1. Introduction

iWarp [5] is a distributed parallel computing system under joint development by Carnegie Mellon University and Intel Corporation since 1986. The architecture is derived from the original Warp architecture developed by Carnegie Mellon [2]. The building block of an iWarp system is the *iWarp cell*, made out of a single-chip *iWarp processor* (or *iWarp component*) connected to a local memory. Parallel systems of different scales and topologies can be built cost-effectively by simply linking together iWarp cells. Figure 1 illustrates one possible configuration.

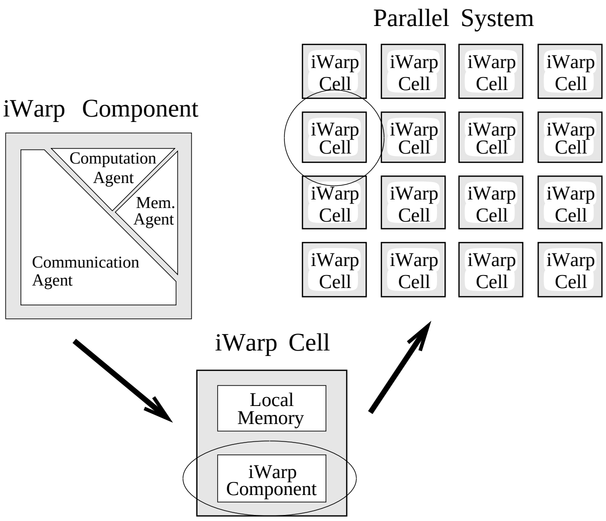

Figure 1. iWarp cell: a building block for parallel systems

The iWarp processor integrates both a high-speed computation and communication capability in a single component. The processor is a powerful computation engine that employs instruction-level parallelism to allow simultaneous operation of multiple functional units. What makes iWarp unique, however, is its interprocessor communication capability. An iWarp processor can simultaneously communicate with a number of other iWarp processors at high speeds. More importantly, the iWarp processor has a highly flexible communication mechanism that can support different programming models, including the tightly coupled computing found in systolic arrays and the message passing style of computation found in many existing distributed memory machines. These communication capabilities allow the effective use of iWarp for a wide range of applications.

The iWarp component consists of three autonomous subsystems, as depicted in Figure 1. The *computation agent*, which executes programs, can deliver 20 (or 10) MFLOPS for single (or double) precision calculations plus 20 MIPS for integer/logic operations. The *communication agent*, which implements the iWarp's communication system, can sustain an aggregate intercell communication bandwidth of 320 MBytes/sec by using four input and four output busses. The *memory agent*, which provides a high-bandwidth interface to the local memory, can transfer streams of data into or out of the communication agent at a rate of 160 MBytes/sec.

The first silicon of the iWarp component was fabricated in December 1989. It consists of approximately 650,000 transistors and measures about 1.4cm (551mil) on a side. Figure 2 shows a photo of the component, together with a floor plan that highlights the major units. The iWarp component operates at a frequency of 20 MHz, with the exception that the data is transferred between processors at twice that frequency. Three iWarp demonstration systems will be delivered to Carnegie Mellon by the Fall of 1990. Each of these systems consists of an 8×8 torus of iWarp cells, delivering more than 1.2 GFLOPS. The system can be readily expanded to include up to 1,024 cells for an aggregate computing power of over 20 GFLOPS and intercell communication bandwidth of 160 GBytes/sec.

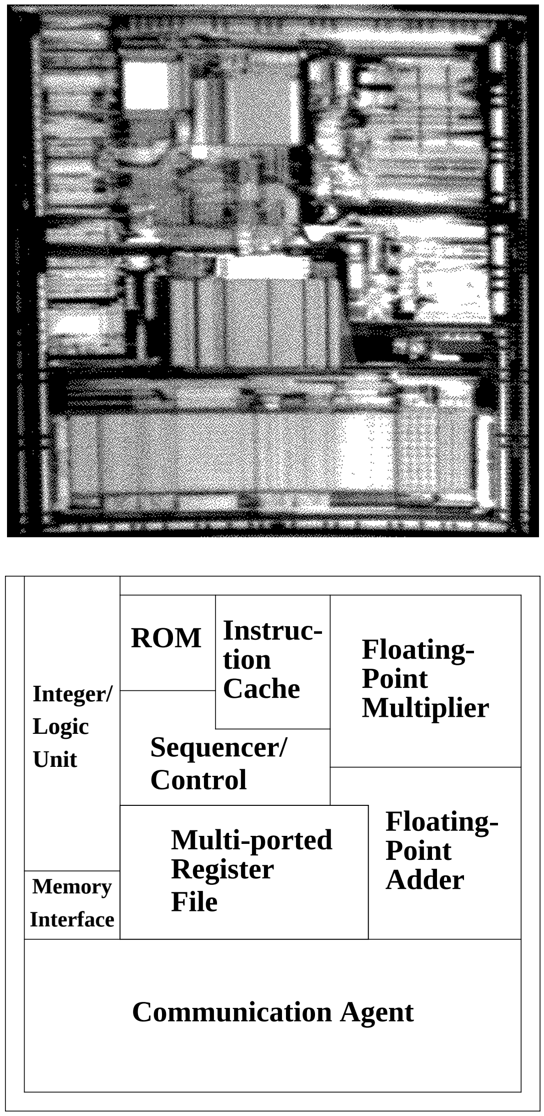

Figure 2. Photo and floor plan of iWarp component

The software for the initial iWarp systems includes optimizing compilers for C and FORTRAN for the programming of single cells. In addition, parallel program generators, such as Apply [11] for image processing, are available for the programming of an array of cells. If cells are programmed individually, the user must insert explicit communication directives (e.g., "send_block" or "receive_block") into the code for each cell. In contrast, parallel program generators can manage the communication between cells as well as the partitioning of data and computation over multiple cells for the user. More precisely, parallel program generators automatically produce for each cell a C program augmented with explicit communication instructions. The C program is then subsequently compiled by the single-cell C compiler.

A resident run-time system on each cell allows multiple light-weight threads to share physical resources, and supports systolic and memory communication. Included in this run-time system are the message-passing services of the Nectar communication system, originally developed for Carnegie Mellon's Nectar network [3].

This paper describes in depth the rationale, concepts, and realization of the iWarp communication agent. In particular, we describe the common design to support both systolic and memory communication, and the innovative architectural features needed to efficiently support these different types of communication.

This paper complements earlier iWarp papers on other topics: iWarp overview [5], architecture and compiler tradeoffs for the computation agent [6], and networks that can be formed on an iWarp array [9]. General discussions on interprocessor communication methods can be found in [15], which describes a taxonomy of communication methods and uses iWarp communication methods as part of the examples. Further discussions on systolic communication can be found in [13].

The organization of the paper is as follows. We first describe the fundamental differences between systolic and memory communication and point out that these two styles of communication each has its own merit. We then discuss the two unique architectural concepts in the iWarp communication system: (1) program access to communication and (2) logical channels. These innovations were motivated originally by systolic communication needs, but as described in Section 3, they are also useful in improving the performance of memory communication. We discuss the details of the iWarp communication system in Sections 4 through 7, starting with the physical intercell connections, the implementation of logical channels, routing and bandwidth reservation, and finally, communication agent interaction with the computation and the memory agents. We close the paper with some performance figures on the latency of communication, and some concluding remarks.

2. Systolic vs. memory communication

An iWarp cell is said to perform *systolic communication* if the program has direct, fine-grain access to the tail or head of the message queue as the message is being sent or received; it is said to perform *memory communication* otherwise. That is, in systolic communication, a receiving cell consumes a message by retrieving data words individually from the head of a queue that receives the message. Similarly, a sending cell generates a message by writing data words individually to the tail of a queue. The sending and receiving cells of the same message do not necessarily use the same communication style; that is, one cell may perform systolic communication while the other performs memory communication. Moreover, a cell may simultaneously use systolic communication for receiving and memory communication for sending, and vice versa. In the following we motivate and elaborate on these two styles of intercell communication.

