Warp Architecture and Implementation

Marco Annaratone, Emmanuel Arnould, Thomas Gross, H. T. Kung, Monica S. Lam,
Onat Menzilcioglu, Ken Sarocky, and Jon A. Webb

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

A high-performance systolic array computer called *Warp* has been designed and constructed. The machine has a systolic array of 10 or more linearly connected cells, each of which is a programmable processor capable of performing 10 million floating-point operations per second (10 MFLOPS). A 10-cell machine therefore has a peak performance of 100 MFLOPS. Warp is integrated into a UNIX host system. Program development is supported by a compiler.

The first 10-cell machine became operational in 1986. Low-level vision processing for robot vehicles is one of the first applications of the machine.

This paper describes the architecture and implementation of the Warp machine, and justifies and evaluates some of the architectural features with system, software and application considerations.

1. Introduction

Warp is a high-performance systolic array computer designed to provide computation power for signal, image and low-level vision processing: the machine's first applications are vision-based control algorithms for robot vehicles, and image analysis for large image databases [3].

A full-scale Warp machine consists of a linear systolic array of 10 or more identical cells, each of which is a 10 MFLOPS programmable processor. The processor array is integrated in a powerful host system, which provides an adequate data bandwidth to sustain the array at full speed in the targeted applications, and a general purpose computing environment, specifically UNIX, for application programs.

As an example of performance, a 10-cell Warp can process 1024-point complex fast Fourier transforms at a rate of one FFT every 600 microseconds. This programmable machine can also perform many other primitive computations, such as two-dimensional convolution, and real or complex matrix multiplication, at a peak rate of 100 MFLOPS. Warp can be described as an array of conventional array processors; the machine can efficiently implement not only systolic algorithms where communication between adjacent cells is intensive, but also non-systolic algorithms where each cell operates on its own data independently.

While achieving a high computation throughput usually expected only from a special-purpose systolic array, Warp has a high degree of programmability. Each processor is a horizontal microengine; the user has complete control over the various functional units. To overcome the complexity in managing this fine-grain parallelism, we are developing an optimizing compiler to support a high-level programming language [2]. To the programmer, Warp is an array of simple sequential processors, communicating asynchronously. Initial results indicate that reasonable performance can be achieved with the compiler.

Carnegie Mellon designed, assembled and tested a 2-cell prototype machine (in use since Fall of 1985). Production of the full-scale machine is contracted to two industrial partners, General Electric and Honeywell. The first 10-cell machine was delivered by General Electric in early 1986. It is used in vision research; for example, it performs the low-level vision processing for a robot vehicle built by Carnegie Mellon [8]. At least eight additional 10-cell Warp machines will be built in 1986 and 1987 for applications in areas such as robot vehicles and image analysis.

This paper describes the Warp architecture and implementation, justifies and evaluates the major architectural features with system, software and application considerations. We first present an overview of the architecture of the machine. We next discuss the architecture of the Warp processor array in more detail: the linear configuration of the array, our programming model of the array and its implications, and the architecture of the individual cells in the array. A description of the hardware implementation then follows. We then present the hardware and software architecture of the host system that controls the Warp machine. The last

The research was supported in part by Defense Advanced Research Projects Agency (DOD), monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539, and Naval Electronic Systems Command under Contract N00039-85-C-0134, and in part by the Office of Naval Research under Contracts N00014-80-C-0236, NR 048-659, and N00014-85-K-0152, NR SDRJ-007. T. Gross is also supported by an IBM Faculty Development Award, H. T. Kung by a Shell Distinguished Chair in Computer Science.

0884-7495/86/0000/0346$01.00 © 1986 IEEE

section includes some general discussions of the machine and concluding remarks.

2. Warp machine overview

The Warp machine has three components—the Warp processor array (*Warp array*), the interface unit (*IU*), and the *host*, as depicted in Figure 2-1. The Warp array performs the computation-intensive routines, for example, low-level vision routines. The IU handles the input/output between the array and the host, and generates addresses and control signals for the Warp array. The host executes the parts of the application programs that are not mapped onto the Warp array and supplies the data to and receives the results from the array.

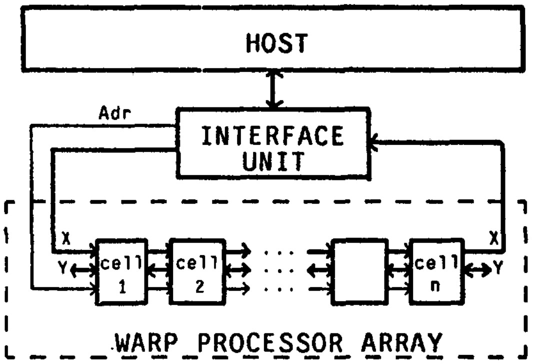

Figure 2-1: Warp machine overview

The Warp array is a programmable, one-dimensional systolic array with identical cells called Warp cells. Data flow through the array on two data paths (X and Y), while addresses and systolic control signals travel on the Adr path (as shown in Figure 2-1).

Each Warp cell is implemented as a programmable horizontal microengine, with its own program memory and microsequencer. A Warp cell has a 32-bit wide data path, as depicted in Figure 2-2. The data path consists of two 32-bit floating-point processing elements: one multiplier and one ALU, a 4K-word memory for resident and temporary data, a 128-word queue for each communication channel, and a 32-word register file to buffer data for each floating-point unit. All these components are interconnected through a crossbar switch as shown in Figure 2-2.

