The Warp Computer: Architecture, Implementation, and Performance

MARCO ANNARATONE, EMMANUEL ARNOULD, THOMAS GROSS, MEMBER, IEEE, H. T. KUNG, MONICA LAM, ONAT MENZILCIOGLU, AND JON A. WEBB

Abstract—**The Warp machine is a systolic array computer of linearly connected cells, each of which is a programmable processor capable of performing 10 million floating-point operations per second (10 MFLOPS). A typical Warp array includes ten cells, thus having a peak computation rate of 100 MFLOPS. The Warp array can be extended to include more cells to accommodate applications capable of using the increased computational bandwidth. Warp is integrated as an attached processor into a Unix host system. Programs for Warp are written in a high-level language supported by an optimizing compiler.**

The first ten-cell prototype was completed in February 1986; delivery of production machines started in April 1987. Extensive experimentation with both the prototype and production machines has demonstrated that the Warp architecture is effective in the application domain of robot navigation as well as in other fields such as signal processing, scientific computation, and computer vision research. For these applications, Warp is typically several hundred times faster than a VAX 11/780 class computer.

This paper describes the architecture, implementation, and performance of the Warp machine. Each major architectural decision is discussed and evaluated with system, software, and application considerations. The programming model and tools developed for the machine are also described. The paper concludes with performance data for a large number of applications.

Index Terms—**Computer system implementation, computer vision, image processing, optimizing compiler, parallel processors, performance evaluation, pipelined processor, scientific computing, signal processing, systolic array, vision research.**

I. Introduction

The Warp machine is a high-performance systolic array computer designed for computation-intensive applications. In a typical configuration, Warp consists of a linear systolic array of ten identical cells, each of which is a 10 MFLOPS programmable processor. Thus, a system in this configuration has a peak performance of 100 MFLOPS.

The Warp machine is an attached processor to a general purpose host running the Unix operating system. Warp can be accessed by a procedure call on the host, or through an interactive, programmable command interpreter called the Warp shell [8]. A high-level language called W2 is used to program Warp; the language is supported by an optimizing compiler [12], [23].

The Warp project started in 1984. A two-cell system was completed in June 1985 at Carnegie Mellon. Construction of two identical ten-cell prototype machines was contracted to two industrial partners, GE and Honeywell. These prototypes were built from off-the-shelf parts on wire-wrapped boards. The first prototype machine was delivered by GE in February 1986, and the Honeywell machine arrived at Carnegie Mellon in June 1986. For a period of about a year starting from early 1986, these two prototype machines were used on a daily basis at Carnegie Mellon.

We have implemented application programs in many areas, including low-level vision for robot vehicle navigation, image and signal processing, scientific computing, magnetic resonance imagery (MRI), image processing, radar and sonar simulation, and graph algorithms [3], [4]. In addition, we have developed an image processing library of over 100 routines [17]. Our experience has shown that Warp is effective in these applications; Warp is typically several hundred times faster than a VAX 11/780 class computer.

Encouraged by the performance of the prototype machines, we have revised the Warp architecture for reimplementation on printed circuit (PC) boards to allow faster and more efficient production. The revision also incorporated several architectural improvements. The production Warp machine is referred to as the PC Warp in this paper. The PC Warp is manufactured by GE, and is available at about $350 000 per machine. The first PC Warp machine was delivered by GE in April 1987 to Carnegie Mellon.

This paper describes the architecture of the Warp machine, the rationale of the design and the implementation, and performance measurements for a variety of applications. The organization of the paper is as follows. We first present an overview of the system. We then describe how the overall organization of the array allows us to use the cells efficiently. Next we focus on the cell architecture: we discuss each feature in detail, explaining the design and the evolution of the feature. We conclude this section on the cell with a discussion of hardware implementation issues and metrics. We then describe the architecture of the host system. To give the reader some idea of how the machine can be programmed, we describe the W2 programming language, and some general methods of partitioning a program onto a processor array that has worked well for Warp. To evaluate the Warp machine

Manuscript received February 2, 1987; revised June 15, 1987 and July 27, 1987. This work was supported in part by the Defense Advanced Research Projects Agency (DOD) monitored by the Space and Naval Warfare Systems Command under Contract N00039-85-C-0134, and in part by the Office of Naval Research under Contracts N00014-87-K-0385 and N00014-87-K-0533. Warp is a service mark of Carnegie Mellon. UNIX is a trademark of AT&T Bell Laboratories. Sun-3 is a trademark of Sun Microsystems.

M. Annaratone was with the Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. He is now with the Institute for Integrated Systems, ETH Zentrum, 8092 Zurich, Switzerland.

E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Menzilcioglu, and J. A. Webb are with the Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

IEEE Log Number 8717037.

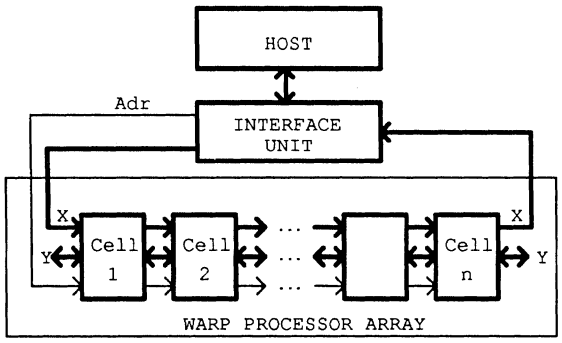


Fig. 1. Warp system overview.

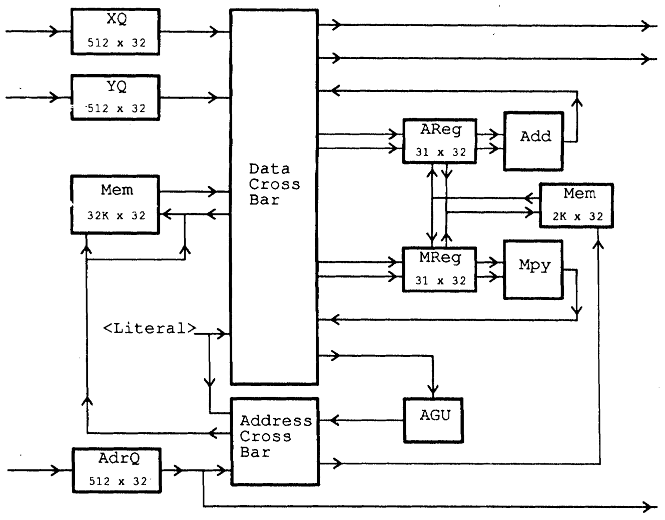


Fig. 2. Warp cell data path.

architecture, we present performance data for a variety of applications on Warp, and a comparison of the architecture of Warp to other parallel machines.

II. Warp System Overview

There are three major components in the system—the Warp processor array (*Warp array*), the interface unit (*IU*), and the *host,* as depicted in Fig. 1. The Warp array performs the computation-intensive routines such as low-level vision routines or matrix operations. The IU handles the input/output between the array and the host, and can generate addresses (Adr) and control signals for the Warp array. The host supplies data to and receives results from the array. In addition, it executes those parts of the application programs which are not mapped onto the Warp array. For example, the host may perform decision-making processes in robot navigation or evaluate convergence criteria in iterative methods for solving systems of linear equations.

The Warp array is a linear systolic array with identical cells, called Warp cells, as shown in Fig. 1. Data flow through the array on two communication channels (X and Y). Those addresses for cells' local memories and control signals that are generated by the IU propagate down the Adr channel. The direction of the Y channel is statically configurable. This feature is used, for example, in algorithms that require accumulated results in the last cell to be sent back to the other cells (e.g., in back-solvers), or require local exchange of data between adjacent cells (e.g., in some implementations of numerical relaxation methods).

Each Warp cell is implemented as a programmable horizontal microengine, with its own microsequencer and program memory for 8K instructions. The Warp cell data path, as depicted in Fig. 2, consists of a 32-bit floating-point multiplier (Mpy), a 32-bit floating-point adder (Add), two local memory banks for resident and temporary data (Mem), a queue for each intercell communication channel (XQ, YQ, and AdrQ), and a register file to buffer data for each floating-point unit (AReg and MReg). All these components are connected through a crossbar. Addresses for memory access can be computed locally by the address generation unit (AGU), or taken from the address queue (AdrQ).

The Warp cell data path is similar to the data path of the Floating Point Systems AP-120B/FPS-164 line of processors [9], which are also used as attached processors. Both the Warp cell and any of these FPS processors contain two floating-point units, memory and an address generator, and are oriented towards scientific computing and signal processing. In both cases, wide instruction words are used for a direct encoding of the hardware resources, and software is used to manage the parallelism (that is, to detect parallelism in the application code, to use the multiple functional units, and to pipeline instructions). The Warp cell differs from these earlier processors in two key aspects: the full crossbar of the Warp cell provides a higher intracell bandwidth, and the X and Y channels with their associated queues provide a high intercell bandwidth, which is unique to the Warp array architecture.

The host consists of a Sun-3 workstation that serves as the *master* controller of the Warp machine, and a VME-based multiprocessor *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. It also transfers data to and from the Warp array and performs operations on the data when necessary, with low operating system overhead.

Both the Warp cell and IU use off-the-shelf, TTL-compatible parts, and are each implemented on a 15×17 in^2 board. The entire Warp machine, with the exception of the Sun-3, is housed in a single 19 in rack, which also contains power supplies and cooling fans. The machine typically consumes about 1800 W.

III. Warp Array Architecture

In the Warp machine, parallelism exists at both the array and cell levels. This section discusses how the Warp architecture is designed to allow efficient use of the array level parallelism. Architectural features to support the cell level parallelism are described in the next section.

