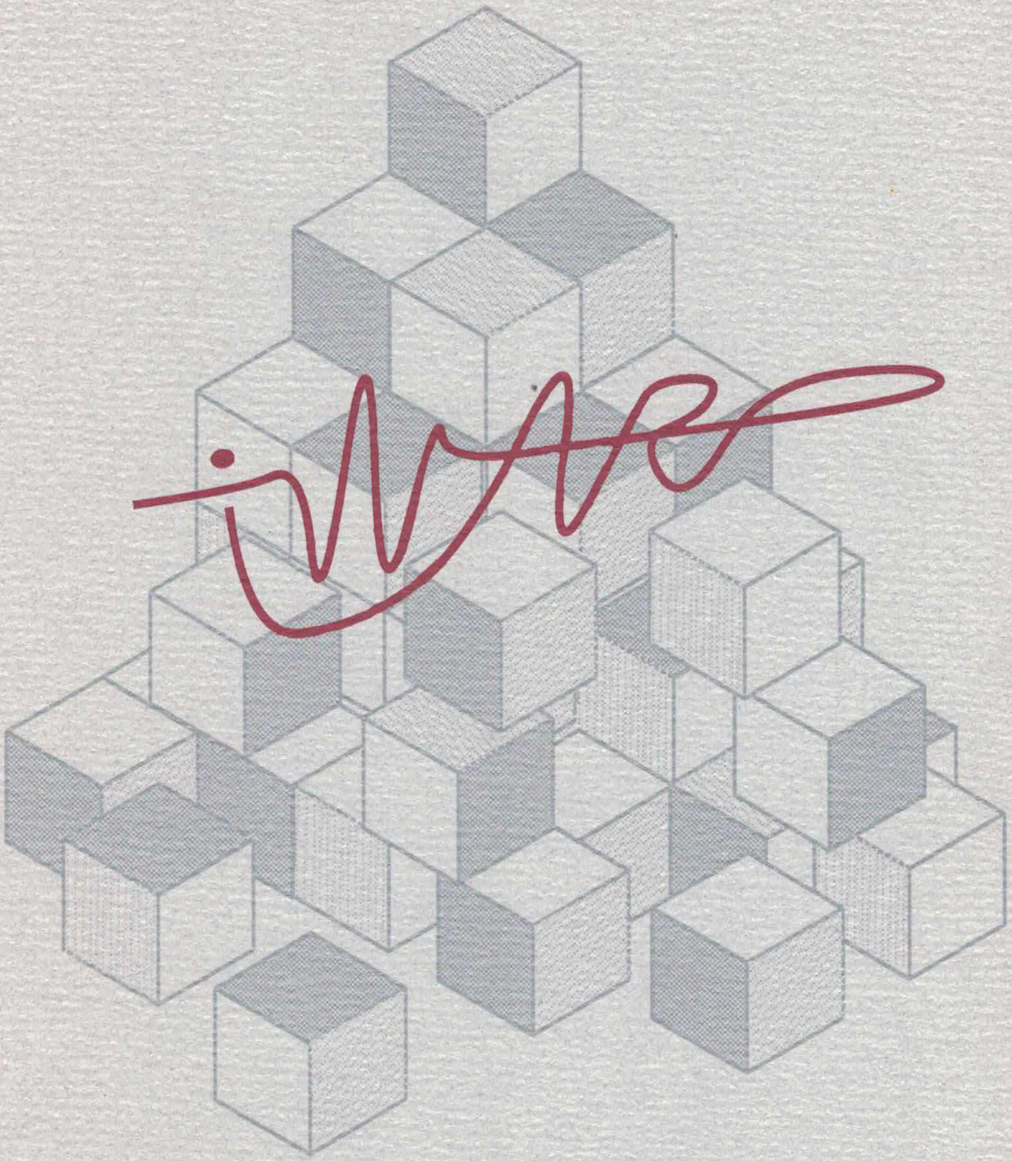
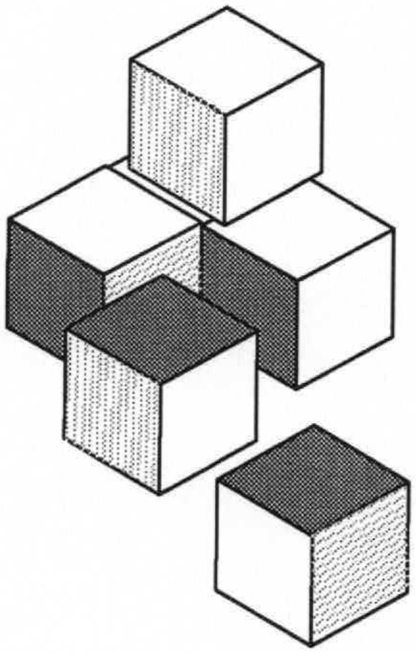
INTRODUCTION TO iWarp



intel

Introduction to iWarp



DISCLAIMER

ACKNOWLEDGEMENT

The work resulting in the concepts and systems described in this manual is supported, in part, by Defense Advanced Projects Agency (DoD), ARPA Order No. 4864, monitored by the Space and Naval Warfare Systems Command under Contract number 00039-C-0134 and Contract Number 00039-87-C-0254 and, in part, by Intel Corporation.

We particularly acknowledge the efforts of the Carnegie Mellon University, School of Computer Science, Warp team. This team included, Dr. H.T. Kung, his associates, and his students who provided the primary technical direction for iWarp technology.

TRADEMARKS

CONTENTS

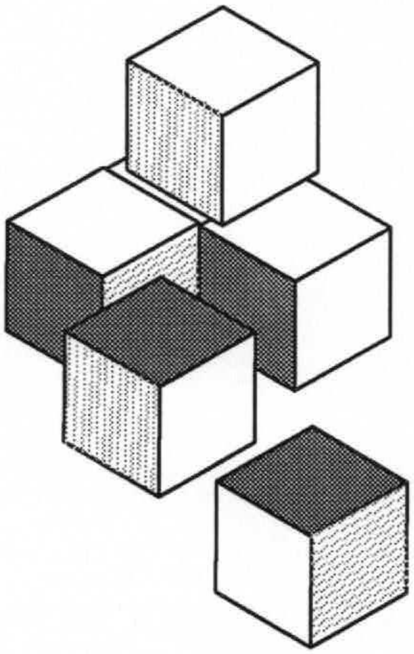
Figures

1 Introducing iWarp

intel

WARP ℠

Carnegie Mellon



iWarp: System Building Blocks for High Performance Systems

iWarp is a parallel computer system that supports a broad range of high-performance computational algorithms including matrix operations and signal and image processing. The system architecture is modular, using a single VLSI component plus memory as a building block to support a variety of connection topologies. Systems are configured as a two-dimensional array of processors, as in Figure 1-1, with performance that scales from 20 to 20,000 millions of floating-point operations per second (MFLOPS). Each iWarp cell consists of an iWarp component and its local memory. Cells communicate by passing messages over the communication channels that connect them, and the I/O capacity for the system scales with the computation power.

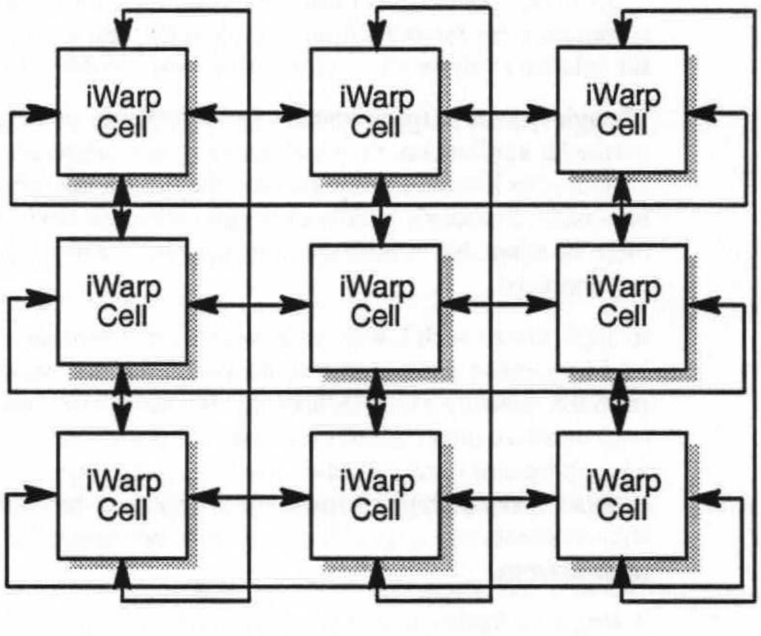


Figure 1-1: An example of a two-dimensional iWarp torus array

The Development Partnership

Intel has developed the iWarp system with Carnegie Mellon University under a four-year cost shared development contract sponsored by the Defense Advanced Research Projects Agency (DARPA). The iWarp system is a substantial advancement in high-performance computer technology created through the partnership of industry, academia, and government. Industry provided VLSI development and systems and production technology, academia provided research basis and conceptual insight, and government provided the vision and support of sponsorship.

The iWarp program is supported under DARPA's Strategic Computing Program for the development of programmable systems that use systolic array technology. The systolic architectural concept is a fine-grain (few calculations per I/O operation) computational technique pioneered by Professor H.T. Kung and his students at Carnegie Mellon University. Systolic architectures achieve high computational efficiency and performance for large arrays of processors and are particularly well suited to the computational needs of signal, image, and matrix processing.

The Need

Historically, high-performance signal and image processing applications users have been forced to use special-purpose system designs to obtain the billions of floating-point operations per second (GFLOPS) performance required. Typical applications like adaptive beam-forming for sonar, image analysis and recognition for factory automation systems, and elastic wave equation modeling for seismic analysis all require performance in the 1-10 GFLOPS range.

Though special-purpose systems can effectively meet specific requirements for a particular application, they are generally not adaptable to other needs. Fielded systems can be obsolete by the time they reach operational status. More important, in today's rapidly changing, complex environment, these systems must often be adaptable to both dynamic operational needs and changing requirements.

In applications with lower performance requirements, the use of commercial VLSI microcomputer components has been a real benefit. This benefit is derived from the maturity of the technology, the economies gained by large-scale, cost-sensitive manufacturing, and the extensive software support that comes from a broad base of users. Further, these systems also benefit from the evolutionary enhancements of market-driven microprocessor technology. System enhancements and upgrades can often be accomplished with simple board replacements.

A similar infrastructure is provided by the iWarp VLSI component for high-performance systems. Signal and image processing applications are most dramatically benefited.

iWarp meets these needs with the following capabilities:

Communication level

- **high communication performance and I/O capability**

  iWarp communication capability scales with increasing computational power to meet the needs of I/O intensive applications.

- **low overhead,course-grain, message-based, communication**

  iWarp's cut-through and worm-hole routing mechanisms provide efficient point-to-point data message delivery to any destination in the array without intervention by intermediate processing elements.

- **multiple logical connections on each physical I/O bus**

  A number of logical connections can be maintained on each physical bus connecting two iWarp cells. This substantially expands the set of intercell communication models that an iWarp array can support.

- **low overhead, fine-grain systolic communication**

  iWarp is the first commercial processor to support fine-grain systolic communication. By using data directly from the communication pathway, computational algorithms avoid memory bottlenecks, latency is reduced, and large parallel systems perform closer to peak performance expectations.

- **support for multi-functional systems**

  iWarp's combination of fine-grain systolic and coarse-grain message-based communication allow the array to be divided into independent functional sub-arrays that interact asynchronously for applications with high complexity.

Computation level

- **high computational power at low cost**

  iWarp computational power scales to 20 GFLOPS, and high capacity VLSI production technology brings new economics to high performance computing.

- **high computational density**

  iWarp supports a computational density of 1 to 3 GFLOPS per cubic foot, opening up general purpose functionality to application-specific systems.

- **low latency, high performance scalar computations**

  iWarp floating-point arithmetic elements in each processor optimize scalar performance by executing a complete arithmetic operation in a single instruction. The single-cycle long instruction word architecture allows iWarp to achieve high performance without vectorization.

System level

- **modularity**

  iWarp's single component requires only memory chips to form a complete processing element that can be combined to form a variety of general-purpose processor arrays of various sizes. System building blocks provide flexibility for rapid prototyping and feasibility demonstration.

- **broad applicability and flexibility**

  iWarp's balance between low-latency communication and high-speed computational performance gives it applicability over a broad range of computational algorithms. These features also provide the architectural basis for optimized compilers and development tools for array-level programming.

- **high-level programming environment**

  iWarp provides a high-level programming environment using C and FORTRAN languages, with parallel application development tools to speed development and minimize life cycle support and maintenance costs.

- **immunity to fault and failure**

  iWarp's on-chip error detection and reporting, logical connections, and source-routed communication allows isolation of faults and supports the development of reconfigurable systems with fault tolerance and graceful degradation properties.

Computation and Communication Requirements

Signal and image processing algorithms are characterized by relatively few computations per data element, as compared with scientific algorithms. This relationship is illustrated in Figure 1-2, where the number of arithmetic calculations per data element is compared to data base sizes for business applications, signal and image processing applications, and scientific applications.

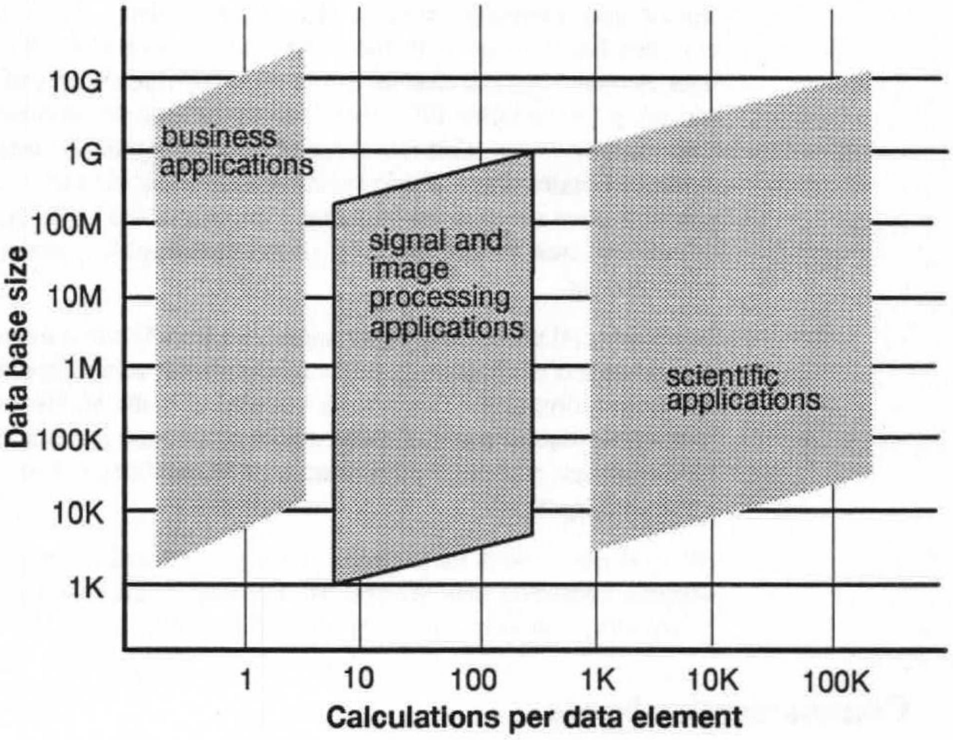


Figure 1-2: Applications with different computation and communication requirements

Comparing Requirements

The number of calculations per data element is a primary design issue that impacts the architecture of all computer systems. For instance, business applications, like transaction processing, perform relatively little computation but are dominated by I/O operations. Systems must support a large number of I/O channels with many peripheral devices. Thus, the principal design goals are maximizing I/O capacity and providing the ability to handle many independent tasks.

Conversely, scientific applications, particularly those of the supercomputer class, require thousands of calculations per data element. Additionally, the number of calculations per data element increases rapidly with the size of the problem. For example, there is a 30-fold increase in the number of calculations per data element between an order 32 Gaussian Elimination problem containing 1024 data elements and an order 1000 problem (one million data elements). Architectures for scientific computers are, therefore, focused on raw computational performance and memory capacity. The highest level of performance and greatest efficiency are obtained when the entire problem is resident, and the machine can operate for hours or days until the problem is completed.

Signal and image processing systems differ in that a typical algorithm requires only 10 to 100 calculations per data element. The calculations per data element remain nearly constant with the size of the problem. For example, a 1024-point complex Fast Fourier Transform (FFT) requires only about 20 calculations per data element. An increase in problem size by three orders of magnitude to a one million-point complex FFT only doubles the required number of calculations per data element to 40. This low reuse of data for signal and image processing systems dictates that scalable systems must be capable of matching increases in compute power with corresponding increases in I/O capacity. High performance computing must be supported by a corresponding I/O capacity with the external environment.

Typical signal processing applications require I/O performance that exceeds by several orders of magnitude the requirements of scientific machines having similar performance. This need is coupled with the additional real-time interactive requirements of many signal processing applications. These requirements amplify the importance of I/O architecture to signal and image processing systems.

iWarp systems meet these needs by supporting an I/O performance of 320 MBytes/s between two iWarp cells. For interfacing with the outside world, an iWarp array can have a number of 40 MBytes/s external I/O interfaces.

Communication Needs

iWarp supports both the coarse-grain message-based communication model of traditional parallel systems and a new fine-grain systolic communication model that is particularly well suited to high performance signal and image processing applications. The systolic model allows the computational element of each processor to use data directly from any of the four communication pathways without sacrificing memory access bandwidth. The term systolic illustrates the concept that data from external sensors can flow, or be pumped, through the array of processors as it is used simultaneously in cell computations.

The coexistence of these two forms of communication is essential for building efficient and flexible high-performance real-time systems. The coarse-grain message-based communication provides the means for interaction between independent heterogeneous tasks and cooperating elements in practical systems. The fine-grain systolic model provides the most efficient use of system resources for tightly coupled compute-intensive tasks.

Application Examples

iWarp's ability to simultaneously handle fine-grain and coarse-grain communication is fundamental for supporting a broad application base and for achieving high utilization from the cells in an iWarp array. The sections that follow illustrate these concepts.

A Sonar Example

A simple sonar application illustrates the benefits of iWarp fine-grain and coarse-grain communication models. Figure 1-3 shows a functional breakdown of the example.

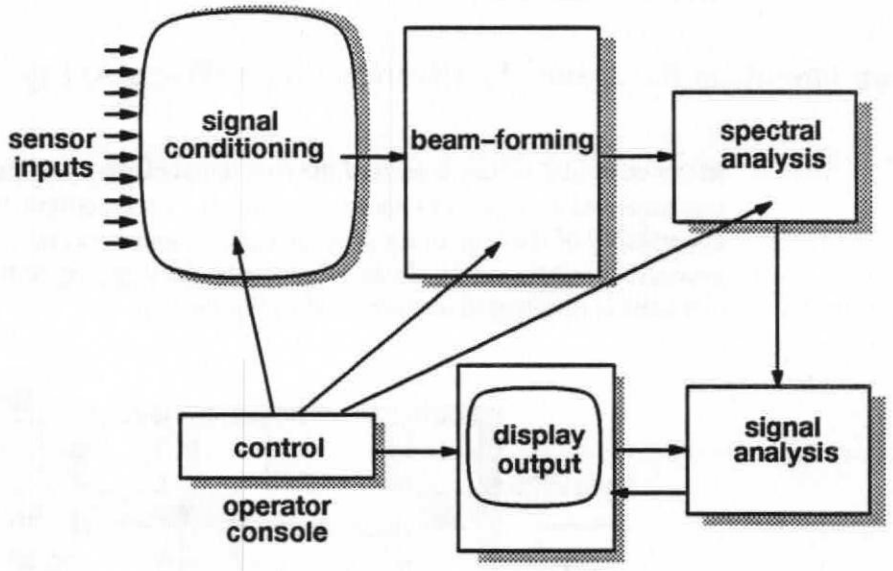


Figure 1-3: Sonar application

The processing task divides into several functional blocks that are performed in a multi-function pipelined fashion. The sensor inputs are signals that might be derived from an array of hydrophones being towed behind a ship. For the first stage of processing, these signals are filtered and digitized by the signal conditioning section of the system and sent to the beam-forming section.

The beam-forming section can use a variety of computational algorithms for forming steerable acoustic monitoring beams. Sophisticated techniques can account for the changing shape of the array as the ship maneuvers and adapt to the noise environment by nulling out undesired signals while enhancing other regions of interest. These techniques depend heavily on matrix linear algebra using QR decomposition and singular value decomposition as the basic computational algorithms.

The output of the beam-forming section is passed to the spectral analysis section. This section computes a spectrum for each of the beams. The rotating machinery of ships, submarines, and other marine devices produces acoustic energy that has characteristic spectral patterns, which help to detect and identify the source. Typically, the FFT and related signal processing algorithms are the core functions for this task.

The spectral output then passes to the signal analysis section where analysis of the spectral data is performed. Detection of a suspect event can be used to alert an operator or generate control functions that change the operational parameters for the other sections of the processor. For example, detection of energy in a specific area of interest might trigger a high resolution mode for the spectral analysis section so the structure of the signal can be more accurately evaluated. Additionally, it could direct the beam-former to lay a tight beam on the region of interest for improved noise immunity. Computational requirements vary depending on operational mode and signals being analyzed. The ability to dynamically adapt computational resources to operational needs is a fundamental need of modern systems.

Implementing the Sonar Application on an iWarp Array

In conventional systems, each of the functional elements of Figure 1-3 is implemented as a separate hardware element. This configuration constrains the adaptability of the system for new operational scenarios, new algorithms, and processing requirements. In an iWarp array, the mapping of the functional elements is configured as illustrated in Figure 1-4.

