parallel processing capability (like iWarp). Because of this, for nearest-neighbor communication, messages can be sent rapidly, with a linearly decreasing amount of overhead included in the per-word message transmission time (i.e., the amount of overhead is constant for any length message, and thus the percentage of overhead included in a per-word message transmission time decreases linearly with message length). However, the C40 has no capability to automatically route messages on a multi-hop path, thus non-nearest neighbor communication becomes a rather more difficult problem to manage. For non-nearest neighbor communication, the programmer must include routines for such communication directly into the program (limiting flexibility) or must include the provision for having the program call an interrupt service routine to handle the message routing (which also must be created by the programmer). Because of this, the usefulness of a large array of C40s for general-purpose processing must be evaluated carefully, because the ease of programming will decrease rapidly with increasing the size of the array.

However, for certain applications, such as image processing, where communication is generally limited to nearest neighbors over the entire scope of the algorithm, a large array of C40s can be used quite efficiently. It was shown in this report that image processing tasks can easily be partitioned over a large variety of topologies constructed from C40s, and the speedups obtained from these partitionings were, in fact, close to linear.

In order to evaluate an array of C40s with a planned avionics system, comparisons were made with the Configurable Hardware Algorithm Mappable Preprocessor (CHAMP) program being run from Wright Laboratories, System Avionics Division, Information Processing Technology Branch. It was shown that CHAMP is capable of a higher data throughput than an array of C40s on a per-chip, per-dollar, and per-watt basis. However, the level of effort required to reprogram CHAMP is considerably more than that required to program the C40s, so a user must make a decision based upon the intended application as to whether the increased cost, power, and chip count of the C40 is worth the ease of programming.

As mentioned in the Introduction, the goal of this report was not to perform an exhaustive evaluation of the C40 in avionics programs, since there is such a large variety of avionics applications that it would have been impossible to test them all. The goal was to evaluate the C40s computing and communicating capabilities in light of certain avionics requirements, and it has been demonstrated that the C40 is capable of avionics processing to a high degree of efficiency. While the C40 cannot compete equally with custom hardware designs like CHAMP, its built-in capability for parallel networking can be a significant asset in avionics, as long as the algorithms being implemented can make proper use of its computing and communicating capacity.

7.0 References

[1] Texas Instruments, Inc. *TMS320C4x Parallel Processing Development System Technical Reference*. Texas Instruments, 1992.

[2] Texas Instruments, Inc. *TMS320C4x User's Guide*. Texas Instruments, 1991.

[3] G. Almasi, A. Gottlieb. *Highly Parallel Computing*, pp. 162-170. Benjamin/Cummings, 1989.

[4] H. Kung, et. al. "iWarp Macroarchitecture Specifications," Version 3.0 CMU and Intel Corporation, 1990.

[5] T. Gross, S. Hinrichs. *External Product Specification for the iWarp Programmed Communication Services (PCS) Array Combiner*. Version 2.0, CMU, August 1991.

[6] W. Dally. "Performance of k-ary n-cube interconnection networks," *IEEE Transactions on Computers*, 39(6):775-785, June 1990.

[7] Xilinx, Inc. *The Programmable Gate Array Data Book*. Xilinx, 1991.

[8] Texas Instruments, Inc. *TMS320C4x C Source Debugger User's Guide*. Texas Instruments, 1992.

[9] Texas Instruments, Inc. *Parallel Debug Manager Addendum*. Texas Instruments, 1993.

[10] Texas Instruments, Inc. *TMS320 Floating-Point DSP Optimizing C Compiler User's Guide*. Texas Instruments, 1991.

[11] Texas Instruments, Inc. *TMS320 Floating-Point DSP Assembly Language Tools User's Guide*. Texas Instruments, 1991.

[12] S. Balakrishnan, D. Panda. "Impact of Multiple Consumption Channels on Wormhole Routed k-ary n-cube Networks," in *Proc. Seventh Int. Parallel Proc. Symp.*, pp. 163-169, 1993.

[13] J. Hennesy, D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kauffman, 1990.

[14] S. Hary, J. McKeeman, D. Van Cleave. "Evaluation of Infrared Missile Warning Suites". in *Proc. IEEE National Aerospace & Electronics Conference (NAECON)*, 1993, pp. 1060-1066.

[15] Intel Corp. *i860XP Microprocessor Data Book*. Intel, 1991.

[16] L. Ni, and P. McKinley. "A Survey of Wormhole Routing Techniques in Direct Networks," *IEEE Computer*, Feb. 1993, pp. 62-76.

Appendix
Listings of Program Code Used in Testing

A.0 Release Information

The program code presented in this appendix has not been cleared for release outside the Department of Defense. The authors may be contacted regarding this policy at Wright Laboratories, Avionics Directorate, System Avionics Division, Information Processing Technology Branch:

WL/AAAT BLDG 635
2185 Avionics Circle
Wright-Patterson AFB, OH 45433-7301
(513) 255-4949