The key features in the architecture that support the array level parallelism are simple topology of a linear array, powerful cells with local program control, large data memory for each cell, and high intercell communication bandwidth. These features support several program partitioning methods important to many applications [21], [22]. More details on the partitioning methods are given in Section VII-B, and a sample of applications using these methods is listed in Section VIII.

A linear array is easier for a programmer to use than higher dimensional arrays. Many algorithms in scientific computing and signal processing have been developed for linear arrays [18]. Our experience of using Warp for low-level vision has also shown that a linear organization is suitable in the vision domain as well. A linear array is easy to implement in hardware, and demands a low external I/O bandwidth since only the two end cells communicate with the outside world. Moreover, a linear array consisting of powerful, programmable processors with large local memories can efficiently simulate other interconnection topologies. For example, a single Warp cell can be time multiplexed to perform the function of a column of cells, so that the linear Warp array can implement a two-dimensional systolic array.

The Warp array can be used for both fine-grain and large-grain parallelism. It is efficient for fine-grain parallelism needed for systolic processing, because of its high intercell bandwidth. The I/O bandwidth of each cell is higher than that of other processors with similar computational power. Each cell can transfer 20 million 32-bit words (80 Mbytes) per second to and from its neighboring cells, in addition to 20 million 16-bit addresses. This high intercell communication bandwidth permits efficient transfers of large volumes of intermediate data between neighboring cells.

The Warp array is efficient for large-gain parallelism because it is composed of powerful cells. Each cell is capable of operating independently; it has its own program sequencer and program memory of 8K instructions. Moreover, each cell has 32K words of local data memory, which is large for systolic array designs. For a given I/O bandwidth, a larger data memory can sustain a higher computation bandwidth for some algorithms [20].

Systolic arrays are known to be effective for local operations, in which each output depends only on a small corresponding area of the input. The Warp array's large memory size and its high intercell I/O bandwidth enable it to perform *global* operations in which each output depends on any or a large portion of the input [21]. The ability of performing global operations as well significantly broadens the applicability of the machine. Examples of global operations are fast Fourier transform (FFT), image component labeling, Hough transform, image warping, and matrix computations such as LU decomposition or singular value decomposition (SVD).

Because each Warp cell has its own sequencer and program memory, the cells in the array can execute different programs at the same time. We call computation where all cells execute the same program *homogeneous,* and *heterogeneous* otherwise. Heterogeneous computing is useful for some applications. For example, the end cells may operate differently from other cells to deal with boundary conditions. Or, in a multifunction pipeline [13], different sections of the array perform different functions, with the output of one section feeding into the next as input.

IV. Warp Cell Architecture

This section describes the design and the evolution of the architectural features of the cell. Some of these features were significantly revised when we reimplemented Warp on PC boards. For the wire-wrapped prototype, we omitted some architectural features that are difficult to implement and are not necessary for a substantial fraction of application programs [1]. This simplification in the design permitted us to gain useful experience in a relatively short time. With the experience of constructing and using the prototype, we were able to improve the architecture and expand the application domain of the production machine.

A. Intercell Communication

Each cell communicates with its left and right neighbors through point-to-point links, two for data and one for addresses. A queue with a depth of 512 words is associated with each link (XQ, YQ and AdrQ in Fig. 2) and is placed in the data path of the input cell. The size of the queue is just large enough to buffer one or two scan-lines of an image, which is typically of size 512 $\times$ 512 or 256 $\times$ 256. The ability to buffer a complete scan line is important for the efficient implementation of some algorithms such as two-dimensional convolutions [19]. Words in the queues are 34 bits wide; along with each 32-bit data word, the sender transmits a 2-bit control signal that can be tested by the receiver.

Flow control for the communication channels is implemented in hardware. When a cell tries to read from an empty queue, it is blocked until a data item arrives. Similarly, when a cell tries to write to a full queue of a neighboring cell, the writing cell is blocked until some data are removed from the full queue. The blocking of a cell is transparent to the program; the state of all the computational units on the data path freezes for the duration the cell is blocked. Only the cell that tries to read from an empty queue or to deposit a data item into a full queue is blocked. All other cells in the array continue to operate normally. The data queues of a blocked cell are still able to accept input; otherwise, a cell blocked on an empty queue will never become unblocked.

The implementation of run-time flow control by hardware has two implications. First, we need two clock generators—one for the computational units whose states freeze whenever a cell is blocked, and one for the queues. Second, since a cell can receive data from either of its two neighbors, it can block as a result of the status of the queues in either neighbor, as well as its own. This dependence on other cells adds serious timing constraints to the design since clock control signals have to cross board boundaries. This complexity will be further discussed in Section V.

The intercell communication mechanism is the most revised feature on the cell; it has evolved from primitive programmable delay elements to queues without any flow control hardware, and finally to the run-time flow-controlled queues. In the following, we step through the different design changes.

1) Programmable Delay: In an early design, the input buffer on each communication channel of a cell was a programmable delay element. In a programmable delay element, data are latched in every cycle and they emerge at the output port a constant number of cycles later. This structure is found in many systolic algorithm designs to synchronize or delay one data stream with respect to another. However, programmable high-performance processors like the Warp cells require a more flexible buffering mechanism. Warp programs do not usually produce one data item every cycle; a clocking discipline that reads and writes one item per cycle would be too restrictive. Furthermore, a constant delay through the buffer means that the timing of data generation must match exactly that of data consumption. Therefore, the programmable delays were replaced by queues to remove the tight coupling between the communicating cells.

2) Flow Control: Queues allow the receiver and sender to run at their own speeds provided that the receiver does not read past the end of the queue and the sender does not overflow the queues. There are two different flow control disciplines, run-time and compile-time flow-control. As discussed above, hardware support for run-time flow control can be difficult to design, implement, and debug. Alternatively, for a substantial set of problems in our application domain, compile-time flow control can be implemented by generating code that requires no run-time support. Therefore, we elected not to support run-time flow control in the prototype. This decision permitted us to accelerate the implementation and experimentation cycle. Run-time flow control is provided in the production machine, so as to widen the application domain of the machine.

Compile-time flow control can be provided for all programs where data only flow in one direction through the array and where the control flow of the programs is not data dependent. Data dependent control flow and two-way data flow can also be allowed for programs satisfying some restrictions [6]. Compile-time flow control is implemented by skewing the computation of the cells so that no receiving cell reads from a queue before the corresponding sending cell writes to it. For example, suppose two adjacent cells each execute the following program:

```
dequeue (X);
output (X)  ;
dequeue (X);
compute    ;
compute    ;
output (X)  ;
```

In this program, the first cell removes a data item from the *X* queue (**dequeue (X)**) and sends it to the second cell on **X** (**output (X)**). The first cell then removes a second item, and forwards the result to the second cell after two cycles of computation. For this program, the second cell needs to be delayed by three cycles to ensure that the **dequeue** of the second cell never overtakes the corresponding **output** of the first cell, and the compiler will insert the necessary **nops**, as shown in Fig. 3.

Run-time flow control expands the application domain of the machine and often allows the compiler to produce more efficient code; therefore, it is provided in the production machine. Without run-time flow control, WHILE loops and FOR loops and computed loop bounds on the cells cannot be implemented. That is, only loops with compile-time constant bounds can be supported. This restriction limits the class of

```
First cell                                  Second cell

dequeue(X);                                 nop       ;
output(X)  ;                                nop       ;
dequeue(X);                                 nop       ;
compute    ;                                dequeue(X);
compute    ;                                output(X)  ;
output(X)  ;                                dequeue(X);
                                            compute    ;
                                            compute    ;
                                            output(X)  ;
```

Fig. 3. Compile-time flow control.

programs executable on the machine. Moreover, many programs for the prototype machines can be made more efficient and easier to write by replacing the FOR loops with WHILE loops. For example, instead of executing a fixed number of iterations to guarantee convergence, the iteration can be stopped as soon as the termination condition is met. The compiler can produce more efficient code since compile-time flow control relies on delaying the receiving cell sufficiently to guarantee correct behavior, but this delay is not necessarily the minimum delay needed. Run-time flow control will dynamically find the minimum bound.

3) Input Control: In the current design, latching of data into a cell's queue is controlled by the sender, rather than by the receiver. As a cell sends data to its neighbor, it also signals the receiving cell's input queue to accept the data.

In our first two-cell prototype machine, input data were latched under the microinstruction control of the receiving cell. This implied that intercell communication required close cooperation between the sender and the receiver; the sender presented its data on the communication channel, and in the same clock cycle the receiver latched in the input. This design was obviously not adequate if flow control was supported at run time; in fact, we discovered that it was not adequate even if flow control was provided at compile time. The tight coupling between the sender and the receiver greatly increased the code size of the programs. The problem was corrected in subsequent implementations by adopting the design we currently have, that is, the sender provides the signal to the receiver's queue to latch in the input data.