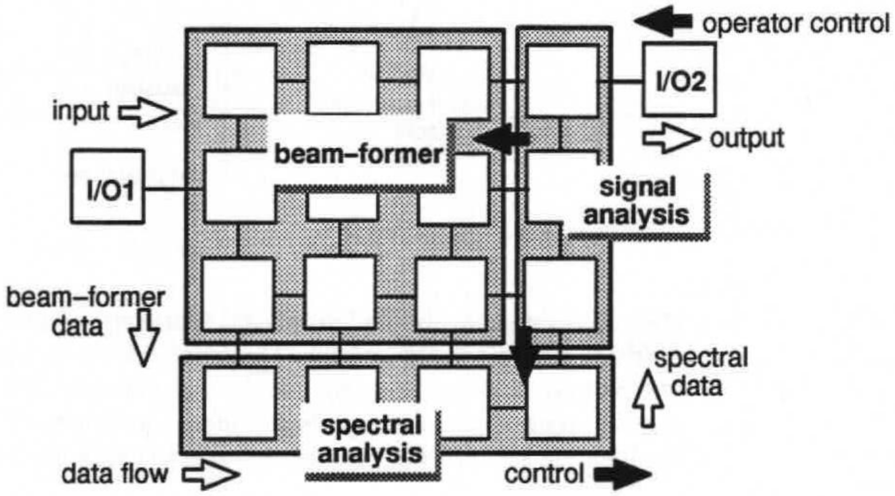


Figure 1-4: Sonar application on a 16-cell iWarp array

Synergy between Communication Models

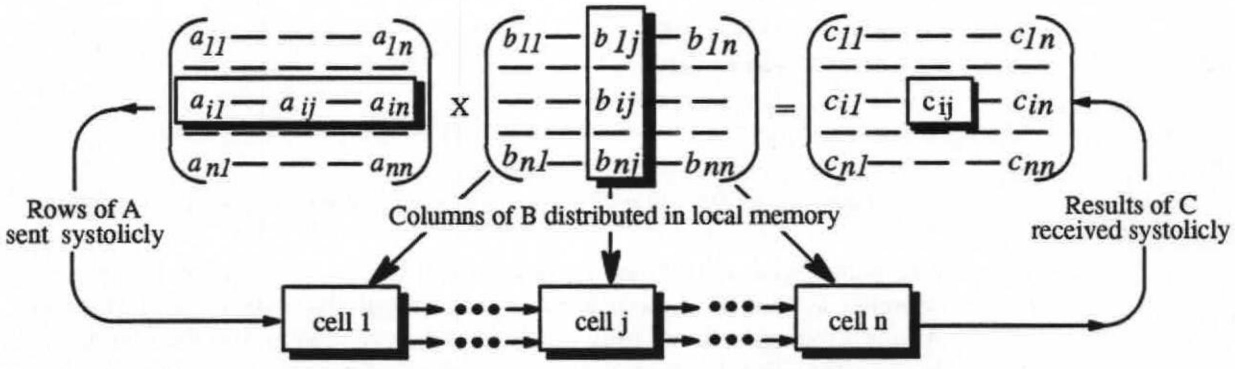
Processors in each of the functional groups use fine-grain systolic communication to achieve the greatest efficiency and performance for the computational task. Processors, in effect, work together as a single high-performance functional element. Data and control signals that pass between the functional elements use coarse-grain message-based communication. This loosely coupled, asynchronous form of communication allows each functional element of the array to work independently of other functional elements and still interact in a timely fashion required for real-time applications.

Fine-Grain Systolic Communication

To illustrate the benefits of fine-grain systolic communication, consider the matrix multiply example of Figure 1-5. Two matrices, *A* and *B*, are multiplied to form the matrix *C*. The familiar dot product method is used, which requires that we take the dot product of each row of *A* against each column of *B*. The entire calculation requires n^3 multiply-accumulate operations.

$$A \times B = C$$

$$\text{Where } C_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$$



a_{11} — — — a_{1n}
a_{i1} — a_{ij} — a_{in}
a_{n1} — — — a_{nn}
X
b_{11} — b_{1j} — b_{1n}
b_{ij}
b_{n1} — b_{nj} — b_{nn}
=
c_{11} — — — c_{1n}
c_{i1} — c_{ij} — c_{in}
c_{n1} — — — c_{nn}
Rows of A sent systolicly
Columns of B distributed in local memory
Results of C received systolicly
cell 1
cell j
cell n

Figure 1-5: Matrix multiply example

Consider a linear array of *n* iWarp cells into which the *n* columns of matrix *B* are distributed so that each cell has a column of *B*. Since each row of matrix *A* is applied against each column of *B*, we can pass each row of *A* down the array and perform the corresponding dot product operation as elements flow past the cells. Thus, each cell j performs the computation, $a_{ik} \times b_{kj} + c_{ij}$, as each a_{ik} value flows through the array. The first result, c_{11}, is complete and can flow out of the array when the dot product between a_{1n} and b_{n1} has completed in cell 1. Other results of matrix *C* are completed in turn as the rows of *A* proceed through the iWarp array.

Performance Analysis

As a general rule, parallel algorithms go through a three stage process of initialization, computation, and cleanup. The initialization process distributes data to processors and gets the computation going. The computation stage occurs when all processors are involved in the process and the peak performance of the system is achieved. The clean-up stage completes the computational process and collects the results at the final destination.

Systolic architectures have a performance advantage over other parallel architectures because the initialization and clean up stages occur as fast as data can flow through the network of processors. This process can be illustrated by viewing performance of the matrix multiply application as a function of time, as in Figure 1-6.

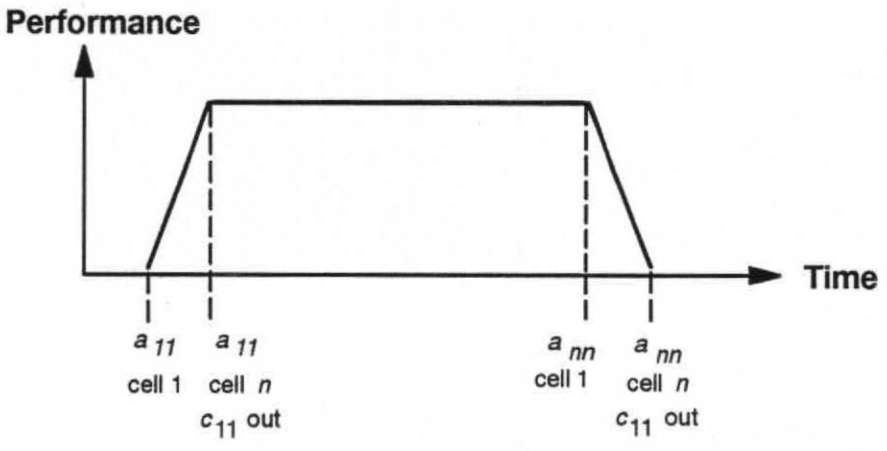


Figure 1-6: Time line of performance peak in matrix multiply example

The computation starts when a_{11} reaches cell 1. The initialization phase is complete as soon as a_{11} completes its path through the array and reaches cell n. At this point, all cells are fully involved in the calculation, and the first result, c_{11}, is complete. The iWarp array runs at full performance until a_{nn} reaches cell 1. Then the clean-up phase begins. As a_{nn} continues down the array, each cell in turn completes its computation, and the performance drops as these cells send out their results and become idle.

Advantages of the Systolic Model

The matrix multiply example illustrates another benefit of systolic computing overcoming memory bottlenecks of Von Neumann and Harvard architectures. iWarp addresses these limitations by augmenting memory bandwidth with comparable I/O bandwidth to sustain peak performance of arithmetic elements. The benefits are twofold. First, operands used in common by all cells are broadcast over the I/O path as was done for the rows of matrix A. Second, more effective use is made of memory because intermediate storage accesses are avoided. Using matrix A directly from the communication pathway avoids four memory accesses that would be required by conventional memory-to-memory message-passing techniques.

Fundamental iWarp Concepts

The following concepts are central to the iWarp system design:

- Integrated communication and computation elements for minimizing communication latency, maximizing communication bandwidth, and maximizing computational performance.
- Balanced ratio between communication and computation for supporting fine-grain systolic computation models.
- Independent communication and computation elements for supporting coarse-grain heterogeneous computation models.
- Scalar computational capability to minimize latency, improve the efficiency of compiled code, and broaden the base for supported applications without impacting vector performance.

Integrated Communication and Computation

The iWarp architecture supports tightly coupled integrated communication and computation elements that provide high-level hardware support for low-latency and high bandwidth sophisticated communications operations. Examples of these operations include word level, intercell, flow control synchronization, automatic spooling and streaming of message data, buffer management, and logical connections that interleave multiple message streams over the finite physical buses.

Balanced Communication to Computation

iWarp supports the one-to-one communication to computation ratio that is an essential element of fine-grain systolic computing concepts. During a single floating-point multiply-add cycle, two operands can be used from the communication pathways, and two operands can be sent to other cells. This capability broadens the range of algorithms and applications that can benefit from iWarp technology by supporting algorithms that require very few computations per data element. Additionally, the systolic computation model provides near linear speedup for scalable systems, to the limit of the parallelism in the application, by making more effective use of memory bandwidth and I/O capacity.

Independent Communication and Computation Elements

Independent communication and computation elements provide support for heterogeneous computation models. This ability allows individual processors or groups of processors to communicate asynchronously and supports transparent overlapping of computation with communication in hardware. Practical applications require a combination of communication capability: fine-grain for high performance and efficiency, and coarse-grain for interaction between loosely coupled processes and external events.

Scalar and Vector Performance

A design goal for iWarp was to reduce latency at all levels of the cell architecture. For the arithmetic elements of the processor, this goal resulted in exceptional scalar performance while maintaining excellent vector performance. The benefit was greater computational efficiency from high-level language compilers and broader applicability of iWarp in a wide range of algorithms and applications. Reduced latency also yields benefits in the handling of exception conditions and external events. In a tightly coupled parallel system, large delays in one processor caused by an exception condition can affect all other processors in the array.

iWarp Architecture Overview

Let us consider how these features are reflected in the iWarp communication architecture. An iWarp system is made up of an array of iWarp cells connected by communication pathways. Each iWarp cell consists of an iWarp component plus memory. As shown in Figure 1-7, the iWarp component contains independent communication and computation agents. Closely coupled yet independently controlled agents make it possible to efficiently overlap communication and computation, and provide greater efficiency for random communication. Nonadjacent cells in the array communicate without disturbing the computation on intermediate cells.

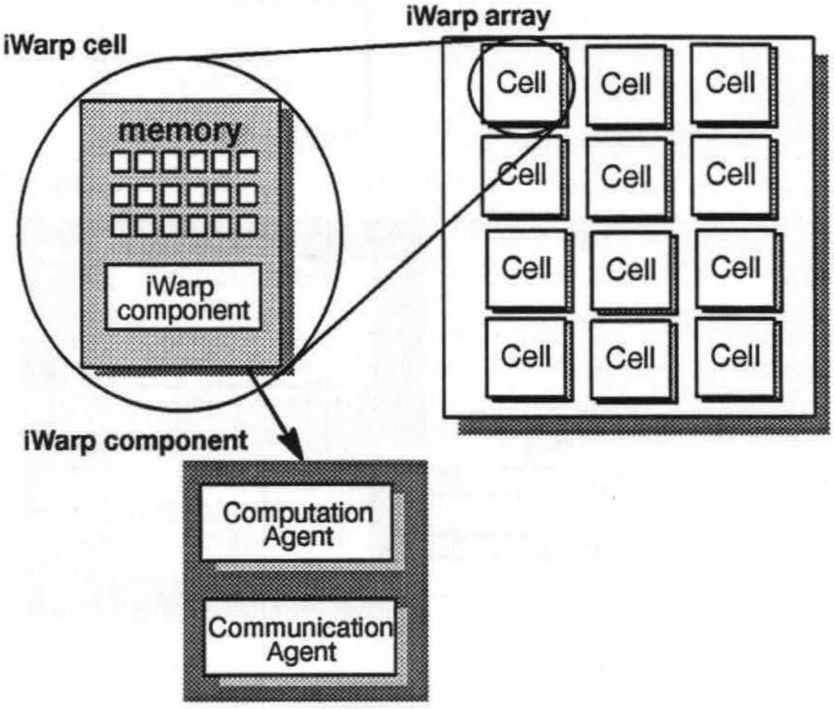


Figure 1-7: iWarp system architecture

The use of source cell routing and logical connection mechanisms provides the capability of reconfiguring the array for fault tolerance and graceful degradation on complex systems. Dead or suspect cell segments can be routed around, and tasks can be redistributed to meet the needs of demanding operational scenarios.

As shown in Figure 1-8, each iWarp cell supports four full duplex I/O channels. Each I/O channel is labelled with a unique name XLeft, YUp, XRight, or YDown. Each channel input or output bus has a sustained performance of 40 MBytes/s. This configuration gives a combined input data bandwidth of 160 MBytes/s plus an output bandwidth of 160 MBytes/s per iWarp cell. The Computation Agent can use half of this bandwidth, 80 MBytes input and 80 MBytes output, while sustaining a similar 160 MBytes/s data I/O capacity with local memory.

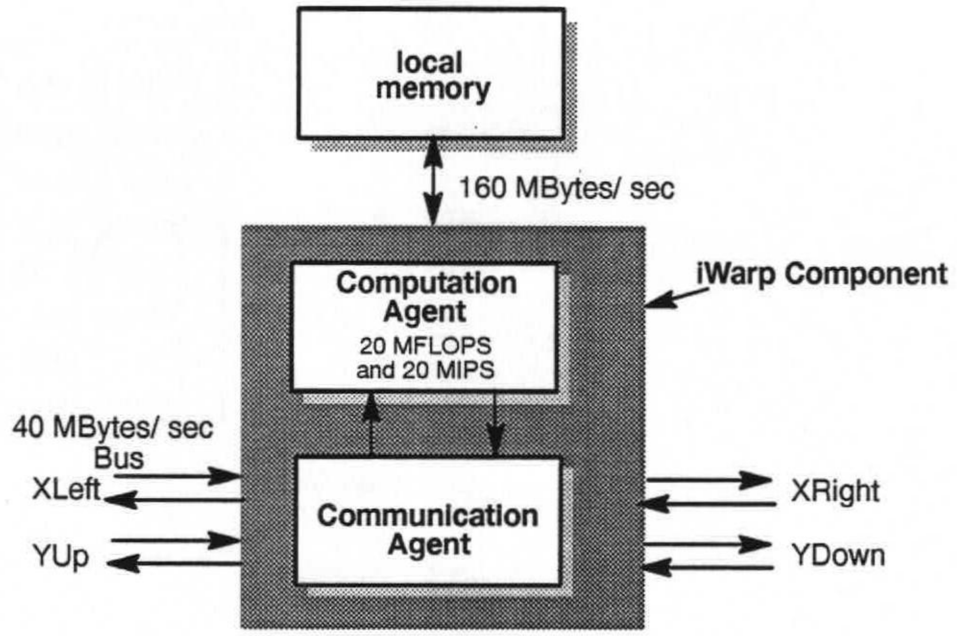


Figure 1-8: iWarp cell I/O capacity

Each cell has a peak performance of 20 MFLOPS for 32-bit single precision and 10 MFLOPS for 64-bit double precision. The iWarp component contains 600,000 transistors on a 540 mil square die and is packaged in a 280-pin, pin grid array (PGA) package. The component and its local memory (typically 18 static RAM components) take up the space of a 3"x5" index card (7.6 cm x 12.7 cm), approximately 15 square inches (96 square cm) of circuit board space.

iWarp Communication and Computation Models

Figure 1-9 shows the primary functional elements of the iWarp component architecture. The computation engine performs the computation and control tasks and is essentially serviced by the other functional elements. The communication pathway provides the interface to other cells in the array and handles message traffic between cells. The memory interface and spooler/streamer elements provide the interface between memory and both the computation engine and communication pathway. The XL channel of a cell's Communication Agent connects to the XR channel of a neighboring cell and vice versa. YU and YD connections are used in the same way.

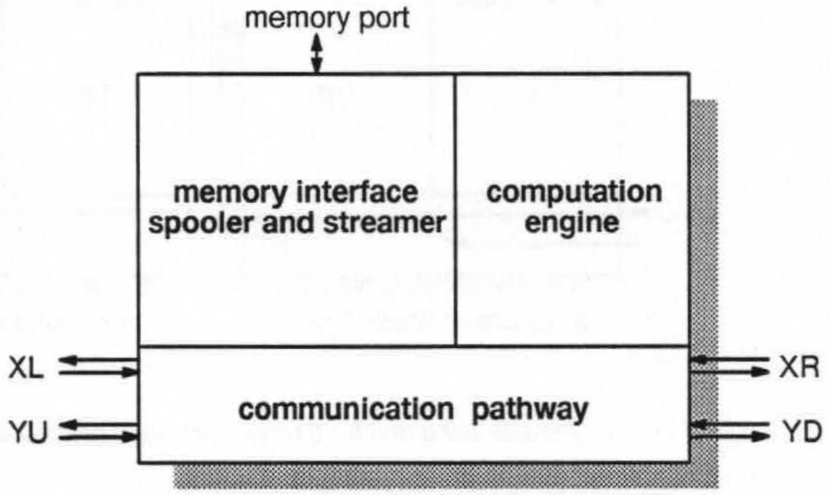
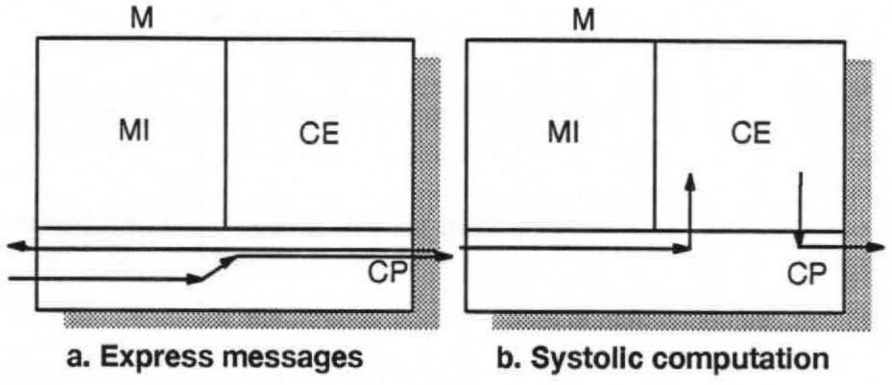


Figure 1-9: Data pathways

These functional units work together to perform a variety of sophisticated communication/computation functions that are classified as follows:

- **Express messages -** Messages that route directly through a cell.
- **Systolic computation -** Computation uses operands directly off the pathway.
- **Spooling -** DMA between memory and pathway.
- **Streaming** - Buffering systolic data through memory.
- **Memory-to-memory message passing -** Standard message-based communication.
- **Memory-based computation -** Conventional method of computation with operands in memory.

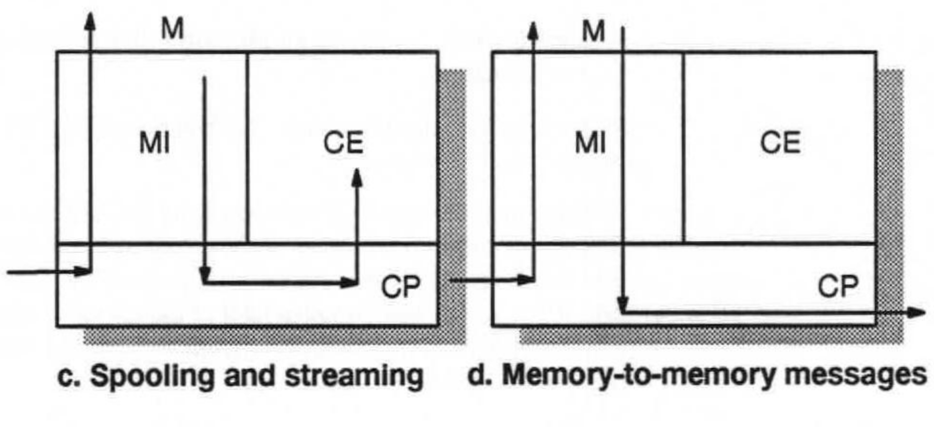
Figure 1-9a illustrates the movement of messages that are not intended for the current cell and pass through unhindered. The cell pathway hardware automatically expresses them through the communication pathway and on to an adjacent cell. Routing information provided at the message level supports corner-turning or transfer of the message from an X connection to a Y connection, in effect, a ninety-degree turn.



Figures 1-9 a and b : Express messages and systolic computation

Figure 1-9b illustrates the movement of data for systolic computation. Data is taken by the computation engine directly off the pathway, used by the computational task, and results sent back on the pathway to another processor in the array.

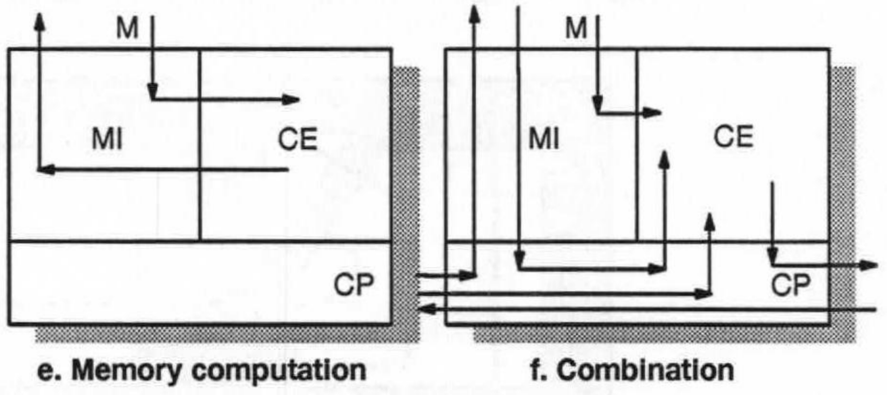
Figure 1-9c illustrates a variation on systolic computing that supports buffering messages in memory. If the computation engine is busy, and a systolic message is received, the spooler can perform a DMA transfer of the data directly into a preassigned buffer in memory, holding it until the processor is available to respond. The data is then streamed out of memory and presented to the processor by the streamer as if it had been received over the communication pathway. This feature ensures consistency in software, even though the data has been treated differently in hardware.



Figures 1-9 c and d: Spooling and streaming and memory-to-memory messages

Figure 1-9d indicates the movement of data for standard memory-to-memory message passing. In this case, the received message automatically goes into memory to be used by the processor at a later time, and messages sent from memory are spooled out in a like manner. The cell runtime system software supports spooling directly between process data spaces without using intermediate buffering.

Figure 1-9e shows standard memory-based computation. A combination of communication activity is shown in Figure 1-9f. Express operations, memory-to-memory message passing, spooling, memory access, systolic, and so on, are all shown. Note that iWarp component hardware supports all of these communication/computation models simultaneously.



Figures 1-9 e and f: Memory computation and combination

Logical Connections

The physical buses that connect iWarp cells are time multiplexed into logical buses that allow several connections to share the same physical pathway. This division improves the use of physical buses, avoids deadlock, and minimizes data starvation problems for unbalanced tasks. As shown in Figure 1-10, logical buses can be viewed as a 20 x 20 crossbar. Logical buses are statically allocated to physical buses under software control. In both Figures 1-10 and 1-11, logical buses have been evenly distributed among the four physical channels and the Computation Agent. Different logical-to-physical mappings can be supported according to the application requirements.