To reduce the inherent risk in prototyping a new architecture, the current implementation of the Warp machine uses conservative design principles and no custom-made parts. TTL-compatible parts are used throughout. Each Warp cell is implemented on a large board (dimensions 15" X 17"), and so is the IU. Figure 2-3 shows a photograph of a Warp cell.

The host consists of a VME-based workstation (currently a Sun 2/160), that serves as the master controller of the Warp machine, and an "external host", so named because it is *external* to the workstation. The workstation provides a UNIX environment for running application programs and the external host provides a high data transfer rate for communicating with the Warp array.

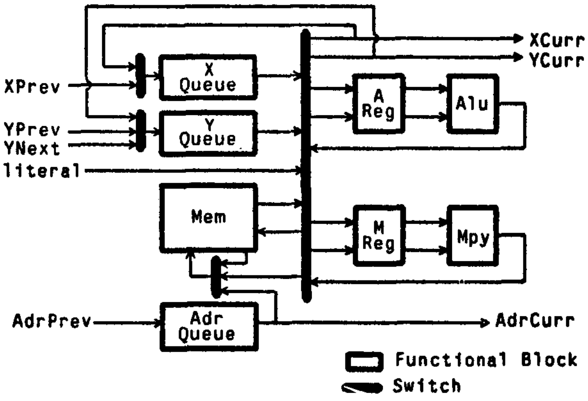

Figure 2-2: Warp cell data path

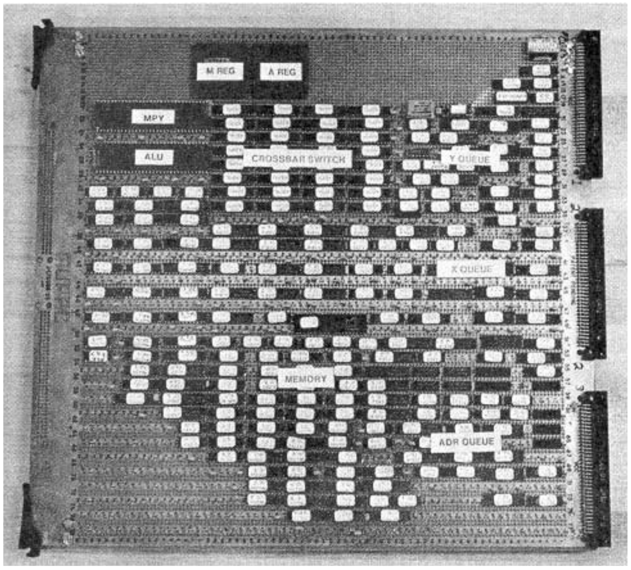

Figure 2-3: Warp cell layout

Figure 3-1 is a picture of the Warp machine. A single 19" rack hosts the IU, a Warp array of 10 cells, the external host, as well as associated power supplies and fans. The workstation is not visible in this picture.

The first 10-cell machine uses wire-wrap technology. Future Warp machines will be implemented with PC boards; the 19" rack will be able to host the IU and up to 24 Warp cells. For the PC board version, each cell will have increased local memory (at least 32K or 64K words) and enlarged queues.

3. Linear array of powerful cells

The Warp array is a linear array of identical cells with only neighboring cells communicating directly. The linear configuration was chosen for several reasons. First, it is easy to implement. Second, it is easy to extend the number of cells in the array. Third, a linear array has modest I/O requirements since only the two end-cells communicate with the outside world.

The advantages of the linear interconnection are outweighed, however, if the constraints of interconnection render the machine too difficult to use for programmers. The concern is whether or not we can efficiently map applications on the linear array of cells. While many algorithms have been designed specifically for linear arrays, computations designed for other interconnection topologies can often be efficiently simulated on the linear array mainly because the Warp cell, the building-block of the array, is itself a powerful engine. In particular, a single Warp cell can be time multiplexed to perform the function of a column of cells, and therefore the linear array can, for example, implement a two-dimensional systolic array effectively.

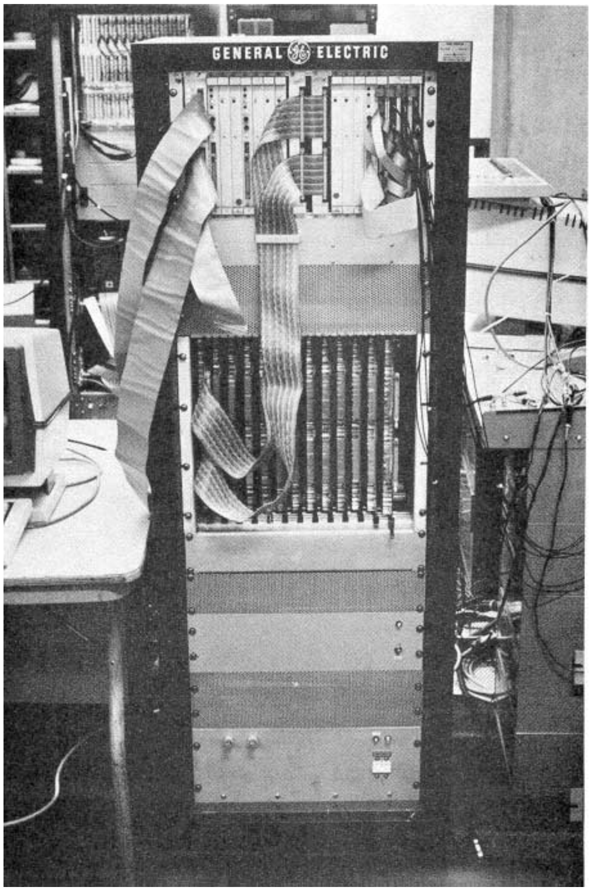

Figure 3-1: The Warp machine, serial number 1

A feature that distinguishes the Warp cell from other processors of similar computation power is its high I/O bandwidth. Each Warp cell can transfer up to 20 million words (80 Mbytes) to and from its neighboring cells per second. (In addition, 10 million 16-bit addresses can flow from one cell to the next cell every second.) We have been able to implement this high bandwidth communication link with only modest engineering efforts because of the simplicity of the linear interconnection structure. This high inter-cell communication bandwidth makes it possible to transfer large volumes of intermediate data between neighboring cells and thus supports fine-grain problem decomposition.

