Supercomputing '88
Kissimmee, Florida, November 14-18, 1988

iWarp: An Integrated Solution to High-Speed Parallel Computing

Shekhar Borkar, Robert Cohn, George Cox, Sha Gleason, Thomas Gross,
H. T. Kung, Monica Lam, Brian Moore, Craig Peterson, John Pieper,
Linda Rankin, P. S. Tseng, Jim Sutton, John Urbanski, and Jon Webb

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Intel Corporation, JF1-60
5200 N.E. Elam Young Pkwy
Hillsboro, Oregon 97124

Abstract

iWarp is a system architecture for high speed signal, image and scientific computing. The heart of an iWarp system is the iWarp component: a single chip processor that requires only the addition of memory chips to form a complete system building block, called the iWarp cell. Each iWarp component contains both a powerful computation engine (20 MFLOPS) and a high throughput (320 MBytes/sec), low latency (100-150 ns) communication engine for interfacing with other iWarp cells. Because of its strong computation and communication capabilities, the iWarp component is a versatile building block for various high performance parallel systems. These systems range from special purpose systolic arrays to general purpose distributed memory computers. They are able to support both fine-grain parallel and coarse-grain distributed computation models simultaneously in the same system. An iWarp system can include a large number of cells; the initial iWarp demonstration system consists of an 8×8 torus of iWarp cells, delivering more than 1.2 GFLOPS. It can be expanded to include up to 1,024 cells. This paper describes the iWarp architecture and how it supports various communication models and system configurations.

1. Introduction

iWarp is a product of a joint effort between Carnegie Mellon University and Intel Corporation. The goal of the effort is to develop a powerful building block for various distributed memory parallel computing systems and to demonstrate its effectiveness by building actual systems. The building block is a custom VLSI single chip processor, called iWarp, which consists of approximately 600,000 transistors.

The iWarp component contains both a powerful computation processor (20 MFLOPS) and a high throughput (320 MBytes/sec), low latency (100-150 ns) communication engine. Using nonpipelined floating-point units, the computation processor will sustain high computation speed for vectorizable as well as non-vectorizable codes.

The research was supported in part by Defense Advanced Research Projects Agency (DOD) monitored by the Space and Naval Warfare Systems Command under Contract N00039-87-C-0251.

Authors' affiliations: S. Borkar, G. Cox, S. Gleason, B. Moore, C. Peterson, L. Rankin, J. Sutton, J. Urbanski: Intel Corporation; R. Cohn, T. Gross, H. T. Kung, M. Lam, J. Pieper, P. S. Tseng, J. Webb: Carnegie Mellon University

An iWarp component connected to a local memory forms an iWarp cell; up to 64 MBytes of memory are directly addressable. A large array of iWarp cells will deliver an enormous computing bandwidth never before realized in distributed memory parallel systems. Because of the strong computation and communication capabilities and because of its commercial availability, iWarp is expected to be an important building block for a diverse set of high performance parallel systems.

The iWarp architecture evolved from the Warp machine [1], a programmable systolic array developed at Carnegie Mellon and produced by General Electric. All applications of Warp, including low-level vision, signal processing, and neural network simulation [2, 18], can run efficiently on iWarp. But systems made of the iWarp building block can achieve at least one order of magnitude improvement over Warp in cost, reliability, power consumption, and physical size. Much larger arrays can be easily built. The clock speed of iWarp is twice as high as Warp; the increase in computation throughput is matched by a similar increase in I/O bandwidth. Therefore we expect iWarp to achieve the same high efficiency as Warp. For example, the NETtalk neural network benchmark [20] runs at 16.5 million connections per second and 70 MFLOPS on a 10 cell Warp array; the same benchmark runs at 36 million connections per second and 153 MFLOPS on an iWarp array of the same number of cells.

Although the design of the iWarp architecture profited greatly from programming and applications experiences gained from many Warp machines in the field, iWarp is not just a straightforward VLSI implementation of Warp. iWarp is intended to have a much more expanded domain of applications than Warp. The following summarizes the goals of iWarp as a system building block:

- iWarp is useful for the implementation of both special purpose arrays, which require high computation and I/O bandwidth, and general purpose arrays where programmability and programming support are essential.
- iWarp is useful for both high performance processors attached to general purpose hosts and autonomous processor arrays capable of performing all the computation and I/O by themselves. That is, iWarp can be used for both ''host centric'' and ''array centric'' processing.
- iWarp supports both tightly and loosely coupled parallel processing, and both systolic [12] and message passing models of communication.

- iWarp can implement a variety of processor interconnection topologies including 1-dimensional (1D) arrays, rings, 2-dimensional (2D) arrays, and tori.
- iWarp is intended for systems of various sizes ranging from several processors to thousands of processors.

This paper will explain how the iWarp architecture addresses these objectives.

Besides conventional high level languages such as C and FORTRAN, the programming of iWarp arrays will be supported by programming tools such as parallel program generators. Previous experience and current research on Warp indicates that parallel program generators are one of the most promising approaches to programming distributed memory parallel computers. In this approach, a specialized, machine independent language is created, which embodies a particular parallel computation model (for example, input partitioning, domain partitioning, or task-queuing [13]). The compiler for that language then maps the program onto a target parallel architecture. This approach can allow efficient parallel programs to be generated automatically for large processor arrays.

Automatic parallel program generators have been developed for iWarp in two applications areas: scientific computing and image processing. The scientific computing language, called AL (Array Language) [21], incorporates the domain partitioning model and allows programmers to transfer data between a common space and a partitioned space, perform computation in parallel in the partitioned space, and then transfer data back. For scientific routines such as those found in LINPACK [5], the AL compiler generates efficient code for iWarp and Warp, as well as for uniprocessors. The image processing language, called Apply [9], incorporates the input partitioning model: the input images are partitioned among the processors, each of which generates part of the corresponding output image. Apply compilers exist for iWarp, Warp, uniprocessors, and the Meiko Computing Surface, as well as several other computer architectures. Benchmark comparisons on Apply programs have validated the above claims [22].