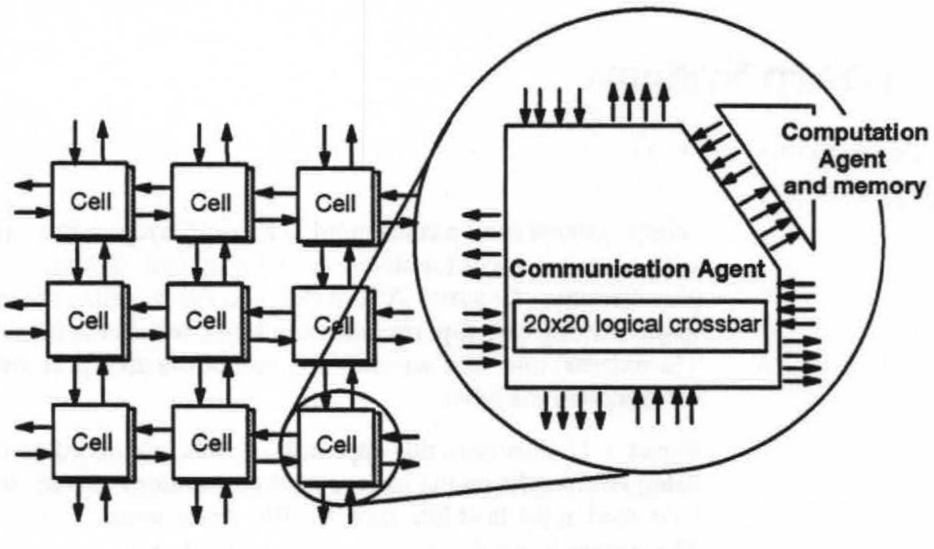


Figure 1-10: Physical and logical buses

Figure 1-11 illustrates the use of logical connections for an isolated segment of a two-dimensional iWarp array. Note the sharing of the physical bus between cells 3 and 4 using two independent logical connections. Logical connections share the physical communication path in a time multiplexing manner. Priority is given on a round-robin word-level basis, taking into account active logical buses only. Physical bus bandwidth is not consumed by idle logical buses.

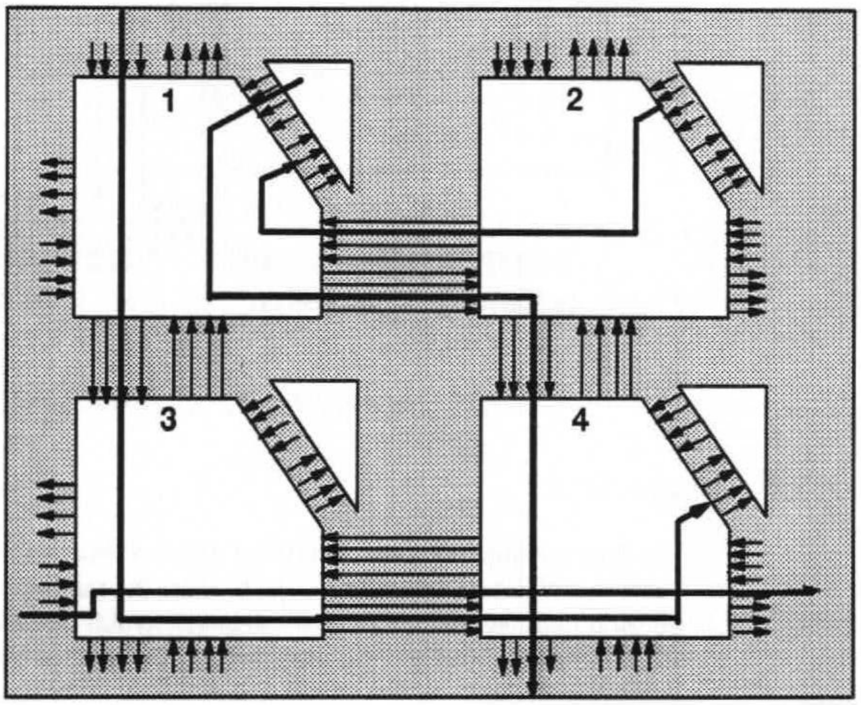


Figure 1-11: Logical buses support sharing of physical resources

iWarp Systems

iWarp systems can be configured in a linear array or in a two-dimensional array with a mesh or torus topology. Each iWarp cell consists of an iWarp component plus memory. External I/O interfaces to iWarp cells can be provided by implementing a dual-ported memory block within the memory space of a cell. The external interface accesses one port of the dual-port memory, and the iWarp cell accesses the other.

Figure 1-12 illustrates this approach. In this model, interfaces are connected using iWarp cells on the loop-around connections. iWarp interface cells are contained in the host interface, the file server interface, and the video interface. The system is configured using Quad Cell Boards for the main processing array. Single Cell Boards, with their large memory capacity, are used for staging data between external interfaces. Network access is provided to multiple users via the host local area network. Both Quad Cell Boards and Single Cell Boards are discussed in Chapter 2.

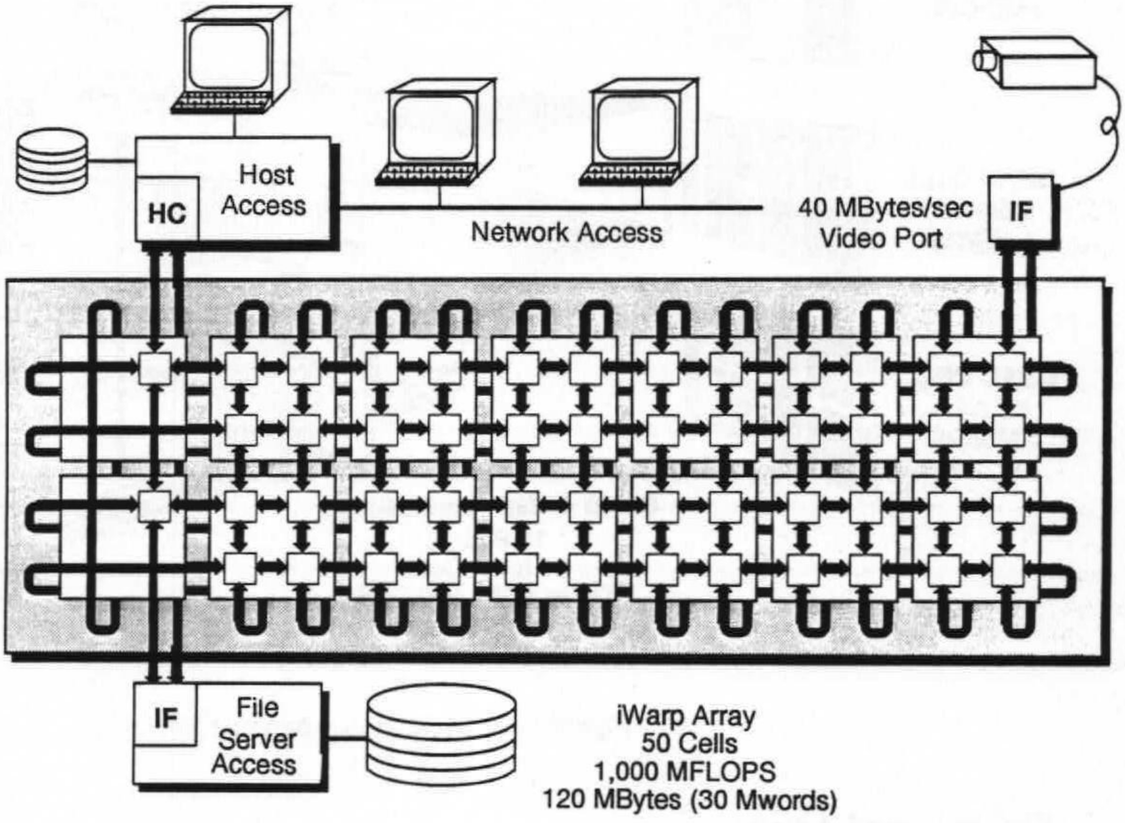


Figure 1-12: iWarp system configuration concept

Configurations

Configurable iWarp systems and system building blocks can be used for applications research, system development, and rapid prototyping. These systems provide a wide range of configuration options, using boards, Cardcage Assemblies, and System Cabinets, as shown in Figure 1-13. Quad Cell Boards support four iWarp cells with .5 to 1.5 MBytes of memory per cell, and Single Cell Boards support 6 MBytes of memory per cell . Boards can be used in custom systems or configured in the standard iWarp Cardcage Assembly.

Cardcage Assemblies are self-contained chassis, including a power supply, fans and a clock distribution board. Using the loop-around cables and interconnects, a wide range of system configurations are supported. System enclosures are 19-inch RETMA racks. iWarp configured systems comply with UL, FCC, VDE, CSA, IEC, and GS regulations for safety and emissions.

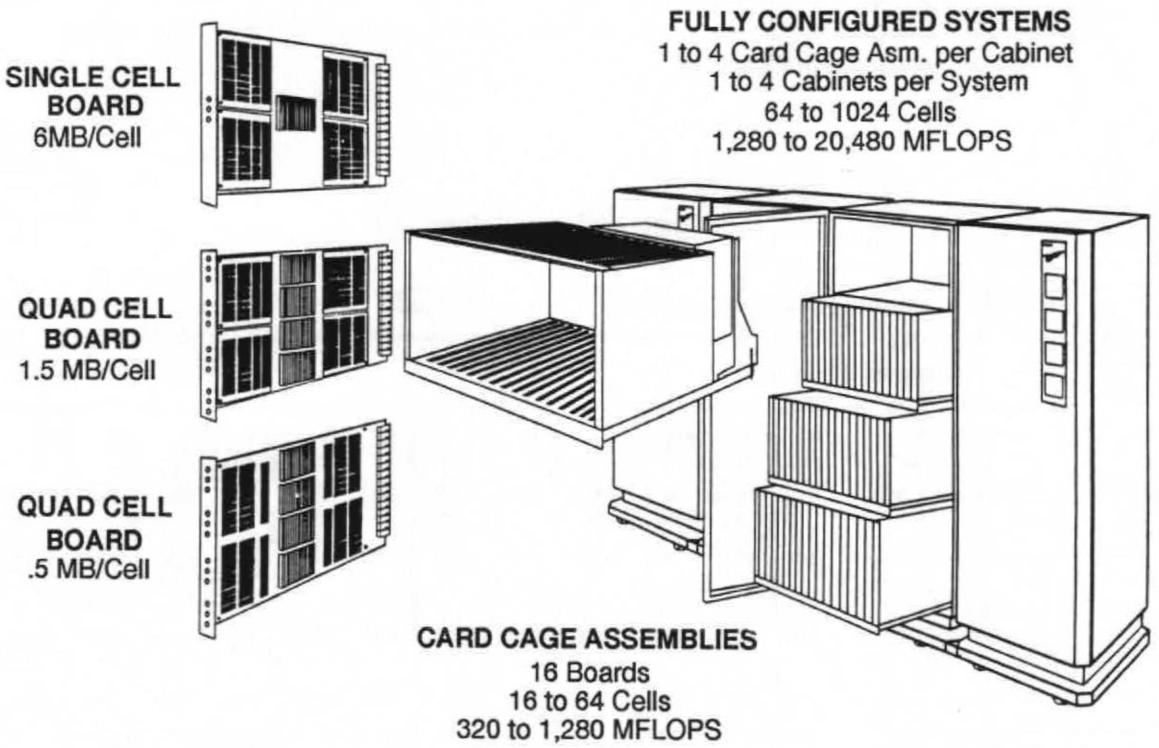


Figure 1-13: iWarp system building blocks

Single Board Array

A Single Board Array is provided for Sun workstations. This system provides an effective development environment for use throughout a development group or as an application accelerator for dedicated systems, as shown in Figure 1-14.

iWarp Single Cell and Quad Cell Boards are supported in Single Board Array systems. Each Single Board Array supports one or four iWarp cells configured as a linear or 2x2 array, respectively. Up to eight boards can be configured in a 32-cell array within a workstation.

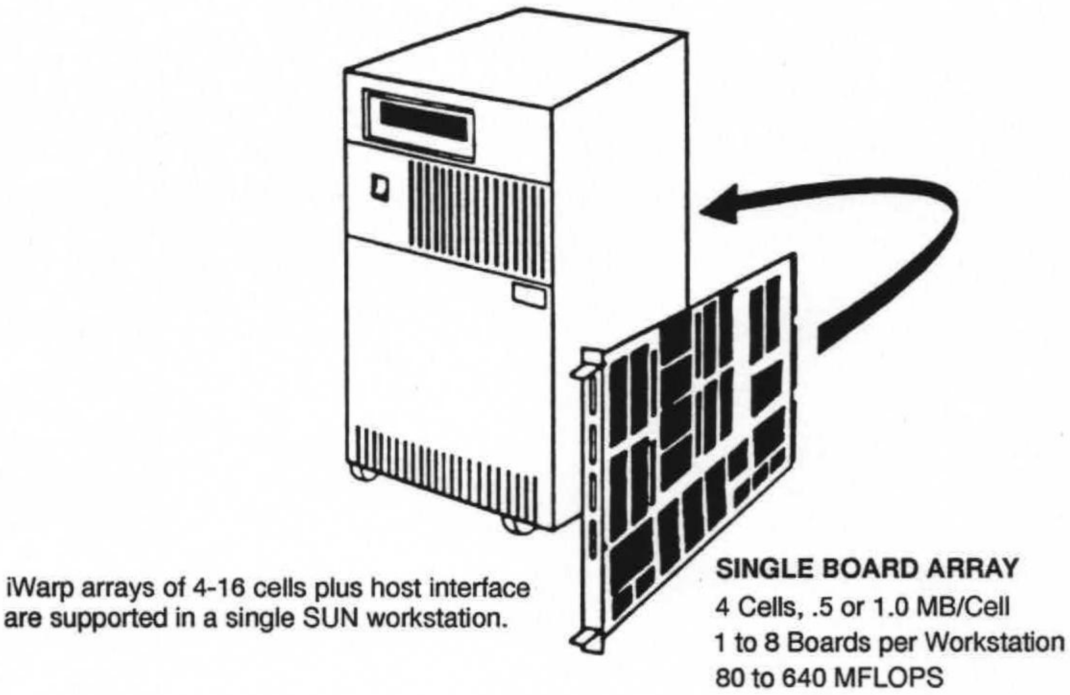

iWarp arrays of 4-16 cells plus host interface are supported in a single SUN workstation.
SINGLE BOARD ARRAY
4 Cells, .5 or 1.0 MB/Cell
1 to 8 Boards per Workstation
80 to 640 MFLOPS

Figure 1-14: iWarp Single Board Arrays and development systems

The Significance of iWarp

iWarp represents a major step forward in high-performance computer architectures and will have its greatest impact on the broad base of signal and image processing applications. For these applications, performance beyond the GFLOPS plus range has historically been accomplished only through special-purpose systems. iWarp provides for the first time a programmable system architecture that can be configured to meet both the computational needs and the I/O needs of these signal and image processing applications. iWarp's building block approach to construct very reprogrammable, high-performance, general-purpose systems, as well as special-purpose systems introduces a new era in parallel computing technology.

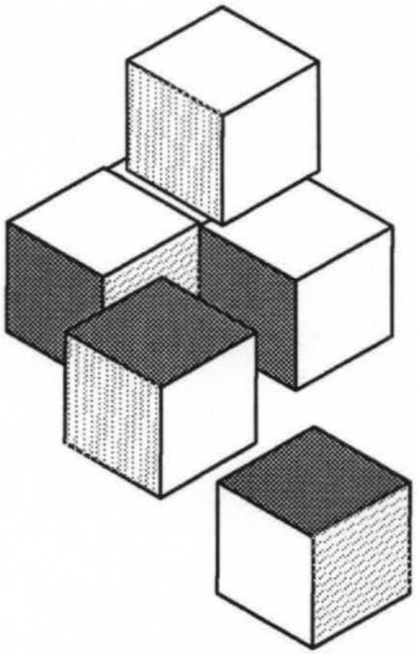
2 iWarp Hardware

System

Board

Component

Cardcage



The iWarp Component

The iWarp component is the basic building block of the iWarp system. Each iWarp component is a complete computer that includes I/O interfaces to connect many devices in a large array of processors. The iWarp component, combined with local memory, forms the iWarp cell, as shown in Figure 2-1.

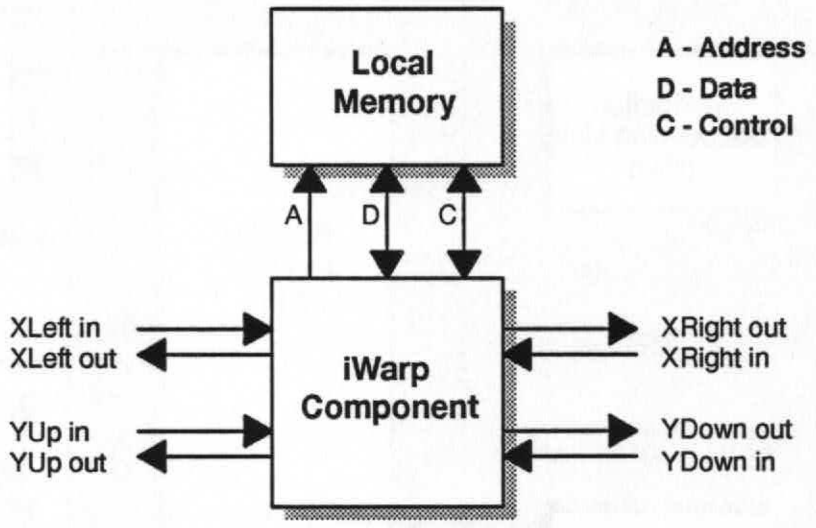


Figure 2-1: iWarp cell

The initial iWarp component package is a pin-grid array.

Component Architecture

The iWarp component architecture is divided into a Computation Agent and a Communication Agent. The Computation Agent and the Communication Agent function independently, so the processor does not have to participate in the communication process. This allows the communication activities of a cell to proceed without disturbing computation. Figure 2-2 shows a block diagram of the iWarp component architecture.

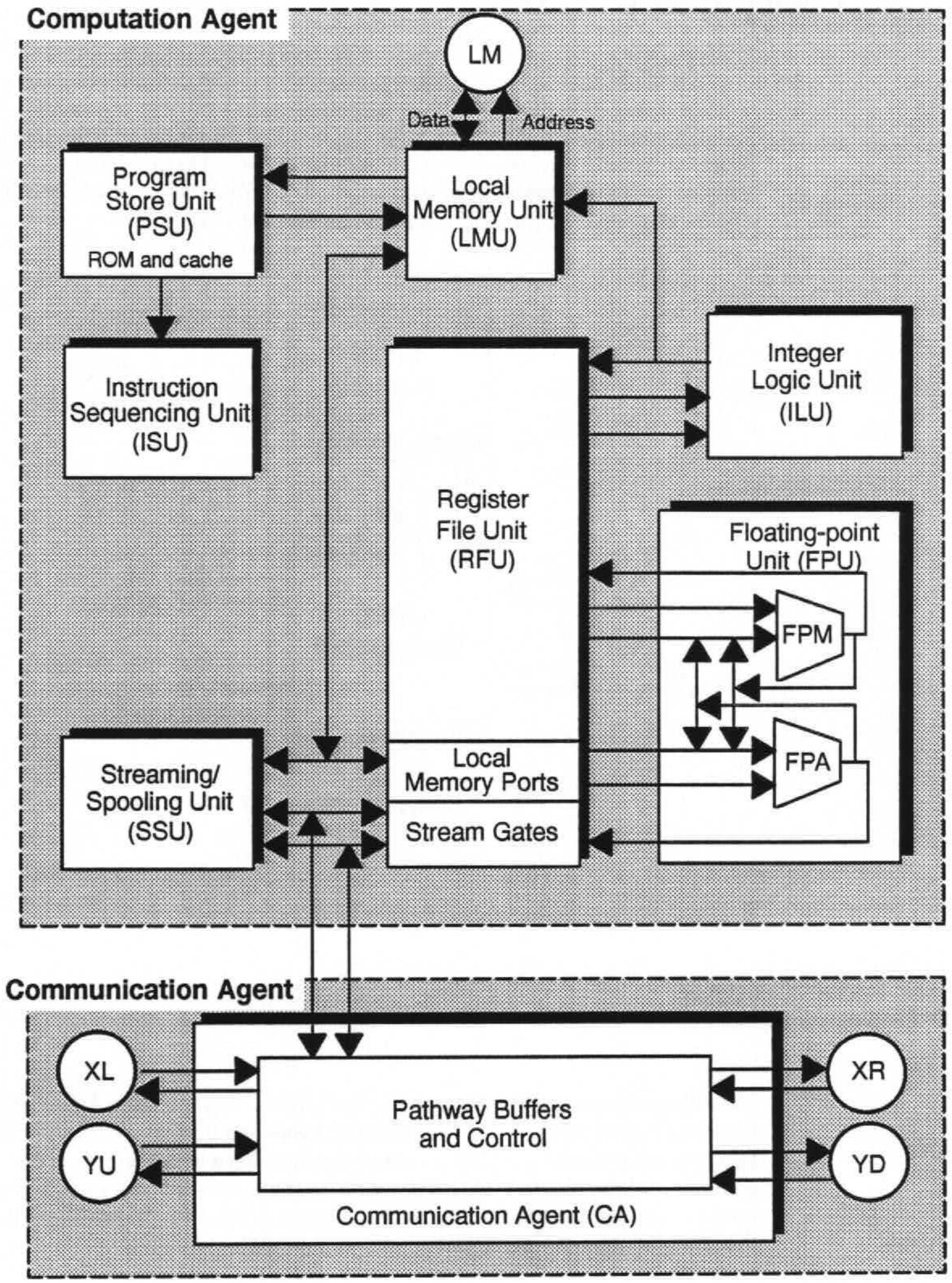


Figure 2-2: iWarp component architecture block diagram

Computation Agent

The Computation Agent executes an individual cell's portion of an algorithm that has been distributed over an iWarp array of cells. The Computation Agent consists of the following functional units:

- Register File Unit
- Local Memory Unit
- Instruction Sequencing Unit
- Program Store Unit
- Integer Logic Unit
- Floating-point Unit
- Streaming/Spooling Unit

Register File Unit

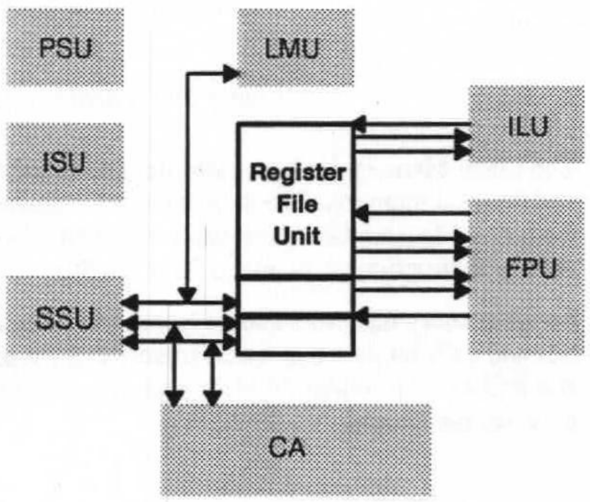


Figure 2-3: Register File Unit

The 15-port Register File is the central element of the iWarp component architecture, routing data between functional units. The Register File is a general-purpose, multi-ported shared RAM containing 128 32-bit locations. Register File access is by bytes, half-words, words, or double words, depending on the instruction used.