For communicating with the outside world, the Warp array can sustain a 80 Mbytes/sec peak transfer rate. In the current setup, the IU can communicate with the Warp array at a rate of 40 Mbytes/sec. This assumes that the Warp array inputs and outputs a 32-bit word every (200 ns) instruction cycle. However the current host can only support up to 10 Mbytes/sec transfer rates. The smaller transfer rate supported by the host is not expected to affect the effective use of the Warp array for our applications for the following reasons: First, the Warp array typically performs one input and one output every two or more cycles. Second, for most signal and image processing applications, the host deals with 8-bit integers rather than 32-bit floating-point numbers, and therefore the I/O bandwidth for the host needs only be a quarter of that for the Warp array. This implies that 10 Mbytes/sec transfer rate for the host is sufficient. This I/O issue and the use of the IU for integer/float-point conversions will be discussed later in the paper.

Each cell has a large local data memory; this feature is seldom found in special-purpose, systolic array designs, and is another reason why Warp is powerful and flexible. It can be shown that by increasing the data memory size, higher computation bandwidth can be sustained without imposing increased demand on the I/O bandwidth [4]. The large memory size, together with the high I/O bandwidth, makes Warp capable of performing *global* operations in which each output depends on any or a large portion of the input [6]. Examples of global operations are FFT, component labeling, Hough transform, image warping, and matrix computations such as matrix multiplication. Systolic arrays are known to be effective for local operations such as a 3 X 3 convolution. The additional ability to perform global operations significantly broadens the applicability of the machine.

In summary, the simple linear processor array used in Warp is a powerful and flexible structure largely because the array is made of powerful, programmable processors with high I/O capabilities and large local memories.

4. Warp architecture

The domains of signal processing, image processing, and scientific computing share the characteristic that the computation is often regular and repetitive. That is, the control flow through a computation is independent from the input data; similar operations must be performed for all input sets. For these applications, a one-dimensional processor array can be used in two major modes:

- *Pipeline mode*: each processor constitutes a stage of the pipeline, and data are processed as they flow through the array; repetitive computation can often be decomposed into a number of identical pipeline stages.
- *Parallel mode*: the data are partitioned among the proces-

sors, and each processor performs the same function on data resident in its local memory.

In both the pipeline and parallel modes of operation, the cells execute identical programs. However, unlike the SIMD model of computation, where all the processing elements execute the same instruction in the same time step, the execution time of the cells may be skewed. That is, all cells perform the same computation; however, the computation of a cell is delayed with respect to that of the preceding cell by a constant amount of time. This delay gives valid data and results from preceding cells time to travel to the cell before the computation on the cell begins. We refer to this model, where all the cells execute the same function, but with a time delay between neighboring cells, as the *skewed model of computation.* Some of the optimizations and design decisions for Warp were based on our intended application domain and the model of how the machine would be used.

4.1. Local program control

Although the functions executed by the different cells of the array are identical, each cell has its own local program memory and sequencer. First, since the microinstruction words are very wide, it is not easy to broadcast them to all the cells, or to propagate them from cell to cell. Second, the skewed model of computation can easily be supported with local program control. Finally, the local sequencer supports conditional branching more efficiently. In machines where all cells execute the same sequence of instruction words, branching is achieved by masking. Therefore, the execution time is equivalent to the summation of the execution time of each 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 maximum execution time of the different branches.

4.2. Computing addresses and loop controls in the IU

Since all the cells compute the same function, and the addresses used in the computation are often data independent, the cells typically have identical addressing patterns and control flow. For example, when multiplying two matrices, each cell computes some columns of the result. All cells access the same *local* memory location, which has been loaded with different columns of one of the argument matrices. This characteristic of the computation allows us to factor out address generation and loop termination signals from the individual cells; we migrate these functional blocks to the IU. Moreover, it is desirable that each Warp cell can make two memory references per cycle. To sustain this high local memory bandwidth, the cell demands powerful address generation capabilities, which were expensive to provide and therefore became a target for optimization. We can dedicate much more hardware resources to address generation if it is implemented only once in the IU, and not replicated on all Warp cells.

The IU has separate functional units for address generation and loop control. In each cycle, the IU can compute up to two array addresses, modify two registers, test for the end of a loop, and update a loop counter. In addition, to support complex addressing schemes for important algorithms such as FFT, the IU contains a table of pre-stored addresses; this table can be initialized when the microcode is loaded.

Data dependent addresses are sometimes necessary. They can be computed locally on the Warp cells, but at a significant cost since the address computation competes with other computations for use of the floating-point processing elements.

4.3. Compile-time flow control

In our application domains of signal and image processing, data independent control flow is typical; the same operation is performed on all input data. Hence, a bound on the timing of the input and output actions of each cell can be obtained at compile-time. Our decision is to let the compiler synchronize the actions of the communicating cells; that is, the compiler ensures that no data is fetched from, or stored into, the queues if they are empty, or full, respectively. An alternative design is to provide the flow control at run-time by suspending a cell whenever it tries to read from an empty queue, or write to a full queue, until the status of the queue changes. Providing this capability would make the machine much more difficult to design, implement and debug.