As of July 1988, the architecture and logic designs for iWarp have been completed. In the software area, an optimizing compiler developed for Warp [8, 17] has been retargeted to generate code for iWarp. Using this compiler, the iWarp performance on real programs, including those generated by the parallel program generators mentioned above, have been evaluated on an iWarp architecture simulator. A prototype iWarp system is expected to be operational by the end of 1989. Three demonstration systems, each consisting of an 8×8 torus of iWarp cells, are scheduled to be operational in the middle of 1990.

The organization of this paper is as follows. In the next section, we give an overview of iWarp and how iWarp systems can be constructed from them. Some sample iWarp usages and system configurations are described in Section 3. Sections 4 and 5 deal with one of the most innovative features of the iWarp architecture — the iWarp intercell communication models and mechanism. The computation part of the iWarp component is discussed in Section 6, which also includes some preliminary iWarp performance figures on the Livermore Loops benchmark. Finally, a summary of the paper and some concluding remarks are given.

2. iWarp overview

An iWarp system is composed of a collection of iWarp cells, each of which consists of an iWarp component and its local memory. This section first gives an overview of the iWarp component and then summarizes how an iWarp cell physically interfaces with the external world so that various iWarp systems can be constructed.

2.1. iWarp component

The iWarp component has a *communication agent* and a *computation agent*, as depicted in Figure 1.

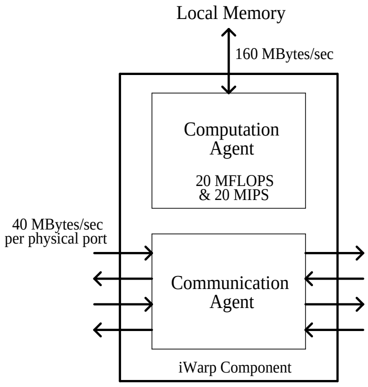

Figure 1. iWarp component overview

The computation agent can carry out computations independently from the operations being performed at the communication agent. Therefore a cell may perform its computation while communication through the cell from and to other cells is taking place, and the cell program does not need to be involved with the communication. While separating the control of the two agents makes programming easy, having the two agents on the same chip allows them to cooperate in a tightly coupled manner. The tight coupling allows several communication models to be implemented efficiently, as to be discussed in Section 4. The major blocks in iWarp are shown in Figure 2.

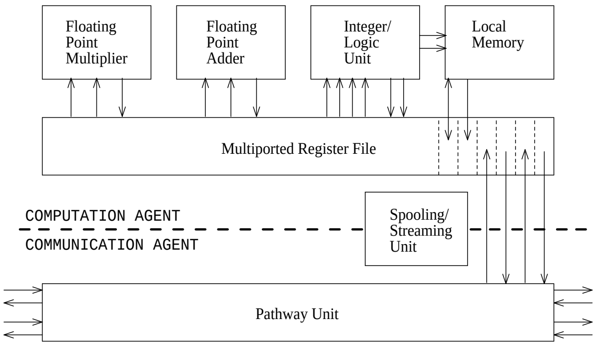

Figure 2. Major functional units in iWarp

In the following we summarize the major features in the two agents and their interface. The performance numbers are based on the expected clock speed of 20 MHz, i.e., a clock is 50 ns. Further discussions on iWarp communication and computation features are provided in Sections 5 and 6, respectively.

Communication agent

Four input and four output ports

- 40 MBytes/sec data bandwidth per port
- Word by word, hardware flow control at each port
- An output port can be connected to an input port of another iWarp cell via a point-to-point *physical bus*.

Multiple logical busses multiplexed on each physical bus.

- Maintaining up to 20 incoming pathways simultaneously in an iWarp component
- Idle logical busses do not consume any bandwidth of the physical bus.

Pathway unit

- Routing for 1D and 2D configurations
- Capable of implementing wormhole and streetsign routing schemes

Both message passing and systolic communication are supported for coarse-grain and fine-grain parallel computation.

Computation agent

Computational units

- Floating-point adder
  - 10 and 5 MFLOPS for 32- and 64-bit additions (IEEE 754 standard), respectively
  - Nonpipelined
- Floating-point multiplier
  - 10 and 5 MFLOPS for 32- and 64-bit multiplications (IEEE 754 standard), respectively
  - Nonpipelined
  - Full divide, remainder, and square root support
- Integer/logical unit
  - 20 MIPS peak performance on 8/16/32-bit integer/ordinal data
  - Arithmetic, logical, and bit operations

All the above three units may be scheduled to operate in parallel in one instruction, generating a peak computing rate of 20 MFLOPS plus 20 MIPS.

Internal data storage and interconnect

- A shared, multiported, 128 word register file
- Special register file locations for local memory and communication agent access

Memory units

- Off-chip local memory for data and instructions
  - Separate address and data busses (24-bit word address bus, 64-bit data bus)
  - 20 million memory accesses/sec peak performance
  - 160 MBytes/sec peak memory bandwidth
  - Read, write, and read/modify/write support
- On-chip program store
  - 256 word cache RAM
  - 2K word ROM (built-in functions)
  - 32- and 96-bit instructions

Communication and computation interface

Communication agent notifies computation agent on message arrival.

Dynamic flow control: Computation agent spins when reading from an empty queue or writing to a full queue in communication agent.

Hardware spools data between queues and local memory.

2.2. Forming iWarp systems

Various iWarp systems can be constructed with the iWarp cell. We describe how copies of the iWarp cell can be connected together, and how an iWarp cell can connect to peripherals to form these systems.

There are two ways that an iWarp cell, consisting of an iWarp component and its local memory, physically interfaces with the external world. Recall that the iWarp component has four input ports and four output ports. The first interface method is to use a physical *bus* to connect an output port of an iWarp to an input port of another. The former and latter port can write to and read from the bus, respectively. Thus this is a unidirectional bus between the two components, as represented by the arrowed edge in Figure 3 (a). Usually another unidirectional bus in opposite direction is also provided, so that bidirectional data communication between the two component is possible. This is illustrated in Figure 3 (b).