The Register File Unit supports nine read and six write operations in a single 50ns clock cycle. The Register File has nine standard ports, three each to the Integer Logic Unit, the Floating-point Adder, and the Floating-point Multiplier. The Register File also has two local memory ports and four special-purpose ports, or gates, that allow data from memory or the pathway to be placed in predetermined locations in the Register File. This process of using a programmable gate to access the communication pathways is called streaming and is discussed further in the description of the Streaming/Spooling Unit.

Local Memory Unit

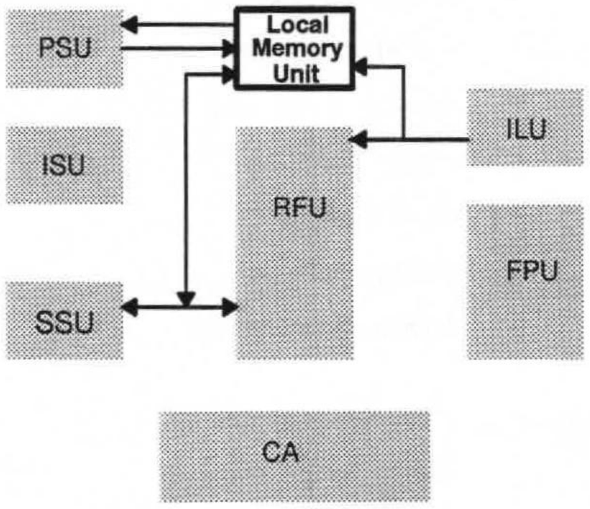


Figure 2-4: Local Memory Unit

The Local Memory Unit provides the interface between the iWarp component and its local memory. The local memory contains both data and instructions, so the Local Memory Unit provides a direct interface to both the Register File Unit and the instruction cache in the Program Store Unit.

Local memory has two separate buses to maximize performance: a 24-bit address bus and a 64-bit data bus. Local memory can support up to 64 Mbytes in RAM and ROM and provides 20 MHz performance with up to 20 million memory accesses per second.

Program Store Unit

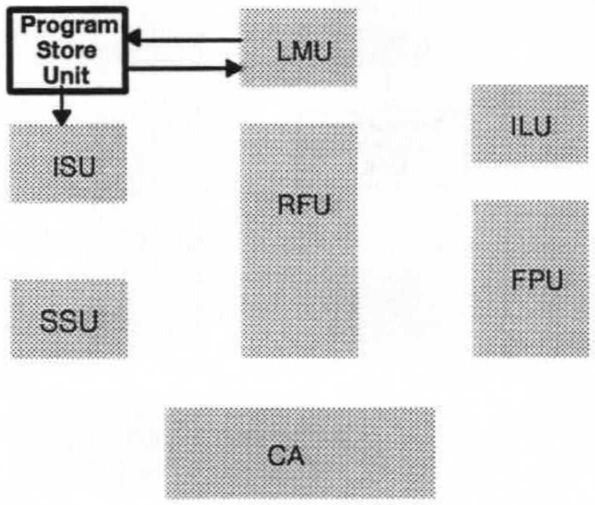


Figure 2-5: Program Store Unit

The Program Store Unit fetches instructions from Local Memory, stores them in an instruction cache, and provides them to the Instruction Sequencing Unit. The Program Store Unit contains a 1 Kbyte instruction cache and an 8 Kbyte instruction ROM.

The instruction cache is divided into four sectors, with each sector containing four 64-byte blocks. The mapping of sectors is fully associative, but blocks within a sector contain contiguous addresses. Management of the instruction cache is transparent once it is initialized.

The instruction ROM is divided into eight 1 Kbyte sections. The ROM contains initialization and start-up programs, as well as system routines.

Instruction Sequencing Unit

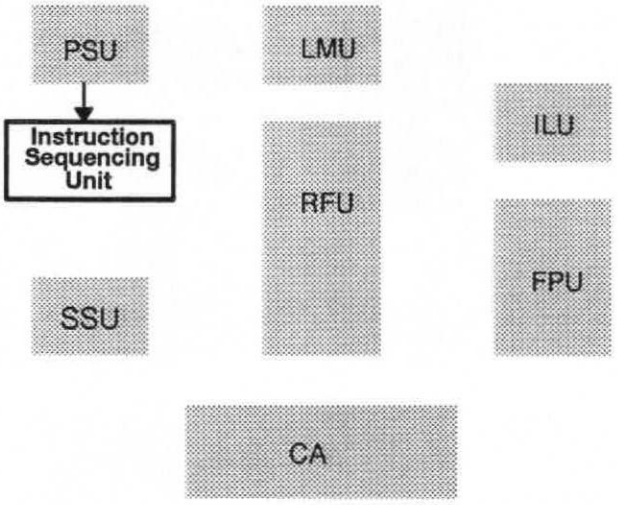


Figure 2-6: Instruction Sequencing Unit

The Instruction Sequencing Unit controls the flow of program execution by decoding all instructions, producing and distributing control signals to other functional units, evaluating their responses, and scheduling execution. The Instruction Sequencing Unit receives instructions from the Program Store Unit.

Integer Logic Unit

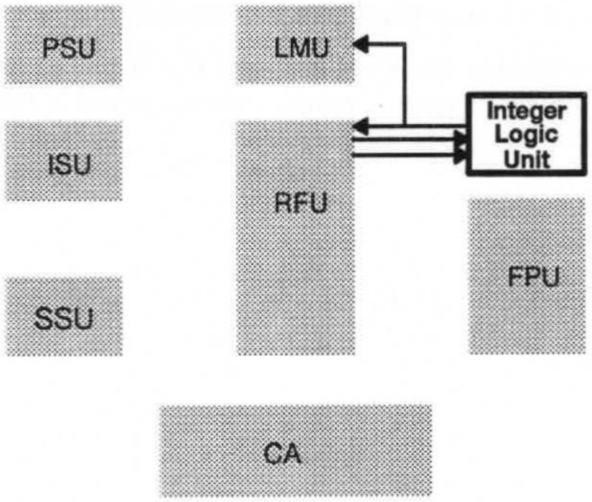


Figure 2-7: Integer Logic Unit

The Integer Logic Unit is a full 32-bit processor, providing integer arithmetic and logical operations on 8, 16, and 32-bit data, and generating addresses for data access to local memory. The Integer Logic Unit runs at 20 million instructions per second (MIPS). In a single 50-nanosecond instruction cycle, the Integer Logic Unit accesses two operands, performs a computation, and writes the result back into the Register File Unit. The long instruction word architecture of the iWarp processor allows the Integer Logic Unit to operate in parallel with the Floating-point Unit, generating a peak computing rate of 20 MFLOPS and 20 MIPS for the iWarp cell.

Floating-point Unit

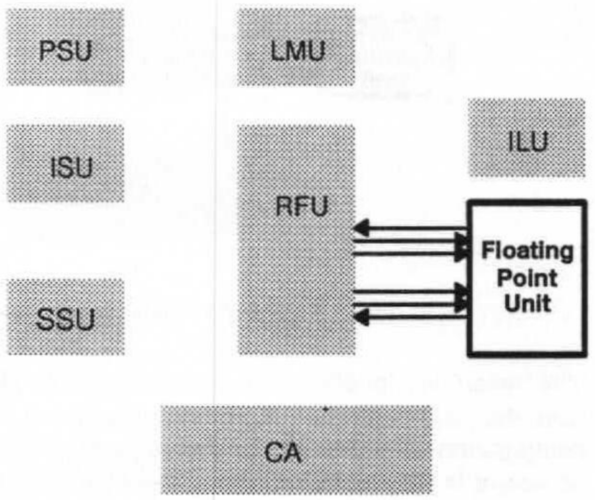


Figure 2-8: Floating-point Unit

The Floating-point Unit contains a non–pipelined Floating-point Adder and Floating-point Multiplier. The Floating-point Adder and Multiplier each provide a peak performance of 10 MFLOPS on 32-bit operations and 5 MFLOPS on 64-bit operations. The Adder and Multiplier run on a two-clock instruction cycle, each producing a result every 100 nanoseconds. In this time, the Adder and Multiplier can each access two operands from the Register File Unit, perform a computation, and write the result back to the Register File Unit. Double precision operations require a four-clock (200 ns) instruction cycle. The Floating-point Unit also contains bypass paths, so the result of a computation can be used as an operand in the next instruction, eliminating the need for an intermediate store and read of the result.

Streaming/Spooling Unit

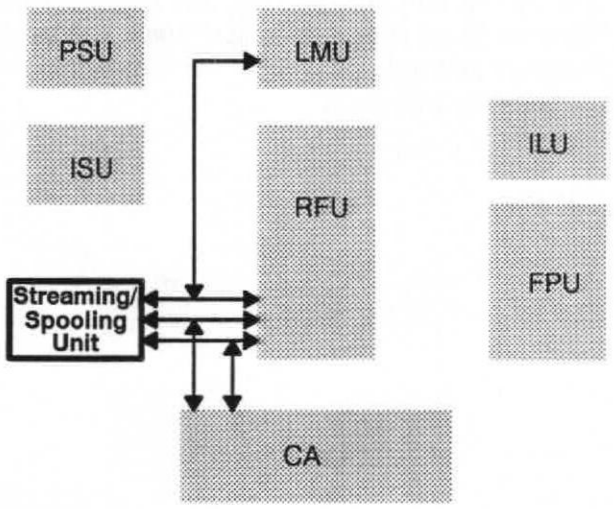


Figure 2-9: Streaming/Spooling Unit

The Streaming/Spooling Unit is a sophisticated DMA controller that spools data from the Communication Agent to memory and streams data from memory to the computation units through the stream gates of the Register File Unit. If the processor is busy and it receives a message, the Streaming/Spooling Unit can direct the data into a preassigned buffer in memory, holding it until the processor is available. When the processor becomes available, the data streams out of memory to the processor.

By removing blocked messages from the pathways and allowing unblocked messages behind them to proceed, spooling helps to relieve pathway congestion and improve overall performance.

Communication Agent

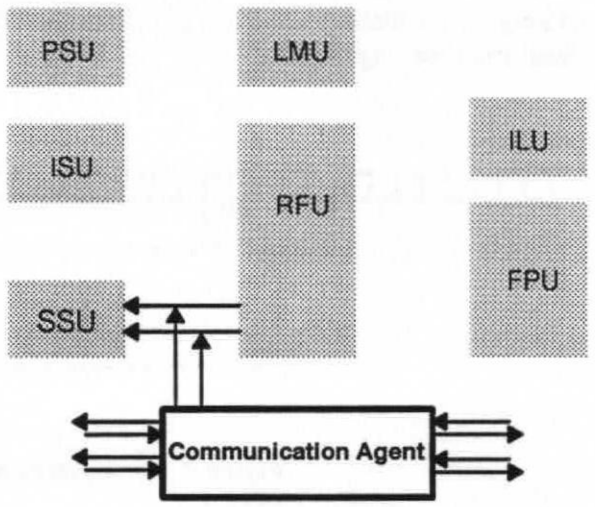


Figure 2-10: Communication Agent

The Communication Agent provides communication between cells in an iWarp array by controlling the transfer of data over the physical links between cells. Each cell in an iWarp array connects to its neighbors through four physical links, as shown in Figure 2-11.

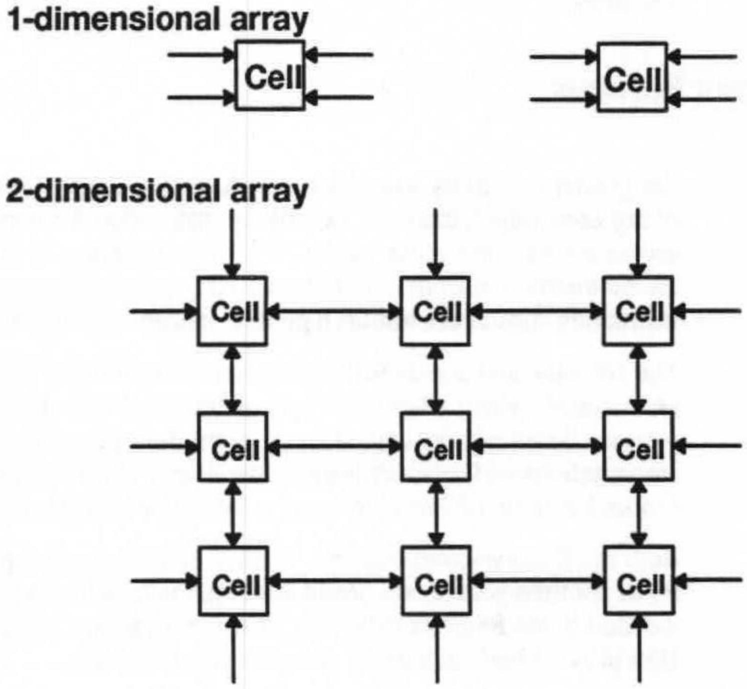


Figure 2-11: iWarp cell links

Each of the physical links has two unidirectional buses, one to transmit data and the other to receive data. Data is transmitted between cells in units called messages. Each message contains a header that has address and control information for that message. Bits 0-19 of the header contain the destination address for the message, and bits 20-23 contain control information. Figure 2-12 illustrates a message header.

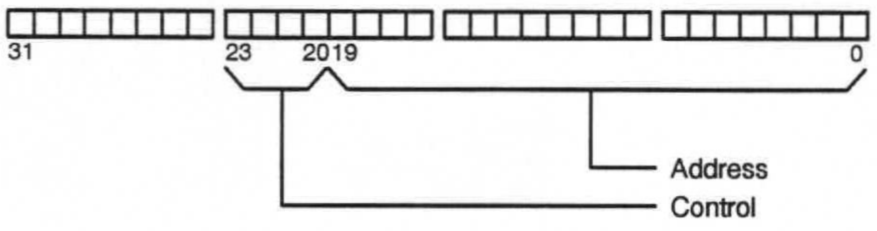


Figure 2-12: Message header

If an incoming message is addressed for the local cell, the Communication Agent delivers the message to the proper destination. If the message is addressed for another cell, the Communication Agent routes the message through to the next cell.

In a two-dimensional array, it may be necessary for a message to change from an X to a Y, or from a Y to an X direction. If a change in direction is required, the appropriate control bit in the message header is set, and the Communication Agent performs the corner-turning operation that allows a message to change direction.

Instruction Formats

For greater flexibility and efficiency in controlling the functional units of the iWarp component, there are two iWarp instruction formats: a 96-bit compute and access instruction format, and a 32-bit general-purpose instruction format. The 96–bit instruction format includes all frequently used operations, while the 32–bit instruction format constitutes a general–purpose RISC instruction set.

The compute and access instruction has the benefits of a long instruction word architecture, which allows multiple operations in parallel. This instruction can perform floating-point add, floating-point multiply, two memory address computations and memory access operations, plus loop decrement and branch evaluation in two 50 ns clock cycles for single precision data.

Both the floating-point add and floating-point multiply operations support 7-bit fields for their source and destination operands, allowing random access to any location in the Register File Unit. Reserved register locations in the Register File Unit provide high efficiency access to the pathways.

Word three of the 96-bit instruction can specify either a 32–bit general–purpose instruction or two memory operations. The 32–bit instruction option is useful when a general-purpose operation is needed during a compute and access cycle. Figure 2-13 illustrates the compute and access instruction format.

Word 1

	-(2)-	-(4)-	-(4)-	-(7)-	-(7)-	-(7)-
J	1 1	Data Mode	FADD	B operand reg	A operand reg	K operand reg

Word 2

-(9)-	-(2)-	-(7)-	-(7)-	-(7)-
Memory Control	FMUL	M operand reg	N operand reg	R operand reg

Word 3 (Option 1)

-(16)-	-(16)-
Operand for 1st Read Access	Operand for 2nd Read/Write Access

Word 3 (Option 2)

-(32)-
General–purpose Instruction

Figure 2-13: Compute and access instruction format

The general-purpose instruction format supports general control functions such as timer operations, pathway control, and event handling flow control.

The floating-point adder supports operations in hardware, including add, subtract, compare, maximum, minimum, and binary log. Similarly, the floating-point multiplier supports multiply, divide, square root, and remainder in hardware. The general-purpose instruction format also supports data conversion operations and a full range of logical operations. This format also supports byte, half-word (16-bit), full-word, and double-word memory operations, as well as automatic read-modify-write in hardware for byte and half-word operations. Table 2-1 summarizes the general-purpose instruction format functions.

Table 2-1: general-purpose Instruction Summary

FUNCTION	OPERATIONS	
General Control	timer operation pointer control pathway control	event control spool request execute instruction

Flow Control	call
	push
	pop
	branch
		return
		break
		enter loop
		stack control

Extended Flow Control	absolute call/branch indirect call/branch	
Floating-point Operations	add subtract compare maximum minimum	binary log multiply divide square root remainder
Data Conversion Operations	integer to floating-point floating-point to integer	
Integer/Logical Operations	logical arithmetic bit	rotate find MSB
Memory Access Operations	byte half-word	full-word double-word

The iWarp Boards

There are two types of iWarp cell boards, the Quad Cell Board and the Single Cell Board. As their names imply, the Quad Cell Board contains four iWarp cells, and the Single Cell Board contains one iWarp cell. Each cell on an iWarp board consists of an iWarp component and its associated local memory. Both the Quad and Single Cell Boards are approximately 9 by 11 inches, and they operate at a minimum clock rate of 40 MHz, or 20 MFLOPS per cell.

Each cell on the Quad and Single Cell Boards controls three LEDs mounted on its front panel. Two of the LEDs indicate cell status, and the third indicates an error condition. The boards also contain clock circuitry that synchronizes cells to within 5ns of each other in a Cardcage Assembly and within 25ns across multiple Cardcage Assemblies.

In addition to the Quad and Single Cell Boards, iWarp offers a Single Board Array that plugs directly into a Sun 3 or Sun 4 system. The Single Board Array is a Quad or Single Cell Board combined with a Sun interface board to form a complete iWarp array. The Single Board Array is approximately 14 by 14 inches for direct fit into the Sun workstation.

Quad Cell Board

The Quad Cell Board contains four iWarp components and four banks of local memory. Figure 2-14 shows the physical layout of the Quad Cell Board.

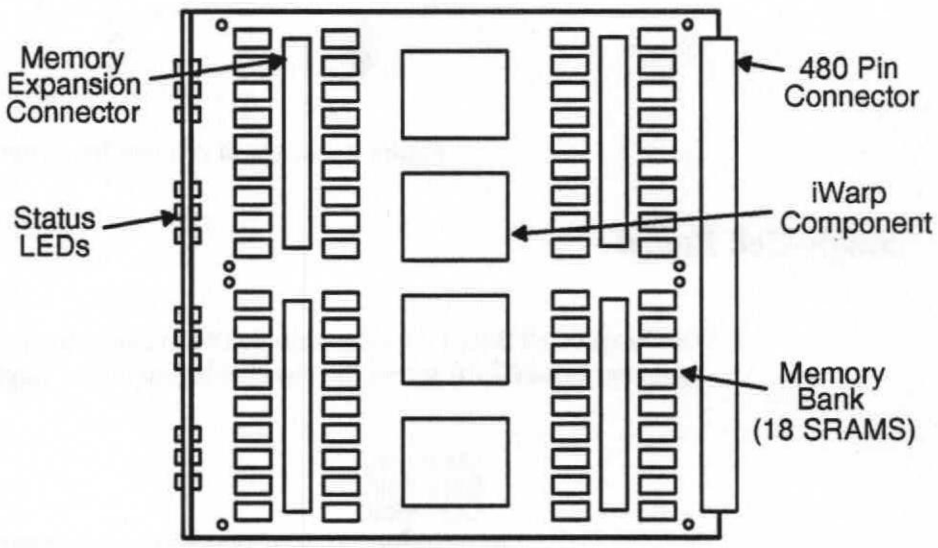


Figure 2-14: iWarp Quad Cell Board

The systolic pathways that extend off the Quad Cell Board are connected directly from the appropriate cell to a board edge connector. Each pathway coming off the board is capable of sustaining transfer rates of 80 Mbytes/s (40 MBytes/s in each direction). Figure 2-15 shows the pathway configuration for the Quad Cell Board.

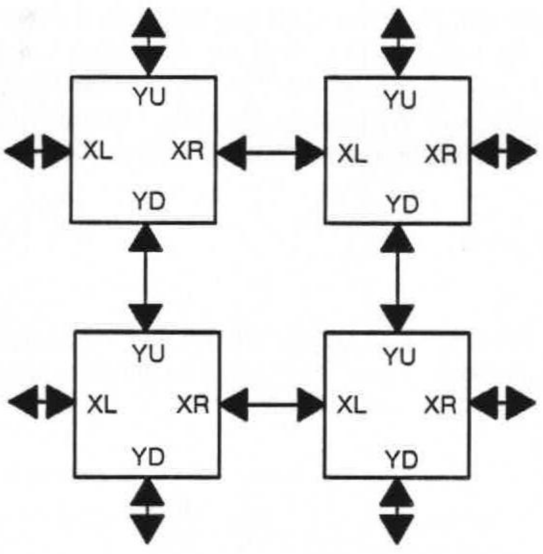


Figure 2-15: Quad cell configuration

Single Cell Board

The Single Cell Board contains a single iWarp component and four banks of local memory. Figure 2-16 shows the physical layout of the Single Cell Board.