While compile-time flow control is adequate in most of the cases in our application domain, run-time flow control is required in some occasions—for example, in three-dimensional graphics clipping algorithms, the number of points to be created or deleted is determined dynamically. Minimal support is provided to handle these cases: the status of the communication queues is accessible to the user. Therefore, the user can loop and continually test the status before each I/O action until the queue stops being empty or full. We note that although run-time flow control is possible, it is expensive since all the pipes in this heavily pipelined machine need to be emptied before each I/O operation.

4.4. Input control

The latching of data into a cell's queue is controlled by the cell that sends the data, rather than by the local microinstruction word. As a cell sends data to its neighbor, it tags the data with a control signal to instruct the receiving cell's queue to accept the data.

Our original design was that input data is latched under the microinstruction control of the receiving cell. This implies that intercell communication requires close cooperation between the sender and the receiver; the sender presents its data on the communication channel, and in the same clock cycle the receiver must latch in the input. This tight-coupling of cells appeared not to present any additional complexity to the compiler, since the compiler has full knowledge of the timing of all the I/O actions to ensure that no queue overflow or underflow results. We did not realize that such a design would cause an intolerable increase in code size.

To explain the increase in code size if the receiving cell were to provide the control to latch in the data, we first describe the compilation process. Consider a simple example where each cell receives a data item from its left neighbor and passes it to its right. The scheduling steps of the compiler are as follows:

1. Schedule the computation, disregarding the control to latch in the data. This sequence of instructions forms the computation process for a given cell. In the simple example, the computation consists of two steps: remove a data item from the X queue and output it on the X channel for the next cell. See Figure 4-1 (a).
2. Extract the output actions and construct the input counterpart of latching in the data. These input operations are executed by the next cell. As shown in Figure 4-1 (b), all output operations are simply turned into input operations. All the control constructs are retained; all other instructions are mapped into idle cycles (nops). This is the input process for the next cell.
3. Determine the minimum skew between neighboring cells such that no inputs are performed before the corresponding data are output from the preceding cell. (For simplicity, we assume that the queue is sufficiently large). Since the input process must execute in lockstep with the computation of the preceding cell, the computation process in the cell is delayed with respect to its own input process.

   In the example of passing a data item down the array, the queue cannot be dequeued until the data are output by the preceding cell. Therefore, the skew between cells in this case is 1. The result of merging the input and the delayed computation is shown in Figure 4-1 (c).

```
(a)          (b)         (c)

dequeue(X).  nop.        nop.
output(X).   input(X).   dequeue(X), input(X).
                         output(X).
```

Figure 4-1: Scheduling a cell program that passes data from cell to cell: (a) computation process, (b) input process, and (c) merged process

The problem of codesize increase shows up in the last step where the computation and the input processes are merged. The code length of the program that combines both the computation and input activities can be significantly greater than either of the original processes if they contain iterative statements. Merging two identical loops, with an offset between their initiation times, requires loop unrolling and can result in a three-fold increase in code length. Figure 4-2 illustrates this increase in code length when merging two identical loops of *n* iterations. Furthermore, if the offset is so large that 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.

Having the sender control the latching of input data not only reduces the complexity of the compiler and avoids increasing the code size, but also allows us to handle conditional statements better. For example, if a data item is output to the next cell in each of the two branches of a conditional statement, the compiler needs only to ensure that the input action of the receiving cell is performed after the output actions of both branches. The data from each branch of the condition need not be sent out in the very same clock cycle.

4.5. Control path

The microcode for the Warp cell and the IU is completely horizontal. The user has full control over every component, and thus can tailor the schedule of the micro-operations according to the needs of specific applications. Each component 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 caused by conflicts in the micro-instruction field assignment.

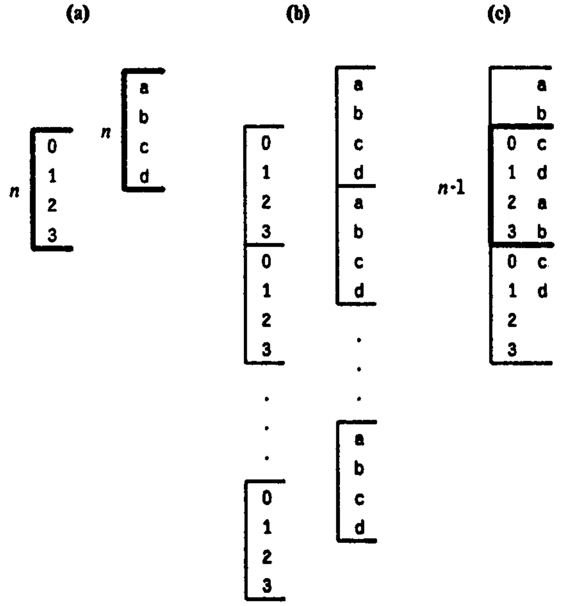

Figure 4-2: Merging two iterative processes (a) original programs (b) execution trace, and (c) merged programs

4.6. Data path

The data path of the Warp cell, shown in Figure 2-2, is carefully designed to ensure that the arithmetic units can be fully utilized. We now discuss each of the components in detail.

4.6.1. Arithmetic units

Each Warp cell has two floating-point processors, one multiplier and one ALU [9]. Floating-point capabilities are necessary for signal processing and scientific computation, and we believe that they will be important in vision as well. Until now most vision computation is done using fixed-point arithmetic, and this has meant careful analysis of algorithms to find upper bounds for the magnitude of intermediate results. If the intermediate results exceed the boundaries imposed by the word size available, the data must be scaled, or the algorithm must be modified. Using floating-point arithmetic, none of this analysis is necessary, so that the vision researcher can concentrate on the algorithmic aspects of the computation.