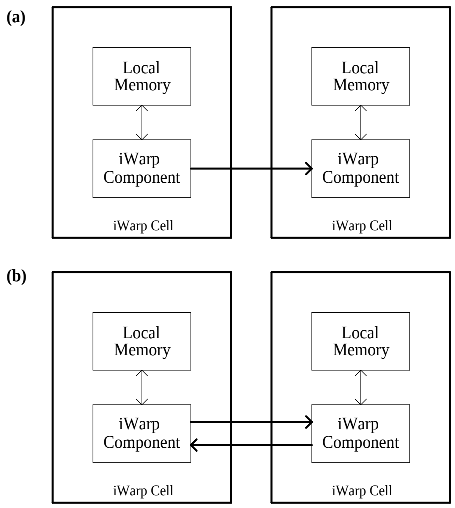

Figure 3. Intercell connection via ports of iWarp components: (a) unidirectional bus and (b) two unidirectional busses in opposite directions

The second interface method is via the local memory of the iWarp cell, as depicted by Figure 4.

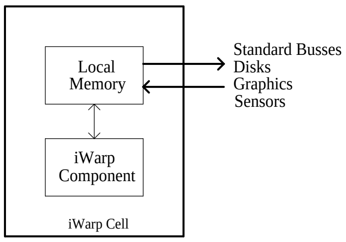

Figure 4. Connection with peripherals via local memory of an iWarp cell

Using this interface the iWarp cell can reach peripherals such as standard busses, disks, graphics devices and sensors. Therefore the iWarp cell's connection with peripherals uses the local memory, while its intercell connection uses ports of iWarp. Since these two functions use different physical resources of the iWarp cell, they can be implemented independently from each other. This implies, for example, that peripherals can be attached to any set of iWarp cells in an array of iWarp cells, independently from the array interconnection topology. With these two interface methods many system configurations can be implemented, as will be shown in Section 3.

3. iWarp usages and system configurations

The iWarp cell, consisting of the iWarp component and local memory, is a building block for a variety of system configurations. These systems can be used as general and special-purpose computing engines. This section describes some of these usages and system configurations.

3.1. General purpose arrays

With its four pairs of input and output ports, the iWarp cell is a convenient building block for a 2D array or torus. Figure 5 depicts a 3×3 torus. Peripherals can be attached to any of the iWarp cells via its local memory.

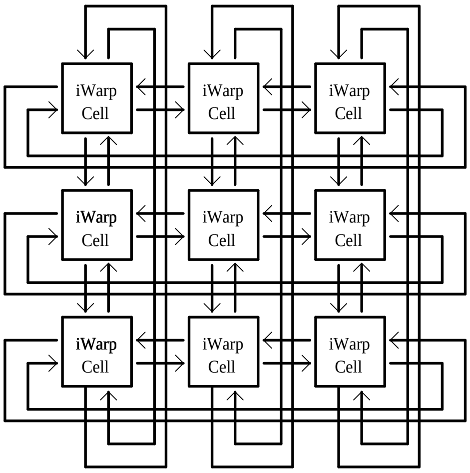

Figure 5. 3×3 torus

The initial demonstration iWarp system in 1990 is an 8×8 torus, with a total of 32 MBytes SRAM. It has a peak performance of 1,280 MFLOPS. The memory of each cell can be expanded up to 1.5 MBytes, and with different memory components, a memory of up to 64 MBytes per cell is possible. The same system design can be extended to a 32×32 torus, giving an aggregate peak performance of 20,480 MFLOPS.

The iWarp cell is a building block for 1D arrays or rings as well. Figure 6 depicts a 6-cell ring.

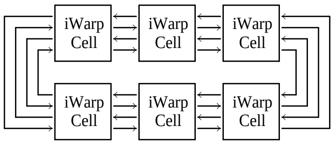

Figure 6. 6-cell ring

A 1D array or ring of a moderate number of iWarp cells, delivering on the order of hundreds of MFLOPS, can be an effective attached processor to a workstation. This has been demonstrated by the 10-cell Warp array. Using the same approach with iWarp, we will achieve an order of magnitude improvement in cost-performance over Warp. To meet the requirement of lower cost (and lower performance) applications, one or a few iWarp cells can also form a single-board accelerator for low-end workstations or PCs.

3.2. Special-purpose arrays

Many systolic algorithms can make effective use of large processors arrays for applications such as signal processing and graphics [10]. With the iWarp cell, various special-purpose arrays that execute only a predetermined set of these algorithms can easily be built. For example, a hexagonal array (as depicted in Figure 7) with unidirectional physical busses between cells can be built to execute some classical systolic algorithms for matrix operations [15]. For such an array, sensors and array output ports may be connected to the local memories of a number of cells, so that I/O can be carried out in parallel. In areas such as high-speed signal processing, special-purpose arrays can effectively use hundreds or even thousands of iWarp cells.

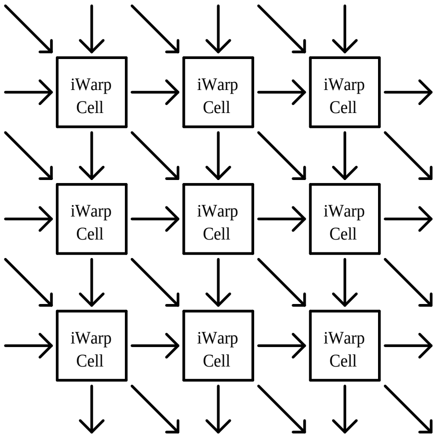

Figure 7. Hexagonal array with unidirectional physical busses between cells

In some systolic algorithms, cells on the array boundary may execute a different function from cells inside the array [7]. In this case, individual iWarp cells can be programmed to perform different functions according to their locations in the array.