2.1. Memory communication

In conventional message passing, messages are delivered from the local memory of the sending cell to the local memory of the receiving cell. That is, a message is first built in the local memory of the sending cell and then delivered (as a unit) to that of the receiving cell. Only when the full message is available in the local memory of the receiving cell is it ready to be operated upon by its program. Thus, in conventional message passing, both the sending and receiving cells perform memory communication.

In memory communication, the program running on the cell is insulated from communication. In the case of a sending cell, the program just needs to build the message in its local memory. After the complete message has been built, delivering it over the network is handled independently by some network software. Similarly, in the case of a receiving cell, the program is not involved in receiving the message, and will operate on the data in the message only after the entire message has been delivered to the local memory by the network software.

Memory communication has the advantage that communication is decoupled from computation. While the message is being delivered and buffered through memory, the program at the sending or receiving cell can operate autonomously on its local data. Moreover, communication protocols can be developed independently from the program to handle communication-specific issues such as deadlock avoidance and recovery from transmission failures. This makes memory communication the method of choice for applications which do not assume detailed knowledge about intercell communication. For these applications, message passing which uses memory communication at both sending and receiving cells is widely used.

2.2. Systolic communication

Systolic communication was motivated by systolic algorithms. In a systolic algorithm, an array of cells perform computations on long data streams flowing through the array. To achieve high efficiency, each cell processes the data immediately as each item arrives. We can view all data sent along each directed connection in a systolic array as belonging to one message. However, instead of waiting until all the data in the message have arrived, each cell operates on the data items within a message as they arrive individually. Similarly, each cell sends out computed results individually as they are generated. Therefore, each cell performs systolic communication as defined in the beginning of Section 2.

Systolic communication has the following advantages over memory communication:

- *Fine-grain communication.* The program at the sending cell can send out data items individually as soon as they are produced; similarly the program at the receiving cell can use data items individually as soon as they are received. This allows programs to communicate and synchronize with each other at word-level rather than message-level granularity. The message routing and header overheads are not paid with each unit of synchronization. This low communication cost makes it possible for the cells to cooperate in fine-grain parallel processing.

- *Reduced access to local memory.* Incoming and outgoing data need not be buffered in the cell's local memory unless it is required by the computation. Since memory access is typically a bottleneck in the cell's performance, the reduced access to local memory may translate into increased computation performance.

- *Increased instruction-level parallelism.* At each cell, systolic inputs and outputs provide additional parallel sources of operands for instructions. These operands can help keep the multiple functional units busy and increase instruction-level parallelism. Optimizing compilers for wide-word instruction set architectures, such as the compilers for Warp and iWarp [6, 16], have been developed to take advantage of this instruction-level parallelism.

- *Reduced size for local memory.* Avoiding buffering data in the local memory also reduces the memory size requirement for some applications.

However, systolic communication is harder to use than memory communication with respect to the flexibility of data access by a cell's program. The local memory of a cell can be accessed *randomly*, while message queues in the communication agent can only be accessed *sequentially*. Consequently, in systolic communication, one must make sure that the reads and writes of message queues are properly sequenced. That is, whenever the cell's program reads from an input queue, the right data item will appear at the front of the queue. Similarly, whenever the program writes a data item to an output queue, one must make sure that when the data item emerges from the head of an input queue of the receiving cell, that cell's program will be ready to read it. Parallel program generators have been developed to handle this strict sequencing requirement [11, 21, 18].

Furthermore, in systolic communication after an item has been sent, it will no longer be available on the sending cell and cannot be re-transmitted. Therefore the communication system must guarantee reliable transmission.

3. Two iWarp architectural innovations in communication

In the area of intercell communication, iWarp introduces two architectural innovations: *program access to communication,* and *logical channels*. These innovations were motivated by the desire to support systolic communication.

In addition, iWarp has many of the more ''traditional'' communication features [5] found in previous distributed memory machines [4, 19], such as support for non-neighborhood communication, message routing hardware, word-level flow control between neighboring cells, and spooling (a DMA-like mechanism to provide high-bandwidth access to local memory). Together, the traditional features and our two innovations make iWarp an effective processor for both systolic and memory communications.

3.1. Program access to communication

iWarp's communication is unique in that its low-level communication mechanisms are exposed and accessible by programs. First, the communication agent supports word-level flow control between connecting cells and transfers messages word by word to implement wormhole routing [7, 8]. Exposing this mechanism to the computation agents allows programs to communicate systolically. Second, a communication agent can automatically route messages to the appropriate destination without the intervention of the computation agent. By allowing the computation agent to modify the routing of messages in midstream, the program can implement some common message operations such as message concatenation or distribution efficiently.

3.1.1. Program access to data in communication

To implement systolic communication, iWarp allows programs running on the computation agent to have direct access to the inputs and outputs of message queues in the communication agent. These locations are bound to special registers, called *gates*, in the register space of the instruction set architecture. Reading from the gate corresponds to receiving data from the queue; similarly, writing to the gate corresponds to sending data to the queue. Data are transferred in the FIFO order and reading from an empty queue or sending to a full queue will block the operation. We call this *program-to-program* communication, since data words generated by a program at the sending cell are directly consumed by a program at the receiving cell.

Applications typically use message queues to smooth the flow of data between cells and to delay one stream of data with respect to others. The size of such queues is application-specific and can be larger than the message queues that the communication agent can provide in hardware. iWarp overcomes this problem by providing the option of extending the queue into the local memory of the cell (see Section 7.2). Although using this mechanism increases the demand for memory bandwidth, it provides a software transparent method for providing queues that are too long to implement with dedicated buffer space.

Besides supporting systolic communication, the ability for the program to access message queues directly can also speed up memory communication. In conventional message passing for distributed memory machines, messages are usually copied from the user space to system space at the sending cell, transmitted, and then copied from system space to user space at the receiving cell. Reliable and safe service routines are used to transfer messages between the system spaces. We call this *station-to-station* communication [5]. However, making copies of data back and forth between user and system spaces incurs considerable overhead.

Direct access to message queues can be used to increase the performance of the communication protocol. That is, the application can transmit the data itself using an application-specific protocol; the data are sent directly between the user spaces of the sending and receiving cells. We call this *door-to-door* communication.

To implement door-to-door communication, the application program at the receiving cell needs to read the message header before the message is buffered in the local memory. Using the information in the header, the program will explicitly control the memory allocation and tell the communication system where to deposit messages.

Again, these details can be readily handled by parallel program generators for image processing [10], scientific computing [21], or regular nested loops [18]. The extra protection provided by service routines for station-to-station communication is not needed by such tools, since the programs generated by the tools can be trusted to be correct in their interactions with the run-time system. Furthermore, parallel program generators can achieve additional efficiency by computing and communicating concurrently, with the use of instruction-level parallelism in iWarp. In summary, the iWarp architecture supports a range of methods of accessing data in communication at the sending and receiving cells: program-to-program, station-to-station, and door-to-door, or hybrid methods such as station-to-program and station-to-door.

3.1.2. Program access to data routing

Under normal operation, the communication agent establishes a route between the sender and the receiver, and all the data in the message follow the same route. In iWarp, a program at a cell on the route can alter the routing during processing of a message so that the rest of the message data can be forwarded to another cell, along a new route. Program access to data routing, for example, allows a cell to distribute (or collect) a block of data to (or from, respectively) a number of cells with a single message. This eliminates the need of generating or receiving multiple messages by the sender or receiver, respectively, and the need of buffering messages at various cells.

This mechanism can be illustrated by the ‘‘GetRow’’ and ‘‘PutRow’’ I/O methods [1], which have been extensively used on Warp for image processing applications. GetRow is an input method of distributing data (e.g., a row of an image) to a group of cells. The first cell, the left-most cell in Figure 3 (a), sends out the data as a single outgoing message to cells to the right. Each receiving cell in turn takes its portion of the message, and then forwards the remainder, if any, to cells to the right. Instead of buffering through the local memory and generating a separate message, the receiving cell alters the destination for the remaining message. More precisely, after having read its portion of the incoming message, the program at the cell will instruct the communication agent to *redirect* the remaining portion of the message to the next cell. Note that only one message is generated by the first cell; and this cell does not have to know which cells will receive the data that it sends out, nor how the data will be distributed among them.