In the above discussion of the example of Fig. 3, it was assumed that the control for the second cell to latch in input was sent with the output data by the first cell. If the second cell were to provide the input control signals, we would need to add an **input** operation in its microprogram for every **output** operation of the first cell, at exactly the cycle the operation takes place. Doing so, we obtain the following program for the second cell:

```
             nop         ;
input (X)                ;
             nop         ;
             dequeue (X);
             output (X)  ;
input (X),   dequeue (X);
             compute     ;
             compute     ;
             output (X)  ;
```

Each line in the program is a microinstruction; the first column

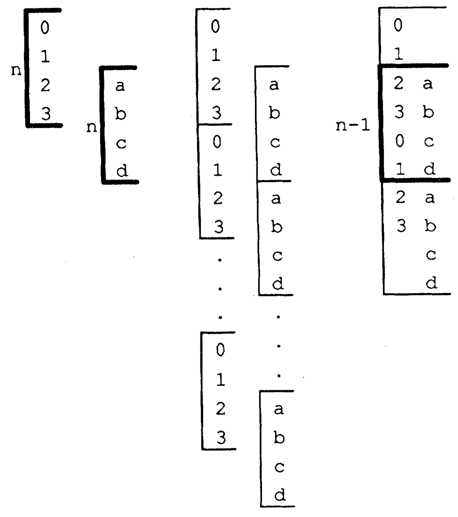


Fig. 4. Merging equal-length loops with an offset. (a) Original loops. (b) Execution trace. (c) Merged loop.

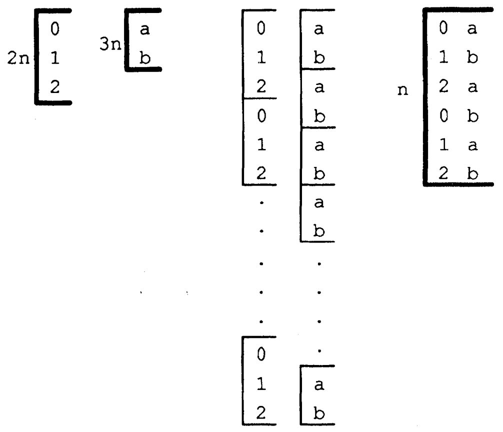


Fig. 5. Merging loops with different lengths. (a) Original loops. (b) Execution trace. (c) Merged loop.

contains the **Input** operations to match the **Output** operations of the first cell, and the second column contains the original program.

Since the input sequence follows the control flow of the sender, each cell is logically executing two processes: the input process, and the original computation process of its own. These two processes must be merged into one since there is only one sequencer on each cell. If the programs on communicating cells are different, the input process and the cell's own computation process are different. Even if the cell programs are identical, the cell's computation process may need to be delayed with respect to the input process because of compile-time flow control as described above. As a result, we may need to merge control constructs from different parts of the program. Merging two equal-length loops, with an offset between their initiation times, requires loop unrolling and can result in a threefold increase in code length. Fig. 4 illustrates this increase in code length when merging two identical loops of n iterations. Numbers represent operations of the input process, and letters represent the computation process. If two iterative statements of different lengths are overlapped, then the resulting code size can be of the order of the least common multiple of their lengths. For example, in Fig. 5, a two-instruction loop of $3n$ iterations is merged with a three-instruction loop of $2n$ iterations. Since 6 is the minimum number of cycles before the combined sequence of operations repeats itself, the resulting merged program is a six-instruction loop of n iterations.

4) Randomly Accessible Queues: The queues in all the prototype machines are implemented with RAM chips, with hardware queue pointers. Furthermore, there was a feedback path from the data crossbar back to the queues, because we intended to use the queues as local storage elements as well [1]. Since the pointers must be changed when the queue is accessed randomly, and there is only a single pair of queue pointers, it is impossible to multiplex the use of the buffer as a communication queue and its use as a local storage element. Therefore, the queues in the production machine are now implemented by a FIFO chip. This implementation allows us to increase the queue size from 128 to 512 words, with board space left over for other improvements as well.

5) Queue Size: The size of the queues is an important factor in the efficiency of the array. Queues buffer the input for a cell and relax the coupling of execution in communicating cells. Although the average communication rate between two communicating cells must balance, a larger buffer allows the cells to receive and send data in bursts at different times.

The long queues allow the compiler to adopt a simple code optimization strategy [23]. The throughput for a unidirectional array is maximized by simply optimizing the individual cell programs provided that sufficient buffering is available between each pair of adjacent cells. In addition, some algorithms, such as two-dimensional convolution mentioned above, require large buffers between cells. If the queues are not large enough, a program must explicitly implement buffers in local memory.

B. Control Path

Each Warp cell has its own local program memory and sequencer. This is a good architectural design even if the cells all execute the same program, as in the case of the prototype Warp machine. The reason is that it is difficult to broadcast the microinstruction words to all the cells, or to propagate them from cell to cell, since the instructions contain a large number of bits. Moreover, even if the cells execute the same program, the computations of the cells are often skewed so that each cell is delayed with respect to its neighboring cell. This skewed computation model is easily implemented with local program control. The local sequencer also supports conditional branching efficiently. In SIMD machines, branching is achieved by masking. The execution time is equivalent to the sum of the execution time of the then-clause and the else-clause of a branch. With local program control, different cells may follow different branches of a conditional statement depending on their individual data; the execution time is the execution time of the clause taken.

The Warp cell is horizontally microcoded. Each component in the data path is controlled by a dedicated field; this orthogonal organization of the microinstruction word makes scheduling easier since there is no interference in the schedule of different components.

C. Data Path

1) Floating-Point Units: Each Warp cell has two floating-point units, one multiplier and one adder, implemented with commercially available floating-point chips [35]. These floating-point chips depend on extensive pipelining to achieve high performance. Both the adder and multiplier have five-stage pipelines. General purpose computation is difficult to implement efficiently on deeply pipelined machines because data-dependent branching is common. There is less data dependency in numerical or computer vision programs, and we developed scheduling techniques that use the pipelining efficiently. Performance results are reported in Section VIII.

2) Crossbar: Experience with the Programmable Systolic Chip showed that the internal data bandwidth is often the bottleneck of a systolic cell [11]. In the Warp cell, the two floating-point units can consume up to four data items and generate two results per cycle. Several data storage blocks interconnected with a crossbar support this high data processing rate. There are six input and eight output ports connected to the crossbar switch; up to six data items can be transferred in a single cycle, and an output port can receive any data item. The use of the crossbar also makes compilation easier when compared to a bus-based system since conflicts on the use of one or more shared buses can complicate scheduling tremendously.

Custom chip designs that combine the functionality of the crossbar interconnection and data buffers have been proposed [16], [28]. In the interconnection chip designed for polycyclic architectures [28], a "queue" is associated with each cross point of the crossbar. In these storage blocks, data are always written at the end of the queue; however, data can be read, or removed, from any location. The queues are compacted automatically whenever data are removed. The main advantage of this design is that an optimal code schedule can be readily derived for a class of inner loops [27]. In the Warp cell architecture, we chose to use a conventional crossbar with data buffers only for its outputs (the AReg and MReg register files in Fig. 2), because of the lower hardware cost. Near-optimal schedules can be found cheaply using heuristics [23].

3) Data Storage Blocks: As depicted by Fig. 2, the local memory hierarchy includes a local data memory, a register file for the integer unit (AGU), two register files (one for each floating-point unit), and a backup data memory. Addresses for both data memories come from the address crossbar. The local data memory can store 32K words, and can be both read and written every (200 ns) cycle. The capacity of the register file in the AGU unit is 64 words. The register files for the floating-point units each hold 31 usable words of data. (The register file is written to in every cycle so that one word is used as a sink for those cycles without useful write operations.) They are five-ported data buffers and each can accept two data items from the crossbar and deliver two operands to the functional units every cycle. The additional ports are used for connecting the register files to the backup memory. This backup memory contains 2K words and is used to hold all scalars, floating-point constants, and small arrays. The addition of the backup memory increases memory bandwidth and improves throughput for those programs operating mainly on local data.

4) Address Generation: As shown in Fig. 2, each cell contains an integer unit (AGU) that is used predominantly as a local address generation unit. The AGU is a self-contained integer ALU with 64 registers. It can compute up to two addresses per cycle (one read address and one write address).

The local address generator on the cell is one of the enhancements that distinguish the PC Warp machine from the prototype. In the prototype, data independent addresses were generated on the IU and propagated down the cells. Data dependent addresses were computed locally on each cell using the floating-point units. The IU of the prototype had the additional task of generating the loop termination signals for the cells. These signals were propagated along the Adr channel to the cells in the Warp array.

There was not enough space on the wire-wrapped board to include local address generation capability on each Warp cell. Including an AGU requires board space not only for the AGU itself, but also for its environment and the bits in the instruction word for controlling it. An AGU was area expensive at the time the prototype was designed, due to the lack of VLSI parts for the AGU functions. The address generation unit in the prototype IU uses AMD2901 parts which contain 16 registers. Since this number of registers is too small to generate complicated addressing patterns quickly, the ALU is backed up by a table that holds up to 16K precomputed addresses. This table is too large to replicate on all the cells. The address generation unit on the PC Warp cells is a new VLSI component (IDT-49C402), which combines the 64-word register file and ALU on a single chip. The large number of registers makes the backup table unnecessary for most addressing patterns, so that the AGU is much smaller and can be replicated on each cell of the production machine.

The prototype was designed for applications where all cells execute the same program with data independent loop bounds. However, not all such programs could be supported due to the size of the address queue. In the pipelining mode, where the cells implement different stages of a computation pipeline, a cell does not start executing until the preceding cell is finished with the first set of input data. The size of the address queue must at least equal the number of addresses and control signals used in the computation of the data set. Therefore, the size of the address queues limits the number of addresses buffered, and thus the grain size of parallelism.

For the production machine, each cell contains an AGU and can generate addresses and loop control signals efficiently. This improvement allows the compiler to support a much larger class of application. We have preserved the address generator and address bank on the IU (and the associated Adr channel, as shown in Fig. 1). Therefore, the IU can still support those homogeneous computations that demand a small set of complicated addressing patterns that can be conveniently stored in the address bank.