In general, the performance of special-purpose arrays made of iWarp cells will be comparable to that of those arrays made of custom hardware using similar VLSI technology. Although the iWarp array will probably have a larger physical size, it can be readily programmed to implement the target algorithms and will incur a much shorter development time.

4. Communication models

Interprocessor communication is an integral part of parallel computing on a distributed memory processor array. To balance the high numerical processing capability on the processor, iWarp must be equally efficient in communication. The development of efficient parallel software is simplified if the communication cost is low and can be estimated reliably.

To motivate the communication agent design on iWarp, in this section we first describe two important communication models commonly used on distributed memory processor arrays: message passing and systolic communication. We will study the requirements to implement these models efficiently, from the data transport level all the way to the integration of communication with the data processing. We then describe a set of unifying programming abstractions to support these models. The next section shows how they are supported by the iWarp communication facility, while meeting the performance requirements of their usages.

4.1. Communication models

We have identified two communication models used for distributed memory parallel systems: *message passing* and *systolic*. They differ primarily in the granularity of communication and computation. In message passing mode, as in computer networks, the unit of processing is a complete message. That is, a message is accumulated in the source cell memory, transmitted (as a unit) to the destination cell, and only when the full message is available in the local memory of the destination cell is it ready to be operated upon. Conversely, in systolic mode [12], the unit of communication and processing can be as fine grained as a single word in a message.

4.1.1. Message passing

Message passing is a commonly used model for coarse-grain parallel computation. Processes at each cell operate independently on the cell's local data and only occasionally communicate with other cells. The timing, order, and even the communication partner are often determined at run time. The dynamic nature makes certain communication overheads unavoidable, such as routing the message across the array and asynchronously invoking the answering party. To efficiently support message passing, we need the capabilities described below.

Hardware support for 1D and 2D configurations. 1D arrays and rings are easier to build than 2D arrays and tori. However, computations on a 2D configuration can be more efficient than those on a 1D configuration for large systems with many cells. Suppose that there are n cells in the system. Using a 2D configuration, not only is the distance between cells reduced from $O(n)$ to $O(\sqrt{n})$ hops, but also is the effective bandwidth of the communication network increased since a transfer takes up fewer hops.

Spooling. Suppose a process wants to send a message to a destination cell. Since the communication network is shared, a process does not have guaranteed instantaneous access. Ideally, the sender process can simply specify the destination and message location, and continue with its processing regardless of the availability of the network, and a separate thread of control spools the data out of the memory. Similarly, another spooling process can take the data from the network and store it into the receiver's memory, with minimal interference with the computational process in progress.

Separate communication support hardware. In an iWarp array, a message may be routed through intermediate cells before reaching the final destination. The routing of data through a cell is logically unrelated to the process local to the cell. It can be supported in dedicated hardware to handle the high bandwidth of the array.

Word-level synchronization. Although the granularity of communication is a message, this does not mean that the intermediate hops should forward the data at the same grain size. In wormhole routing [3], the routing information in the header of the message can be used to set up the next leg of the communication path even before the rest of the data arrives. The contents of the message can be forwarded word by word, without having to be buffered in entirety on intermediate cells. Wormhole routing reduces the latency of communication and does not take up any of the memory bandwidth of the intermediate cells. Since the communication path is built link by link, a word-by-word handshake is necessary to throttle the data flow in case the next link is temporarily unavailable.

Multiplexing messages on a physical bus. Since wormhole routing may use up multiple links at the same time, prevention of deadlock is necessary in ring or torus architectures. One scheme to prevent deadlock is the virtual channel method, which uses another set of links when routing beyond a certain cell [4, 19]. If in each direction only one physical bus is available between two connecting cells, this deadlock prevention scheme requires that multiple communication paths be multiplexed on a physical bus. Multiplexing can also be used to keep a long message from monopolizing the physical bandwidth for an indefinitely long time.

Door-to-door message passing. When a message arrives at the destination cell, it is generally first buffered in a system memory space and then copied into the user's memory space. This extra copy can be eliminated if the data is stored directly into the desired memory location. Using the data throttling mechanism above, the receiving process can first examine the message header to determine the memory address for the message. We call this scheme of shipping data directly from a sender's data structures to a receiver's data structures, without any system memory buffering, *door-to-door* message passing.

4.1.2. Systolic communication

Systolic communication supports efficient, fine-grain parallelism. In this model, the source cell program sends data items to the destination cell as it generates them, and the destination cell program can start processing the data as soon as the first word of input has arrived. For example, the outputs of an adder in one cell can be used as operands to the adder of another, without going through the memories of either cells, in a matter of several clocks. This mode provides tight coupling and synchronization between cooperating processes.

Systolic algorithms rely on the ability to transfer long streams of intermediate data between processes at high throughput and with low latency. More importantly, the communication cost must be consistently small, because cost variations can greatly increase delays in the overall computation. This implies that dedicated communication paths are desirable, which may be neighboring or non-neighboring paths depending on communication topologies of the algorithm.

Raw data words are sent along a communication path, identified only by their ordering in the data stream. The sender appends data to the end of the data stream and the receiver must access the words in the order they arrive. Our experience with the Warp systolic array [1] shows that FIFO queuing along a communication path is useful in relaxing the coupling between the sender and the receiver. A sender does not need to wait for the receiver unless the queue is full; similarly, the receiver can process the queued data until they run out. Word-level synchronization is provided by stalling a process that tries to read from an empty queue or write to a full queue.

A special-purpose systolic array can be tailored to a specific algorithm by implementing the dedicated communication paths directly in hardware and providing long enough queues to ensure a steady flow of data. As a programmable array, iWarp processors should implement the common systolic algorithms well, but can also degrade gracefully to cover other algorithms. We have identified the following requirements for efficient support of systolic communication.

Hardware support for 1D and 2D configurations. Many systolic algorithms in signal and image processing and in scientific computing use 1D and 2D processor arrays [10]. iWarp can directly support such configurations in hardware.