The convenience of using floating-point arithmetic does not imply a high cost in hardware implementation because high-performance floating-point arithmetic chips are commercially available. However at the time of design for the Warp machine, these floating-point chips depended on extensive pipelining to achieve high performance. Specifically, both of the ALU and the multiplier have 5-stage pipelines [9]. Deep pipelining has a devastating effect on the performance of general-purpose computation, where data-dependent branching is common. Fortunately, our applications include few conditional branches and consist primarily of tight data-independent loops. Previous results have shown that pipelining is an effective optimization for systolic array algorithms [5]: while pipelining reduces the cycle time of a machine, the throughput rate of results per cycle can often be maintained.

4.6.2. Internal data bandwidth

Experience with the programmable systolic chip [1] showed that the internal data bandwidth is often the bottleneck of a systolic cell. In the Warp cell, the two functional 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; 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: conflicts on the use of one or more shared busses can complicate scheduling tremendously.

4.6.3. Data storage

The local memory hierarchy includes a local data memory and two register files, one for each functional unit. The local data memory can store 4K words of input data or intermediate results, and can be both read and written every (200 ns) cycle. The register files are 4-ported data buffers capable of accepting two data items and delivering two operands to the functional units every cycle.

4.6.4. Queues

Delay elements are often used in systolic algorithm designs to synchronize or delay a data stream with respect to another. In the original design of Warp, the queues were intended to serve simply as programmable delays; data were latched in every cycle and they emerged at the output port of the queue a constant number of cycles later. During the compiler development, the inadequacy of programmable delays was discovered. The Warp cell is a programmable processor with a relatively large local data storage; it is much more flexible than the typical inner-product nodes that appear in many systolic algorithms. Often, Warp programs do not produce one data item every cycle; a clocking discipline that reads/writes one item per cycle restricts the use of the machine. Furthermore, a constant delay through the queues means that the timing of the production of results must match exactly the timing of the consumption of data. Therefore, the architecture was modified to allow the user to conditionally enqueue or dequeue the data as they are produced or consumed.

The size of a queue is 128 words. This size was determined by the available RAM chips with sufficient speed to implement the queues. The queues are not large enough for vision algorithms that require buffering entire rows of images, which are at least 256 elements wide. The overflow data have to be stored in the local memory of a Warp cell in the current implementation.

The overflow problem for the address queue is much more severe than that for the data queue. First, while it is possible to use the memory as an extended storage for overflow data, this technique does not apply to addresses, as addresses are needed to access the memory. Second, the size of the address queue severely limits the grain size of parallelism in programs executable on Warp. In the pipeline mode of operation, a cell does not start executing until the preceding cell is finished with the first set of input data. The size of the data queue needs to be at least equal to that of the data set; the size of the address queue, however, needs to be at least equal to the number of addresses and control signals used in the computation of the data set. This problem would not exist if the Warp cells had the capability to generate addresses effectively.

Architectural revisions to deal with the above problems will be included in the PC board implementation of the Warp cell.

4.7. Compilation

While the parallelism potentially available in this machine is tremendous, the complexity of using it effectively is correspondingly overwhelming. To make the machine truly programmable, an optimizing compiler is developed [2]. The programmer simply thinks of the machine as a uni-directional array of sequential processors, communicating asynchronously. A sample program that implements the evaluation of a polynomial on an array of ten cells is included in the appendix. Since all cells perform the same function in our programming model, the user simply supplies the program for one cell. The programming language is Algol-like, with additional receive and send statements as communication primitives. The semantics of the communication protocol is that a cell is blocked when it tries to read from an empty queue or write to a full queue.

The compiler divides the operations in the cell program into three parts, one for each of the different components of the system: the host, the IU and the Warp cells. The input and output actions with the external environment are performed on the host and the IU. The data independent addresses and loop control are computed by the IU and are sent to the Warp cells via the address path. The rest of the operations are performed on the Warp cells.

The compiler currently accepts only programs where data flow uni-directionally through the array; flow control between cells is achieved by skewing the execution of the cells by the necessary amount to ensure that the input data is in the queue before it is used. Overflow of the queue is detected and reported, but not currently handled. The high degree of pipelining in the functional units of the Warp cell is another cause of concern for the compiler. Scheduling techniques based on software pipelining techniques are used [7]. The utilization obtained is reasonable; as an example, full utilization of the functional units is achieved for the sample program of polynomial evaluation in the appendix.

5. Implementation

5.1. Implementation of the Warp cell

The Warp architecture calls operates on 32-bit data. In the implementation, all data channels in the Warp array, including the internal data path of the cell (except for the floating-point processors), are implemented as 16-bit wide channels operating at 100 ns. There are two reasons for choosing a 16-bit time-multiplexed implementation. First, a 32-bit wide hardware path would not allow implementing one cell per 15" × 17" board. Second, the 200 ns cycle time dictated by the Weitek floating-point processors (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 were under 160 ns.

High speed and parallel cell operations coupled with the tight synchronization of the Warp array presented a challenge to logic design. In particular, the design is required to address sensitive timing issues. Most of them were dealt with by adding pipeline stages to critical paths. Given that the floating-point units impose a multi-stage pipeline, the addition of yet another stage seems to be a good tradeoff when compared to added design complexity, increased chip count, and timing problems. To achieve high programmability without degrading performance, the timing of the control path must be balanced to the timing of the data path. Therefore, the microengine also operates at 100 ns and supports high and low cycle operations of the data path separately.