V. Warp Cell and IU Implementation

The Warp array architecture operates on 32-bit data. All data channels in the Warp array, including the internal data path of the cell, are implemented as 16-bit wide channels operating at 100 ns. There are two reasons for choosing a 16-

TABLE I
IMPLEMENTATION METRICS FOR WARP CELL

Block in Warp cell	Chip count	Area contribution (Percent)
Queues	22	9
Crossbar	32	11
Processing elements and registers	12	10
Data memory	31	9
Local address generator	13	6
Microengine	90	35
Other	55	20
Total for Warp cell	255	100

TABLE II
IMPLEMENTATION METRICS FOR IU

Block in IU	Chip count	Area contribution (Percent)
Data-converter	44	19
Address generator	45	19
Clock and host interface	101	31
Microengine	49	20
Other	25	11
Total for IU	264	100

bit time-multiplexed implementation. First, a 32-bit wide hardware path would not allow implementing one cell per board. Second, the 200 ns cycle time dictated by the Weitek floating-point chips (at the time of design) allows the rest of the data path to be time multiplexed. This would not have been possible if the cycle time of the floating-point chips were under 160 ns. The microengine operates at 100 ns and supports high and low cycle operations of the data path separately.

All cells in the array are driven from a global 20 MHz clock generated by the IU. To allow each cell to block individually, a cell must have control over the use of the global clock signals. Each cell monitors two concurrent processes: the input data flow (*I* process) and the output data flow (*O* process). If the input data queue is empty, the *I* process flow must be suspended before the next read from the queue. Symmetrically, the *O* process is stopped before the next write whenever the input queue of the neighboring cell is full. Stopping the *I* or *O* process pauses all computation and output activity, but the cell continues to accept input. There is only a small amount of time available between detection of the queue full/empty status and blocking the read/write operation. Since the cycle time is only 100 ns, this tight timing led to race conditions in an early design. This problem has been solved by duplicating on each cell the status of the I/O processes of the neighboring cells. In this way, a cell can anticipate a queue full/empty condition and react within a clock cycle.

A large portion of the internal cell hardware can be monitored and tested using built-in serial diagnostic chains under control of the IU. The serial chains are also used to download the Warp cell programs. Identical programs can be downloaded to all cells at a rate of 100 μs per instruction from the workstation and about 67 μs per instruction from the external host. Starting up a program takes about 5 ms.

The Warp cell consists of six main blocks: input queues, crossbar, processing elements, data memory, address generator, and microengine. Table I presents the contribution of these blocks to the implementation of the Warp cell. The microengine includes the program memory (8K instruction words of 272 bits, including parity). The Warp cell consumes 94 W (typical) and 136 W (maximum).

The IU handles data input/output between the host and the Warp array. The host–IU interface is streamlined by implementing a 32-bit wide interface, even though the Warp array has only 16-bit wide internal data paths. This arrangement is preferred because data transfers between the host and IU are slower than the transfers between IU and the array. Data transfers between the host and IU can be controlled by interrupts; in this case, the IU behaves like a slave device. The IU can also convert packed 8-bit integers transferred from the host into 32-bit floating-point numbers for the Warp array, and vice versa.

The IU is controlled by a 96-bit wide programmable microengine, which is similar to the Warp cell controller in programmability. The IU has several control registers that are mapped into the host address space; the host can control the IU and hence the Warp array by setting these registers. The IU has a power consumption of 82 W (typical) and 123 W (maximum). Table II presents implementation metrics for the IU.

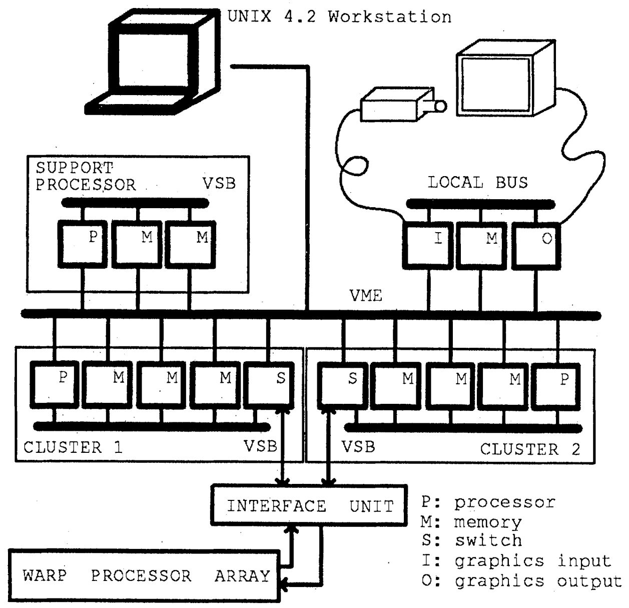


Fig. 6. Host of the Warp machine.

VI. HOST SYSTEM

The Warp host controls the Warp array and other peripherals, supports fast data transfer rates to and from the Warp array, and also runs application code that cannot easily be mapped on the array. An overview of the host is presented in Fig. 6. The host is partitioned into a standard workstation (the master) and an external host. The workstation provides a Unix programming environment to the user, and also controls the external host. The external host consists of two *cluster processors,* a subsystem called *support processor,* and some graphics devices.

Control of the external host is strictly centralized: the workstation, the master processor, issues commands to the cluster and support processors through message buffers local to each of these processors. The two clusters work in parallel, each handling a unidirectional flow of data to or from the Warp processor through the IU. The two clusters can exchange their roles in sending or receiving data for different phases of a computation, in a ping-pong fashion. An arbitration mechanism transparent to the user has been implemented to prohibit simultaneous writing or reading to the Warp array when the clusters switch roles. The support processor controls peripheral I/O devices and handles floating-point exceptions and other interrupt signals from the Warp array. These interrupts are serviced by the support processor, rather than by the master processor, to minimize interrupt response time. After servicing the interrupt, the support processor notifies the master processor.

The external host is built around a VME bus. The two clusters and the support processor each consist of a standalone MC68020 microprocessor (P) and a dual-ported memory (M), which can be accessed either via a local bus or via the global VME bus. The local bus is a VSB bus in the production machine and a VMX32 bus for the prototype; the major improvements of VSB over VMX32 are better support for arbitration and the addition of DMA-type accesses. Each cluster has a *switch* board (S) for sending and receiving data to and from the Warp array, through the IU. The switch also has a VME interface, used by the master processor to start, stop, and control the Warp array. The VME bus of the master processor inside the workstation is connected to the VME bus of the external host via a bus-coupler (bus repeater). While the prototype Warp used a commercial bus-coupler, the PC Warp employs a custom-designed device. The difference between the two is that the custom-designed bus repeater decouples the external host VME bus from the Sun-3 VME bus: intrabus transfers can occur concurrently on both buses.

There are three memory banks inside each cluster processor to support concurrent memory accesses. For example, the first memory bank may be receiving a new set of data from an I/O device, while data in the second bank are transferred to the Warp array, and the third contains the cluster program code.

Presently, the memory of the external host is built out of 1 Mbyte memory boards; including the 3 Mbytes of memory on the processor boards, the total memory capacity of the external host is 11 Mybtes. An expansion of up to 59 Mbytes is possible by populating all the 14 available slots of the VME card cage with 4 Mbyte memory boards. Large data structures can be stored in these memories where they will not be swapped out by the operating system. This is important for consistent performance in real-time applications. The external host can also support special devices such as frame buffers and high-speed disks. This allows the programmer to transfer data directly between Warp and other devices.

Except for the switch, all boards in the external host are off-the-shelf components. The industry standard boards allow us to take advantage of commercial processors, I/O boards, memory, and software. They also make the host an open system to which it is relatively easy to add new devices and interfaces to other computers. Moreover, standard boards provide a growth path for future system improvements with a minimal investment of time and resources. During the transition from prototype to production machine, faster processor boards (from 12 to 16 MHz) and larger memories have been introduced, and they have been incorporated into the host with little effort.

A. Host I/O Bandwidth

The Warp array can input a 32-bit word and output a 32-bit word every 200 ns. Correspondingly, to sustain this peak rate, each cluster must be able to read or write a 32-bit data item every 200 ns. This peak I/O bandwidth requirement can be satisfied if the input and output data are 8-bit or 16-bit integers that can be accessed sequentially.

In signal, image, and low-level vision processing, the input and output data are usually 16- or 8-bit integers. The data can be packed into 32-bit words before being transferred to the IU, which unpacks the data into two or four 32-bit floating-point numbers before sending them to the Warp array. The reverse operation takes place with the floating-point outputs of the Warp array. With this packing and unpacking, the data bandwidth requirement between the host and IU is reduced by a factor of two or four. Image data can be packed on the digitizer boards, without incurring overhead on the host.

The I/O bandwidth of the PC Warp external host is greatly improved over that of the prototype machine [5]. The PC Warp supports DMA and uses faster processor and memory boards. If the data transfer is sequential, DMA can be used to achieve the transfer time of less than 500 ns per word. With block transfer mode, this transfer time is further reduced to about 350 ns. The speed for nonsequential data transfers depends on the complexity of the address computation. For simple address patterns, one 32-bit word is transferred in about 900 ns.

There are two classes of applications: those whose input/output data are pixel values (e.g., vision), and those whose input/output data are floating-point quantities (e.g., scientific computing). In vision applications, data are often transferred in raster order. By packing/unpacking the pixels and using DMA, the host I/O bandwidth can sustain the maximum bandwidth of all such programs. Many of the applications that need floating-point input and output data have nonsequential data access patterns. The host becomes a bottleneck if the rate of data transfer (and address generation if DMA cannot be used) is lower than the rate the data are processed on the array. Fortunately, for many scientific applications, the computation per data item is typically quite large and the host I/O bandwidth is seldom the limiting factor in the performance of the array.