Multiplexing communication paths on a physical bus. iWarp can also support other configurations, with degraded performance, if necessary. A systolic algorithm may call for more communication paths between a pair of connecting cells than those provided directly by hardware. It may require extra communication paths for configurations such as a hexagonal array, or to implement deadlock avoidance schemes [14]. Divide-and-conquer algorithms may require communication between powers of two distances away at different times of the algorithm. All these considerations motivate the need to multiplex multiple communication paths on a physical bus.

Coupling of computation and communication. The computation part of a cell needs to access the communication part directly without going through the cell's local memory. This extra source of data is a key to systolic algorithm's efficiency. For example, fine-grained systolic algorithms for important matrix operations can consume and produce up to four data words per clock. Memory bandwidth cannot match this high communication bandwidth.

Spooling. Regardless of the size of the hardware queue available on each cell, there is always some systolic algorithm that requires deeper queues. For example, a systolic algorithm for convolving a kernel with a 2D image requires some cells to store the entire row of the image [11]. Therefore, it is desirable to provide an automatic facility to overflow the data to the cell's local memory if necessary.

4.1.3. Reserving a communication subnetwork

In both message passing and systolic communication models, there is a need to multiplex multiple communication paths onto a physical bus. Efficient support of communication paths requires dedicated hardware resources, thus only a small number of paths can be provided. This resource limitation raises the issue of resource management.

The need for managing the communication resource is more pronounced in the systolic communication model. This is because the production and consumption rates of a data stream are tied directly to the computation rates of the cells. As the computation on a cell can stall and even deadlock while waiting for data, the lifetime of a communication path can be arbitrarily long. Although an idle communication path does not consume any communication bandwidth, if all the multiplexed paths on a physical bus are occupied, no other traffic can get through.

In message passing, an entire message is first prepared and buffered in the sender cell's local memory, and the message is stored into the destination cell's local memory directly. Once a communication path becomes available, the data can be spooled in and out of the memories. With a proper routing scheme to avoid deadlocks a message can always get through, although it may have to wait for a while if the network is backed up with long messages.

Both models can benefit from a mechanism to reserve a set of communication paths for a class of messages. For instance, we can reserve a set of communication paths for system messages for purposes such as synchronization, program debugging and code downloading. First of all, this guarantees that the system can reach all the cells even if the user uses up all other paths. Moreover, since the system has full control over all messages on the reserved network, the behavior of the network is more predictable, and attributes such as a guaranteed response time are possible.

4.2. Messages and pathways

Message passing and systolic communication are two kinds of communication. The former supports coarse-grain parallelism where processes at different cells behave independently, and the latter supports fine-grain parallelism where processes at different cells cooperate synchronously. However, on examining the requirements to make message passing *efficient* and systolic communication *general,* they are not that dissimilar. For example, wormhole routing uses up multiple hardware links simultaneously, much like a communication path in systolic communication that connects two non-neighboring processes. On iWarp, both communication models can be unified and supported efficiently by the same programming abstractions of a *pathway* and a *message*, defined below.

A *pathway* is a direct connection from a cell (called the source cell) to another cell (called the destination cell). Each segment of the pathway that connects the communication agent of a cell to the computation agent of the same cell or to the communication agent of another cell is called a *pathway segment*. (See Figure 9 for examples of pathways.)

A *message* consists of a header, a sequence of data words, and a marker denoting the end of the message. Messages can be sent from the source cell to the destination cell over a pathway. The pathway is initiated and terminated by the source cell. It assembles a header containing a destination address and additional routing information and hands it to the communication agent. The source cell closes the pathway by sending a special marker to signal the end.

Normally, one pathway is set up for each individual message. The source cell opens a pathway to the destination cell, sends its message, and then closes the pathway. In the message passing model, the sending process dynamically creates a new pathway and message for each data transfer. Not all intermediate links of a pathway need to exist at the same time. In wormhole routing, the marker denoting the end of the pathway may have reached an intermediate cell even before the header and data reach the destination. In the systolic communication model, the cells typically set up required pathways for a longer duration. The sending program transmits individual data words along a pathway as they are generated without sending any additional headers or markers. On termination, the cell programs close the message and the pathway.

However, it is possible that a pathway is set up for multiple messages. That is, the sender cell does not take down the pathway immediately after the first message has passed through, so the sender can send further messages over the same pathway. The sender cell has reserved the pathway for its future use.

Reservation of multiple pathways is also possible on iWarp. Two pathways are said to be connected if the destination cell of one is the source of the other. The sender of the first pathway can send messages to the destination of the second using both pathways. In this way, a cell can send messages to multiple destinations through a set of reserved pathways.

5. iWarp communication

This section describes how the communication agent on iWarp implements the above programming abstractions and satisfies the performance requirements of both the message passing and systolic communication models. We break down the functionality of the communication agent into four categories. The categories and the requirements they fulfill are summarized as follows:

1. ***Physical communication network***: Hardware support for 1D and 2D configurations.
2. ***Logical communication network***: A mechanism to multiplex multiple pathway segments on a physical bus on a word level basis.
3. ***Pathway unit***: A mechanism to establish pathways.
4. ***Streaming and spooling unit***: Direct access of communication agent from the computation agent, and a spooling mechanism to transfer data from and to local memory.

5.1. Physical communication network

The communication network of iWarp is based on a set of high bandwidth point-to-point physical busses, linking the input and output ports of a pair of cells. Each cell has four input and four output ports, allowing cells to be connected in various topologies. Figure 8(a) illustrates the 2D array configuration where each cell is connected bidirectionally to four cells.

Each physical bus can transmit one 32-bit word data every 100 ns. The VLSI, custom chip implementation made it possible to have fine-grain, word level handshaking without any synchronization delay. Thus, each physical bus has a data bandwidth of 40 MBytes/sec, giving an aggregate data transfer rate of 320 MBytes/sec.