The cell is controlled by a pipelined microengine with a 112-bit wide microinstruction. Logically related micro-operations are typically grouped into the same microinstruction, although some data path element's require that control signals are presented at later cycles. For example, to initiate a multiplication, the microinstruction specifies the opcode and two source registers. First, the operands are retrieved from the register file; then, a cycle later, the opcode is fed to the multiplier unit. These delay adjustments are made in hardware using pipeline delays rather than in software to make code generation easier. However, the programmability of the microengine suffered from the limitations of the available microsequencer (Am2910A), especially in implementing nested loops.

A major portion of the internal cell hardware can be monitored and tested using built-in serial diagnostic chains. The serial chains access 36 registers of the Warp cell, covering almost all the registers outside the register files. This feature, along with the fully static operation of the cell, made testing easier and led to a fast debug time. In the best case, a wire-wrapped board was populated, debugged, and checked out in five days. The serial chains are also used to download the Warp cell programs. Initializing the Warp array with the same program for all the cells takes no more than 100 ms.

The Warp cell consists of five main blocks: input queues, crossbar, processing elements, data memory and microengine. Table 1 presents the contribution of these blocks to the implementation of the prototype with 4K-word data memory. The Warp cell contains 5422 pins in 6525 nets, and it consumes 94W (typ.) and 136W (max.).

Part	Chip count	Contribution (%)
Queues	63	25.1
Crossbar	32	12.7
Processing elements	8	3.1
Data memory (4K words)	30	11.9
Micro-engine	79	31.4
Other	40	15.8
Total for the Warp cell	252	100.0

Table 5-1: Metrics for Warp cell

5.2. Implementation of the IU

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 reduces the number of transfers. Data transfer to and from the IU may be controlled by interrupts; in this case, the IU always behaves like a slave device. During data transfers, the IU can convert 8-bit or 16-bit integers from the host into 32-bit floating-point numbers for the Warp array, and vice versa.

The IU generates addresses and control signals for the Warp array. It supplies all the necessary signals to drive the cells of the Warp array, requiring little control and data bandwidth on the host side. For the address generation, the IU has an integer ALU capable of generating two addresses every 200 ns. All these operations are 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 memory space; the host can control the IU and hence the Warp array by setting these registers.

As each Warp cell, the IU is implemented on a single 15"×17" board. There are 5692 pins and 6170 nets on the board. The IU has a power consumption of 82W (typ.) and 123W (max.). Table 5-2 presents implementation metrics for the IU.

Part	Chip count	Contribution (%)
Data-converter	52	18.9
Address generation	64	23.2
Clock and host interface	74	26.9
Micro-engine	44	16.1
Other	41	14.9
Total for the IU	275	100.0

Table 5-2: Metrics for the IU

The IU generates the 20 MHz clock and plays a special role in system diagnostics; it generates the diagnostic signals for the Warp array. The host has total control of the clock generator and can run the Warp array in step mode. After each step, most of the internal state of the array can be monitored using the built-in serial diagnostic chains. The IU can reach the serial chains of each Warp cell separately. This allows the debugging of each Warp cell in the array independently from any other cell.

6. Host system

Input and output are critical issues for any high-performance architecture. At its peak I/O rate, the Warp array can communicate with the outside world at the rate of 40 Mbytes/sec through the IU. This is a stringent requirement for a typical off-the-shelf, microcomputer-based system.

The system must also be able to execute those parts of an application that do not map well onto the Warp array, and to coordinate all the peripherals, which include the Warp array, digitizers, graphics displays, and similar devices. Moreover, for vision applications, the system must have a large memory to store images. These images are fed through the Warp array, and results must be stored in memory until used again. In addition, for vision research at Carnegie Mellon, it is crucial that the Warp machine exists within a UNIX environment.

These goals are achieved by integrating the Warp array with a host. This distinguishes Warp from other high-performance machines that define the interface close to the special-purpose hardware and leave the problem of system integration to the user.

6.1. Host architecture

Integrating Warp into a UNIX environment is accomplished by partitioning the host into a standard workstation supporting UNIX and an "external host." The workstation is also *master* and controls the external host. Figure 6-1 presents an overview of the Warp host.

The external host consists of two *cluster processors*, a sub-system called *support processor* and graphics devices. The external host is built around a VME bus. The two cluster processors and the support processor (all Motorola 68020 microprocessors) have dual-ported memories, with a private access on a local VMX32 bus and a shared access on the global VME bus. These three processors run stand-alone. Two *switch boards* allow the clusters to send and receive data to and from the Warp array, through the IU. Each switch has also a VME interface, used for debugging purposes and to start/stop/control the Warp array. The master (the workstation processor) is connected to the external host via a VME bus-coupler. All the boards with the exception of the switch are off-the-shelf components.

Application programs execute on the workstation and invoke functions for execution on the Warp array. The external host acts as intermediary in the interaction between workstation program and Warp array. Control of the external host is strictly centralized: the master issues commands to all external processors through message queues that are local to each external processor.

The support processor controls peripheral I/O devices and handles floating-point exception 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. This organization does not conflict with the fundamental paradigm of centralized control inside the host.