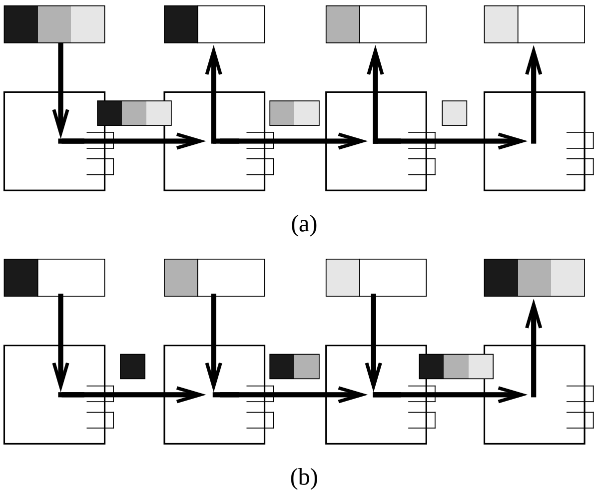

Figure 3. (a) GetRow and (b) PutRow

Corresponding to GetRow is the‘‘PutRow’’ output method of concatenating multiple messages from a group of cells to form one long output message. In PutRow, the last cell, the right-most cell in Figure 3 (b), receives the data from all the other cells. Each of the other cells sends out its data as a separate outgoing message to the next cell. After having sent out its message, the program at the cell, without closing the message, will peel off the header of the next incoming message and instruct the communication agent to redirect this incoming message as the remaider of the original outgoing message. Note that only one message is received by the last cell; and this cell does not need to know which cells participated in computing its incoming message.

3.2. Logical channels

iWarp’s second innovation in communication is logical channels. These have two important functions. First, in mapping computations onto iWarp arrays, logical channels provide a higher degree of connectivity than that achievable by physical means. Second, they provide a mechanism for delivering guaranteed communication bandwidth for classes of messages.

3.2.1. Increasing connectivity

When mapping computations onto iWarp arrays, it is desirable for the cells to be highly interconnected. However, the number of physical connections is limited by hard constraints such as the number of available pins and pads on the iWarp component. Logical channels overcome this problem by providing multiple ‘‘logical’’ connections over the same physical connection. In iWarp, multiple logical channels can time-multiplex a physical bus at word-level granularity (see Section 5). A total of twenty logical channels can be multiplexed over the eight external physical busses in each cell.

A high degree of connectivity is useful for systolic communication. In this case a cell may need to have simultaneous connections to several cells. Without logical channels, algorithms that require more physical connections than those provided in hardware cannot be implemented. Consider, for example, mapping a hexagonal systolic array onto a 2-dimensional grid of iWarp processors. Whereas the X and Y connections of the hexagonal array map directly onto those of the iWarp array, each of the diagonal connections of the hexagonal array can be implemented on the iWarp array with one horizontal and one vertical logical channel.

In general, a high degree of connectivity is required when mapping computations onto a physical array which has quite a different intercell communication topology. Even when the computation and the physical array have exactly the same communication topology, extra connections may still be needed to route around congested or faulty cells. Extensive simulation has shown that a moderate number of logical channels (on the order of 10) can be highly effective in avoiding faulty cells [17].

3.2.2. Delivering guaranteed communication bandwidth

Logical channels can be used to guarantee communication bandwidth for special classes of messages between a set of selected cells. The time-multiplexing of logical channels onto physical busses uses a schedule that guarantees that no active logical channel suffers from starvation. Therefore some minimum bandwidth is guaranteed to be available to each logical channel, and thus to the messages carried by the channel, since the total number of logical channels sharing the same physical bus is bounded. Moreover, the multiplexing of logical channels to physical busses is designed such that idle logical channels do not consume any physical bandwidth. That is, when a logical channel is inactive, the physical bandwidth reserved for it is not wasted and can be used by other logical channels.

The ability to deliver guaranteed communication bandwidth is important for both systolic and memory communication. The need in the case of systolic communication is obvious. The connection for systolic communication requires some guaranteed minimum performance to ensure effective low cost fine-grain communication. A systolic connection may exist for an indefinitely long period of time, possibly for the duration of an entire application program. If these connections exclude other communication on the same bus, then cells engaged in systolic communication can potentially lock out all other messages. It is important that some bandwidth be made available for memory communication to support system-related functions such as monitoring.

Guaranteeing communication bandwidth in the case of memory communication is less clear but nonetheless important. Messages received and sent using memory communication will complete in a bounded amount of time for a given available communication bandwidth. Provided that at least one connection from any cell to any cell can be made at any one time, all messages will eventually arrive at their destinations. However, there is little guarantee as to when a particular message will be delivered. Reserving a set of logical channels for a class of messages guarantees that some minimum bandwidth is reserved for them. For example, it is useful to guarantee that special system messages can be delivered in a timely fashion. This is especially useful for debugging and diagnostic purposes.

The reservation of communication resources in iWarp is modeled by the notion of *pathways*, each being a chain of linearly connected logical channels (see Section 6). Logical channels in a given set of pathways can be reserved to transport a class of messages between the cells connected by the pathways. Conversely, all these messages are confined to use only those logical channels within the pathways, guaranteeing the availability of the rest of the resources for other usages.

4. Physical busses

In the next few sections, we describe the architecture and implementation of the iWarp communication system and, in particular, show how they implement the two communication concepts described in Section 3. We describe the system in a bottom-up fashion. We start with the physical connections and the logical channels, then proceed to describe how logical channels guarantee minimum communication bandwidth for classes of messages. Lastly, we describe how the low level communication mechanisms are made accessible to the computation and memory agents.

Each cell is connected via eight physical busses to the outside world, each delivering a bandwidth of 40 MBytes/sec. The busses are unidirectional; four are input busses and the other four are output busses. We refer to these external busses as XRight, YUp, XLeft, and YDown as shown in Figure 4. The subscripts ‘‘in’’ and ‘‘out’’ are attached when necessary to distinguish between input and output busses.

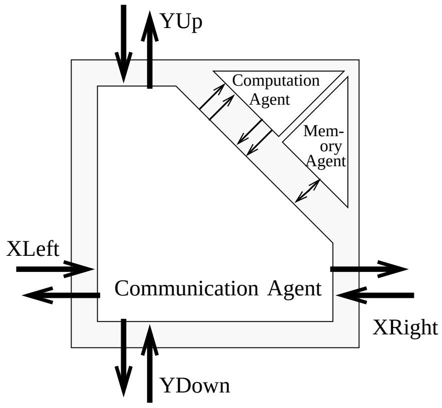

Figure 4. Physical busses

The design of the physical busses is a tradeoff between performance goals and implementation constraints. The component is limited by the number of pins in the package and the switching speed of the signals. Each of these eight external busses consists of eight data lines and five control lines. The data busses are unidirectional because they can operate at higher frequencies than bidirectional busses. The unit of transfer over a bus is a 32-bit data word; it takes four phases of 25 ns each to complete a transfer. That is, the external interface of the iWarp processor operates at a frequency of 40 MHz, yielding a bandwidth of 40 MBytes/sec for each bus.

The partitioning of the total 64 data lines into byte parallel busses is motivated by the desire to provide a high peak bandwidth per bus. Dividing the data lines into more, yet narrower, busses would increase the connectivity of the system. However, narrower busses reduce the available bandwidth for an individual message, and penalize programs that need only a low dimension of connectivity. Instead of taking this approach, we achieve both high individual bus bandwidth and high connectivity by the use of logical channels, as described below.