B. Host Software

The Warp host has a run-time software library that allows the programmer to synchronize the support processor and two clusters and to allocate memory in the external host. The run-time software also handles the communication and interrupts between the master and the processors in the external host.

The library of run-time routines includes utilities such as copying and moving data within the host system, subwindow selection of images, and peripheral device drivers. The compiler generates program-specific input and output routines for the clusters so that a user need not be concerned with programming at this level; these routines are linked at load time to the two cluster processor libraries.

The application program usually runs on the Warp array under control of the master; however, it is possible to assign subtasks to any of the processors in the external host. This decreases the execution time for two reasons: there is more parallelism in the computation, and data transfers between the cluster and the array using the VSB bus are twice as fast as transfers between the master processor and the array through the VME bus repeater. The processors in the external host have been extensively used in various applications, for example, obstacle avoidance for a robot vehicle and singular value decomposition.

Memory allocation and processor synchronization inside the external host are handled by the application program through subroutine calls to the run-time software. Memory is allocated through the equivalent of a Unix *malloc*( ) system call, the only difference being that the memory bank has to be explicitly specified. This explicit control allows the user to fully exploit the parallelism of the system; for example, different processors can be programmed to access different memory banks through different busses concurrently.

Tasks are scheduled by the master processor. The application code can schedule a task to be run on the completion of a different task. Once the master processor determines that one task has completed, it schedules another task requested by the application code. Overhead for this run-time scheduling of tasks is minimal.

VII. Programming Warp

As mentioned in the Introduction, Warp is programmed in a language called W2. Programs written in W2 are translated by an optimizing compiler into object code for the Warp machine. W2 hides the low-level details of the machine and allows the user to concentrate on the problem of mapping an application onto a processor array. In this section, we first describe the language and then some common computation partitioning techniques.

A. The W2 Language

The W2 language provides an abstract programming model of the machine that allows the user to focus on parallelism at the array level. The user views the Warp system as a linear array of identical, conventional processors that can communicate asynchronously with their left and right neighbors. The semantics of the communication primitives is that a cell will block if it tries to receive from any empty queue or send to a full one. This semantics is enforced at compile time in the prototype and at run time in the PC Warp, as explained in Section IV-A-2.

The user supplies the code to be executed on each cell, and the compiler handles the details of code generation and scheduling. This arrangement gives the user full control over computation partitioning and algorithm design. The language for describing the cell code is Algol-like, with iterative and conditional statements. In addition, the language provides receive and send primitives for specifying intercell communication. The compiler handles the parallelism both at the system and cell levels. At the system level, the external host and the IU are hidden from the user. The compiler generates code for the host and the IU to transfer data between the host and the array. Moreover, for the prototype Warp, addresses and loop control signals are automatically extracted from the cell programs; they are generated on the IU and passed down the address queue. At the cell level, the pipelining and parallelism in the data path of the cells are hidden from the user. The compiler uses global data flow analysis and horizontal microcode scheduling techniques, software pipelining and hierarchical reduction to generate efficient microcode directly from high-level language constructs [12], [23].

```
module MatrixMultiply (A in, B in, C out)
float A[10,10], B[10,10], C[10,10];

cellprogram (cid : 0 : 9)
begin
    function mm
    begin
        float col[10]; /* stores a column of the B matrix */
        float row;     /* accumulates the result of a row */
        float element;
        float temp;
        int i,j;

        /* first load a column of B in each cell */
        for i := 0 to 9 do begin
            receive (L, X, col[i], B[i,0]);
            for j := 1 to 9 do begin
                receive (L, X, temp, B[i,j]);
                send (R, X, temp);
            end;
            send (R, X, 0.0);
        end;

        /* calculate a row of C in each iteration */
        for i := 0 to 9 do begin
            /*  each cell computes the dot product
                between its column and the same row of A */
            row := 0.0;
            for j := 0 to 9 do begin
                receive (L, X, element, A[i,j]);
                send (R, X, element);
                row := row + element * col[j];
            end;

            /*  send out the result of each row of C */
            receive (L, Y, temp, 0.0);
            for j := 0 to 8 do begin
                receive (L, Y, temp, 0.0);
                send (R, Y, temp, C[i,j]);
            end;
            send (R, Y, row, C[i,9]);
        end;
    end
    call mm;
end
```

Fig. 7. Example W2 program.

Fig. 7 is an example of a 10 × 10 matrix multiplication program. Each cell computes one column of the result. We first load each cell with a column of the second matrix operand, then we stream the first matrix in row by row. As each row passes through the array, we accumulate the result for a column in each cell, and send the entire row of results to the host. The loading and unloading of data are slightly complicated because all cells execute the same program. **Send** and **receive** transfer data between adjacent cells; the first parameter determines the direction, and the second parameter selects the hardware channel to be used. The third parameter specifies the source (send) or the sink (receive). The fourth parameter, only applicable to those channels communicating with the host, binds the array input and output to the formal parameters of the cell programs. This information is used by the compiler to generate code for the host.

B. Program Partitioning

As discussed in Section III, the architecture of the Warp array can support various kinds of algorithms: fine-grain or large-grain parallelism, local or global operations, homogeneous or heterogeneous. There are three general program partitioning methods [4], [22]: *input partitioning, output partitioning,* and *pipelining.*

1) Input Partitioning: In this model, the input data are partitioned among the Warp cells. Each cell computes on its portion of the input data to produce a corresponding portion of the output data. This model is useful in image processing where the result at each point of the output image depends only on a small neighborhood of the corresponding point of the input image.

Input partitioning is a simple and powerful method for exploiting parallelism—most parallel machines support it in one form or another. Many of the algorithms on Warp make use of it, including most of the low-level vision programs, the discrete cosine transform (DCT), singular value decomposition [2], connected component labeling [22], border following, and the convex hull. The last three algorithms mentioned also transmit information in other ways; for example, connected components labeling first partitions the image by rows among the cells, labels each cell's portion separately, and then combines the labels from different portions to create a global labeling.

2) Output Partitioning: In this model, each Warp cell processes the entire input data set or a large part of it, but produces only part of the output. This model is used when the input to output mapping is not regular, or when any input can influence any output. Histogram and image warping are examples of such computations. This model usually requires a lot of memory because either the required input data set must be stored and then processed later, or the output must be stored in memory while the input is processed, and then output later. Each Warp cell has 32K words of local memory to support efficient use of this model.

3) Pipelining: In this model, typical of systolic computation, the algorithm is partitioned among the cells in the array, and each cell performs one stage of the processing. The Warp array's high intercell communication bandwidth and effectiveness in handling fine-grain parallelism make it possible to use this model. For some algorithms, this is the only method of achieving parallelism.

A simple example of the use of pipelining is the solution of elliptic partial differential equations using successive overrelaxation [36]. Consider the following equation:

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=f(x, y).$$

The system is solved by repeatedly combining the current values of u on a two-dimensional grid using the following recurrence.

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

where ω is a constant parameter.

In the Warp implementation, each cell is responsible for one relaxation, as expressed by the above equation. In raster order, each cell receives inputs from the preceding cell, performs its relaxation step, and outputs the results to the next cell. While a cell is performing the kth relaxation step on row i, the preceding and next cells perform the k − 1st and k + 1st relaxation steps on rows i + 2 and i − 2, respectively. Thus, in one pass of the u values through the ten-cell Warp array, the above recurrence is applied ten times. This process is repeated, under control of the external host, until convergence is achieved.

VIII. Evaluation

Since the two copies of the wire-wrapped prototype Warp machine became operational at Carnegie Mellon in 1986, we have used the machines substantially in various applications [2]-[4], [10], [13], [22]. The application effort has been increased since April 1987 when the first PC Warp machine was delivered to Carnegie Mellon.

The applications area that guided the development of Warp most strongly was computer vision, particularly as applied to robot navigation. We studied a standard library of image processing algorithms [30] and concluded that the great majority of algorithms could efficiently use the Warp machine. Moreover, robot navigation is an area of active research at Carnegie Mellon and has real-time requirements where Warp can make a significant difference in overall performance [32], [33]. Since the requirements of computer vision had a significant influence on all aspects of the design of Warp, we contrast the Warp machine with other architectures directed towards computer vision in Section VIII-B.

Our first effort was to develop applications that used Warp for robot navigation. Presently mounted inside of a robot vehicle, Warp has been used to perform road following and obstacle avoidance. We have implemented road following using color classification, obstacle avoidance using stereo vision, obstacle avoidance using a laser range-finder, and path planning using dynamic programming. We have also implemented a significant image processing library (over 100 programs) on Warp [30], to support robot navigation and vision research in general. Some of the library routines are listed in Table IV.