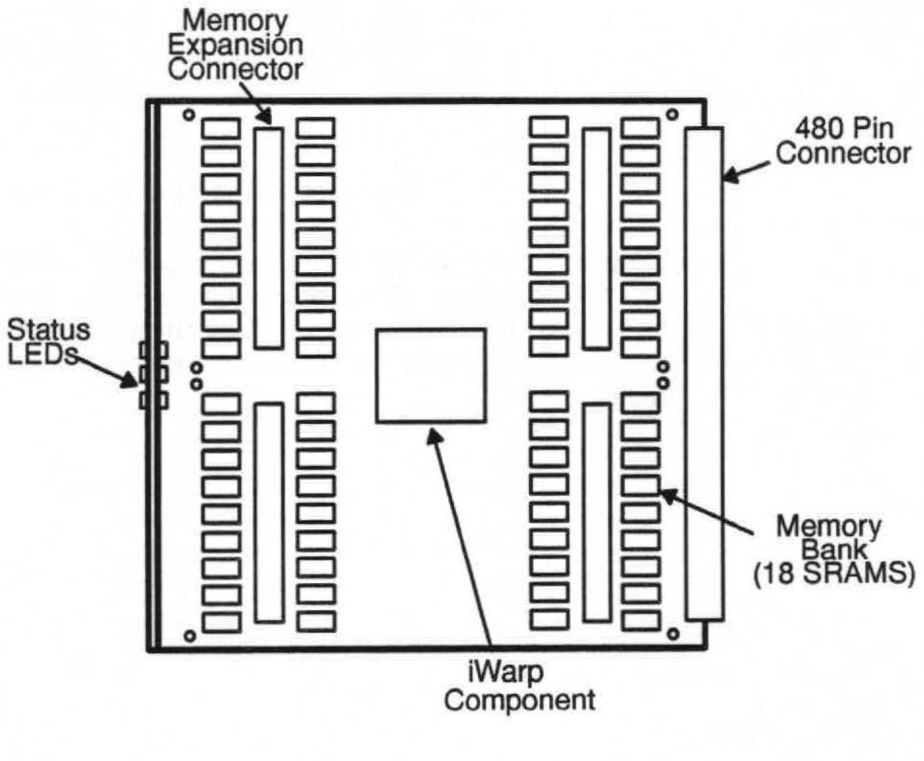


Figure 2-16: iWarp Single Cell Board

As with the Quad Cell Board, the pathways that extend off the Single Cell Board are connected directly from the appropriate cell to a board edge connector. Each pathway coming off the board can sustain transfer rates of 80 MBytes/s (40 MBytes/s in each direction). Figure 2-17 shows the pathway configuration for the Single Cell Board.

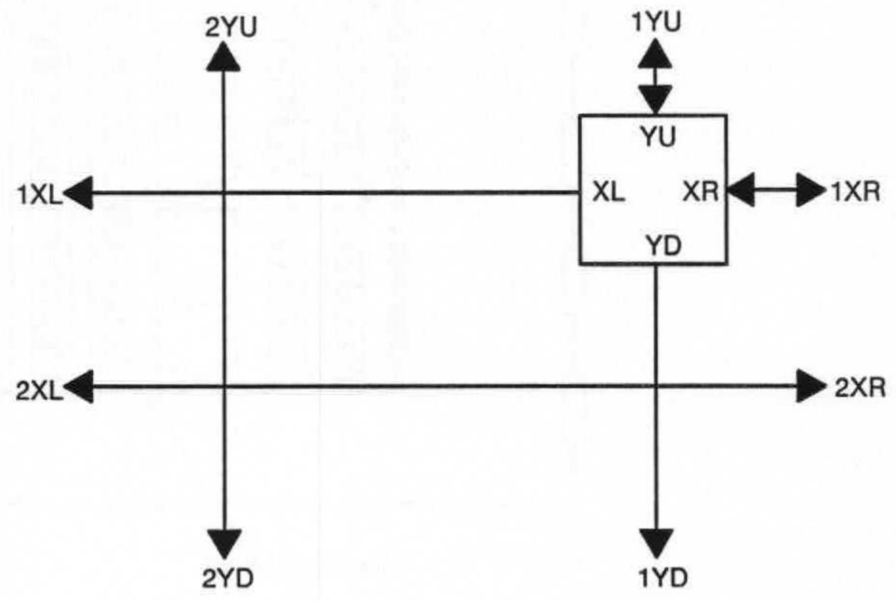


Figure 2-17: Single cell configuration

The pathways that are not connected to another cell are bypass connections that directly connect 2XL to 2XR and 2YU to 2YD on the board.

Single Board Array

The Single Board Array is a Single or Quad Cell Board combined with a Sun interface board to form a complete iWarp array. The Single Board Array plugs directly into a Sun 3 or Sun 4 system, providing a dedicated program development environment or a network performance accelerator. Up to eight Single Board Arrays can be connected in a single workstation to form a 32-cell, 2 by 16 array. Figure 2-18 shows the physical layout of a Single Board Array with a Quad Cell Board.

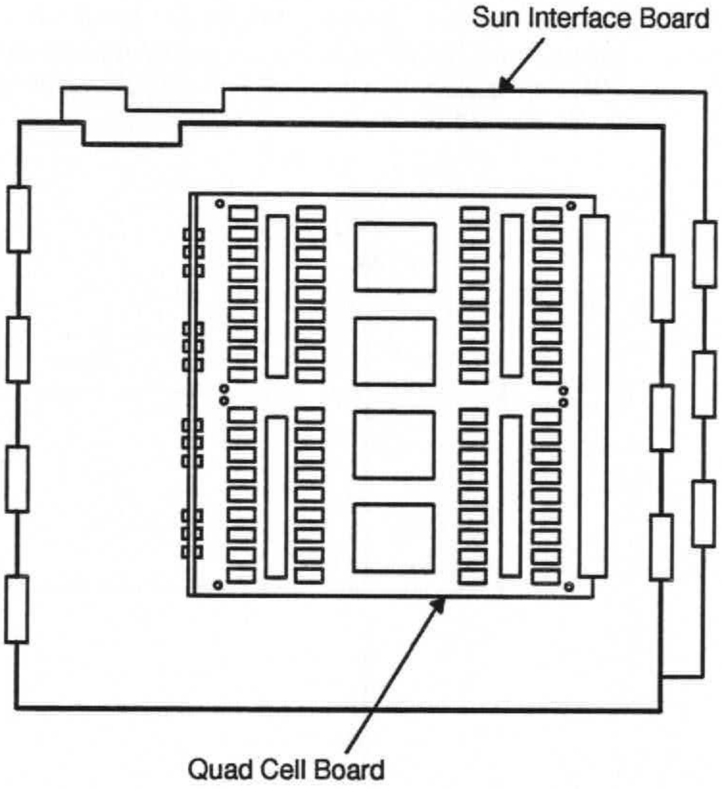


Figure 2-18: Single Board Array with Quad Cell Board

Local Memory

Each iWarp board contains four banks of local memory. On the Single Cell Board, the single component has access to all four banks. On the Quad Cell Board, each cell has access to one bank. Each bank of local memory consists of 18 SRAMs.

Local memory can be expanded with the Memory Expansion Module. Memory Expansion Modules connect to a Single or Quad Cell Board through a connector residing on the board. Each Memory Expansion Module contains up to two banks of additional local memory, and up to four Memory Expansion Modules can be connected per board, allowing three times the local memory. In addition, each bank of local memory can be configured using one of two SRAM densities, allowing even greater local memory flexibility. Table 2-2 lists the available memory for each iWarp cell. The maximum amounts represent full use of Memory Expansion Modules.

Table 2-2: iWarp Cell Memory

Available Memory per iWarp Cell (bytes)				
	Quad Cell Board		Single Cell Board	
SRAM density	minimum	maximum	minimum	maximum
256K	512K	1.5M	2M	6M
1M	2M	6M	8M	24M

The 1M SRAM density is not available until 1991.

The iWarp Cardcage Assembly

The iWarp Cardcage Assembly is a standard 19-inch rack-mountable open-frame chassis that combines the following into a single assembly:

- 17-slot cardcage
- backplane
- external device interface
- power supply
- fans

Sixteen of the slots in the Cardcage assembly are available for Single or Quad Cell Boards. A single Cardcage Assembly can hold up to 64 iWarp cells using Quad Cell Boards or up to 16 iWarp cells using Single Cell Boards. Single and Quad Cell Boards can also be mixed within a Cardcage Assembly to allow even greater flexibility.

The remaining slot in the Cardcage Assembly is reserved for the Clock/Sync Board. This board provides the necessary circuitry for synchronizing all cells within an iWarp array, even if the array extends to multiple Cardcage Assemblies. Figure 2-19 shows the iWarp Cardcage Assembly.

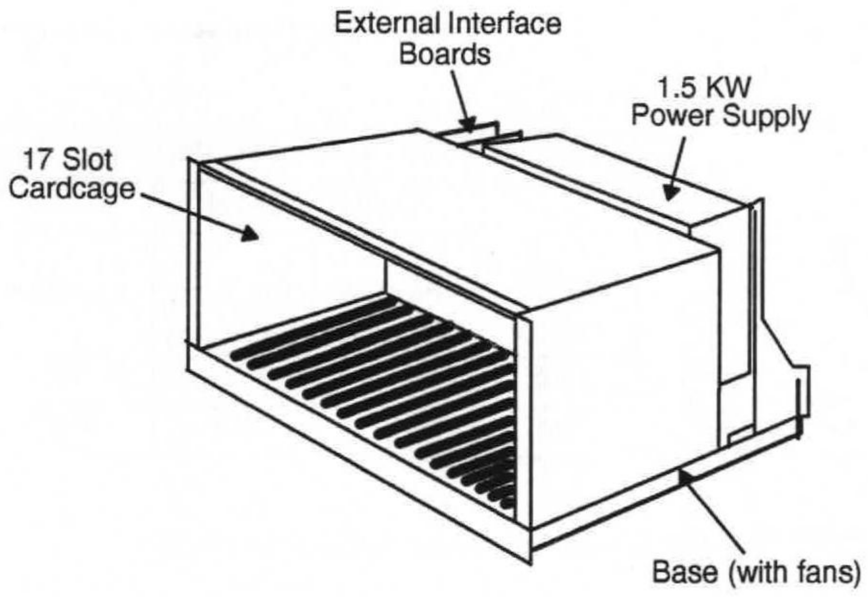


Figure 2-19: iWarp Cardcage Assembly

Cardcage Assemblies can be connected to form larger iWarp arrays by using iWarp External Interface Boards. These boards plug directly into the appropriate cell pathways on the backplane. External Interface Boards also allow connection of other external devices to an iWarp array. Figure 2-20 shows an example of a 4 by 8 iWarp torus array with a single external connection.

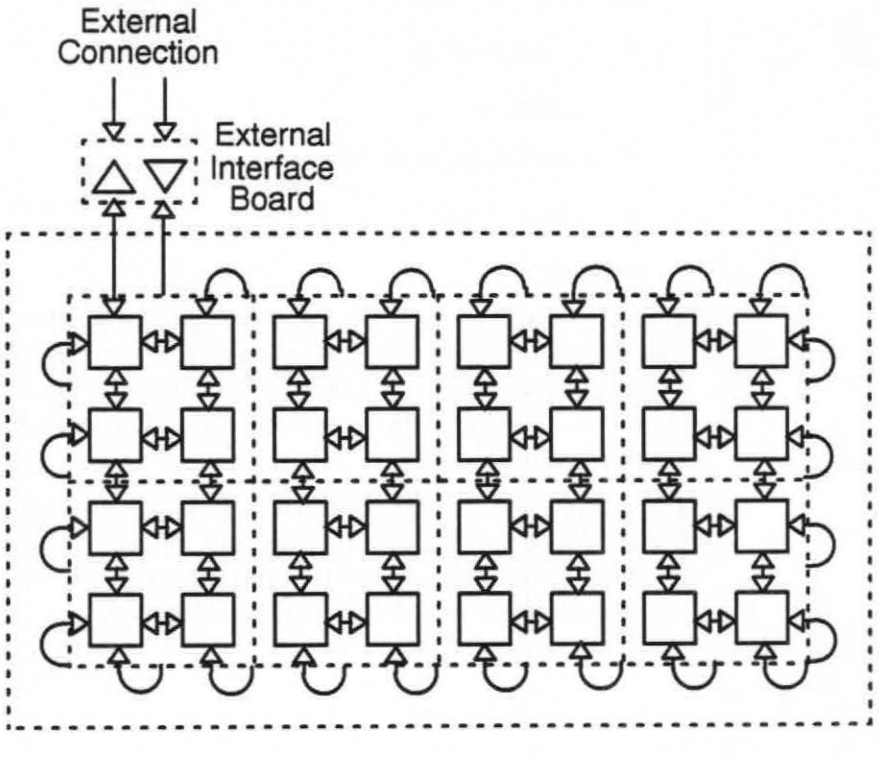


Figure 2-20: 4 by 8 array with one external connection

The external connection to the iWarp array can be made at any of the loops in either the X or the Y direction.

The iWarp System

Flexibility is the key characteristic of the iWarp system. From one to four iWarp Cardcage Assemblies reside in a single System Cabinet, and up to four cabinets can be connected to form even larger arrays. With a system of four cabinets, an iWarp system can be extended to a 32 by 32 array of 1024 iWarp cells. Figure 2-21 shows the iWarp System Cabinet, which contains up to four Cardcage Assemblies.

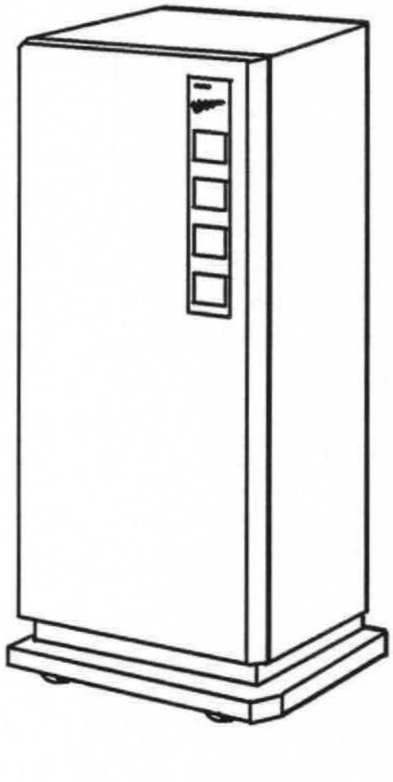


Figure 2-21: iWarp System Cabinet

The front door of the System Cabinet contains an LED display that shows status conditions for each iWarp cell housed in the cabinet. The LED display consists of four 8 by 8 LED arrays, with each array corresponding to one of the cardcages in the cabinet. Each pair of LEDs in the array corresponds to the status of a specific cell. There is also an error LED and a power LED for each array.

Diagnostics

The iWarp diagnostics consist of self-tests for the interface board and a set of tests to ensure the integrity of the entire system. The interface tests are run at power-up or when a system reset is done. These tests ensure that the interface board is functioning correctly. The system test checks the data paths from the host to the interface and from the interface to the processor array. These tests can be run interactively or in batch mode.

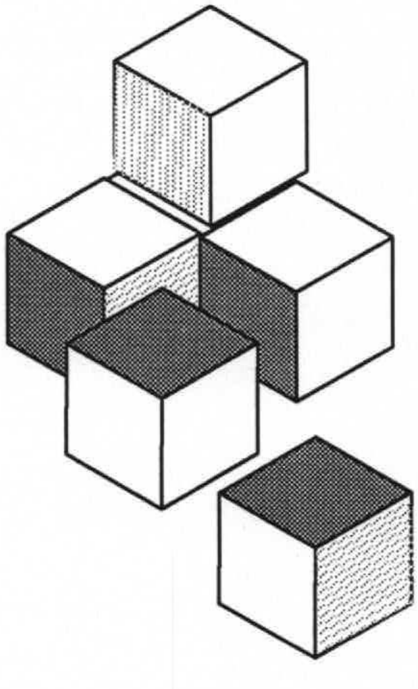
3 iWarp Software

C

UNIX

FORTRAN

Sun Host



iWarp Software Architecture

iWarp software architecture supports high-performance computation in signal and image processing and scientific applications. iWarp's software environment is composed of a program development environment that resides on the host workstation and execution support that resides on the iWarp array. This software, which is closely tied to iWarp's hardware architecture, provides the following:

- a host development environment supported on Sun workstations
- a runtime environment supported on each iWarp cell and the host

Parallel user code is executed on the array of iWarp cells. Sequential code can either run on the host or on a single iWarp cell. The software for the file server can also run on the I/O subsystem (see figure 1-12) or on the host. Figure 3-1 illustrates the relationship between iWarp's software and hardware architecture.

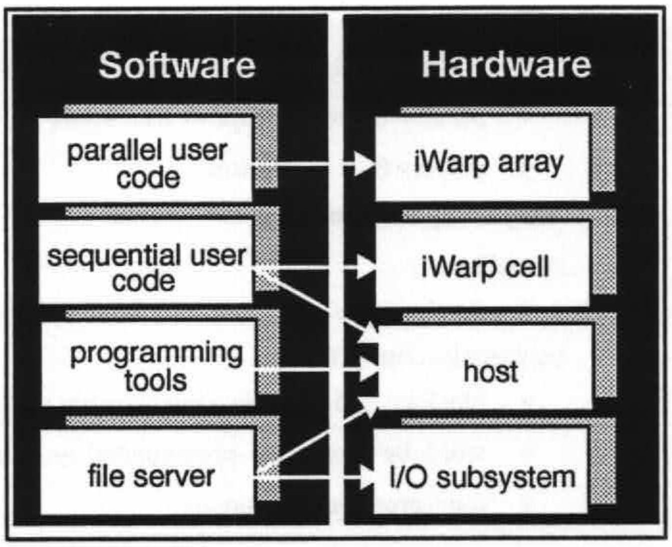


Figure 3-1: Relationship between iWarp software and hardware architecture

The following lists summarize the software for the iWarp cell and iWarp array. The software listed is described in the following sections of this chapter.

iWarp cell software

compilers:

- C
- FORTRAN
- symbolic debugger

utilities:

- linker/combiner
- loader
- librarian

communication software:

- blocking and non–blocking message passing using a mailbox paradigm
- inter–cell communication mechanisms can be used to communicate between processes on a single cell
- remote UNIX I/O

libraries:

- math libraries
- subset of UNIX Sys V Library calls

system functions:

- memory allocation
- user timers
- multi–threaded programs based on Mach C-threads
- priority-based preemptive scheduling
- process (thread) control

iWarp array software

compilers:

- Apply

communication software:

- blocking and non–blocking message passing using a mailbox paradigm
- word–by–word user–programmed systolic communication
- user–programmed spooling
- request–response and RPC protocols

libraries:

- WEB

system functions:

- support for UNIX file I/O to be executed on the attached host
- low-level functional access to iWarp communication engine
- remote thread creation and invocation

iWarp host software

- combiner
- array allocation
- array job management

iWarp Host Environment

The host environment (the Sun system) includes program development software and software that supports communication between the host and the iWarp array. The host software grants access to the iWarp array and maintains the connection between a user's system and the iWarp system after access is granted. The program development environment for iWarp is supported on Sun workstations using SunOS. This environment provides UNIX-based program development tools such as:

- cross compilers/assembler/linker
- loader
- debugger
- diagnostics

Figure 3-2 illustrates the structure of iWarp's host environment software.

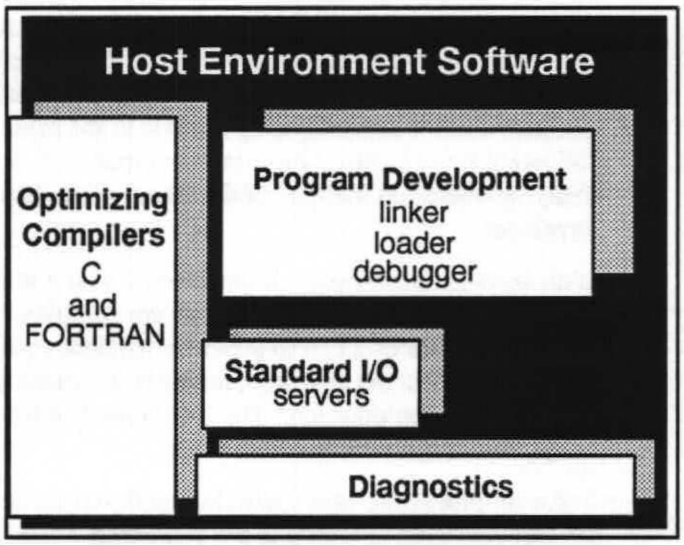


Figure 3-2: iWarp host environment

iWarp Program Development Tools

iWarp provides tools to develop, manage, link, load, and debug programs in a familiar UNIX environment.

The single-cell compilers

The iWarp C and FORTRAN compilers are highly optimizing compilers that run on the Sun workstation and generate iWarp object code for individual iWarp cells. The compilers, assembler, linker, and loader have standard UNIX interfaces. The compilers pack multiple operations into each wide instruction word, allowing the iWarp hardware functional units to execute those operations simultaneously. The iWarp C compiler is an industry standard Kernighan and Ritchie C language compiler with iWarp-specific extensions that provide access to the systolic pathway. The iWarp FORTRAN compiler accepts standard FORTRAN77 source with VMS extensions and iWarp-specific extensions.

The iWarp extensions for C and FORTRAN support:

- systolic communication support (send and receive primitives)
- iWarp condition code checking support
- sophisticated assembly code inlining capability
- pragma support for inlining specification

Getting better performance

To make a program run faster and take less space, the iWarp compilers employ a variety of optimizations to fully use the multiple functional units of iWarp cells. Two of these optimizations important to iWarp users are software pipelining and local code compaction. These code scheduling techniques allow the compilers to generate code with multiple operations in the same machine instruction. The following two examples discuss how pipelining and code compaction, two of the many optimization features of the iWarp compilers, support the iWarp program developer.

With software pipelining, an iteration of a loop in the source program can be initiated before preceding iterations are completed. This technique exploits the repetitive nature of loops to generate efficient code for processors with multiple functional units. At any time, multiple iterations are simultaneously in different stages of the computation. The steady state of this pipeline constitutes the loop body of the object code.

Software pipelining uses multiple functional units to perform the calculations of several iterations of a loop at the same time. The following example shows source code and pseudo assembler output that illustrate software pipelining of a simple loop.

```
DO I = 1, 10
          A(I) = A(I) * C
ENDDO
```

Register r_1 contains the iteration count minus 2 (10 - 2 = 8), and register r_2 contains the constant C. Registers r_3 and r_4 contain intermediate results.

```
{     load a1,r3                                               }
{     fmul r2,r3,r4;     load a2,r3                            }
```

```
loop  r_1
el {  store r_4,a_{i-2};    fmul r_2,r_3,r_4;    load a_i,r_3        }

{                           store r_4,a_{n-1}    fmul r_2,r_3,r_4    }
{                                                store r_4,a_n       }
```

In this example, three iterations of the loop are computed simultaneously. The "steady state" of the loop is shown on the line beginning with "el" (for, "end loop"). On the *i-th* iteration, the load of A(I) takes place in parallel with the multiplication of A(I-1) and C, and the store into A(I-2).