5.2. Logical communication network

Besides the four input and four output *external* busses described above for connecting to other cells, a communication agent is also connected to the computation agent in its cell through two input and two output *internal* busses. Each of these busses can be multiplexed on a word level basis to support a number of *logical busses* in the same direction, whereas each logical bus can implement one pathway segment of a pathway at a time.

To the communication agent on a cell, there are four kinds of logical busses:

1. incoming busses from communication agent of a neighboring cell,
2. incoming busses from computation agent of the same cell,
3. outgoing busses to communication agent of a neighboring cell, and
4. outgoing busses to computation agent of the same cell

The mapping of the logical busses to physical busses is performed statically, under software control. The hardware allows the total number of incoming logical busses in the communication agent of each cell to be as large as 20. For example, in a 2D array, the logical busses can be evenly distributed among the four neighbors and the computation agent, as shown in Figure 8(b). In this case, the heart of the communication agent is a 20×20 crossbar that links incoming logical busses to outgoing logical busses. Logical busses are managed by the source, i.e., the sending cell. The sender can initiate communication using any of its pre-allocated free logical busses without consulting the receiver. This design minimizes the time needed to set up a pathway between cells.

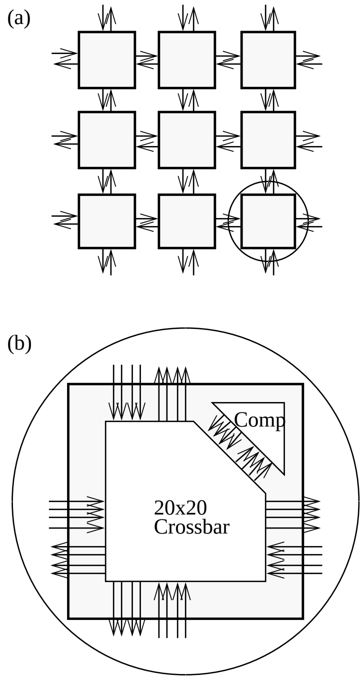

Figure 8. (a) Physical communication network
(b) logical busses of a cell

5.3. The pathway unit

A pathway is formed by connecting a sequence of pathway segments together. Figure 9 contains an example of three pathways through some cells in a 2D array. Pathway 1 connects the computation agent of B to that of A through a pathway segment between the communication agents of the two cells. Pathway 2 passes through cell A, turns a corner at cell C and finally reaches the destination D. Lastly, pathway 3 passes through both cells C and D. Two pathway segments are multiplexed on the physical bus from cell C to cell D.

A new pathway is established by the use of a special *open pathway marker*. As the communication agents pass the open pathway marker along from cell to cell, they allocate resources to form the pathway.

Open pathway markers carry *addresses* to tell how to route them from their source to their destination, using *streetsign* routing (e.g., ‘‘go to Jones and stop’’). There are two components of such a streetsign: the *streetname* (e.g., Jones) and the associated *action* (e.g., stop). The controller provides special hardware support for address recognition with a multiple entry *Address Match CAM*. For example, a pathway route might consist of ‘‘go to Jones, turn right, go to Smith, turn left, go to Johnson, and stop’’. Each pathway unit is responsible for recognizing addresses of open pathway markers requiring service or attention at that cell. Given the sequential nature of streetsign routing interpretation, a given communication agent needs to deal with only the ‘‘next’’ streetname on passing open pathway markers.

Upon the arrival of an open pathway marker, the pathway unit interprets the address to see if it is addressed to this cell, and, if so, posts an event to the computation agent to invoke the appropriate routine. Otherwise, it finds a free outgoing logical bus along the route given by the header and connects it to the incoming logical bus. The pathway is dismantled, one link at a time, by the flow of a *close pathway marker* along the pathway, cell to cell, from the source to the destination.

The latency of communication through a cell is 100 ns normally, and 150 ns in the case of corner turning. The interpretation of addresses and the establishment of pathways are completely performed by hardware. Creating a new pathway segment does not incur any additional time delay.

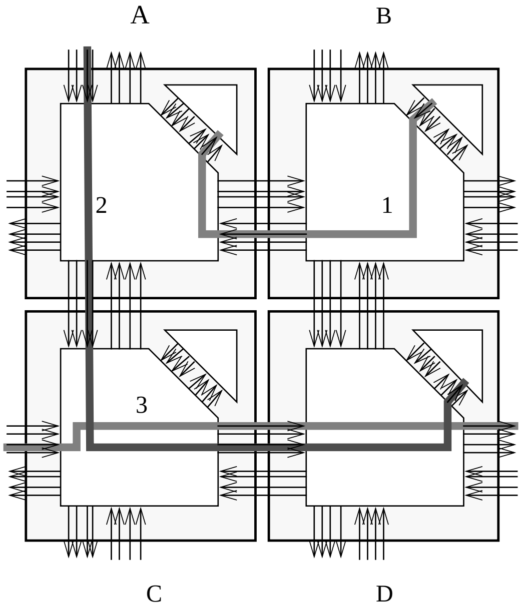

Figure 9. Pathways in a 2D array

5.4. The streaming and spooling unit

The computation agent can get access to the communication data by (1) directly accessing the communication agent a word at a time, or (2) spooling the data in and out of local memory using special hardware support.

Programs can read data from a message or write data to a message via the side effects of special register references. These special registers are called *streaming gates*, because they provide a ‘‘gating’’ or ‘‘windowing’’ function allowing a stream of data to pass, word by word, between the communication agent and the computation agent. There are two input gates and two output gates. These gates can be bound to different logical busses dynamically. A read from the gate will consume the next word of the associated input message; correspondingly, a write to an output gate will generate the next word of the associated output message. Data word-at-a-time synchronization is expressed in algorithms by the side effects of gate register references (e.g., a read of an input gate at which no data is available causes the instruction to spin until the data arrives).

iWarp also provides a transparent, low overhead mechanism for transferring data between the pathway unit and the local memory via *spooling gates*. Spooling has low overhead to avoid significant reduction of the efficiency of any ongoing or parallel computation. Spooling is transparent except for delays incurred due to either cycle stealing (i.e., for address computation) or local memory access interference from other memory references (i.e., due to concurrent cache or instruction activities).