A second interest was in using Warp in signal processing and scientific computing. Warp's high floating-point computation rate and systolic structure make it especially attractive for these applications. We have implemented singular value decomposition (SVD) for adaptive beam forming, fast two-dimensional image correlation using FFT, successive overrelaxation (SOR) for the solution of elliptic partial differential equations (PDE), as well as computational geometry al-

TABLE III
MEASURED SPEEDUPS ON THE WIRE-WRAPPED PROTOTYPE WARP MACHINE

Task (All images are 512×512. All code compiler generated.)	Time (ms)	Speedup over Vax 11/780 with floating-point accelerator
Quadratic image warping	400	100
Warp array generates addresses using quadratic form in 240 ms.		
Host computes output image using addresses generated by Warp.		
Road-following	6000	200
Obstacle avoidance using ERIM, a laser range-finder	350	60
Time does not include 500 ms for scanner I/O.		
Minimum-cost path, 512×512 image, one pass	500	60
Host provides feedback.		
Detecting lines by Hough Transform	2000	387
Host merges results.		
Minimum-cost path, 350-node graph	16000	98
Convex hull, 1,000 random nodes	18	74
Solving elliptic PDE by SOR, 50,625 unknowns (10 iterations)	180	440
Singular value decomposition of 100×100 matrix	1500	100
FFT on 2D image	2500	300
Warp array takes 600 ms. Remaining time is for data shuffling by host.		
Image correlation using FFT	7000	300
Data shuffling in host.		
Image compression with 8×8 discrete cosine transforms	110	500
Mandelbrot image, 256 iterations	6960	100

TABLE IV
PERFORMANCE OF SPECIFIC ALGORITHMS ON THE WIRE-WRAPPED PROTOTYPE WARP MACHINE

Task All images are 512×512. All code compiler generated.	Time (ms)	MFLOPS (Upper bound)	MFLOPS (Achieved)
100×100 matrix multiplication.	25	100	79
3×3 convolution.	70	94	66
11×11 symmetric convolution.	367	90	59
Calculate transformation table for non-linear warping.	248	80	57
Generate matrices for plane fit for obstacle avoidance using ERIM scanner.	174	62	49
Generate mapping table for affine image warping.	225	67	43
Moravec's interest operator.	82	60	36
3×3 maximum filtering.	280	67	30
Sobel edge detection.	206	77	30
Label color image using quadratic form for road following.	308	87	27
Image magnification using cubic spline interpolation.	8438	66	25
7×7 average gray values in square neighborhood.	1090	51	24
5×5 convolution.	284	52	23
Calculate quadratic form from labelled color image.	134	58	22
Compute gradient using 9×9 Canny operator.	473	92	21
Discrete cosine transform on 8×8 windows.	175	94	21
3×3 Laplacian edge detection.	228	94	20
15×15 Harwood-style symmetric edge preserving smoothing.	32000	50	16
Find zero-crossings.	179	78	16
Calculate (x,y,z) coordinates from ERIM laser range scanner data.	24	75	13
Histogram.	67	50	12
Coarse-to-fine correlation for stereo vision.	12	77	11
3×3 median filter.	448	50	7
Levialdi's binary shrink operation.	180	71	7
31×31 average gray values in square neighborhood.	444	61	5
Convert real image to integer using max, min linear scaling.	249	66	4
Average 512×512 image to produce 256×256.	150	58	3

gorithms such as convex hull and algorithms for finding the shortest paths in a graph.

A. Performance Data

Two figures of merit are used to evaluate the performance of Warp. One is overall system performance, and the other is performance on specific algorithms. Table III presents Warp's performance in several systems for robot navigation, signal processing, scientific computation, and geometric algorithms, while Table IV presents Warp's performance on a large number of specific algorithms. Both tables report the performance for the wire-wrapped Warp prototype with a Sun-3/160 as the master processor. The PC Warp will in general exceed the reported performance, because of its improved architec-

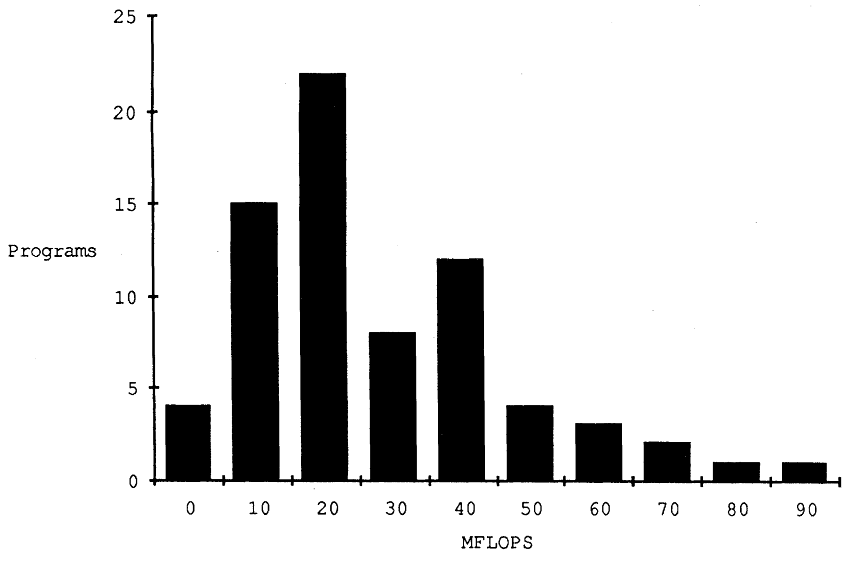


Fig. 8.

ture and increased host I/O speed as described earlier. Table III includes all system overheads except for initial program memory loading. We compare the performance of Warp to a VAX 11/780 with floating-point accelerator because this computer is widely used and, therefore, familiar to most people.

Statistics have been gathered for a collection of 72 W2 programs in the application areas of vision, signal processing, and scientific computing [23]. Table IV presents the utilization of the Warp array for a sample of these programs. System overheads such as microcode loading and program initialization are not counted. We assume that the host I/O can keep up with the Warp array; this assumption is realistic for most applications with the host of the production Warp machine. Fig. 8 shows the performance distribution of the 72 programs. The arithmetic mean is 28 MFLOPS, and the standard deviation is 18 MFLOPS.

The Warp cell has several independent functional units, including separate floating-point units for addition and multiplication. The achievable performance of a program is limited by the most used resource. For example, in a computation that contains only additions and no multiplications, the maximum achievable performance is only 50 MFLOPS. Table IV gives an upper bound on the achievable performance and the achieved performance. The upper bound is obtained by assuming that the floating-point unit that is used more often in the program is the most used resource, and that it can be kept busy all the time. That is, this upper bound cannot be met even with a perfect compiler if the most used resource is some other functional unit, such as the memory, or if data dependencies in the computation prevent the most used resource from being used all the time.

Many of the programs in Tables III and IV are coded without fine tuning the W2 code. Optimizations can often provide a significant speedup over the times given. First, the W2 code can be optimized, using conventional programming techniques such as unrolling loops with few iterations, replacing array references by scalars, and so on. Second, in some cases in Table III the external host in the prototype Warp is a bottleneck, and it is possible to speed up this portion of the Warp machine by recoding the I/O transfer programs generated by the W2 compiler in MC68020 Assembly language. Moreover, the external host for the PC Warp is faster and supports DMA, so that even with the compiler generated code it will no longer be the bottleneck. Third, since restrictions on using the Warp cells in a pipeline are removed in PC Warp as explained in Section IV-B-4, it will be possible to implement many of the vision algorithms in a pipelining fashion. This can lead to a threefold speedup, since input, computation, and output will be done at the same time. Fourth, in a few cases we have discovered a better algorithm for the Warp implementation than what was originally programmed.

In Table III, the speedup ranges from 60 to 500. With the optimizations we discuss above, all systems listed should show at least a speedup of about 100 over the VAX 11/780 with a floating-point accelerator.

B. Architectural Alternatives

We discuss the architectural decisions made in Warp by contrasting them with the decisions made in bit-serial processor arrays, such as the Connection Machine [34] and MPP [7]. We chose these architectures because they have also been used extensively for computer vision and image processing, and because the design choices in these architectures were made significantly differently than in Warp. These differences help exhibit and clarify the design space for the Warp architecture.

We attempt to make our comparison quantitative by using benchmark data from a DARPA Image Understanding (''DARPA IU'') workshop held in November 1986 to compare various computers for vision [29]. In this workshop, benchmarks for low and midlevel computer vision were defined and programmed by researchers on a wide variety of computers, including Warp and the Connection Machine [25].

We briefly review salient features of the Connection Machine, called CM-1, used in these benchmarks. It is a SIMD machine, consisting of an array of 64K bit-serial processing elements, each with 4K bits of memory. The processors are connected by two networks: one connects each processor to four adjacent processors, and the other is a 12-dimensional hypercube, connecting groups of 16 processors. The array is controlled by a host, which is a Symbolics 3640 Lisp machine. CM-1 is programmed in an extension to Common Lisp called *Lisp [24], in which references to data objects stored in the CM-1 array and objects on the host can be intermixed.

Although our intention is to illustrate architectural decisions made in Warp, not to compare it to the Connection Machine, we should not cite benchmark performance figures on two different computers without mentioning two critical factors, namely cost and size. CM-1 is approximately one order of magnitude more expensive and larger than Warp.

1) Programming Model: Bit-serial processor arrays implement a *data parallel* programming model, in which different processors process different elements of the data set. In the Connection Machine, the programmer manipulates data objects stored in the Connection Machine array by the use of primitives in which the effect of a Lisp operator is distributed over a data object.

In systolic arrays, the processors individually manipulate words of data. In Warp, we have implemented data parallel programming models through the use of input and output partitioning. We have encapsulated input partitioning over images in a specialized language called Apply [14]. In addition to these models, the high interprocessor bandwidth of the systolic array allows efficient implementation of pipelining, in which not the data, but the algorithm is partitioned.

2) Processor I/O Bandwidth and Topology: Systolic arrays have high bandwidth between processors, which are organized in a simple topology. In the case of the Warp array, this is the simplest possible topology, namely a linear array. The interconnection networks in the Connection Machine allow flexible topology, but low bandwidth between communicating processors.