Each bus is complete in the sense that it contains all the necessary control lines to transfer data between two adjacent cells. This includes the ability of the receiver to provide status information to the transmitter, and vice versa. Thus the busses are completely independent. For example, there is no need to connect $XRight_{in}$ to the same neighbor cell as $XRight_{out}$. This feature is necessary to create, for example, a special-purpose hexagonal array in which each cell is connected to six neighboring cells, as seen in Figure 5.

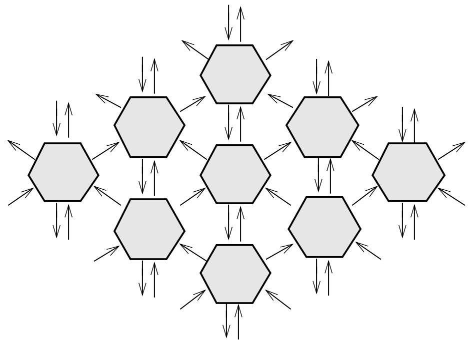

Figure 5. Hexagonal array

Internal to each processor, the communication agent interfaces with the computation agent through four unidirectional busses, two in and two out, each with a bandwidth of 40 MBytes per second. It interfaces with the memory agent via a bidirectional bus that can deliver 160 MBytes per second.

5. Implementation of logical channels

Logical channels allow the multiplexing of distinct, independent streams of data over a single physical bus.

5.1. Queues and logical channels

The communication agent of each iWarp processor contains a set of message queues. In its current implementation, the iWarp component includes 20 queues, numbered 0, . . . ,19, of eight 32-bit words each. The communication agent maintains for each queue some state including information about the source and destination, as shown in Figure 6.

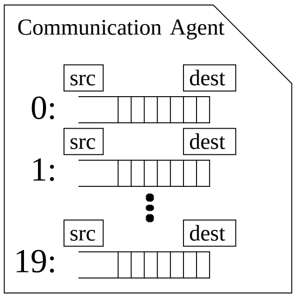

Figure 6. Queues in communication agent

There are two phases in managing the queues—*static allocation of queue sources*, which specifies the source of each queue before execution, and *dynamic assignment of queue destinations*, which binds the destination of a queue at program execution time.

In the first phase, each queue is statically allocated to receiving data either externally (i.e., from an adjacent cell) or internally (i.e., from the computation and memory agent of the current cell). *A logical channel is a queue that has been allocated to receiving data externally.* It is also called an *input logical channel* for the current cell, and an *output logical channel* for the adjacent cell from which the channel receives data.

5.2. Static allocation of queue sources

Allocating queue sources is to set the source record (i.e., src in Figure 6) of each queue, which indicates who can deposit data into the queue. Suppose that the queue is a logical channel. Then the src record of the queue is set to one of $\text{XRight}_{\text{in}}$, XLeft_{in}, YUp_{in}, and YDown_{in}. Only data words that are transmitted via this specific external bus can be deposited in this queue. The allocation divides the set of logical channels for one cell up among the different input directions, thereby creating disjoint groups of logical channels for each of these external input busses. That is, each logical channel is allocated to carry input from a specific input direction.

For a queue that has been allocated to receiving data internally, Internal or simply I is recorded in src. One example of an internal use of a queue is the local buffering of data before they are transmitted via a logical channel to another cell, as explained later. Section 7.2 discusses other internal usages.

Figure 7 shows one logical channel (highlighted by light gray shading) leading from Cell 0 to Cell 1. This channel is allocated by Cell 1 to the XLeft_{in} direction, as indicated by the source record for this queue in Cell 1. This logical channel is an input logical channel for Cell 1 and an output logical channel for Cell 0. A logical channel can be thought of as a unidirectional separate path (visualized by an ‘‘arrow’’ to indicate the flow of data from one cell to another) with a queue in the receiving cell. Figure 7 shows also another logical channel leading into Cell 2 from Cell 0.

Figure 8 shows a possible allocation of logical channels of two cells in a 2-dimensional array. Of the 20 queues of Cell A, 14 implement logical channels and 6 are reserved for internal use. The 14 logical channels are represented by 14 arrows pointing from the four neighbor cells at the communication agent of this cell. Figure 8 also shows which queue is associated with each of the logical channels of Cell A. Cell B has allocated 16 logical channels; of those, 6 are allocated to the bus that connects Cell A with Cell B.

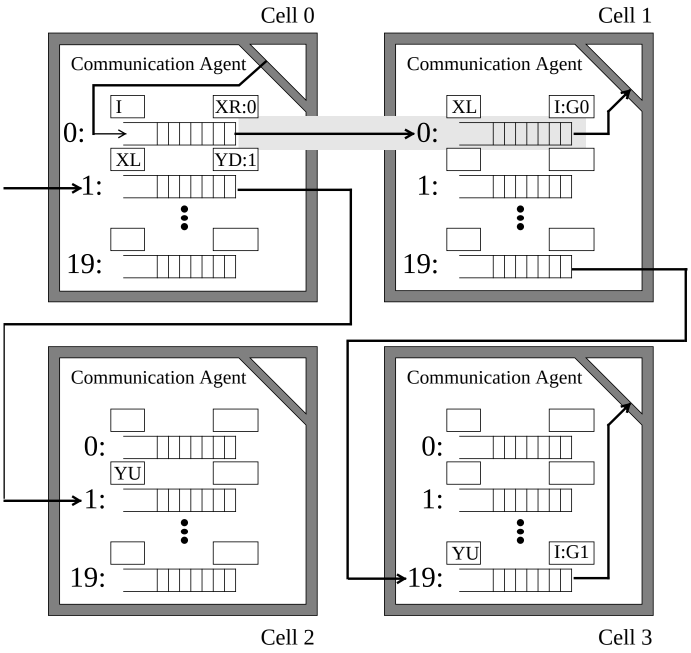

Figure 7. Logical channels

Table 1 shows the source records for the queues on this cell. The 6 queues that are reserved for internal usage can be used by either the computation agent or memory agent.

Some corresponding information is also recorded in a table in the adjacent cells. Each cell maintains a table of output logical channels. Each logical output channel is identified by the number of the *remote* queue that is associated with the logical channel in the recipient cell, as illustrated in Figure 8. The logical output channels of Cell A in the $\text{YDown}_{\text{out}}$ direction are associated with queues 1, 4, 2, 7, 3, 8 of Cell B. The allocation of logical channels between two adjacent cells can be asymmetric in the two opposite directions. The cell in Figure 8 has allocated four of its logical channels (7, 8, 9, 14) to YDown_{in}, but the lower neighbor has allocated six of its logical channels to its YUp_{in} bus.

5.3. Dynamic assignment of queue destinations

We now discuss the second phase in managing queues—dynamic queue destination assignment during program execution. There are no restrictions on the assignment. Any possible destination, either internal or external, can be assigned to any queue destination, and these assignments can be changed during the execution of the program. If the destination of a queue is internal (see Cell 1 in Figure 7), then this cell consumes the words from the queue. If the destination of a queue is external, then via an output logical channel an adjacent cell receives the data from the queue (see Cell 0 in Figure 7).