6. iWarp computation

The iWarp processor is designed to execute numerical computations with a high sustained floating-point arithmetic rate. The iWarp cell has a high peak computation rate of 20 MFLOPS for single precision and 10 MFLOPS for double precision. More importantly, iWarp can attain a high computation rate consistently. This is because the multiple functional units in the computation agent are directly accessible through a long instruction word (LIW) instruction. By translating user’s code directly into these long instructions using an optimizing compiler [17], a high computation rate can be achieved for all programs, vectorizable or not.

6.1. The computation agent

The computation agent has been optimized for LIW controlled, parallel operation of multiple functional units. Chief among these optimizations are:

- nonpipelined floating-point arithmetic units,
- inter-unit and intra-unit, output to input, operand bypassing,
- parallel, hardware supported, zero-overhead looping,
- large, shared, multi-ported register file,
- a high bandwidth, low latency (no striding penalty) memory,
- high bandwidth, low latency interface with the communication agent.

The LIW workhorse instruction of iWarp is called the *ComputeAndAccess* (C&A) instruction. As an example of the parallelism available, a loop with code

```
FOR i := 0 TO n-1 DO BEGIN
    f:=(A[i]*B[3*i])+f;
END;
```

is compiled into a loop that initiates one iteration every cycle surrounded by a loop prologue and epilogue to get the iterations started. Similarly, a loop body that reads a value **`V1`** from one message, **`V2`** from another message, computes **`V1 * V2 + C[i]`** and sends the result as well as **`V2`** on to the next processor is also translated into a single C&A instruction in the loop body. The single precision C&A instruction executes in two clocks, and the double precision C&A instruction executes in four clocks, so both loops execute at the peak computation rate of the processor.

A C&A instruction requires up to 8 operands and produces up to 4 results. Memory accesses may produce or consume up to two of those operands: either a read and a write or two reads. Each memory reference includes an address computation (e.g., an indexing operation with a non-unit stride). The C&A instruction employs a read-ahead/write-behind pipeline that makes memory read operands from one instruction available for use in the next. Conversely, computational results of one instruction are written to memory during the next.

Those operand references that are not satisfied by the memory read operation or read from a gate (see Section 5.4) must be to the register file. These operands may themselves be the results of previous operations (e.g., intermediate results held in the register file). To avoid any interinstruction latencies, the results of the integer/logical unit or a floating-point unit may be ‘‘bypassed’’ directly back to that unit as an input operand, without waiting for the destination register file location to be updated. Also, the results of either floating-point unit may be ‘‘bypassed’’ directly to the other floating-point unit (e.g., to support multiply-accumulate sequences).

Thus, the execution of a single C&A instruction can include up to one floating-point multiplication, one floating-point addition, two memory accesses (including two integer operations for addressing), four gate accesses, several more register accesses (enough to provide the rest of the required operands), and branching back to the beginning of the loop.

Incremental to the single ‘‘long’’ C&A instruction, the iWarp computation agent provides a full complement of ‘‘short’’ instructions. They can be thought of as 2 and 3 address RISC-like instructions. These ‘‘short’’ instructions are provided to make iWarp a generally programmable processor. They usually control only a single functional unit.

6.2. Livermore Loops performance

The Livermore Loops [6], a set of computational kernels typically found in scientific computing, have been used since the 1960’s as a benchmark for computer systems. The loops range from having no data dependence between iterations (easily vectorizable) to having only a single recurrence (strictly sequential). This combination of vector and scalar code provides a good measure on the performance of a machine across a spectrum of scientific computing requirements.

The Livermore loops were manually translated from FORTRAN to W2. (W2 is a Pascal-like language developed for the Warp machine. The retargeted W2 compiler [8, 17, 16] has been used as a tool in developing and evaluating the iWarp architecture.) The translation into W2 was straightforward, preserving loop structures and changing only syntax, except for kernels 15 and 16, which were translated from Feo’s restructured loops [6].

The performance of Livermore Loops (double precision) on a single iWarp processor is presented in Table 6-1. The unweighted mean is 4.2 MFLOPS, the standard deviation is 2.6 MFLOPS and the harmonic mean is 2.7 MFLOPS. Since the machine’s peak double-precision performance is 10 MFLOPS, these numbers demonstrate a highly effective use of the raw computation power of iWarp.

The iWarp cell is a scalar processor, and does not require that loops be vectorizable for full utilization of its floating-point units. This is why it does not exhibit the same tremendous disparity in MFLOPS rates for the different loops as do vector machines. Nonetheless, the variation between the MFLOPS rates obtained is still significant. Near peak performance can be achieved (using a high-level language and an optimizing compiler), as demonstrated by kernels 3 and 7. On the other hand, performance of near 1 MFLOPS is also observed. The factors that limit iWarp performance are data dependency and the critical resource bottleneck.

Kernel	MFLOPS	Kernel	MFLOPS
1	8.2	13	1.8
2	3.3	14	3.2
3	9.8	15	1.6
4	3.2	16	0.8
5	3.3	17	1.6
6	5.0	18	6.6
7	9.9	19	4.0
8	7.4	20	3.1
9	6.7	21	4.1
10	2.0	22	3.1
11	2.0	23	6.3
12	2.5	24	0.9

Table 6-1: Double precision performance of Livermore Loops on a single iWarp cell

Data dependency. Consider kernel 5:

```
FOR i := 0 TO n-1 DO BEGIN
    X[i] := Z[i] * (Y[i] - X[i-1]);
END;
```

The multiplications and additions are serialized because of the data dependencies. Just by this consideration alone, iWarp is limited to a peak performance of 5 MFLOPS on this loop. However, iWarp still executes data dependent code better than vector machines. The floating-point units are not pipelined, and there is no penalty on non-unit stride memory accesses. More importantly, not all data dependencies force the code to be serialized. As long as the loop contains other independent floating-point operations, the floating-point units can still be utilized. This is a unique advantage an LIW architecture has over vector machines.