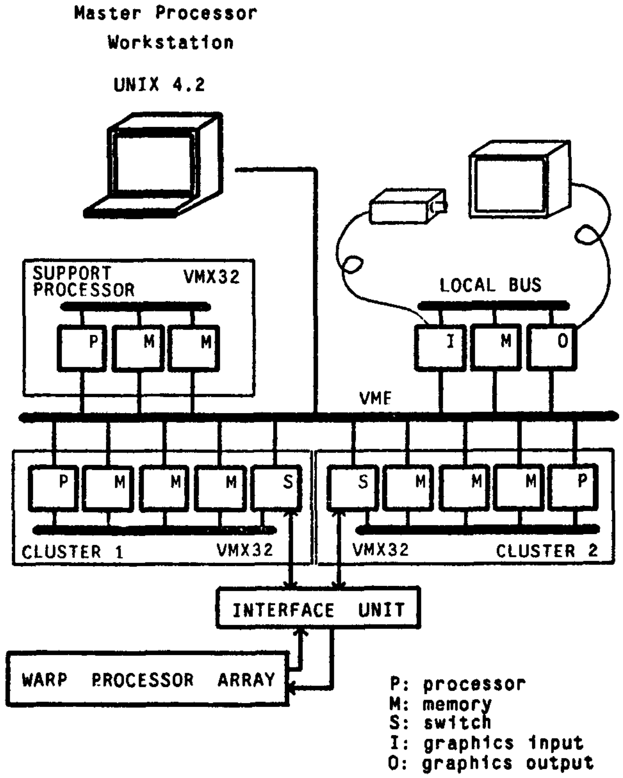

Figure 6-1: Host of the Warp machine

During computation, two clusters work in parallel, each handling a uni-directional 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. When the two clusters exchange their roles, there could be a danger that both clusters write to, or read from, the Warp array at the same time. To prevent this from happening, an arbitration mechanism transparent to the user has been implemented.

There are three memory banks inside each cluster processor to provide storage as well as efficiency. For example, the first memory bank may be receiving a new set of data from an I/O device, while the second contains previous data to be transferred to the Warp array, and the third has the cluster program code. This arrangement allows overlapping the transfer of an image to the Warp array with the acquisition of the next image. Presently, the total memory in the external host is 8 Mbytes, and it can be upgraded to 16 Mbytes. In the current configuration, physical constraints limit the total storage to 36 Mbytes.

6.2. Host I/O bandwidth

To meet the stringent I/O bandwidth required by the Warp array, the host design exploits the fact that input/output data for signal, image and low-level vision processing are usually 8-bit or 16-bit integers. Because the data items in the host are either 8- or 16-bit integers, they can be *packed* into 32-bit words before being transferred to the IU. These 32-bit words are unpacked and expanded by the IU into four or two 32-bit floating-point numbers, before being shipped to the Warp array. The opposite process takes place with the floating-point outputs of the Warp array. Therefore, the data bandwidth requirement between the host and IU is reduced by a factor of 4 or 2, respectively. This I/O bandwidth reduction allows the Warp processor array to be fully utilized under the present microcomputer-based system, without having to rely on other custom-made hosts with higher I/O bandwidth capabilities. For example, if the input/output data are 8-bit pixels, then to support the peak Warp I/O bandwidth via the IU (40 Mbytes/sec) the host needs only to transfer 10 Mbytes/sec to and from Warp. This transfer rate can be sustained with two clusters each capable of handling a 5 Mbytes/sec transfer rate, which is within their current capabilities.

6.3. Host software

The Warp host features a run-time software library that allows the programmer to synchronize the support processor and two clusters and to perform the allocation of the memories in the external host. The run-time software handles also communication and interrupts between the master and the processors in the external host. The library of run-time routines includes utilities such as "copy", "move", windowing, and peripheral device drivers. The compiler generates program-specific input and output routines for the clusters so that a user needs not be concerned with programming at this level; these routines are linked at compile time to the two cluster processor libraries.

The application program usually runs on the master; however, it is possible to assign certain sub-tasks (e.g., finding the maximum in an output vector from the Warp array) to any of the external processors. This increases parallelism and boosts performance, since processing can now take place locally without transferring data from the external host to the master. Each transfer from the Warp array to the master processor memory passes through the VME bus repeater, which roughly doubles the transfer time (compared to direct VME access).

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 indicated. This is necessary for reasons of efficiency. In fact, the programmer must be aware of the different memory banks, that can at any time be accessed simultaneously by different processors through different buses, thus achieving full parallelism.

Conditional execution of tasks inside each stand-alone processor is provided by *conditional requests* issued by the master processor. In other words, any task queued into the queue of each processor is always unconditionally executed; it is the issuing of the specific command that must satisfy a certain condition. A typical example is the execution of a certain task conditioned by the completion of a different task in a different processor: one call in the application code to the run-time software achieves this purpose.

7. Concluding remarks

Some of the Warp design decisions are influenced by the technology we use to implement the machine. These design choices must be re-evaluated if a different technology is used. For example, in the Warp architecture, the cells mostly rely on the IU to generate the addresses and loop controls. If the Warp cells are implemented in custom VLSI, however, the external data path for passing addresses through the cells is far more expensive than an on-chip address generation unit.

Direct hardware support for flow control should be considered. Currently, this functionality is provided in software; the compiler prevents a cell from reading from an empty queue, or writing into a full queue. Compile-time synchronization makes the hardware design easier; however, it complicates the compiler and limits the applicability of the machine.

The Warp project is not complete at this time, but our experiences so far support four conclusions. First, an early identification of the application area is important for experimental special-purpose machines with radically different architectures. 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.

Second, software support tools are crucial for a programmable systolic machine. A compiler development effort contributes in two ways: it makes user programming easier and provides feedback on the architecture. To write efficient code for such a machine by hand is time-consuming, error-prone and requires a detailed understanding of the implementation. Only a compiler can free the user from the management of low-level details. Developing a compiler also benefits the architecture. Designing and implementing a compiler requires a thorough study of the functionality of the machine; this analysis uncovers the sources of real problems. Furthermore, a compiler enables the designer to get a larger set of test programs to validate architecture and implementation decisions. It is our conclusion that the compiler component is essential for architecture development.