There are a total of 10 internal destinations. Represented by 10 arrows in Figure 8, these are two systolic gates and eight spooling gates (see Section 7.1). There can be at most 80 external destinations. If all 4 neighbors allocate all logical channels to the same cell, then this cell has the maximum of 80 output logical channels. At any given point in time a cell can assign the destination of a queue to at most one output

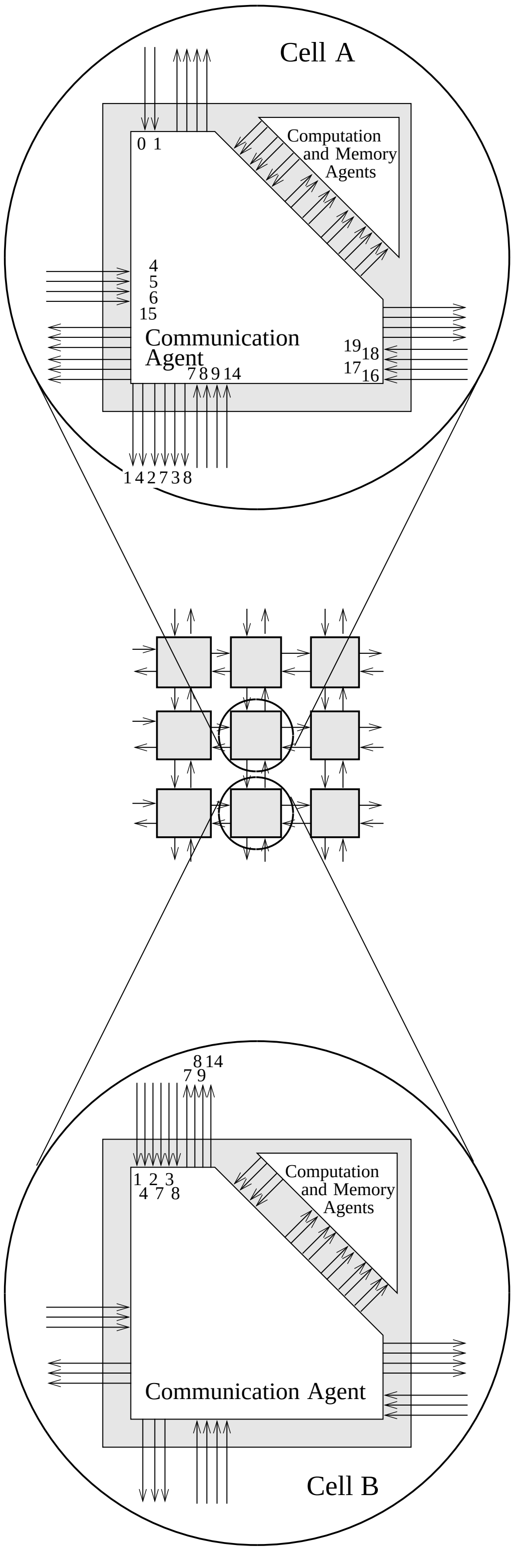

Figure 8. Example of logical channel allocation in 2D array

channel, and therefore up to 20 of these channels can be in use at any given moment.

When the destination of a queue needs to be assigned to one of the output logical channels allocated to a physical bus, the *transmitter* of the bus is responsible for selecting one of these channels. Since each communication agent maintains a table of the pre-allocated output logical channels (as shown above), together with the status of these logical channel (i.e., free or in use), each cell has complete knowledge about the availability of logical channels. The transmitter of a bus can initiate communication using any of the free logical channels allocated to the bus without first consulting the receiver, since these logical channels can receive data only from this physical bus. This design minimizes the time needed to initiate external communication. When an output logical channel is assigned, its direction as well as its queue number in the receiving cell are recorded in the destination record of the queue, as shown in Cell 0 of Figure 7.

Queue	Src	Queue	Src	Queue	Src
0	YU	7	YD	14	YD
1	YU	8	YD	15	XL
2	I	9	YD	16	XR
3	I	10	I	17	XR
4	XL	11	I	18	XR
5	XL	12	I	19	XR
6	XL	13	I		

Table 1. Allocation of logical channels

A cell that wants to send locally generated data to another cell deposits the data into one of the queues allocated for internal use and then assigns an output logical channel to this queue, as shown in Cell 0 of Figure 7. This arrangement has several advantages over writing directly to the output logical channel. The internal queue provides additional buffering, which is especially important if programs on adjacent cells use systolic communication. And having queues on both the transmitter as well as the receiver side of an external bus will ease flow control.

5.4. Multiplexing logical busses onto a physical bus

The multiplexing of logical channels over a physical bus is designed to maximize utilization; when only one logical channel is active, it must be able to take advantage of the full bandwidth of the physical bus. For example, the bandwidth should not be wasted on trying to send data to a full queue.

On iWarp, the transmitter keeps a count of the free slots in each of the receiver's queues. With every word it sends along a logical channel, the transmitter decrements the free space counter for the logical channel. Every time the receiver removes data from one of its input queues, it informs the transmitter with a dequeue signal that contains the index of the logical input channel from which a word was read. The transmitter then increments the free space counter for its corresponding logical output channel. The queue size of eight 32-bit words is designed to tolerate the feedback delay so that if only a single logical channel is active the maximum bandwidth can be obtained by this single logical channel.

The logical channel manager includes a round robin scheduler that multiplexes data from the logical channels over the physical bus. To preserve bandwidth, only those logical channels that have a non-empty transmitter queue and non-full receiver queue participate in the scheduling decision. That is, logical channels that are currently idle do not waste any physical bus bandwidth. Consequently, if only one of the logical channels allocated to the same bus is active, it can utilize the full bandwidth of 40 MBytes/sec of the underlying physical bus.

6. Routing and bandwidth reservation

The logical channels in the communication agent of an iWarp cell are statically divided into two pools. The first pool, called the *reservation pool*, is to implement ‘‘pathways’’ which can be reserved over a long period of time for transporting classes of messages with some guaranteed bandwidth. The second pool, called the *open pool*, is to implement traditional message passing. For this pool, there is no reservation of pathways; the logical channels are dynamically acquired and released for transporting each message. As described in Section 3.2.2, these messages do not hold onto resources indefinitely. By dedicating a pool for such messages, and using standard deadlock avoidance protocols, the run-time system can guarantee that there is no deadlock to prevent these messages from being delivered.

Although the usages between the two pools are different, they use the same basic hardware mechanism. For example, the same hardware is used to route pathways for the reservation pool and messages for the open pool. In the following we first describe the support for the reservation pool, then show briefly how the same mechanism implements the open pool.

6.1. The reservation pool

Intercell communication using the reservation pool consists of two phases: (1) reserving the logical channels for communication, and (2) sending the data as messages on those reserved channels. The reservation is done *dynamically* by setting up ‘‘pathways’’. This can be likened to a railway transportation system: first connect the track segments (logical channels) to form a pathway from a source to a destination, and then run trains (messages) over the pathway.

6.1.1. Setting up a pathway

A *pathway* is a unidirectional connection, built out of logical channels, that leads from a source cell to a destination cell. Pathways are initialized using wormhole routing [7, 8]. The source cell generates a header containing a destination address and additional routing information. As the header is passed along from the source to the destination according to the route specified, the logical channels are linked up to build the pathway. It is not necessary for the pathway to be completely established before sending messages over the pathway starts; the sending cell can start sending a message as soon as the pathway header leaves the cell.

6.1.2. Pathway markers

Each data word that is transmitted between cells can carry with it a tag. If a tag is present, we call the data word plus tag a marker; the absence of a tag indicates a normal data item. Markers are recognized by the communication system. There are two markers for pathways, the *pathway begin marker*, which includes a data field that carries pathway routing, and the *pathway end marker*.

6.1.3. Specifying pathway route

iWarp uses ‘‘street-sign’’ routing. Pathway begin markers have a default course of travel. For example, markers arriving on a logical input channel allocated to the XLeft bus default to continuing onto a logical output channel leaving via the XRight bus, and vice versa. Similarly, markers arriving on the YUp bus default to continuing onto the YDown bus, and vice versa.

The source cell can change this default course of travel by including in the header the addresses of all the cells at which a different action is to be taken. There are two possible actions: the pathway has either reached the destination, or it has to ‘‘turn a corner’’ and head in a new direction. This is analogous to city street navigation where each cell is a street intersection. The scheme is to follow the road in the same direction until you reach the destination, or make a turn when you come to a particular corner. For each corner turned, the pathway must include a word in the header containing the cell address and the direction to turn, in the order in which the cells are reached.

Street-sign routing takes advantage of the underlying topology of the system. By incorporating the concept of a default direction, headers can be kept short. A header contains only the addresses of those cells where a specific action is to take place (i.e., corner turning points and destination). Therefore the header takes less time to generate, and fewer routing decisions have to be made during the routing. In addition, a shorter header means a smaller overhead to the load of the communication system.