Bit-serial processor arrays may suffer from a serious bottleneck in I/O with the external world because of the difficulty of feeding a large amount of data through a single simple processor. This bottleneck has been addressed in various ways. MPP uses a "staging memory" in which image data can be placed and distributed to the array along one dimension. The I/O bottleneck has been addressed by a new version of the Connection Machine, called CM-2 [31]. In this computer, a number of disk drives can feed data into various points in the array simultaneously. The CM-1 benchmark figures do not include image I/O: the processing is done on an image which has already been loaded into the array, and processing is completed with the image still in the array. Otherwise, image I/O would completely dominate processing time. In many cases it is necessary to process an image which is stored in a frame buffer or host memory, which is easier in Warp because of the high bandwidth between the Warp array and the Warp host. All the Warp benchmarks in this section include I/O time from the host.

The high bandwidth connection between processors in the Warp array makes it possible for all processors to see all data in an image, while achieving useful image processing time. (In fact, because of the linear topology, there is no time advantage to limit the passage of an image through less than all processors.) This is important in global image computations such as Hough transform, where any input can influence any output. For example, the transform of a 512 × 512 image into a 180 × 512 Hough space takes 1.7 s on Warp, only 2.5 times as long as on CM-1. The ratio here is far less than for a simple local computation on a large image, such as Laplacian and zero crossing.

In some global operations, processing is done separately on different cells, then combined in a series of pairwise merge operations using a "divide and conquer" approach. This type of computation can be difficult to implement using limited topology communications as in Warp. For example, in the Warp border following algorithm for a 512 × 512 image, individual cells trace the borders of different portions of the image, then those borders are combined in a series of merge operations in the Warp array. The time for border following on Warp is 1100 ms, significantly more than the 100 ms the algorithm takes on CM-1.

3) Processor Number and Power: Warp has only ten parallel processing elements in its array, each of which is a powerful 10 MFLOPS processor. CM-1, on the other hand, has 64K processing elements, each of which is a simple bit-serial processor. Thus, the two machines stand at opposite ends of the spectrum of processor number and power.

We find that the small number of processing elements in Warp makes it easier to get good use of the Warp array in problems where a complex global computation is performed on a moderate-sized data set. In these problems, not much data parallelism is "available." For example, the DARPA IU benchmarks included the computation of the two-dimensional convex hull [26] of a set of 1000 points. The CM-1 algorithm used a brush-fire expansion algorithm, which led to an execution time of 200 ms for the complete computation. The same algorithm was implemented on Warp, and gave the 18 ms figure reported in Table III. Similar ratios are found in the times for the minimal spanning tree of 1000 points (160 ms on Warp versus 2.2 s on CM-1) and a triangle visibility problem for 1000 three-dimensional triangles (400 ms on Warp versus 1 s on CM-1).

Simple algorithms at the lowest level of vision, such as edge detection computations, run much faster on large arrays of processors such as the Connection Machine than Warp. This is because no communication is required between distant elements of the array, and the large array of processors can be readily mapped onto the large image array. For example, the computation of an 11 × 11 Laplacian [15] on a 512 × 512 image, followed by the detection of zero crossings, takes only 3 ms on CM-1, as opposed to 400 ms on Warp.

The floating-point processors in Warp aid the programmer in eliminating the need for low-level algorithmic analysis. For example, the Connection Machine used discrete fixed-point approximation to several algorithms, including Voronoi diagram and convex hull. The use of floating-point made it unnecessary for the Warp programmer to make assumptions about the data range and distribution.

In conclusion, we see that bit-serial processor arrays excel in the lowest level of vision, such as edge detection. The CM-1's performance at this level exceeded Warp's by two orders of magnitude. However, specialized hardware must be used to eliminate a severe I/O bottleneck to actually observe this performance. The use of the router in the Connection Machine allows it to do well also at higher levels of vision, such as border following. We also see that the more general class of programming models and use of floating-point hardware in Warp give it good actual performance in a wide range of algorithms, especially including complex global computations on moderately sized data sets.

IX. Conclusions

The Warp computer has achieved high performance in a variety of application areas, including low-level vision, signal processing, and scientific computation. Currently produced by our industrial partner (GE), Warp is much more powerful and programmable than many other machines of comparable cost.

The effectiveness of the Warp computer results from a balanced effort in architecture, software, and applications. The simple, linear topology of the Warp array naturally supports several useful program partitioning models; the Warp cells' high degree of programmability and large local memory make up for the lack of higher dimensional connectivity. The high-computation rate on each cell is matched by an equally high inter- and intracell bandwidth. The host system provides the Warp array with high I/O bandwidth. The optimizing W2 compiler maps programs from a high-level language to efficient microcode for the Warp array. Integration of the Warp array into Unix as an attached processor makes the Warp machine easily accessible to users. A sizable application library has been implemented to support development of research systems in vision.

The development of a compiler is essential in designing the architecture of a machine. Designing and implementing a compiler require a thorough study of the functionality of the machine; the systematic analysis of the machine allows us to uncover problems that may otherwise be undetected by writing sample programs. The compiler is also an excellent tool for evaluating different architectural alternatives. The development of the W2 compiler has significantly influenced the evolution of the architecture of Warp.

An early identification of an application area is essential for the development of an experimental machine such as Warp whose architecture is radically different from conventional ones. Including the application users in the early phase of the project—the vision research group at Carnegie Mellon in our case—helped us focus on the architectural requirements and provided early feedback.

Prototyping is important for architecture development. An early prototype system gives the designers realistic feedback about the constraints of the hardware implementation and provides a base for the software and application developers to test out their ideas. To speed up implementation of the prototype, we used off-the-shelf parts. To concentrate our efforts on the architecture of the Warp array, we developed the host from industry standard boards.

The Warp machine has demonstrated the feasibility of programmable, high-performance systolic array computers. The programmability of Warp has substantially extended the machine's application domain. The cost of programmability is limited to an increase in the physical size of the machine; it does not incur a loss in performance, given appropriate architectural support. This is shown by Warp, as it can be programmed to execute many well-known systolic algorithms as fast as special-purpose arrays built using similar technology.

Acknowledgment

We appreciate the contributions to the Warp project by our colleagues and visitors at Carnegie Mellon: D. Adams, F. Bitz, C. Bono, M. Browne, B. Bruegge, C. H. Chang, E. Clune, R. Cohn, R. Conde, J. Deutch, P. Dew, B. Enderton, L. Hamey, P. K. Hsiung, K. Hughes, T. Kanade, G. Klinker, P. Lieu, P. Maulik, D. Morris, A. Noaman, T. M. Parng, H. Printz, J. Race, M. Ravishankar, J. Rendas, H. Ribas, C. Sarocky, K. Sarocky, J. Senko, Y. Shintani, B. Siegell, H. Sohn, P. Steenkiste, Y. B. Tsai, P. S. Tseng, R. Wallace, J. K. Wang, I. C. Wu, D. Yam, and A. Zobel. We thank our industrial partners GE and Honeywell for their contribution towards the construction of the wire-wrapped prototypes. We appreciate the continued collaboration with GE for the development of the production Warp machine. In particular, we thank R. Barned, S. Carmel, J. Cokus, J. Condon, D. Crump, R. A. Field, R. Gaus, N. Gearhart, J. Iannuzzi, A. Lock, C. Pickering, A. Pfueller, M. Sarig, S. Sillich, T. Stark, W. Tates, A. Toll, C. Walrath, and J. Weimar of GE in Syracuse for their efforts.

References

[1] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. S. Lam, O. Menzilcioglu, K. Sarocky, and J. A. Webb, "Warp architecture and implementation," in *Proc. 13th Annu. Int. Symp. Comput. Architecture,* IEEE/ACM, June, 1986, pp. 346-356.

[2] M. Annaratone, E. Arnould, H. T. Kung, and O. Menzilcioglu, "Using Warp as a supercomputer in signal processing," in *Proc. ICASSP 86,* Apr. 1986, pp. 2895-2898.

[3] M. Annaratone, F. Bitz, E. Clune, H. T. Kung, P. Maulik, H. Ribas, P. Tseng, and J. Webb, "Applications and algorithm partitioning on Warp," in *Proc. Compcon Spring 87,* San Francisco, CA, Feb., 1987, pp. 272-275.

[4] M. Annaratone, F. Bitz, J. Deutch, L. Hamey, H. T. Kung, P. C. Maulik, P. Tseng, and J. A. Webb, "Applications experience on Warp," in *Proc. 1987 Nat. Comput. Conf.,* AFIPS, Chicago, IL, June 1987, pp. 149-158.

[5] M. Annaratone, E. Arnould, R. Cohn, T. Gross, H. T. Kung, M. Lam, O. Menzilcioglu, K. Sarocky, J. Senko, and J. Webb, "Architecture of Warp," in *Proc. Compcon Spring 87,* San Francisco, CA, Feb. 1987, pp. 274-267.

[6] ——, "Warp architecture: From prototype to production," in *Proc. 1987 Nat. Comput. Conf.,* AFIPS, Chicago, IL, June, 1987, pp. 133-140.

[7] K. E. Batcher, "Design of a massively parallel processor," *IEEE Trans. Comput.,* vol. C-29, pp. 836-840, 1980.

[8] B. Bruegge, C. Chang, R. Cohn, T. Gross, M. Lam, P. Lieu, A. Noaman, and D. Yam, "The Warp programming environment," in *Proc. 1987 Nat. Comput. Conf.,* AFIPS, Chicago, IL, June 1987, pp. 141-148.

[9] A. E. Charlesworth, "An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family," *Computer,* vol. 14, pp. 18-27, Sept. 1981.

[10] E. Clune, J. D. Crisman, G. J. Klinker, and J. A. Webb, "Implementation and performance of a complex vision system on a systolic array machine," in *Proc. Conf. Frontiers Comput.*, Amsterdam, Dec. 1987.