Critical resource bottleneck. The execution speed of a program is limited by the most heavily used resource. Unless *both* the floating-point multiplier and adder are the most critical resources, the peak MFLOPS rate cannot be achieved. Programs containing no multiplications cannot run faster than 5 MFLOPS since the multiplier is idle all the time. For example, for kernel 13 since the integer/logical unit is the most heavily used resource, the MFLOPS measure is naturally low.

7. Summary and conclusions

iWarp is the first of a new class of parallel computer architectures. iWarp integrates both the computation and communication functionalities into a single VLSI component. The communication models supported range from large-grain message passing to fine-grain systolic communication.

The computation agent of an iWarp component contains floating-point units with a peak performance of 20 and 10 MFLOPS for single and double precision operations, respectively, as well as an integer/logical unit that performs 20 million integerk or logical operations per second. The communication agent operates independently of the computation agent; since both are implemented on a single chip, tight coupling between communication and computation is possible. This permits efficient systolic communication, as well as low-overhead message passing.

iWarp is designed to be a building block for high performance parallel systems. Not only does iWarp have impressive computational capabilities, it also has exceptional communication capabilities, making iWarp suitable for both scientific computing and high speed signal processing. The first iWarp based systems will be 1D arrays, rings, 2D arrays or tori, but the iWarp component is flexible enough to be used in numerous other organizations.

We anticipate iWarp to have a significant impact on the practice of parallel computing. Arrays of thousands of cells are feasible, programmable, and much cheaper than many other supercomputers of comparable power. iWarp systems can have a variety of goals: they can be special or general purpose, and experimental or commercial. The support of well accepted languages for the cell like FORTRAN and C, together with parallel program generators to simplify the programming of the array, make it possible to program the diverse parallel machines that can be realized with iWarp components.

The iWarp component has to meet the diverse requirements of fine-grain and coarse-grain communication for various applications including scientific computing and signal processing. The design of the iWarp component has convinced us that these requirements are not incompatible and, in fact, do reinforce each other. The high bandwidth/low latency communication mechanism in iWarp implements both message passing and systolic communication efficiently. This synergy makes iWarp a suitable building block for the affordable supercomputing systems of the future.

References

1. 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.

2. Annaratone, M., Bitz, F., Deutch, J., Hamey, L., Kung, H. T., Maulik, P., Ribas, H., Tseng, P. and Webb, J. Applications Experience on Warp. Proceedings of the 1987 National Computer Conference, AFIPS, 1987, pp. 149-158.

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

4. Dally, W.J., Seitz, C.L. "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks". *IEEE Transactions on Computers C-36,* 5 (May 1987), 547-553.

5. Dongarra, J.J., Bunch, J.R., Moler, C.B. and Stewart, G.W.. *LINPACK Users' Guide.* Society for Industrial and Applied Mathematics, Philadelphia, 1979.

6. Feo, J. T. "An Analysis of the Computational and Parallel Complexity of the Livermore Loops". *Parallel Computing 7,* 2 (June 1988), 163-186.

7. Gentleman, W.M. and Kung, H.T. Matrix Triangularization by Systolic Arrays. Proceedings of SPIE Symposium, Vol. 298, Real-Time Signal Processing IV, Society of Photo-Optical Instrumentation Engineers, August, 1981, pp. 19-26.

8. Gross, T. and Lam, M. Compilation for a High-performance Systolic Array. Proceedings of the SIGPLAN 86 Symposium on Compiler Construction, ACM SIGPLAN, June, 1986, pp. 27-38.

9. 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.

10. Kung, H.T. "Why Systolic Architectures?". *Computer Magazine 15*, 1 (Jan. 1982), 37-46.

11. 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..

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

13. Kung, H. T. "Computational Models for Parallel Computers". *Philosophical Transactions of the Royal Society* (1988).

14. Kung, H. T. Deadlock Avoidance for Systolic Communication. Conference Proceedings of the 15th Annual International Symposium on Computer Architecture, June, 1988, pp. 252-260.

15. Kung, H.T. and Leiserson, C.E. Systolic Arrays (for VLSI). Sparse Matrix Proceedings 1978, Society for Industrial and Applied Mathematics, 1979, pp. 256-282.

16. Lam, M. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. ACM Sigplan '88 Conference on Programming Language Design and Implementation., June, 1988.

17. Lam, M. *A Systolic Array Optimizing Compiler*. Ph.D. Th., Carnegie Mellon University, May 1987.

18. Pomerleau, D. A., Gusciora, G. L., Touretzky, D. S. and Kung, H. T. Neural Network Simulation at Warp Speed: How We Got 17 Million Connections Per Second. Proceedings of 1988 IEEE International Conference on Neural Networks, July, 1988, pp. 143-150.

19. Raubold, E. and Haenle, J. A Method of Deadlock-Free Resource Allocation and Flow Control in Packet Networks. Proceedings of the Third International Conference on Computer Communication, International Council for Computer Communication, August, 1976.

20. Sejnowski, T. J., and Rosenberg, C. R. "Parallel Networks that Learn to Pronounce English Text". *Complex Systems 1*, 1 (1987), 145-168.

21. Tseng, P. S., Lam, M. and Kung, H. T. The Domain Parallel Computation Model on Warp. Proceedings of SPIE Symposium, Vol. 977, Real-Time Signal Processing XI, Society of Photo-Optical Instrumentation Engineers, August, 1988.

22. Wallace, R. S., Webb, J. A. and Wu, I-C. Architecture Independent Image Processing: Performance of Apply on Diverse Architectures. Third International Conference on Supercomputing, International Supercomputing Institute, Inc., Boston, MA, May, 1988, pp. 25-34.