6.1.4. Pathway routing by communication agent

All begin markers arriving at the communication agent are matched against a small content-addressable memory of four entries, called the *match CAM*. The computation agent can ‘‘program’’ the communication agent by loading different values into this match CAM.

One of the uses of the match CAM is pathway routing. At initialization time, the run-time system on each cell preloads the match CAM with the address of this cell. Upon receiving a pathway begin marker, the communication agent presents the data field of the marker to the match CAM. If the marker does not match, the pathway continues in the default direction. If the marker matches, the current cell is either the destination, or the pathway must turn a corner. The information on the action taken is encoded in the marker. If the destination is reached, the computation agent is notified of the arrival of a new pathway. If a corner turning operation is specified, part of the matching marker indicates the new direction. The communication agent discards this marker and converts the next word (i.e., the destination or the next corner at which to turn) into a new pathway begin marker and directs the pathway to follow the specified direction.

The pathway header also indicates the reservation pool from which the free logical channel should be drawn. Therefore, to continue a pathway in a certain direction, the communication agent must assign a free output logical channel which is in the reservation pool and and is allocated to the specified direction. If such an outgoing logical channel is not available, then this request is blocked and repeated continuously until a logical channel becomes available.

If a pathway header reaches the last cell of the array without reaching its destination, then the communication system on this cell can notice the situation and take appropriate action, for example, report an error or discard the data. However, if the topology of the system is a ring or a torus, there is no ‘‘last’’ cell. One way to avoid the ‘‘Flying Dutchman’’ problem (i.e., the pathway header circulating around without ever reaching a destination) is to set up the match CAM of each cell to detect pathways originated by the cell itself.

6.1.5. Dismantling a pathway

Pathways are long-lived in the sense that they exist until explicitly taken down. To dismantle a pathway, i.e., to free up the resources reserved by this pathway, the source cell sends a pathway end marker over the pathway. As this marker is seen by each cell along the pathway, the logical channels used in each cell are returned to the set of free channels.

6.1.6. Joint cells and bandwidth reservation via pathways

A cell that is both a source and destination cell of two or more pathways is called a *joint cell* for the pathways. In Figure 9, Cells 1, 3 and 6 are joint cells.

At a joint cell, the computation agent can configure the communication agent to *link* together a pair of incoming and outgoing pathways. The output pathway is called the *default output pathway* for the input pathway. Any message arriving on the input pathway which is not intended for the cell is automatically forwarded by the communication agent to the default outgoing pathway. However, when a message destined for the joint cell arrives, the communication agent will notify the computation agent to process or route the message. Therefore messages can be sent over a single pathway, or multiple pathways via joint cells.

As stated earlier, pathways are built to reserve bandwidth for classes of messages which are to be sent over the pathways. In Figure 9, there is one pathway connecting Cell 0 to Cell 1, and another one connecting Cell 1 to Cell 5. Over these two pathways two classes of messages can be sent simultaneously, one from Cell 0 to Cell 1 and one from Cell 1 to Cell 5. Via the joint cell (i.e., Cell 1) messages from Cell 0 to Cell 5 can be sent over the two pathways. These two pathways reserve a set of logical channels solely for communication between cells 0, 1 and 5. Conversely, messages designated to use these pathways will not use other resources, and as a result will not block out other messages which critically depend on the other resources. For example, if a message from Cell 0 to Cell 5 passes through Cell 1 in Figure 9, then Cell 1 cannot send messages over the pathway from Cell 1 to Cell 5 until the ongoing message is complete. Figure 9 also illustrates how the queues in the communication agent are linked together to implement a single long message queue between two communicating cells such as cells 4 and 7.

6.1.7. Message routing over pathways

A message header contains the address of the destination cell and information needed to route the message over a given set of pathways. For each joint cell at which the message's route is to depart from the default, the header must include the cell address and information to identify the intended output pathway.

In the simple case of sending a message over a single pathway, the destination of a message is the destination of the pathway. The message simply follows the twists and turns of the pathway route until the destination is reached.

Routing messages over multiple pathways requires special attention at joint cells. When the header of a message arrives at a joint cell, the cell performs one of the following three actions:

1. *Forwarding the message by hardware*. If the destination in the header does not match in the match CAM, the communication agent forwards the message onto the default output pathway.

2. *Receiving the message*. If the destination in the header matches in the match CAM, the communication agent *splits* the incoming pathway from the default outgoing pathway, and notifies the computation agent. After receiving the notification, the computation agent reads the message header, recognizes that the message is intended for the cell, and starts processing the message.
   - After part of the message is consumed, the computation agent redirects the message to another destination. The remainder of the message of the message is forwarded; *or*
   - After the message is completely consumed, the computation agent restores the link between the incoming pathway and the default outgoing pathway.

3. *Routing the message by software*. Continuing onto a pathway other than the default requires software intervention. As above, the communication agent notifies the computation agent that the address on the header matches the cell's, the computation agent then interprets the header and instructs the communication agent to direct the message onto a specific outgoing pathway. Cell 6 in Figure 9 is an example of a cell where such action is necessary.

In summary, routing of messages over reserved pathways is not completely supported in hardware, unless the pathways form a chain so that the default outgoing pathway can be take at every joint cell. If another outgoing pathway other than the default one is desired, the computation agent at the joint cell must serve as a smart router. The computation agent can, in fact, perform arbitrarily complex computation on the beginning of the message before forwarding the rest of the message onto another cell. The usefulness of this scheme is illustrated by the GetRow and PutRow examples in Section 3.1.2.

6.2. The open pool

The open pool is reserved for message passing. Data sent using the open pool of logical channels are encapsulated in *routing* messages. Such messages are called datagrams in local or wide-area network terminology [20]. Each of these messages has its own routing information in the header. These messages are routed in a similar manner as pathways; therefore the routing is completely supported in hardware. To the routing hardware, these messages are identical to pathways, except that the logical channels are assigned from among the open pool instead of the reservation pool.

7. Communication agent interaction with computation and memory agents

There are two types of interaction between the communication agent and the rest of the system: data and control. Data in a message can be accessed directly by the computation agent or it can be spooled through memory by the memory agent. On the control side, the computation agent informs the communication agent of the events it is interested in, and the communication agent notifies the computation agent when an event occurs. In addition, the computation agent can redirect messages by changing the connection of the pathways in the communication agent's logical crossbar.

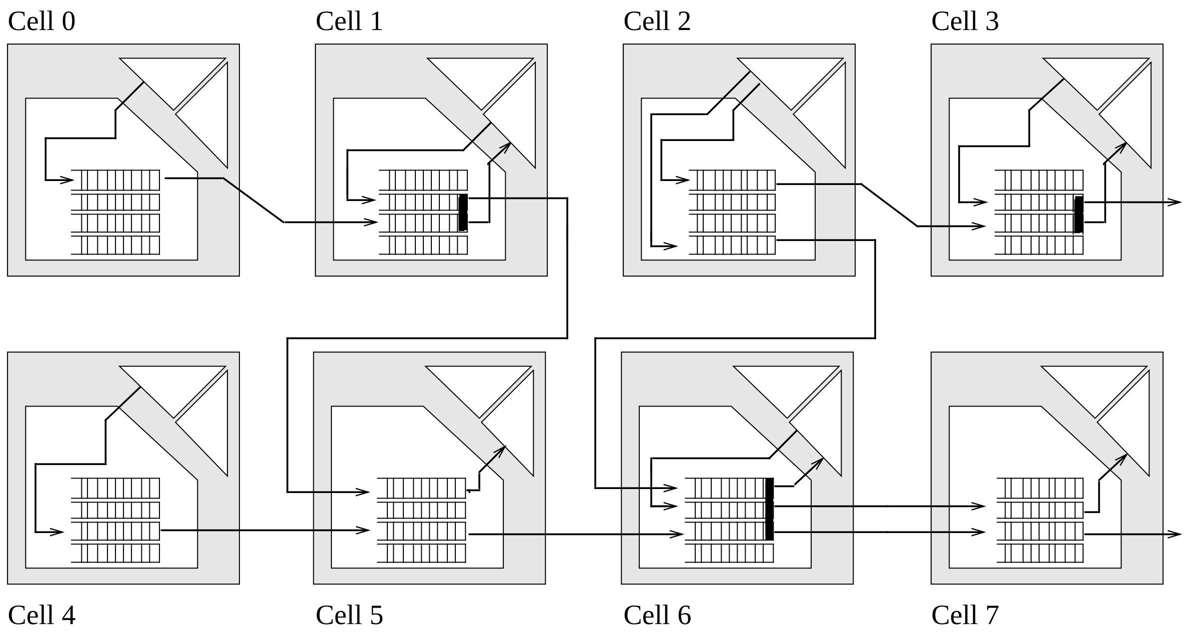

Figure 9. Examples of pathways and joint cells

7.1. Data interface to computation agent

In the computation agent's register address space there are special locations called *systolic gates*. There are two input gates and two output gates. Under program control, these gates can be bound to different logical channels and queues in the communication agent. Reading from an input gate corresponds to receiving data from the message queue associated with the logical channel bound to the gate. Similarly, writing to an output gate has the effect of enqueuing the data for transmission to another cell.