[11] A. L. Fisher, H. T. Kung, and K. Sarocky, "Experience with the CMU programmable systolic chip," *Microarchitecture VLSI Comput.*, pp. 209-222, 1985.

[12] T. Gross and M. Lam, "Compilation for a high-performance systolic array," in *Proc. SIGPLAN 86 Symp. Compiler Construction,* ACM SIGPLAN, June, 1986, pp. 27-38.

[13] T. Gross, H. T. Kung, M. Lam, and J. Webb, "Warp as a machine for low-level vision," in *Proc. 1985 IEEE Int. Conf. Robot. Automat.*, Mar. 1985, pp. 790-800.

[14] L. G. C. Hamey, J. A. Webb, and I. C. Wu, "Low-level vision on warp and the apply programming model," in *Parallel Computation and Computers for Artificial Intelligence,* J. Kowalik, Ed. Hingham, MA: Kluwer Academic, 1987.

[15] R. M. Haralick, "Digital step edges from zero crossings of second directional derivatives," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. PAMI-6, pp. 58-68, 1984.

[16] F. H. Hsu, H. T. Kung, T. Nishizawa, and A. Sussman, "Architecture of the link and interconnection chip," in *Proc. 1985 Chapel Hill Conf., VLSI*, Comput. Sci., Dep., Univ. North Carolina, May, 1985, pp. 186-195.

[17] T. Kanade and J. A. Webb, "End of year report for parallel vision algorithm design and implementation," Tech. Rep. CMU-R1 TR-87-15 Robot. Instit., Carnegie Mellon Univ., 1987.

[18] H. T. Kung, "Why systolic architectures?," *Computer,* vol. 15, pp. 37-46, Jan. 1982.

[19] ——, "Systolic algorithms for the CMU Warp processor," in *Proc. Seventh Int. Conf. Pattern Recognition,* Int. Ass. Pattern Recognition, 1984, pp. 570-577.

[20] ——, "Memory requirements for balanced computer architectures," *J. Complexity,* vol. 1, pp. 147-157, 1985.

[21] H. T. Kung and J. A. Webb, "Global operations on the CMU warp machine," in *Proc. 1985 AIAA Comput. Aerosp. V Conf.*, Amer. Instit. Aeronaut. Astronaut., Oct., 1985, pp. 209-218.

[22] ——, "Mapping image processing operations onto a linear systolic machine," *Distributed Comput.*, vol. 1, pp. 246-257, 1986.

[23] M. S. Lam, "A systolic array optimizing compiler," Ph.D. dissertation, Carnegie Mellon Univ., May 1987.

[24] C. Lasser, *The Complete *Lisp Manual,* Thinking Machines Corp., Cambridge, MA, 1986.

[25] J. J. Little, G. Gelloch, and T. Cass, "Parallel algorithms for computer vision on the connection machine," in *Proc. Image Understanding Workshop,* DARPA, Feb., 1987, pp. 628-638.

[26] F. P. Preparata and M. I. Shamos, *Computational Geometry—An Introduction.* New York: Springer-Verlag, 1985.

[27] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," in *Proc. 14th Annu. Workshop Microprogramming,* Oct., 1981, pp. 183-198.

[28] B. R. Rau, P. J. Kuekes, and C. D. Glaeser, "A statistically scheduled VLSI interconnect for parallel processors," *VLSI Syst. Comput.*, Oct. 1981, pp. 389-395.

[29] A. Rosenfeld, "A report on the DARPA image understanding architectures workshop," in *Proc. Image Understanding Workshop,* DARPA, Los Angeles, CA, Feb., 1987, pp. 298-301.

[30] H. Tamura, S. Sakane, F. Tomita, N. Yokoya, K. Sakaue, and N. Kaneko, *SPIDER Users' Manual,* Joint System Development Corp., Tokyo, Japan, 1983.

[31] Thinking Machines Corp., *Connection Machine Model CM-2 Technical Summary HA 87-4,* Thinking Machines Corp., Apr. 1897.

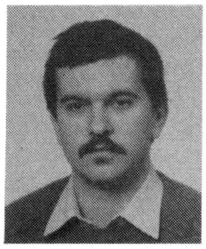
[32] R. Wallace, A. Stentz, C. Thorpe, W. Whittaker, and T. Kanade, "First results in robot road-following," in *Proc. IJCAI,* 1985, pp. 1089-1093.

[33] R. Wallace, K. Matsuzaki, Y. Goto, J. Crisman, J. Webb, and T. Kanade, "Progress in robot road-following," in *Proc. 1986 IEEE Int. Conf. Robot. Automat.*, Apr., 1986, pp. 1615-1621.

[34] D. L. Waltz, "Applications of the connection machine," *Computer,* vol. 20, pp. 85-97, Jan. 1987.

[35] B. Woo, L. Lin, and F. Ware, "A high-speed 32 bit IEEE floating-point chip set for digital signal processing," in *Proc. ICASSP 84,* IEEE, 1984, pp. 16.6.1-16.6.4.

[36] D. Young, *Iterative Solution of Large Linear Systems.* New York: Academic, 1971.

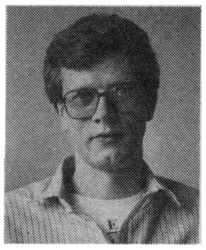


Marco Annaratone received the Dott.Ing. degree in computer science and electrical engineering from Politecnico di Milano, Milan, Italy, in 1980.

From 1982 to 1984 he was Visiting Scientist in the Department of Computer Science at Carnegie Mellon University, Pittsburgh, PA. From 1984 to 1987 he was a faculty member in the same department, first as a Research Associate and then as a Research Computer Scientist. He is the author of *Digital CMOS Circuit Design* (Hingham, MA: Kluwer Academic), a book on VLSI design methodologies. His current research interests include computer architecture, parallel computer architecture, and parallel implementation of algorithms in the field of scientific computation. He is now an Assistant Professor of Computer Science at the Swiss Federal Institute of Technology (ETH), Zurich and can be reached at the Institute for Integrated Systems, ETH Zentrum, 8092 Zurich, Switzerland.

Emmanuel Arnould was born in Paris, France. He received the M.S. degree in electrical engineering from the Universite des Sciences, Paris, France, in 1981, and the M.S. degree in computer science from the Ecole Nationale Superieure des Telecommunications, Paris, France, in 1983.

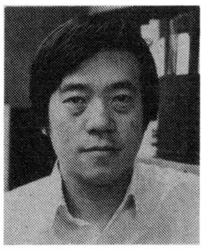
Since February 1984, he has been a Research Engineer in the Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, where he actively participated in the design of the Warp computer. His research interests include computer system architecture, supercomputing, and supercomputer networks.



Thomas Gross (S'79–M'83) received the M.S. degree in computer science from Stanford University, Stanford, CA, the Diplom-Informatiker degree from the Technical University, Munich, Germany, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 1983, where he participated in the MIPS project.

He joined the faculty of the Department of Computer Science, Carnegie Mellon University, in 1984. His current research interests include the practical aspects of high-performance computer systems: computer architecture, processor design, optimizing compilers, and the software systems that are needed to make high-performance computers usable.

Dr. Gross received an IBM Faculty Development Award in 1985.



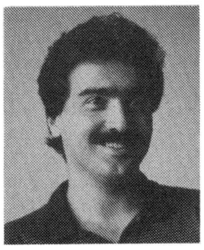
H. T. Kung received the Ph.D. degree from Carnegie Mellon University, Pittsburgh, PA, in 1974.

He joined the faculty of Carnegie Mellon University in 1974 and was appointed to Professor in 1982. He is currently holding the Shell Distinguished Chair in Computer Science at Carnegie Mellon. He was Guggenheim Fellow in 1983–1984, and a full time Architecture Consultant to ESL, Inc., a subsidiary of TRW, in 1981. His current research interests are in high-performance computer architectures and their applications.

Dr. Kung has served on editorial boards of several journals and program committees of numerous conferences in VLSI and computer science.

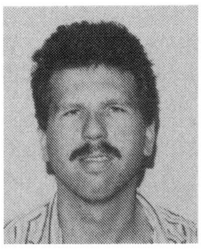
Monica Lam received the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, in 1987, and the B.S. degree in computer science from the University of British Columbia, Vancouver, B.C. in 1980.

She is currently a Research Associate in the Department of Computer Science at Carnegie Mellon University. Her research interests include parallel architectures and optimizing compilers.



Onat Menzilcioglu received the B.S. degree in electrical engineering from Middle East Technical University, Ankara, Turkey, in 1980, and the M.S. degree in computer engineering from Carnegie Mellon University, Pittsburgh, PA, in 1982.

He is a Ph.D. degree candidate in the Department of Electrical and Computer Engineering, Carnegie Mellon University. He has been working on programmable systolic array architectures since 1983. His research interests include computer architecture and design, and fault tolerance.



Jon A. Webb received the B.A. degree in mathematics from The University of South Florida, Tampa, in 1975, the M.S. degree in computer science from The Ohio State University, Columbus, in 1976, and the Ph.D. degree in computer science from The University of Texas, Austin, in 1980.

Since 1981 he has worked on the faculty of the Department of Computer Science at Carnegie Mellon University, where he is currently a Research Computer Scientist. His research interests include the theory of vision and parallel architectures for vision. He has published papers on the recovery of structure from motion, the shape of subjective contours, the design and use of a parallel architecture for low-level vision, and experiments in the visual control of a robot vehicle.

Dr. Webb is a member of the IEEE Computer Society and the Association for Computing Machinery.