Code compaction can be performed anywhere there are enough source level operations to make the optimization worthwhile. The following example shows a sequence of operations typical of complex arithmetic.

```
REAL = (TEMP1 * WREAL) - (TEMP2 * WIMAG)
IMAG = (TEMP1 * WIMAG) - (TEMP2 * WREAL)
```

This example shows six operations, but with compaction, two of the operations can be overlapped, resulting in four instructions actually being used. Loading and storing of values can also be overlapped, giving a greater savings than this example implies.

Some additional optimizations include:

- redundant-instruction elimination
- flow-of-control optimizations
- algebraic simplification
- peephole optimizations
- function-preserving transformations
- common subexpression elimination
- copy propagation
- dead-code elimination
- induction variables and reduction in strength
- constant folding
- branch elimination
- variable renaming
- inline renaming
- loop unrolling

The linker and loader

The iWarp linker acts much like the UNIX linker. It combines multiple, separately compiled modules into one object file ready to be loaded onto the cell. The iWarp linker also supports user-created libraries. The loader places certain linked files into specified cells. In addition, the iWarp loader handles segments of files and acts as an array-level combiner to resolve intercell resource issues.

The debugger

The iWarp debugger helps the programmer monitor the behavior of iWarp cell programs and gather information about the programs being run. It is an interactive symbolic debugger that supports application debugging for C, FORTRAN, and iWarp assembly language programs. Debugger breakpoints allow the programmer to suspend and examine cell or array programs and then continue or terminate execution. The debugger displays both high-level and assembly language program text and can be used from a window-based interface.

iWarp Runtime Environment

The runtime system manages the resources of the hardware on which it runs and provides the application programmer with a set of routines that can be called to use these resources. The iWarp runtime environment provides libraries and communication protocols supported on each cell, the host, and the file server. These libraries include runtime libraries such as mathematical and utility functions, I/O libraries, and pathway libraries. The runtime environment works with the hardware to provide several internode communication paradigms for program development. The runtime environment includes:

- C and FORTRAN runtime libraries
- remote I/O runtime libraries
- basic runtime system support
- message passing protocol support
- systolic protocol support

Figure 3-3 illustrates the iWarp runtime environment software.

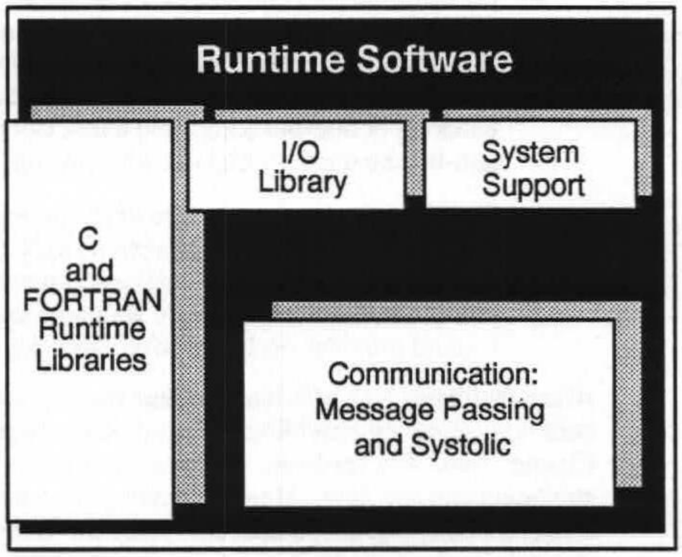

Runtime Software
I/O Library
System Support
C and FORTRAN Runtime Libraries
Communication: Message Passing and Systolic

Figure 3-3: iWarp runtime environment software

iWarp Runtime Libraries

The runtime libraries include the standard C, FORTRAN, and I/O libraries, as well as special pathway runtime libraries. The pathway runtime libraries provide the programmer with a set of routines for low-level access to the pathway hardware. These routines also provide a high-level interface for opening and debugging ports and connections as well as creating and accepting messages.

iWarp System Support

iWarp's runtime system software provides a complete, yet low overhead management of the iWarp component's resources. This software has the basic services associated with general-purpose runtime systems plus some special-purpose services to handle the distributed nature of an iWarp array.

iWarp provides a number of communication protocols that can be used to transfer data between cells or between a cell and host. These include high-level communication protocols, such as remote procedure calls and guaranteed arrival streams for the application programmer. Lower-level communication protocols, such as a data link layer, are for the user who intends to build higher-level protocols. In addition, multitasking provides support for asynchronous communication protocols that allow server processes to run in the background and provide service to communication requests as resources become available.

iWarp communication facilities support both memory-to-memory message passing communication and program-to-program word-by-word systolic communication.

- **Message-based communication** is based on variable length, untyped messages that are sent or received from buffers in cell memory. The user calls send and receive primitives to transfer data, specifying the protocol that should be used to transmit the data. Sends and receives may be blocking or non-blocking, and a task can resynchronize with a non-blocking call to find out when the operation is completed.
- **Systolic communication** allows the user to perform arbitrary computations on operands taken directly from the pathway without the overhead associated with memory-to-memory communication. Since the iWarp hardware and software are optimized for systolic data transfer, this method provides optimal performance for certain applications.

iWarp facilitates both efficient message passing and flexible systolic communication by providing the required hardware support, discussed in Chapter Two. The hardware supports word-level flow control, logical buses, and streaming and spooling. Message passing communication is a commonly used model for coarse-grain parallel computation. Conversely, systolic communication is typically used for fine-grain parallel computation.

Programming an iWarp System

So far in this chapter, we have discussed the overall software environment, how it relates to the hardware environment, and the software tools for programming the iWarp system.

The following program gives an example of what the user can do by programming the array in C. This approach requires that the programmer be familiar with the functioning of the cell array, but it offers great programming flexibility. As an alternative, the program developer can use the Apply language, which is specialized for image processing applications. Such programs are fairly easy to write in Apply.

The following simple example shows segments of an iWarp C program that evaluates a polynomial. It demonstrates the solution of an ***n-th*** order polynomial at ***m*** data points using an ***n+1*** cell ring array of iWarp cell processors. Each iWarp cell processor is connected to its right neighbor with two paths. Each of ***n*** cells computes one step of the Horner algorithm and passes the result to the right. The ***n+1th*** cell serves as a master cell. The master cell provides data and receives, stores, and prints the results. In this example, we are computing the following 5th order polynomial using 6 cells:

$$P(z)=c_0z^4+c_1z^3+c_2z^2+c_3z+c_4$$

Figure 3-4 illustrates how data flows from the master cell around the array of cells.

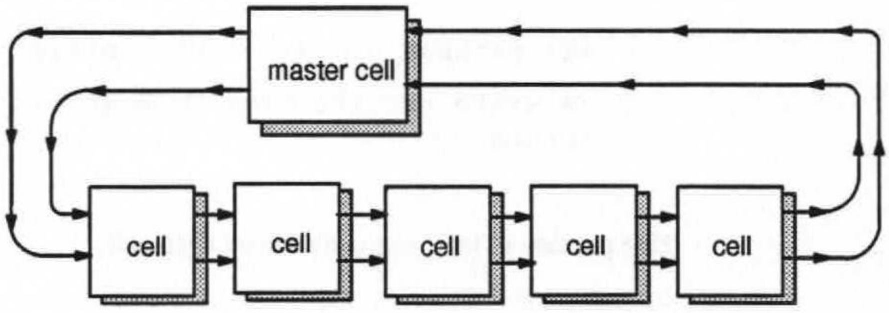


Figure 3-4: Master cell and cell communication configuration

There are two parts to the program: one part that runs on the master cell and one part that runs on each cell. The #include statements and the master cell program segment that reads in the data from the host and writes back the results have been omitted for brevity.

```
static float c[5]; /*coefficients*/
static float z[5]; /*data points*/
static float p[5];/*results*/
static int nc; /*number of coefficients*/
static int nz; /*number of data points*/

main()
{
  register int i, tmp, error = 0;
  register float ftmp, fzero = 0.0;

  /*host input*/

  if( pathway_init() ) return(1);

  _sendi( GATE0, nc );
  tmp = _receivei( GATE0 );
  _sendi( GATE0, nz );
  tmp = _receivei( GATE0 );

  /* send coefficients */

  for ( i = 0; i < nc; i++ )

    _sendf( GATE0, c[i] );

  /* send z and receive p */
  for ( i = 0; i < nz; i++ ) {
    _sendf( GATE0, z[i] );
```

```
      _sendf( GATE1, fzero );
      ftmp = _receivef( GATE0 );
      p[i] = _receivef( GATE1 );
    }

    if( pathway_close() ) return(1);

    /* write results back to host */
    return 0;

  }
```

This portion of the program runs on each cell.

```
  main()
  {
  register int i, nc, nz;
  register float temp, coeff, xin, yin, ans;
  register float fzero = 0.0;

  if( pathway_init() ) return(1);

    nc = _receivei( GATE0 );
    _sendi( GATE0, nc-1 );
    nz = _receivei( GATE0 );
    _sendi( GATE0, nz );

    /* capture the first coefficient
     * and pass the rest on.            */

    coeff = _receivef( GATE0 );
    for ( i = 1; i < nc; i++ ) {
      temp = _receivef( GATE0 );
      _sendf( GATE0, temp );
    }

    /* Horner's rule                    */

    for ( i = 0; i < nz; i++ ) {
      xin = _receivef( GATE0 );
      yin = _receivef( GATE1 );
      _sendf( GATE0, xin );
      ans = coeff + yin * xin;
      _sendf( GATE1, ans );
    }

    if( pathway_close() ) return(1);
    return 0;

  }
```

Apply: for Image Processing Applications

Apply is a special-purpose, high-level language for image processing applications that frees the programmer from having to program low-level inter-processor communication.

The Apply language:

- provides per-pixel generation of the output image
- allows a local operation to be written easily; more easily than with serial languages like C
- has special functions for borders, image expansion, and reduction

This language is designed for writing two-dimensional local operator algorithms. Apply generates parallel code that runs on an array of iWarp cells of any size. Apply and its implementation on iWarp is designed for implementing low-level vision algorithms such as:

- edge detection
- smoothing
- contrast enhancement
- thresholding
- point operators
- image addition, subtraction, multiplication, and division
- image reduction and expansion
- color conversion

The Apply language simplifies programming two-dimensional image processing operations. Data objects are scalars and two-dimensional arrays of scalars having various types. These types include byte, integer, real, and double. Apply uses conventional expression syntax to specify computations on these data objects.

Apply procedures are called from FORTRAN or C language main programs in much the same way as other procedures are called. On the calling side, arguments to the Apply procedures are declared according to syntax and semantics of the FORTRAN or C languages. Inside the Apply procedure, arguments are declared in a slightly different way, as required by the syntax and semantics of the Apply language.

Image data from the calling program is transferred from the calling program to the array of computational cells on which the Apply routine is executed. Results are returned to the calling program before control passes back to the calling program. This data transfer is normally invisible to the Apply programmer.

Apply procedures can be compiled for sequential execution on the Sun workstation for convenience in debugging new code. This code can be run as a single native program on the Sun workstation or can be run on a single iWarp cell that is fitted with a large memory. Afterwards, the Apply procedure can be compiled for high performance parallel execution on an array of iWarp cells.

An Apply program is a procedure that represents the inner loop of an image-to-image operation. The Sobel operator is a simple edge detection operation. The output is a combination of the horizontal and vertical edge values. An Apply implementation of the Sobel edge detector is shown in the following example. In this example, the operation is performed on a 3x3 window. The horizontal edge value is **horiz** and the vertical edge value is **vert.** One of two different methods of computing **imageout** is used, depending on the value of **type.**

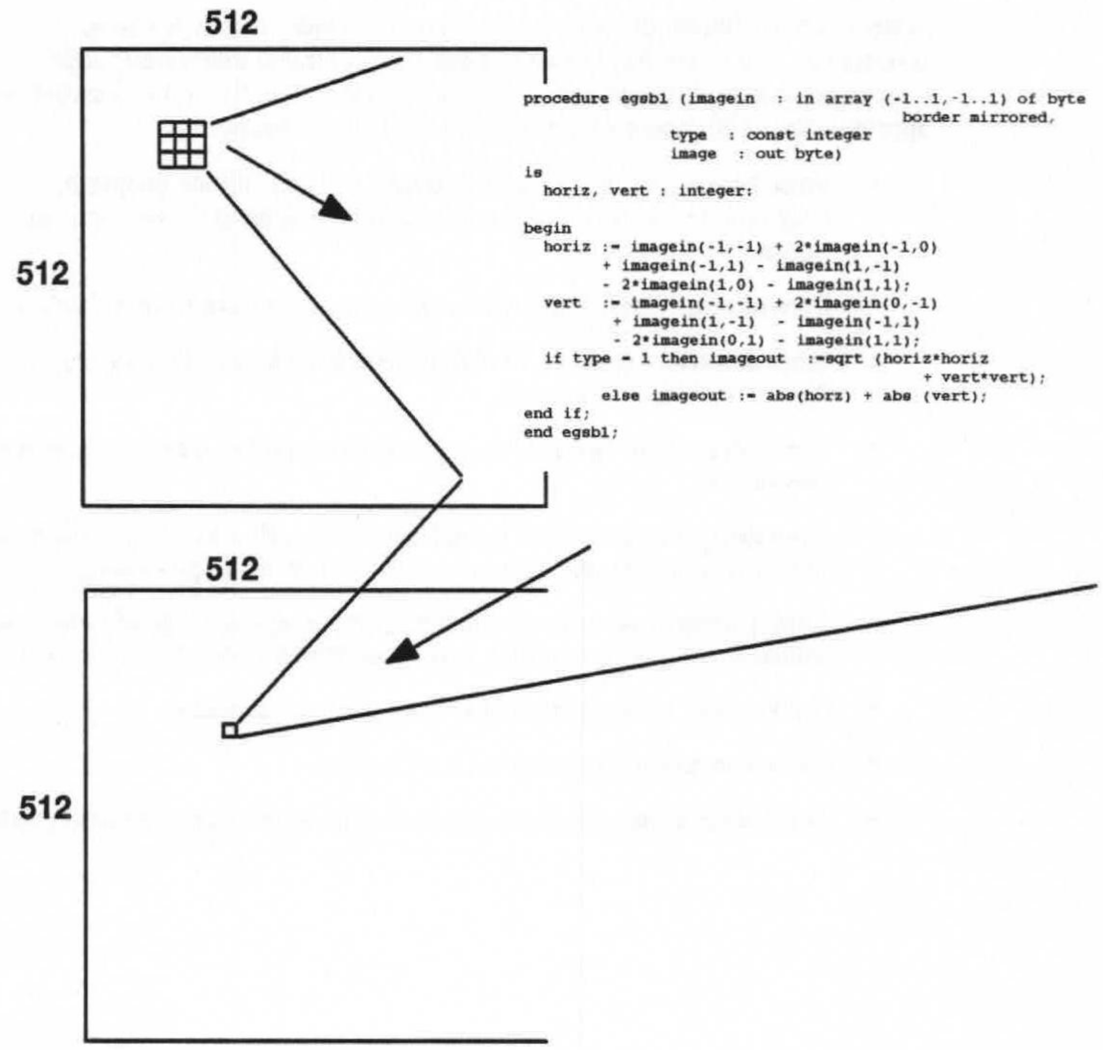
```
procedure egsbl (imagein : in array (-1..1,-1..1)       --1
                        of byte   border mirrored,
                        type  : const integer
                        image  : out byte)
is                                                      --2
   horiz, vert : integer;                               --3

begin                                                   --4
   horiz := imagein(-1,-1) + 2*imagein(-1,0)            --5
           + imagein(-1,1) - imagein(1,-1)
          - 2*imagein(1,0) - imagein(1,1);
   vert  := imagein(-1,-1) + 2*imagein(0,-1)            --6
            + imagein(1,-1)  - imagein(-1,1)
            - 2*imagein(0,1) - imagein(1,1);
   if type = 1 then imageout  := sqrt(horiz*horiz      --7
                                      + vert*vert);
       else imageout := abs(horiz) + abs (vert);
   end if;
end egsbl;                                              --8
```

Line **1** defines the input, output, and constant parameters to the function. The input parameter **imagein** is a window of the input image. Line **1** also defines the input image window. The window is a 3x3 window centered about the current pixel processing position. This position is filled with the value 0 when the window lies outside the image. This same line declares the constant and output parameters to be floating-point scalar variables. Line **3** defines **horiz** and **vert,** which are internal variables used to hold the results of the horizontal and vertical Sobel edge operator.

The straightforward expressions on lines **5** through **7** implement the computation of the Sobel convolutions.

The Apply programmer cannot control the order in which the Apply program is executed over the image. This restriction is the key to the easy mapping of Apply programs onto parallel computers. Because the order is unrestricted, the entire image can be processed in parallel if there are as many processors as pixels, or it can be processed in sections, one section per processor, if there are fewer processors than pixels.

Figure 3-5 illustrates how the edge detection program works.



```
procedure egsbl (imagein  : in array (-1..1,-1..1) of byte
                                                border mirrored,
                 type   : const integer
                 image   : out byte)
is
  horiz, vert : integer:

begin
  horiz := imagein(-1,-1) + 2*imagein(-1,0)
         + imagein(-1,1) - imagein(1,-1)
         - 2*imagein(1,0) - imagein(1,1);
  vert  := imagein(-1,-1) + 2*imagein(0,-1)
          + imagein(1,-1)  - imagein(-1,1)
          - 2*imagein(0,1) - imagein(1,1);
  if type = 1 then imageout  :=sqrt (horiz*horiz
                                          + vert*vert);
        else imageout := abs(horz) + abs (vert);
end if;
end egsbl;
```

Figure 3-5: Using Apply for programming

Special features extend the power of the Apply language and match what is needed to write image processing programs. Apply provides the user with these important features :

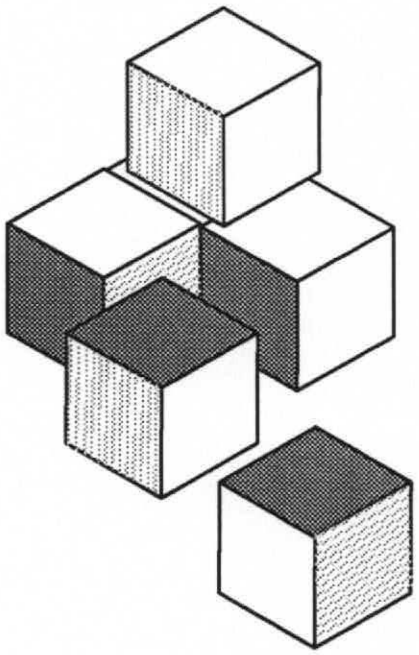
- The apply program is written on a per-pixel basis rather than a per-image basis.
- All the input parameters are input images; all output parameters are output images.
- The **const** parameters are broadcast all across the image; all cells get the same value.

WEB Library Routines

WEB is a basic library of routines, based on the Spider library for image processing. All of the Apply routines can be recompiled easily for C code generation and for different image sizes and number of cells. WEB consists of approximately 100 programs covering the areas listed below:

- **Basic image operations**: add, subtract, multiply, divide images by images and images by constants, assign zeroes, assign constant inside region.
- **Convolution**: convolution with a variable or constant weight window.
- **Edge detection**: Roberts, Frei and Chen, Kirsch, Sobel Laplacian, Prewitt, Robinson, Kasvand.
- **Image greyvalue operations**: clip threshold, remap greyvalues, reduce greyvalues.
- **Smoothing**: adaptive local smoothing, median filtering, local maximum and minimum, iterative enhancements, texture image processing.
- **Binary image processing**: detect borders, compute image of boundary points, connectivity, crossing, expand or contract, shrink components.
- **Conversion**: byte to real, real to byte, polar to cartesian.
- **Color conversion**: color to black and white.
- **Multi-level image processing**: generate pyramid, reduce by half, double.

Appendix: Computational Models



Phil. Trans. R. Soc. Lond. A **326**, 357–371 (1988) [357]
Printed in Great Britain

Computational models for parallel computers

BY H. T. KUNG

Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, *U.S.A.*

Computational models define the usage patterns of a computer. They can be used to derive the architecture of the machine, provide guidelines for programming tools, and suggest how the machine should be used in applications. Identifying computational models is especially important for parallel computers, because their architectures and usages are still not well understood in general.

This paper describes a number of computational models for parallel computers. These models characterize the communication patterns under which processors exchange their intermediate results during computation. Emphases are placed upon models for one-dimensional processor arrays, reflecting Carnegie Mellon's experiences with the Warp systolic array machine. These models include local computation, domain partition, pipeline, multifunction pipeline and ring.

1. INTRODUCTION

Many problems in science and technology are becoming so computationally demanding that conventional sequential computers can no longer provide the required computing power. Parallel computers have the potential to provide that power. A large number of parallel computers are commercially available. Shared-memory parallel computers include MIMD (multiple instruction multiple data) machines such as Alliant, Encore, Sequent, and Cray X-MP. Distributed-memory computers include MIMD machines such as Transputer, Warp, and Hypercube, and SIMD (single instruction multiple data) machines such as Connection Machine and DAP. Many more parallel machines of enhanced capabilities are under development. Successful use of parallel computers has been demonstrated in a number of application areas including scientific computing, signal and image processing, and logic simulation.

It is useful to develop models to capture important ways in which parallel computers are actually used in applications. These models can be used to derive architectures of new parallel machines, provide guidelines for programming tools, and suggest how each machine should be used in applications. There are roughly three stages in solving an application problem on a parallel computer:

step 1, application definition (e.g. by mathematical formula);
step 2, computation specification (e.g. by program);
step 3, computation on the parallel machine.

Computational models described in this paper characterize the interprocessor communication behaviour of step 3.

These computational models are based on our experiences in parallel algorithm design and parallel architecture development at Carnegie Mellon. In 1984–87 Carnegie Mellon developed a programmable systolic array machine called Warp, that has a one-dimensional (1D) array of 10 or more processing elements (Annaratone *et al.* 1987). The machine is currently produced and marketed by General Electric Company. Anticipating the future need for integrated Warp systems, Carnegie Mellon and Intel Corporation have been developing a VLSI (very large scale integrated) Warp chip, called the *i*Warp chip. The *i*Warp system will be available in 1989–90. Our work in Warp and *i*Warp has shown us the importance of being explicit about computational models in the development of a new parallel architecture as well as its applications and programming tools. The paper will mention some of these insights.

In this paper we describe computational models for 1D processor arrays. We use 1D processor arrays because their simple structure makes presentation easy and we have extensive applications experiences with the 1D array in Warp. It should be clear that the concepts presented here generalize to 2D or higher-dimensional processor arrays, and other parallel computer architectures.