Since these gates are in the processor's register space, an input gate can be used as a source operand of an instruction, and an output gate can be used as a result register of an instruction. Any read of an input gate implies an input operation, and any write to an output gate implies an output operation. Specifying input and output instructions implicitly through the use of these special registers greatly reduces the instruction word width. For example, a three-address arithmetic operation using input and output gates as operands will imply two input operations, the arithmetic operation itself and one output operation. If any of the input queues is empty or if the output queue is full, the instruction execution is stalled until the condition of the queues changes. Using the long instruction word of iWarp, all four systolic gates may be used in one instruction. The iWarp hardware can execute all four input/output operations in two 50 ns clock cycles.

Through the systolic gates, data can be transferred between the computation and communication agents at the rate of 160 MBytes per second. As computation can be specified in the same (long) instruction word of the machine, this high communication rate can be accompanied by an equally impressive computation rate. The additional data operands supplied by the input and output gates help reduce the memory bottleneck and increase the utilization of the functional units (see Section 2.2). Using systolic communication requires more programming effort to ensure that cells do not stall frequently on empty or full message queues; however, a well-designed systolic algorithm can be extremely efficient.

7.2. Data interface to memory agent

The memory agent transfers, or spools, data directly from the message queues in the communication agent into consecutive locations in the memory, and vice versa. It is like a DMA device, with special hardware to keep the state and to sequence the spooling operation. The memory agent steals memory and computation cycles when spooling data in or out of memory. The memory agent is useful for memory communication transfers as well as for simulating large queues for systolic communication by buffering data in memory (see Section 3.1).

There are eight 64-bit spooling gates that can be dynamically reconfigured for either input or output and can be dynamically bound to the logical channels in the communication agent (see Section 5.3). The bandwidth of the memory bus is 160 MBytes/sec, while each external physical bus has a bandwidth of only 40 MBytes/sec. Thus spooling one message queue at peak rate consumes one quarter of the total memory and computation bandwidth. The memory agent can spool eight different message queues ''concurrently'' by interleaving the transfers at double word granularity over the 64-bit memory bus. The ability of the spooling unit to dynamically select the next logical channel on a word by word basis is especially useful when multiple messages are being spooled into (or out of) memory. It is likely that the data words of the messages will arrive at varying rates, either because of contention on the physical busses or because systolic messages come directly from the computation agent. Dynamically selecting the next spool ensures that cell memory bus bandwidth is never wasted.

Besides using a counter based termination condition, as in the case of DMA, spooling to memory can also terminate on receiving a message end marker. This is important for spooling data that was generated by systolic communication, because the number of words in a message may not be known in advance. Since spooled messages can be of arbitrary size, a mechanism is needed to ensure that a spooled message does not overflow its buffer. The current spool address is checked against an address limit register to prevent this from happening.

Once spooled to memory, the computation agent can access the data randomly using regular memory operations. Similarly, it can first prepare the message in memory before requesting the memory agent to spool out the message. This provides one way to implement memory communication.

The computation agent can also access data spooled in memory in a systolic manner as if the data just arrived over a logical channel. This is achieved by binding a systolic input gate to a queue that is allocated for internal use and binding that same queue to an output spooling gate. As the computation agent reads data from the systolic input gate, data are spooled from memory and buffered in the queue. This allows the computation agent to use implicit input operations to consume the data that were buffered in memory.

It is not necessary to wait for the entire message to be received before it can be spooled out as described above. A sequence of buffers (not necessarily consecutive) is allocated in memory to hold the incoming message, and the computation agent can start consuming data as soon as a buffer in memory is filled.

This mechanism of buffering through memory logically extends the length of the message queues in the communication agent, at a cost of adding one clock cycle to each word access. This extension in memory is necessary for those computations such as 2D convolutions [12] which require long queues and for those programs which will deadlock if the queues at the receiving cell are too short [14]. Also, a long queue reduces stalling. Since a spooled message can be read from the systolic gates in the same manner as a message received directly from another cell, the decision to buffer a message in memory can be delayed until run time.

Similarly, the data generated by the program can first be spooled into memory by binding an input spooling gate to a queue reserved for internal use and binding a systolic output gate to this queue.

7.3. Control interface to computation agent

The computation agent can inform the communication agent of events that are of interest by storing the appropriate information into the communication agent's match CAM. The communication agent notifies the computation agent of these events when they take place by storing the information in status registers.

Sample events that are of interest to the computation agent are:

- Arrival of a pathway begin marker
- Arrival of a pathway end marker
- Arrival of a message begin marker
- Arrival of a message end marker
- Arrival of an application marker

Some of these events can be registered on a per-logical channel basis; for others, they are either registered for all logical channels or not at all. An application marker is a tagged data word that can be found anywhere within a message. It is used by the application program to mark those points within a message that demand special attention by the computation agent.

When an event occurs for which the computation agent wishes to be notified, the communication agent posts this event by setting the appropriate status register. These registers are monitored by the master sequencer of the computation agent and result in a control transfer to an appropriate service routine.

The computation agent can also instruct the communication agent to modify the connections of existing pathways. One example is the joining of two pathways, as discussed in Section 6.1.6, or to route around a faulty cell.

8. Communication latency summary

When a program initiates a pathway, the originating cell's computation agent instructs the communication agent to bind the destination of a local queue to a output logical channel in the direction of the destination. If a logical channel is available, this request takes 150 ns to complete. Next, the computation agent must generate and send a pathway begin marker, which takes 100 ns. Generating additional addresses for corner turning (if required) takes 50 ns per address. At this point, the computation agent can send messages on the pathway. Tearing down a pathway requires that a pathway end marker be generated and sent; this takes 100 ns. Note, however, that the above numbers assume that the relevant data (i.e., data field of begin marker, additional addresses for corner turning, etc.) are already in registers. In practice, the run-time system imposes additional overheads such as retrieving values from a configuration table, checking for valid cell addresses, and updating various tables.

The communication agent of every intermediate cell in the pathway must decide if the incoming pathway begin marker matches on this cell or must be forwarded. If the marker does not match, it will take 200 ns to forward to the next cell. If the marker matches for corner turning, it will take 250 ns to discard the current marker, convert the next data word into a new marker and forward the new marker to the next cell. Joining two pathways is inexpensive; after the join instruction is issued, it will take 100 ns for the first data word to leave the joint cell for the next cell. Once the pathways are joined together, there is no additional latency involved for data passing through a joint cell.

Generating a message begin marker is fast because all the necessary resources are reserved at the time when the underlying pathway was initialized. The message begin marker and the message end marker are generated in 100 ns each. This time does not include any run-time system overhead to look up the destination address in the message header, or to maintain bookkeeping tables. The latency of a message header is the same as the latency of a pathway header (200 ns, plus 50 ns for corner turning) if the message is routed by the communication agent.