Our third conclusion is that the integration of a special-purpose machine in an open and general purpose host environment is essential. The Warp host provides flexible I/O facilities as well as the UNIX programming environment, which are crucial to the usability of the machine. Moreover, early recognition of the importance of system integration allowed us to balance the I/O bandwidth of the host with the computation power of the Warp array.

Lastly, we conclude that the linear array configuration is suitable for most applications in signal processing, low-level vision, and scientific computation. A linear array with large local memory and powerful cells is efficient for global operations as well as for local operations. The Warp cell entails a high degree of pipelining and parallelism which needs to be managed explicitly. Our experience demonstrates that our compiler can produce effective code for the programs in our application domain.

Warp is a powerful and usable machine. Even with a conservative implementation, the current machine delivers a peak performance of 100 MFLOPS. The host system makes the Warp array accessible to users; it provides an environment for application development, and sustains the Warp cells with an adequate data rate. Program development for Warp is easy since the Warp compiler produces efficient code.

Acknowledgments

We appreciate the contributions to the Warp project by our colleagues at Carnegie Mellon: D. Adams, C. Bono, C. Chang, E. Clune, R. Cohn, P. Dew, J. Deutch, B. Enderton, P.K. Hsiung, K. Hughes, T. Kanade, G. Klinker, P. Lieu, R. Mosur, H. Ribas, J. Senko, Y. Shintani, B. Siegell, P. Steenkiste, Y-B Tsai, R. Wallace, and J-K Wang. We appreciate technical assistance and corporate support from General Electric and Honeywell, our industrial partners for the project. In particular, we thank A. Lock and C. Walrath of General Electric and L. Johnson and D. Kaylor of Honeywell, who spent four months at Carnegie Mellon working with us during the first half of 1985.

I. Sample program

In the sample program below, most statements have their obvious meaning. **Send** and **receive** transfer data between adjacent cells. The first parameter determines the direction of the transfer, and the second parameter selects the hardware channel to be used. The remaining two parameters bind the source (for send) or result (for receive) to local variables and formal parameters.

```
/*******************************************/
/*          Polynomial evaluation          */
/*  A polynomial with 10 coefficients is   */
/*evaluated for 100 data points on 10 cells*/
/*******************************************/

module polynomial (data in, coeffs in,
                                results out)
float data[100], coeffs[10];
float results[100];

cellprogram (cid : 0 : 9)
begin

    function poly
    begin
        float coeff, xin, yin, ans;
        int i;

/*Every cell saves the first coefficient that
reaches it, consumes the data and passes the
remaining coefficients. Every cell generates
an additional item at the end to conserve the
number of receives and sends.              */

         receive (L, X, coeff, coeffs[0]);
         for i := 1 to 9 do begin
             receive (L, X, temp, coeffs[i]);
             send (R, X, temp);
         end;
         send (R, X, 0.0);

/* Implementing Horner's rule, each cell
multiplies the accumulated result yin with
incoming data xin and add the next
coefficient                                */

         for i := 0 to 99 do begin
            receive (L, X, xin, data[i]);
            receive (L, Y, yin, 0.0);
            send (R, X, xin);
            ans := coeff + yin*xin;
            send (R, Y, ans, results[i]);
         end;
    end

    call poly;
end
```

References

1. Fisher, A.L., Kung, H.T., Monier, L.M. and Dohi, Y. "The Architecture of a Programmable Systolic Chip". *Journal of VLSI and Computer Systems 1*, 2 (1984), 153-169. An earlier version appears in *Conference Proceedings of the 10th Annual Symposium on Computer Architecture*, Stockholm, Sweden, June 1983, pp. 48-53.

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

3. Gross, T., Kung, H.T., Lam, M. and Webb, J. Warp as a Machine for Low-level Vision. Proceedings of 1985 IEEE International Conference on Robotics and Automation, March, 1985, pp. 790-800.

4. Kung, H.T. "Memory Requirements for Balanced Computer Architectures". *Journal of Complexity 1*, 1 (1985), 147-157. (A revised version also appears in Conference Proceedings of the 13th Annual International Symposium on Computer Architecture, June 1986)..

5. Kung, H.T. and Lam, M. "Wafer-Scale Integration and Two-Level Pipelined Implementations of Systolic Arrays ". *Journal of Parallel and Distributed Computing 1*, 1 (1984), 32-63. A preliminary version appeared in *Proceedings of the Conference on Advanced Research in VLSI*, MIT, January 1984.

6. Kung, H.T. and Webb, J.A. Global Operations on the CMU Warp Machine. Proceedings of 1985 AIAA Computers in Aerospace V Conference, American Institute of Aeronautics and Astronautics, October, 1985, pp. 209-218.

7. Rau, B. R. and Glaeser, C. D. Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. Proc. 14th Annual Workshop on Microprogramming, Oct., 1981, pp. 183-198.

8. Wallace, R., Matsuzaki, K., Goto, Y., Crisman, J., Webb, J. and Kanade, T. Progress in Robot Road-Following. Proceedings of 1986 IEEE International Conference on Robotics and Automation, April, 1986.

9. Woo, B., Lin, L. and Ware, F. A High-Speed 32 Bit IEEE Floating-Point Chip Set for Digital Signal Processing. Proceedings of 1984 IEEE International Conference on Acoustics, Speech and Signal Processing, 1984, pp. 16.6.1-16.6.4.