Section 2 provides background information on the Warp and *i*Warp systems. Nine computational models for 1D processor arrays are presented in §3. Among them five models are frequently used on Warp. These are models corresponding to local computation, domain partition, pipeline, multifunction pipeline and ring. They will be discussed in more detail than the other models. The last section contains some concluding remarks.

2. Overview of Warp and *i*Warp

2.1. *Warp*

The Warp machine has three components: the Warp array, the interface unit, and the host, as shown in figure 1. We describe the machine only briefly here; details are available from a separate paper (Annaratone *et al.* 1987). The Warp array performs the bulk of the computation. The interface unit handles the input–output between the array and the host. The host supplies data to and receives results from the array, in addition to executing the parts of the application programs that are not mapped onto the Warp array.

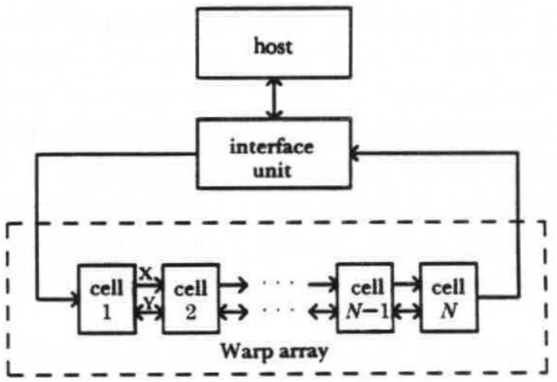


FIGURE 1. Warp machine overview.

The Warp array is a 1D systolic array with identical processing elements called Warp cells. Data flow through the array on two communication channels (X and Y), as shown in figure 1. The direction of the Y channel is statically configurable at compile time. By putting the Y channel in the opposite direction from the X channel, a ring interconnection can be formed inside the 1D array. Another way to form a ring is to use the interface unit to connect the first and last cells of the array.

Each Warp cell is implemented as a programmable horizontal micro-engine, with its own microsequencer and program memory. The cell data path includes a 5 MFLOPS (5×10^6 floating-point operations per second) floating-point multiplier (Mpy), a 5 MFLOPS floating-point adder (Add), a local memory, and two data input queues for the X and Y channels. All these components are connected through a crossbar. An output port of the crossbar can receive the value of any input port in each cycle. Via the crossbar the floating-point units can directly access data at the front of any input queue, and insert computed results at the end of any input queue of the next cell. Data at the front of any input queue can also be sent directly to the next cell. A (much) simplified description of the Warp cell data path is given in figure 2.

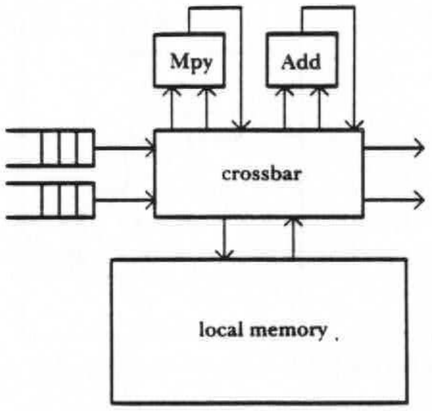


FIGURE 2. Warp cell data path (much simplified).

A feature that distinguishes a Warp cell from many other processors of similar computation power is its high I/O bandwidth, an important characteristic for systolic arrays. Each Warp cell can transfer up to 20×10^6 words (80 Mbytes) to and from its neighbouring cells per second. This high intercell communication bandwidth makes it possible to transfer large volumes of intermediate data between neighbouring cells and support fine-grain parallelism on the Warp array.

The host consists of a Sun-3 workstation that serves as the master controller of the Warp machine, and a VME-based multi-processor 'external host', so named because it is external to the workstation. The workstation provides a UNIX environment for running application programs. The external host controls the peripherals and contains a large amount of memory for storing data to be processed by the Warp array. Its dedicated processors transfer data to and from the Warp array and perform operations on the data, with low operating system overhead.

Warp programs are written in a high level PASCAL-like language called W2, which is supported by an optimizing compiler (Gross & Lam 1986; Lam 1987). To the application programmer, Warp is a 1D array or a ring of simple sequential processors, communicating asynchronously. Based on the user's program for this abstract array or ring, the compiler generates code for the host, interface unit and Warp array automatically. W2 programs are developed in a LISP-based programming environment supporting interactive program development and debugging. A C or LISP program can call a W2 program from any UNIX computer on the local area network.

2.2. *iWarp*

Carnegie Mellon and Intel are jointly developing a large VLSI chip, called the *i*Warp chip, to implement an integrated version of the Warp cell. The *i*Warp chip is a programmable processor capable of delivering at least 20 or 10 MFLOPS for single or double precision floating-point computations, respectively. This chip together with a local memory form the *i*Warp cell, is shown in figure 3. The *i*Warp cell is a powerful building-block cell for a variety of processor arrays, including 1D and 2D arrays. With recompilation, the *i*Warp cell will be able to execute W2 programs originally written for the Warp cell.

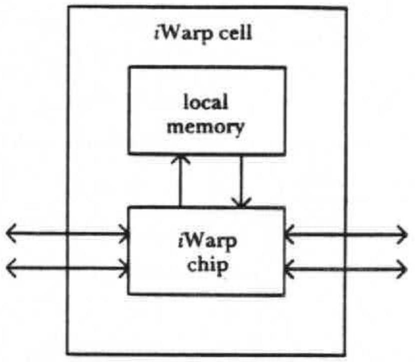


FIGURE 3. *i*Warp cell consisting of *i*Warp chip and local memory.

The initial prototype *i*Warp system will have an array of 72 *i*Warp cells, with a peak performance of at least 1440 MFLOPS. To ensure that a large fraction of this peak performance can actually be realized in real applications, the *i*Warp array supports the following features:

large local memory for the cells (at least 24 address bits);

high bandwidth intercell communication (320 Mbytes s^{-1});

2D or higher-dimensional interconnection;

on-chip message routing hardware.

Passing messages by a cell is handled by its routing hardware, and is transparent to its program. This implies that communication between non-neighbouring cells can now be easily accomplished.

3. Computational models

We will describe the following computational models for 1D processor arrays:

1. local computation;	4. multifunction pipeline;	7. divide-and-conquer;
2. domain partition;	5. ring;	8. query processing; and
3. pipeline;	6. recursive computation;	9. task queue.

These models correspond to different ways in which cells interchange their intermediate results during computation. Under each model there may also be different ways in handling inputting and outputting for the processor array (see discussions below concerning the local computation model). Therefore the computational models are based on the communication behaviour for intermediate results rather than input and output.

The current Warp system uses the first five models mostly, whereas the future *i*Warp system will efficiently support all the models. Because of our experience with Warp, we will give more detailed descriptions for the first five models. The other models will only be briefly touched, mainly to indicate that there are other models which could be important for parallel computers to support.

In the diagrams, cells in a 1D processor array are denoted by square boxes, and named as cell 1, cell 2, ..., cell N from left to right. Solid arrows denote data flows of intermediate results between cells.

3.1. *Local computation model*

The local computation model corresponds to the case where cells do not exchange their intermediate results during computation at all. Many computational problems have the property that elements in the output set are computed independently from each other. The use of the local computation model is natural in solving these problems on a parallel computer. In this model each output is computed entirely within a cell, and all the cells compute different outputs simultaneously. The main characteristic is that the entire computation for each output is done locally at a cell, i.e. the computation does not depend on intermediate results computed by other cells.

Various methods can be used to take care of the inputting and outputting for each cell. For example, before or during computation, the required input to a cell can be shifted in via the cells to the left, and during or after the computation the output produced by a cell can be shifted out via the cells to the right. This is shown by figure 4, where dotted arrows denote the shift-in and shift-out paths for input and output, respectively. To achieve high performance, it is important that the I/O time and computation time can be overlapped as much as possible.

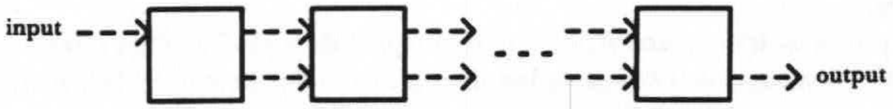


FIGURE 4. Local computation model, with input and output shifted in and out.

Many image processing computations involve transforming an input image to an output image, using a kernel operator defined by, say, a 3×3 window. Figure 5 shows such a transformation, with which each pixel in the output image depends on a neighbourhood of the corresponding pixel in the input image. Clearly, all the pixels in the output image can be computed simultaneously and independently. Therefore the local computation model applies here. The figure illustrates that four cells can work on the four subregions of the output image independently, provided that the input pixels needed by each cell's computation are pre-stored in the cell. Note that cells computing adjacent subregions have overlapped input; the larger is the kernel, the larger is the overlap.

As shown by the figure, the partitioning of the image processing task for the local computation model is straghtforward. All that needs to be done is to partition the output image equally for all the cells. This partitioning has been automated; Carnegie Mellon has developed a compiler called Apply, which can generate W2 programs for image-processing computations based on kernel operators as described above, and other computations of similar kind (Hamey *et al.* 1987).

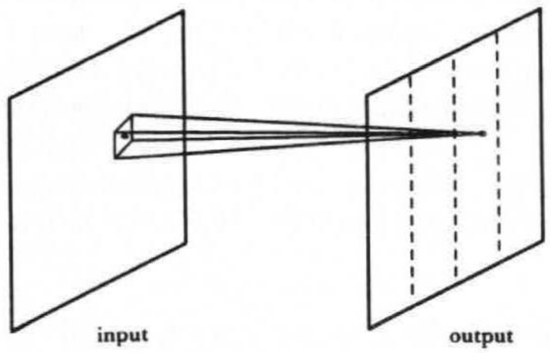


FIGURE 5. Local computation model for image processing using a kernel operator straightforward.

Apply-generated W2 programs are able to overlap I/O with computation. While computing a row of pixels for the output image, a cell can output a previous row of pixels already computed and input a new row of pixels required for future computations. The Warp array supports this overlapping well, because the array has a high intercell communication bandwidth, and each cell is a horizontal micro-engine capable of performing several computation and I/O operations in each cycle. Because with Apply this overlapping is done automatically, Apply-generated Warp programs are often more efficient than the corresponding hand-generated code.

There is another interesting form of overlapping input with computation for the local computation model. Although all the cells compute different parts of the output set, the cells may share some input. In this case the shared input may be pumped systolically from cell to cell during computation. In the following this is illustrated with a matrix multiplication example.

Given $n \times n$ matrices A and B, we wish to compute their product C on a linear processor array of k cells. We assume that k is much less than n, and in the illustration below, $k = 4$. We evenly partition columns of B and C as shown in figure 6*a*. By using the local computation model, cell i will compute the entires of submatrix C_i. As its inputs, cell i needs A and B_i. Therefore input A is shared by all cells. Cell i will first load entries of B_i into its local memory. Then during computation, entries of matrix A will be input to the left-most cell in the row-major ordering, and shifted to the right from cell to cell, as shown in figure 6*b*. Cell i will perform inner products

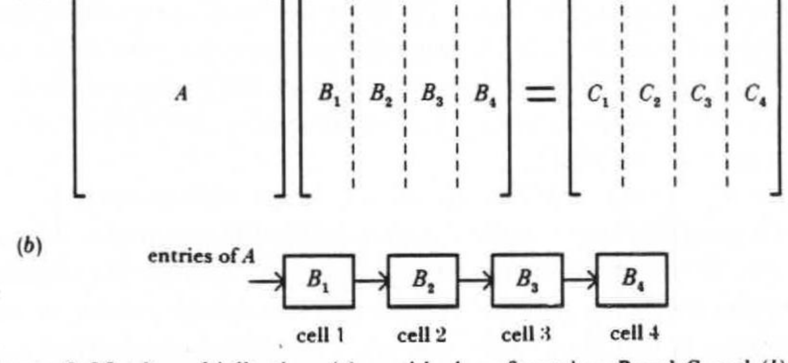


FIGURE 6. Matrix multiplication: (*a*) partitioning of matrices B and C, and (*b*) distribution of the resulting submatrices of B to the cells; entries of A moving to the right during computation.

for all pairs of row and column in A and B_i, respectively. (Each entry of A will be input repeatedly as it will be used by each cell multiple times, one for each of the columns of B that the cell has.) Each inner product involves reading in a row of A from one of its input queues and a column of B_i from the cell's local memory, and performing a sequence of multiply-accumulate operations. By shifting in entries of A on-the-fly, each cell does not have to store the entire matrix. This can significantly save memory storage and access time for each cell (Kung 1988).

There are many other usage examples based on the local computation model. They include the discrete cosine transform (Annaratone *et al.* 1986) and the labelled histogram computation (Kung & Webb 1986).

3.2. *Domain partition model*

For some applications the computation shown in figure 5 is repeated many times; each time a new output image is computed based on the previous output image. This computational process, called successive relaxation (Rosenfeld 1977; Rosenfeld *et al.* 1976), is shown in figure 7, where the grids correspond to the images.

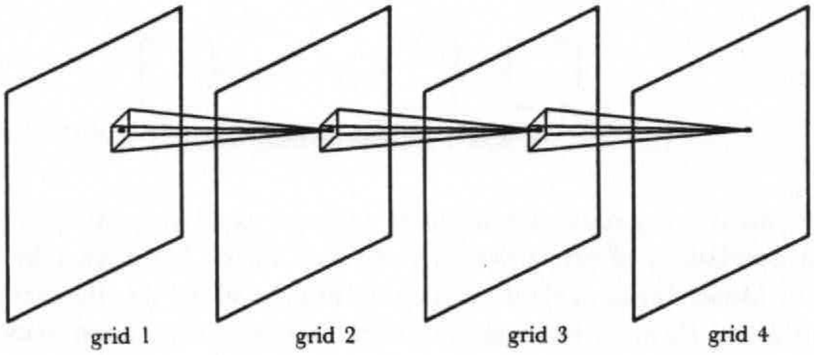


FIGURE 7. Successive relaxation.

The successive relaxation process is often used in scientific computing. Consider, for example, the solution of the following elliptic partial differential equations using successive overrelaxation (Young 1971):

$$\partial^2 u/\partial x^2 + \partial^2 u/\partial y^2 = f(x, y).$$

The system is solved by repeatedly computing values of u on a 2D grid using the following recurrence:

$$u'_{i,j} = (1-\omega)\, u_{i,j} + \tfrac{1}{4}\omega(f_{i,j} + u'_{i,j-1} + u_{i,j+1} + u_{i+1,j} + u'_{i-1,j}),$$

where ω is a constant parameter. In the recurrence, values associated with location (i,j) of the grid have indices (i,j).

Suppose that the partitioning scheme of figure 5 is used. Then when computing a new grid, each cell must import from its neighbouring cells some of the values computed for the previous grid. The required bidirectional data flows between neighbouring cells are shown in figure 8.

With this example, the concept of the domain partition model can be easily introduced. The model arises when a problem domain (such as the grid space corresponding to an image, or to

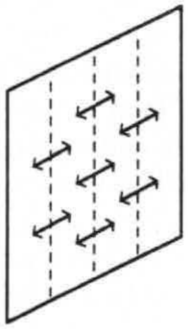


FIGURE 8. Bidirectional data flows for successive relaxation.

a finite-difference or finite-element modelling) is partitioned so that each cell handles a subdomain. This model differs from the local computation model in that each output is not computed entirely by a single cell. That is, once in a while the cell needs to receive intermediate results from its neighbouring cells before it can proceed further with its computation. Figure 9 shows the domain partition model.

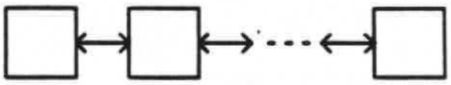


FIGURE 9. Domain partition model.

There are many computations that can be naturally done with the domain partition model. Numerical simulations of properties of a physical object, formulated by either differential equations or Monte Carlo methods, can be partitioned along the physical space. A large file can be sorted on a 1D array by using the bidirectional communication to merge sublists sorted by individual cells. The merging can be done with only nearest-neighbour communications, in a manner similar to that used in the odd–even transposition sort (Baudet & Stevenson 1978). Labelling of connected components in an image can be done by using the bidirectional communication to merge labels in the subimages computed by individual cells (Kung & Webb 1986).

3.3. *Pipeline model*

There is another (elegant) method to carry out the successive relaxation computation shown in figure 7 on a 1D array. This method uses pipelining. Instead of the data space, i.e. the grid, we partition along the time axis. That is, successive relaxation steps are done on successive cells. In the row-major ordering, each cell receives inputs from the preceding cell, performs its relaxation step, and outputs the results to the next cell. Consider, for example, the successive overrelaxation computation described in §3.2. While a cell is performing the kth relaxation step on row i, the preceding and next cells perform the $(k-1)$th and $(k+1)$th relaxation steps on rows $i+2$ and $i-2$, respectively. Thus, in one pass of the u values through a k-cell processor array, k relaxation steps are performed. This process is repeated, until convergence is achieved. In a similar way we can implement many other iterative methods such as Jacobi and Gauss–Seidel methods in a pipelined manner.

In this pipeline model, the computation for each output is partitioned into a sequence of identical stages, and cell i is responsible for stage i. A characteristic of this model is that cell $i+1$ uses computed results of cell i, as shown in figure 10. Intermediate results move in one direction and final results emerge from the last cell. I/O and computation are automatically overlapped; this is a major advantage of the model. The pipeline model is natural when implementing systolic algorithms where the partial results move from cell to cell and get updated at each cell they pass (Kung 1982; Kung & Leiserson 1979).

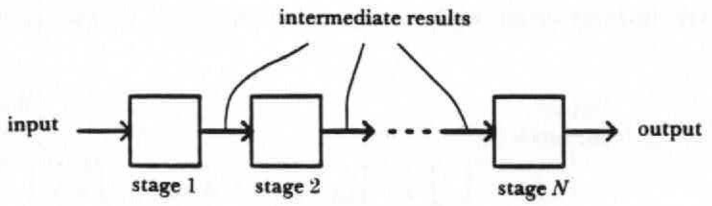


FIGURE 10. Pipeline model.

Under the pipeline model, cell $i+1$ cannot start its operation until cell i completes at least a stage of computation. Thus for this model minimizing the latency between the starting times of adjacent cells is a major concern. This is in contrast with the domain partition model, for which the starting time of a cell does not depend upon any computed results of other cells.

For some computations the pipeline model represents the only efficient parallel implementation. To see such a case, consider a variant of the image processing task shown in figure 5. For this variant, in computing the value of each point, the new values of its neighbours will be used whenever possible. Suppose that using a 3×3 window, the computation follows the row-major ordering. Then computing the value of each new point uses the new values of the left neighbour and the upper three neighbours, which were computed earlier. Local computation and domain partition models will not work here, as subregions of the image cannot be computed independently from each other. A way of using the pipeline model is that cell i computes values of points in row i in the left to right order. Cell i is pre-stored with values of points in rows i and $i+1$. During computation, a copy of each new value cell i computes is sent to cell $i+1$. Note that cell $i+1$ can start its computation as soon as cell i has computed the values of the first two points in row i. We have implemented a version of this pipeline computation on Warp to solve a path planning problem using a dynamic programming technique (Bitz & Kung 1988).

3.4. *Multifunction pipeline model*

A single computation may involve a series of subcomputations each performing a different function. If these functions can be chained together on a 1D array, then a one-pass execution of the entire computation will be possible. This is the basic idea of the multifunction pipeline model (Gross *et al.* 1985). In this model, the 1D array is a pipeline of several groups, each consisting of a number of cells devoted to a different function. The number of cells in each group is adjusted so that every group will take about the same time, to maximize the pipeline throughout.

This model is illustrated in the following example, which is a laser radar simulation implemented on Warp.

Step 1, for every 1024-point input block, perform a 1024-point complex FFT (fast Fourier transform). Partition each FFT output into 30 overlapped 256-element subsequences.

Step 2, for each of the 30×256-element subsequences, perform the following operations:

(i) multiply each element by a weight, which is a complex number;

(ii) perform a 256-point complx inverse FFT;

(iii) compute the amplitude of each element in the output subsequences.

Step 3: threshold the resulting 30×256 image using 3×3 windows.

These steps are implemented with consecutive segments of the Warp array, as shown in figure 11.

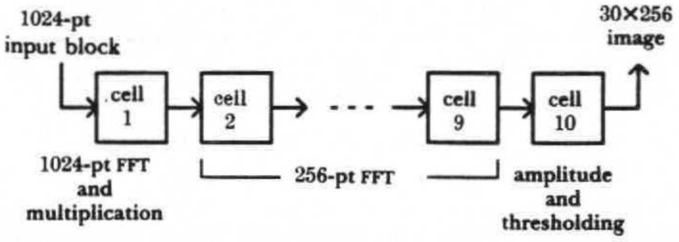


FIGURE 11. Multifunction pipeline model to implement a radar simulation on Warp.

Figure 12 shows another possible use of the multifunction pipeline model in implementing the geometry system portion of 3D computer graphics. The first cell performs the matrix multiplications, the next three cells do clipping, and the last cell does the scaling operation. Three cells are devoted to clipping as it requires more arithmetic operations then either matrix multiplication or scaling (Hsu *et al.* 1985).

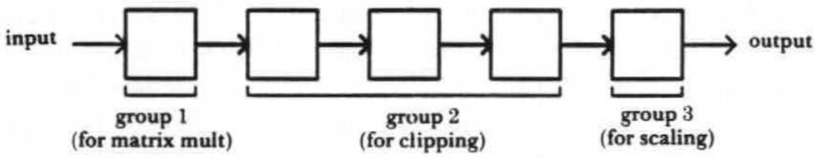


FIGURE 12. Multifunction pipeline model to implement a geometry system.

The multifunction pipeline model is useful when a computation requires a number of small functions, each of which is not large enough to make an effective use of all the cells in a 1D array. Concatenating these functions in a chain offers an opportunity to use more cells effectively. Also, for some computations, it is inherent that one or few cells must perform functions different from the rest. For example, when performing a 2D convolution on a 1D array, some cells need to buffer a row of image and none of the other cells need to do that (Kung 1984). For some computations, the first and last cells of a 1D array carry out special functions such as interface with the outside world or preparation of data for the next phase of computation on the array. An example of this is a neural network simulation on Warp, where only the last cell performs weight updates based on weight changes computed by other cells (Pomerleau *et al.* 1988).

To support the multifunction model, the processor array must allow heterogeneous programming, that is, different programs to be executed at different cells at a given time. Further, the rate of the input to a group may not be compatible to that of the output from the preceding group. Thus some buffering and flow control mechanisms need to be provided between each pair of cells. For the Warp array, all cells can be individually controlled, and dedicated hardware queues capable of performing flow control are available between adjacent cells.