9. Conclusions

An iWarp system is a distributed memory machine, supporting two different styles of communication, systolic and memory communication, fully and efficiently. Housed in one system, the two styles of communication can be easily intermixed to adapt to the application needs.

The iWarp communication architecture provides a wealth of communication services. As a systolic array, iWarp allows data to stream through the cells at high data rates, with each cell cooperating at word-level granularity. This basic systolic functionality is enriched by a set of features that simplify programming without reducing efficiency. Processors can communicate with non-neighboring cells directly without involving programs at intervening cells. Using logical channels, an iWarp system can efficiently implement intercell communication topologies which are quite different from that of the hardware interconnect in the system. This capability is also useful in routing data around faults and congestion. The size of input queues can be extended indefinitely by spooling through memory, a decision that can be made dynamically and is totally transparent to the program. By redirecting data messages, a cell can have messages or portions of them forwarded to an appropriate destination automatically. This is especially useful for overlapping the input/output phase of a systolic algorithm with computation.

As a message passing machine, iWarp routes messages between cells efficiently using wormhole routing. Its ‘‘street-sign’’ routing minimizes routing overhead by imposing a default direction at each hop of the routing. On iWarp, it is possible to reserve communication bandwidth for specific classes of messages. This management of the bandwidth is important to implement system functions such as monitoring. iWarp can implement ‘‘door-to-door’’ communication by allowing data to be stored into the user space directly without buffering through system space.

This myriad of communication functionalities is provided using only a few communication mechanisms in iWarp. The two unique iWarp architectural features are logical channels and program access to the communication system. Having only a small number of basic new ideas keeps the design simple and easily optimized, and makes it possible to integrate the communication agent with the computation and memory agents in a single VLSI component.

The iWarp hardware supports a high communication bandwidth, and more importantly, the iWarp architecture can translate this raw data rate into a high communication rate between programs through the various layers of communication abstractions. First, iWarp has a peak communication bandwidth of 320 MBytes per second; the bandwidth for each bus is 40 MBytes per second. Idle logical channels do not consume bus bandwidth; the peak bandwidth of 40 MBytes per second can be fully utilized by a single logical channel when it is the only active channel. Second, dedicated communication hardware routes the messages through the system with a minimum latency. Third, to realize the efficiency at the program level, the iWarp communication agent has a unique, high bandwidth interface between the computation agent and memory. Data can be spooled into memory at a rate of 160 MBytes per second, or four messages can be accessed directly via long instructions at a rate of 40 Mbytes per second each. This high bandwidth is made possible by the integration of the communication and computation units into a single component.

The complete iWarp communication architecture is designed to deliver a high effective program communication rate to both systolic and memory communication models. Integrating this communication capability with a computation engine that delivers 20 MFLOPS and 20 MIPS into a single component, iWarp is a powerful building block for large-scale distributed memory machines and many computation and communication intensive parallel systems.

Acknowledgements

We appreciate the contributions of Dave Nedwek and An Nguyen, of Intel Corporation to the design and implementation of the iWarp communication system as well as the assistance in design validation and performance evaluation by Abu Noaman and David Yam of Carnegie Mellon University. We also thank Paul Wiley of Intel and David O’Hallaron of Carnegie Mellon for their comments on earlier versions of this paper.

References

1. Annaratone, M., Bitz, F., Clune, E., Kung, H. T., Maulik, P., Ribas, H., Tseng, P. and Webb, J. Applications and Algorithm Partitioning on Warp. COMPCON Spring ’87, IEEE Computer Society, 1987, pp. 272-275.

2. Annaratone, M., Arnould, E., Gross, T., Kung, H. T., Lam, M., Menzilcioglu, O. and Webb, J. A. "The Warp Computer: Architecture, Implementation, and Performance". *IEEE Transactions on Computers C-36*, 12 (December 1987), 1523-1538.

3. Arnould, E. A., Bitz, F. J., Cooper, E. C., Kung, H. T., Sansom, R. D. and Steenkiste, P. A. The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers. Proceedings of Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III), ACM, April, 1989, pp. 205-216.

4. Athas, W. C. and Seitz, C. L. "Multicomputers: Message-Passing Concurrent Computers". *Computer 21*, 8 (August 1988), 9-24.

5. Borkar, S., Cohn, R., Cox, G., Gleason, S., Gross, T., Kung, H. T., Lam, M., Moore, B., Peterson, C., Pieper, J., Rankin, L., Tseng, P. S., Sutton, J., Urbanski, J. and Webb, J. iWarp: An Integrated Solution to High-Speed Parallel Computing. Proceedings of Supercomputing ’88, IEEE Computer Society and ACM SIGARCH, Orlando, Florida, November, 1988, pp. 330-339.

6. Cohn, R., Gross, T., Lam, M. and Tseng, P. S. Architecture and Compiler Tradeoffs for a Long Instruction Word Microprocessor. Proceedings of Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III), ACM, April, 1989, pp. 2-14.

7. Dally, William J. *A VLSI Architecture for Concurrent Data Structures.* Kluwer Academic Publishers, 1987.

8. Dally, W. J., and Seitz, C. L. "The Torus Routing Chip". *Distributed Computing 1*, 4 (1986), 187-196.

9. Gross, T. Communication in iWarp Systems. Proceedings of Supercomputing ’89, November, 1989, pp. 436 - 445.

10. Hamey, L. G. C., Webb, J. A., and Wu, I. C. "An Architecture Independent Programming Language for Low-Level Vision". *Computer Vision, Graphics, and Image Processing 48* (1989), 246-264.

11. Hamey, L. G. C., Webb, J. A., and Wu, I. C. Low-level Vision on Warp and the Apply Programming Model. In *Parallel Computation and Computers for Artificial Intelligence*, Kluwer Academic Publishers, 1987, pp. 185-199. Edited by J. Kowalik.

12. Kung, H.T., Ruane, L.M., and Yen, D.W.L. "Two-Level Pipelined Systolic Array for Multidimensional Convolution". *Image and Vision Computing 1*, 1 (February 1983), 30-36. An improved version appears as a CMU Computer Science Department technical report, November 1982..

13. Kung, H. T. Systolic Communication. Proceedings of the International Conference on Systolic Arrays, San Diego, California, May, 1988, pp. 695-703.

14. Kung, H. T. "Deadlock Avoidance for Systolic Communication". *Journal of Complexity 4*, 2 (June 1988), 87-105. (A revised version also appears in Conference Proceedings of the 15th Annual International Symposium on Computer Architecture, June 1988, pp. 252-260)..

15. Kung, H. T. Network-Based Multicomputers: Redefining High Performance Computing in the 1990s. Proceedings of Decennial Caltech Conference on VLSI, MIT Press, Pasadena, California, March, 1989, pp. 49-66.

16. Lam, M. *A Systolic Array Optimizing Compiler*. Ph.D. Th., Carnegie Mellon University , May 1987. The thesis is published by Kluwer Academic Publishers, Boston, Massachusetts, 1988.

17. Menzilcioglu, O., Kung, H. T. and Song, S. W. Comprehensive Evaluation of a Two-Dimensional Configurable Array. Proceedings of the Nineteenth International Symposium on Fault-Tolerant Computing, 1989, pp. 93-100.

18. Ribas, H. B. *Automatic Generation of Systolic Programs from Nested Loops*. Ph.D. Th., Carnegie Mellon University , June 1990.

19. Seitz, C. L., Athas, W. C., Flaig, C. M., Martin, A. J., Seizovic, J., Steele, C. S. and Su, W-K. The Architecture and Programming of the Ametek Series 2010 Multicomputer. The Third Confererence on Hypercube Concurrent Computers and Applications., Pasadena, California, January, 1988, pp. 33-36.

20. Tanenbaum, A. S. *Computer Networks.* Prentice-Hall, 1988. Second Edition.

21. Tseng, P. S. *A Parallelizing Compiler for Distributed Memory Parallel Computers*. Ph.D. Th., Carnegie Mellon University , May 1989.

Table of Contents