In summary, the multifunction model differs from the pipeline model described earlier in that cells are now allowed to perform different functions. This flexibility in the usage offers the opportunity of effectively using a large number of cells in a 1D array.

3.5. *Ring*

A 1D array becomes a ring when the first cell is connected to the last cell. In the ring model intermediate results flow on a ring of cells.

An important usage of the ring model is the implementation of a large 'logic' array of logical cells, under the pipeline model, with a small 'physical' array of physical cells. One implementation is to have each physical cell handle a group of consecutive logical cells as shown in figure 13*a*. This will incur a large latency between the starting times of two adjacent physical cells, as the latency will be the sum of all the latencies incurred by those logical cells which are assigned to a physical cell. Another implementation is to use the physical array in multiple passes to simulate the function of the logical array, as shown in figure 13*b*. This multiple pass scheme can be implemented with a ring as shown in figure 13*c*. The ring is formed by using a queue to connect the last physical cell to the first. The queue can store outputs from the last physical cell while the first is still busy in doing its computation for the current pass. This ring scheme incurs the minimum latency between the starting times of two adjacent physical cells.

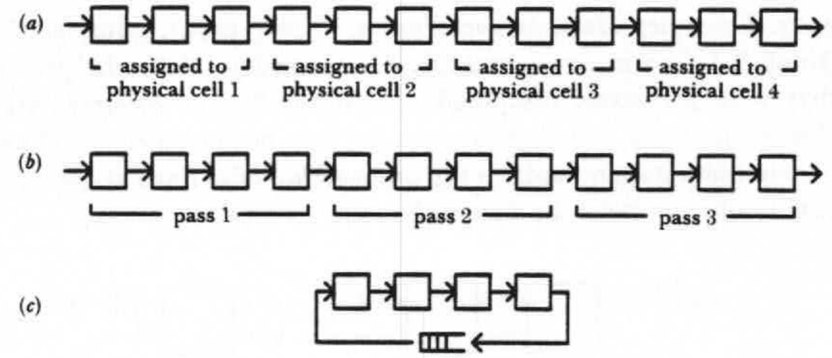

(a)
assigned to physical cell 1
assigned to physical cell 2
assigned to physical cell 3
assigned to physical cell 4
(b)
pass 1
pass 2
pass 3
(c)

FIGURE 13. Implementing a large pipeline with a small physical array: (*a*) each physical cell is assigned to a set of consecutive logical cells, (*b*) using the physical array in multiple passes and (*c*) using a ring to implement the multiple passes on the physical array.

Another major use of the ring model is in the implementation of broadcasting. Many computational problems involve multiple levels of computation as depicted in figure 14*a*. Each value in a level depends on all the values computed in the previous level. For example, in the figure to compute b_1 in level 2 we need all the values in level 1, as indicated by the lines connecting b_1 with a_1, a_2, a_3 and a_4. Therefore all the values computed in a level need to be broadcast to all the cells which will be computing values in the next level. An example of such a computational problem is the back propagation neural network simulation (Rumelhart *et al.* 1986), for which levels of computation correspond to layers of the neural network.

The ring structure can implement the broadcasting in a natural way, provided that the computation for each value is commutative and associative so that inputs in the previous level can be combined in any order. Figure 14*b* illustrates the idea, by considering how values in level 1 can be sent to cells computing values in level 2. Assume that every value in a layer is computed by a separate cell, and for each i the cell which computes a_i will also compute b_i. Then by pumping the a_is around the ring for a full cycle, as shown in figure 14*b*, cell i (for every i) will be able to meet all the a_is so it will have all the inputs to compute b_i. The computation of b_i will occur on-the-fly as each a_i passes by. Therefore computation and I/O are totally overlapped.

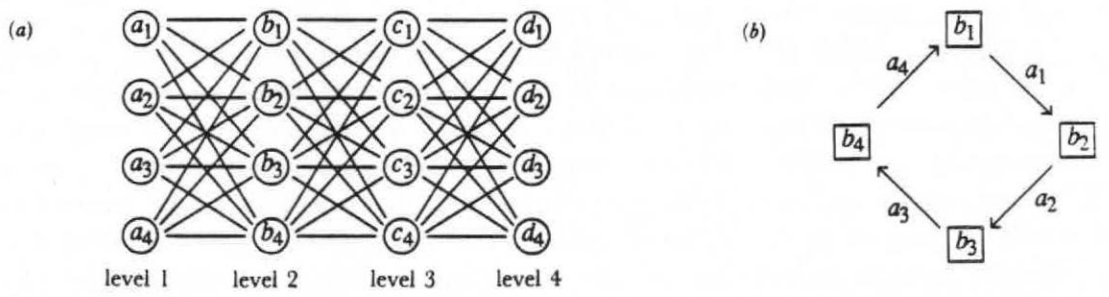


FIGURE 14. (*a*) Multilevel computation where results in one level are broadcast to the next level, and (*b*) use of the ring model to implement the broadcasting.

3.6. *Recursive computation model*

Recursive computations are those where results of the computation are used for computing future results. Examples are recursive filtering (Kung 1979), solution of triangular linear systems (Kung & Leiserson 1979), and QR-decomposition (Heller & Ipsen 1982). By flowing outputs that were previously computed against the flow of intermediate results that are currently being computed, recursive computations can be implemented. The important feature of the recursive computation model is the propagation of outputs in the opposite direction of intermediate results, as shown by figure 15.

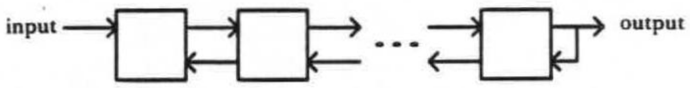


FIGURE 15. Recursive computation model.

3.7. *Divide-and-conquer model*

Divide-and-conquer is a fundamental technique in algorithm design (Aho *et al.* 1975). Under this design paradigm, we solve a problem by (1) partitioning it into subproblems of nearly equal size, (2) solving all the subproblems and (3) merging the solutions to the subproblems; this procedure is applied recursively to all the subproblems. Because of this recursion, this partitioning scheme distinguishes itself from others used, in, for example, the local computation and domain partition models. Figure 16 shows the divide-and-conquer model. Each subproblem is carried out by one cell or a set of consecutive cells. When a (sub)problem is partitioned into subproblems or solutions to subproblems are merged,

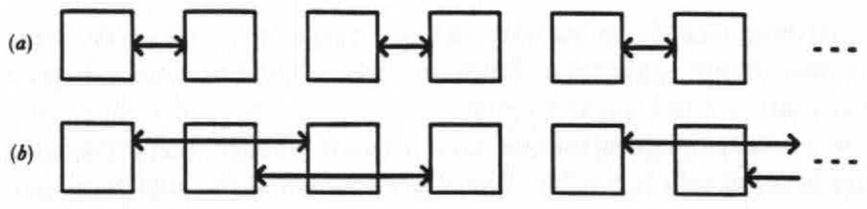


FIGURE 16. Divide-and-conquer model: (a) 1-apart communication; (b) 2-apart communication.

communications between cells that are either 1-apart, 2-apart, ..., or $N/2$-apart take place. These communications are depicted by solid arrows in the figure.

The divide-and-conquer model for example can be used in sorting, and various geometric problems such as computing convex hulls (Preparata & Shamos 1985).

3.8. *Query processing model*

A 1D array can be used to process queries. One way to do this is to have the database partitioned evenly among the cells. Then queries are passed to all the cells. Every cell looks at the arriving query and outputs its reply to the query. The query processing model is shown in figure 17.

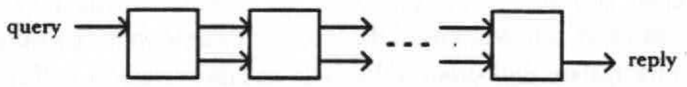


FIGURE 17. Query processing model.

Consider for example the problem of looking for a table in an image. The particular table we are searching for is defined as having a rectangular top, which will appear as a parallelogram in the image. Initially, we do not know anything about the position of the table, except an upper bound on the size of its bounding square in the image. After extracting features such as lines and edges from the image, we partition it into regions whose sizes are at least that of the bounding square for the table. We assign each region to a cell. To balance the computational load between the cells, we define the regions so that there are about the same number of features associated with each region. Regions assigned to the cells are properly overlapped to ensure that the entire table is contained in at least one region. All the cells can work in parallel on their own regions to respond to the query:

'list all sets of four lines that form a parallelogram'.

Given the response to this query, the host or the cell that controls the searching process can predict the position of other sides of the table, and produce queries such as:

'list parallel lines with a given orientation',

to find the other sides of the table.

The query processing model requires that the cells operate asynchronously, as when responding to a query they may have to perform different amounts of computations and may produce variable amounts of outputs.

3.9. *Task-queue model*

For all of the preceding models, cells work together for a common task, whether they are tightly coupled (as in the pipeline model) or loosely coupled (as in the local computation or domain partition model). In contrast, the task queue model allows different cells to work on different tasks in one application. More precisely, a free cell can be dynamically assigned to execute any task in a task queue maintained by a cell or the host, as depicted by figure 18. Cells operate in a totally independent and asynchronous manner. Using this model, dynamic load balancing between cells is possible. The major concern in the implementation of this model is to minimize the latency between when a cell becomes free and when it starts doing a new task sent from the task queue. To use the cell effectively, this latency should not be larger than the time for the cell to execute the task.

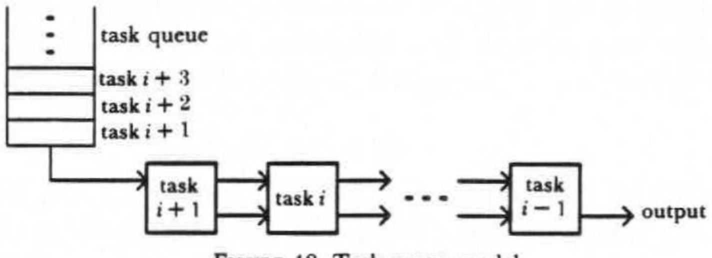


FIGURE 18. Task queue model.

The task queue model will be efficiently supported by the *i*Warp system. The on-chip message router at each cell will allow flexible communication between the cell (or host) that maintains the task queue and other cells. The communication will have low latency because of the available high bandwidth intercell communication channels.

4. CONCLUDING REMARKS

In this paper we have informally described a number of computational models for 1D processor arrays. Among these models, local computation, domain partition, pipeline, multifunction pipeline and ring are frequently used by the Warp users. We have found that in terms of these models various applications usages of the machine can be easily described. Also, we can discuss how architectural features support these models. For example, the 1D systolic array is natural for the pipeline model; and the routing hardware is needed for the efficient support of the divide-and-conquer or task queue model. Moreover, these models provide a way to classify programming tools for the automatic generation of parallel programs. For example, the Apply programming tool is to generate parallel code for the local computation model. There are several ongoing research projects at Carnegie Mellon intended to generate parallel programs for the other computational models such as the pipeline model.

For these reasons, we believe that computational models need to be made as explicit as possible in parallel computing. This paper represents an initial attempt to identify some of the models that seem to be important. Further work is needed to expand this set of models, and characterize them more precisely. Eventually, notations need to be developed to represent computational models.

Many of the ideas presented in this paper were inspired by work done under the Warp project at Carnegie Mellon. I am especially indebted to those members of the project, including F. Bitz, G. Gusciora, H. Ribas, P. S. Tseng and J. Webb, for their implementation of some of the applications examples discussed in this paper.

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, and in part by the Office of Naval Research under Contracts N00014-87-K-0385 and N00014-87-K-0533.

References

Aho, A., Hopcroft, J. E. & Ullman, J. D. 1975 *The design and analysis of computer algorithms*. Reading, Massachusetts: Addison-Wesley.

Annaratone, M., Arnould, E., Gross, T., Kung, H. T., Lam, M., Menzilcioglu, O. & Webb, J. A. 1987 The Warp computer: architecture, implementation and performance. *IEEE Trans. Comput.* **C-36** (12), 1523–1538.

Annaratone, M., Arnould, E., Kung, H. T. & Menzilcioglu, O. 1986*b* Using Warp as a supercomputer in signal processing. *Proc. ICASSP 86*. IEEE.

Baudet, G. & Stevenson, D. 1978 Optimal sorting algorithms for parallel computers. *IEEE Trans Comput.* **C-27(1)**, 84–87.

Bitz, F. & Kung, H. T. 1988 Path planning on the Warp computer: using a linear systolic array in dynamic programming. In *Proc. SPIE Symposium, Vol. 826, Advanced Algorithms and Architectures for Signal Processing II, August 1987*. Society of Photo-Optical Instrumentation Engineers. (Also *Int. comput. Math.* (In the press.).)

Gross, T. & Lam, M. 1986 Compilation for a high-performance systolic array. In *Proc. SIGPLAN 86 Symposium on Compiler Construction, June 1986*. ACM SIGPLAN.

Gross, T., Kung, H. T., Lam, M. & Webb, J. 1985 Warp as a machine for low-level vision. In *Proc. 1985 IEEE International Conference on Robotics and Automation*, March 1985.

Hamey, L. G. C., Webb, J. A. & Wu, I. C. 1987 Low-level vision on Warp and the Apply programming model. In *Parallel computation and computers for artificial intelligence* (ed. J. Kowalik). Kluwer Academic Publishers.

Heller, D. E. & Ipsen, I. C. F. 1982 Systolic networks for orthogonal equivalence transformations and their applications. In *Proc. Conf. Advanced Research in VLSI*, January 1982. Massachusetts Institute of Technology.

Hsu, F. H., Kung, H. T., Nishizawa, T. & Sussman, A. 1985 Architecture of the link and interconnection chip. In *Proc. 1985 Chapel Hill Conference on VLSI*, May 1985 (ed. H. Fuchs). The University of North Carolina, Computer Science Press, Inc.

Kung, H. T. 1979 Let's design algorithms for VLSI systems. In *Proc. Conf. on Very Large Scale Integration: Architecture, Design, Fabrication*, January 1979. California Institute of Technology.

Kung, H. T. 1982 Why systolic architectures? *Comput. Mag.* **15** (1), 37–46.

Kung, H. T. 1984 Systolic algorithms for the CMU Warp processor. In *Proc. 7th Int. Conf. on Pattern Recognition*. International Association for Pattern Recognition. (Revised version: *Systolic signal processing systems* (ed. E. E. Swartzlander, Jr.), chap. 3, pp. 73–95. New York: Marcel Dekker (1987).)

Kung, H. T. 1988 Systolic communication. In *Proc. Int. Conf. on Systolic Arrays*, May 1988. San Diego, California.

Kung, H. T. & Leiserson, C. E. 1979 Systolic arrays (for VLSI). In *Sparse matrix proceedings 1978* (ed. I. S. Duff & G. W. Stewart). Society for Industrial and Applied Mathematics.

Kung, H. T. & Webb, J. A. 1986 Mapping image processing operations onto a linear systolic machine. *Distrib. comput.* **1** (4), 246–257.

Lam, M. S. 1987 A systolic array optimizing compiler. Doctoral dissertation, Carnegie Mellon University.

Pomerleau, D. A., Gusciora, G. L., Touretzky, D. S. & Kung, H. T. 1988 Neural network simulation at warp speed: how we got 17 million connections per second. In *Proc. 1988 IEEE Int. Conf. on Neural Networks*, July 1988, San Diego, California.

Preparata, F. P. & Shamos, M. I. 1985 *Computational geometry: in introduction*. New York: Springer-Verlag.

Rosenfeld, A. 1977 Iterative methods in image analysis. In *Proc. IEEE Computer Society Conference on Pattern Recognition and Image Processing*. International Association for Pattern Recognition.

Rosenfeld, A., Hummel, R. A. & Zucker, S. W. 1976 Scene labelling by relaxation operations. *IEEE Trans. Systems, Man and Cybernetics* **SMC-6**, 420–433.

Rumelhart, D. E., Hinton, G. E. & Williams, R. J. 1986 Learning internal representations by error propagation. In *Parallel distributed processing: explorations in the microstructure of cognition. Vol. I. Foundations* (ed. D. E. Rumelhart & J. L. McClelland). Cambridge Massachusetts: Bradford Books/MIT Press.

Young, D. 1971 *Iterative solution of large linear systems*. New York: Academic Press.

Bibliography

Annaratone, Marcos, R Bitz, J. Deutch, Leonard G.C. Hamey, H. T. Kung, P.C. Maulik, H. Ribas, P. S. Tseng, and JonWebb. *Applications Experiences on Warp*. In Proceedings of the 1987 National Computer Conference. AFIPS, 1987.

Borkar, Shekhar, Robert Cohn, George W. 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, Jon Webb. *iWarp: An Integrated Solution to High-Speed Parallel Computing*. In Proceedings of the Supercomputing Conference, Orlando, FL, November 14 -15, 1988.

*Borkar, Shekhar, George W. Cox, Sha Gleason, Dick Hofsheier, Margie Levine, Greg Meece, Brian Moore, Craig Peterson, Linda Rankin, Jim Sutton, John Urbanski, Intel Corporation; Dan Hammerstrom, Department of Computer Science/Engineering Oregon Graduate Center; and Marco Annaratone, Clair Bono, Robert Cohn, Thomas Gross, H. T. Kung, Monica Lam, P.C. Maulik, John Pieper, P.S. Tseng, Jon Webb. Carnegie Mellon University, Department of Computer Science. *iWarp Macroarchitecture Specification.*

Cohn, Robert, Thomas Gross, Monica Lam, and P.S Tseng. Carnegie Mellon University, Department of Computer Science. *Architecture and Compiler Tradeoffs for a Long Instruction Word Microprocessor*. In proceedings from the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III) Boston, Massachusetts, April 3 - 6, 1989.

Gross, Thomas, Monica Lam, James Reinders. Carnegie Mellon University, Department of Computer Science. *Programming Warp in W2*. 1988.

Hamey, Leonard G.C., Jon A. Webb, I-Chen Wu. Carnegie Mellon University, Department of Computer Science. *Apply, A Programming Language for Low-Level Vison on Diverse Parallel Architectures.*

Kung, H.T. Carnegie Mellon University, Department of Computer Science. *Network-Based Multicomputers: Redefining High Performance Computing in the 1990s.* In proceedings of Decennial Caltech Conference on VLSI Pasadena, California, March 20 - 22, 1989.

Lam, Monica. *Software Pipelining: An Effective Scheduling Technique for VLIW Machines.* In proceedings of ACM Sigplan 1988 Conference on Programming Language Design and Implementation.

Webb, Jon, and I–Chen Wu. Carnegie Mellon University, Department of Computer Science, *A User's Guide to Apply*. 1988.

*** This is a proprietary document, and requires a non–disclosure agreement. Contact Intel iWarp Marketing and Sales Department for more information.**

Glossary

Apply. A special purpose high level language for image processing applications.

cardcage assembly. See *iWarp Cardcage Assembly.*

cell. See *iWarp cell.*

Communication Agent. The Communication Agent controls the pathways that provide inter–cell communications throughout the iWarp System.

component. See *iWarp Component.*

Computation Agent. One of two functional agents in the iWarp component, this agent executes the applications and service process. It consists of the Instruction Sequencing Unit, the Program Store Unit, the Integer Logic Unit, the Floating-point Unit, the Register File Unit, the Local Memory Unit, and the Streaming/Spooling Unit.

compute and access instruction format. A 96–bit instruction format for high performance floating-point computations. This format exhibits the benefits of a long instruction word architecture.

External Interface Board. A small circuit board that connects to the backplane of an iWarp Cardcage Assembly, allowing connection to other cardcage assemblies or external devices.

Floating-point Unit. The Floating-point Adder and Floating-point Multiplier make up the Floating-point Unit.

host. The program development host to which the iWarp array is attached. In certain system configurations, other software such as the file server or the sequential component of a user's application may also run on the host.

general-purpose instruction format. A 32–bit instruction format for general-purpose requirements. This format has many of the features of a reduced instruction set computer architecture.

Integer Logic Unit. This unit performs all integer, ordinal, and logical operations. The Integer Logic Unit is one of the functional units of the Computation Agent.

Instruction Sequencing Unit. This unit provides the instruction execution sequencing and instruction decoding for all other functional units. The Instruction Sequencing Unit is one of the functional units of the Computation Agent.

iWarp. Intel's Integrated Warp processor, a joint venture between Intel and Carnegie Mellon University.

iWarp Cardcage Assembly. A 19-inch rack-mountable chassis that combines a 17-slot cardcage, backplane, external device interface, power supply, and fans. The iWarp Cardcage Assembly can hold up to 64 iWarp cells.

iWarp Cell. An iWarp Cell consists of an iWarp Component and its local memory.

iWarp Component. A component made up of a Communication Agent to handle networking between cells and a Computation Agent to provide floating-point, integer, and logical operations, as well as instruction sequencing control.

local memory. The iWarp component's associated memory. The local memory combines with the iWarp Component to form an iWarp cell.

Local Memory Unit. This unit provides the interface between the iWarp Component and its local memory.

Memory Expansion Module. A small circuit board that allows additional memory to be added to an iWarp Single or Quad Cell Board.

message passing model. A course-grain communication model in which the unit of communication is a complete message.

Program Store Unit. The Program Store Unit fetches instructions from local memory and provides them to the Instruction Sequencing Unit.

Quad Cell Board. An iWarp board that contains four iWarp cells and four banks of local memory.

Register File Unit. The Register File Unit is the central element of the iWarp Component architecture. It routes data between all of the functional elements of the iWarp Component.

Single Cell Board. An iWarp board that contains one iWarp cell and four banks of local memory.

Streaming/Spooling Unit. The Streaming/Spooling Unit removes data from the pathways to memory and retrieves data to the computation units. The process of streaming and spooling helps relieve pathway congestion.

systolic computing. Data flows, or is pumped, through an array of processors as it is used simultaneously in cell computations.

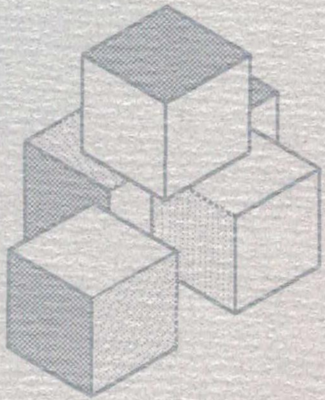
systolic model. A fine-grain communication model in which the unit of communication can be as small as a single word in a message.

Warp. The predecessor of iWarp, developed by Professor H.T. Kung and his associates at Carnegie Mellon University and General Electric.

Index

Intel Corporation
iWarp Marketing, Mailstop: CO4-Ø5
5200 NE Elam Young Pkwy.
Hillsboro, OR 97124-6497

Phone: (503) 629-6300



Order#281005