Compilation for a High-performance Systolic Array

Thomas Gross and Monica S. Lam

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

We report on a compiler for Warp, a high-performance systolic array developed at Carnegie Mellon. This compiler enhances the usefulness of Warp significantly and allows application programmers to code substantial algorithms.

The compiler combines a novel programming model, which is based on a model of skewed computation for the array, with powerful optimization techniques. Programming in W2 (the language accepted by the compiler) is orders of magnitude easier than coding in microcode, the only alternative available previously.

1. Introduction

In systolic arrays, the flow of data through the cells is planned cycle by cycle so that cells never have to wait for input and can compute at their maximum rates. As a result, systolic algorithm design is not easy. Existing systolic algorithms and design tools are directed at simple machine models where all cells perform simple, identical, atomic computations. That is, the cells input a set of data, compute and output the result in each clock cycle. The domain of these simple models is restricted to mathematical recurrences, which are amenable to simple algebraic manipulation.

Carnegie Mellon is currently building a high-performance, programmable systolic machine, known as Warp; each of its ten cells contains a 4K-word memory and can deliver up to 10 million floating-point operations per second [3]. The machine's application domain extends far beyond simple recurrences, which have been the focus of earlier systolic designs. The complexity of the machine as well as the application domain cause previous techniques, be they manual or automatic, to break down.

Our solution to the problem of compilation for high-performance systolic arrays has three aspects: a new computation model for systolic arrays, a programmer's model, and compilation techniques that translate algorithms expressed in terms of the programmer's model to our computation model for the machine. A compiler based on these concepts has been implemented for Warp.

In this paper, we first present an overview of the Warp machine. Next, we describe our computation model, which we call the *skewed computation model*. We then introduce our programmer's model and describe the compilation techniques that allow us to compile user programs for Warp. We conclude the paper with some preliminary results on the performance of the Warp compiler.

2. Warp architecture

Warp is a high-performance systolic array computer designed by Carnegie Mellon for signal and image processing, and scientific computing [3]. This machine has been named *Warp* because of its high speed and throughput. 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. It can also be programmed to perform many other primitive computations, such as two-dimensional convolution, and real or complex matrix multiplication, at a peak rate of 100 million floating-point operations per second. A two-cell prototype has been operational since the Fall of 1985, and the first 10-cell machine was delivered early 1986; at least eight additional machines will be built during the next two years.

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. Thomas Gross is also supported by an IBM Faculty Development Award.

2.1. Warp machine

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 host is a general-purpose workstation (currently a SUN) running UNIX and provides an adequate data bandwidth to sustain the array at full speed in the targeted applications. The Warp array provides the raw computation power for performing computation-intensive routines. It is a 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 a programmable horizontal microengine, with its own local program memory and sequencer. The host controls the Warp array and executes those parts of the application code that cannot be mapped onto the Warp array. The IU connects the Warp array and the host; it also generates addresses and control signals for the Warp array.

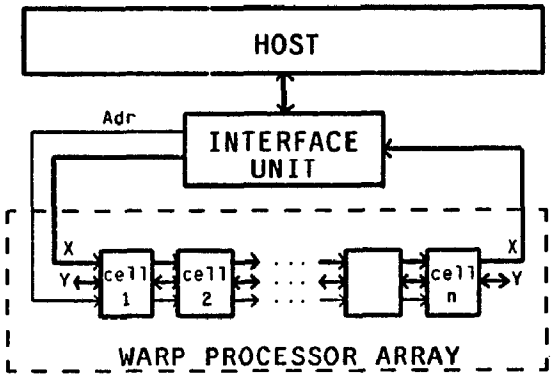

Figure 2-1: Warp machine overview

In the following, we examine the different sources of complexity that must be mastered by a user of this architecture. We describe the interaction between the three components of the machine, the cooperation between the cells in the processor array, and lastly, code generation issues for the Warp cells.

2.2. System issues

Communication between the host and the processor array requires code to be generated for all three components of the system. The IU and the host communicate asynchronously via a standard bus. The I/O processors in the Warp host must be programmed to supply input in the exact sequence as the data is used in the Warp cells. The IU and the processor array communicate synchronously; the IU must be programmed correctly so that data from the host reach the Warp cells in time.

In addition to interfacing the Warp array to the host, the IU is also responsible for generating addresses and loop control signals for the cells. Warp cells are not equipped with integer arithmetic capability. The justification is that systolic cells typically perform identical, data-independent functions, using identical addressing patterns and loop controls. 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. Therefore, common addresses and loop control signals can be generated externally in the IU and propagated to all the cells. 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 also supplies loop control signals to the Warp cells. When the cells execute a loop, the IU sends a signal at the end of each iteration to indicate if another loop iteration is to be executed. Since addresses and loop controls are integral to any computation, the actions on the IU and the Warp cells are strongly coupled.

2.3. Systolic array issues

A feature that distinguishes Warp from other processors of similar computation power is its high I/O bandwidth. Adjacent cells can transfer up to 20 million words (80 Mbytes) per second. This high inter-cell communication bandwidth makes it possible to transfer large volumes of intermediate data and thus supports fine-grain problem decomposition.

The problem of mapping a computation onto a systolic array is an ongoing research topic. Except for very simple mathematical recurrences, automatic tools do not exist. Furthermore, in most cases, insight into the application domain is necessary to partition the computation properly across an array.

2.4. Cell issues

While the parallelism potentially available in this machine is tremendous, the complexity of using it effectively is correspondingly overwhelming. Each Warp cell can be likened to a conventional array processor. Its data path is 32-bit wide and is depicted in Figure 2-2. It contains two floating-point units, both of which are 5-stage pipelined. Each cell contains 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. These storage blocks can supply four words of data per cycle; this rate matches exactly the consumption rate of the arithmetic units. All the functional units are interconnected by a full crossbar, and are controlled individually by separate fields in wide micro-instruction words (over 200 bits).

All cells operate in lock step under control of a global clock. There is no hardware support for dynamic flow control between cells. As a result, the I/O operations of the cells in the array need to be synchronized at compile-time. Although a buffer exists between each I/O link of a pair of communicating cells, the status of the buffer cannot be tested at run time. Therefore, the computation on all cells needs to be scheduled such that no data is removed from the channel buffer before it is sent by the sender of the data.

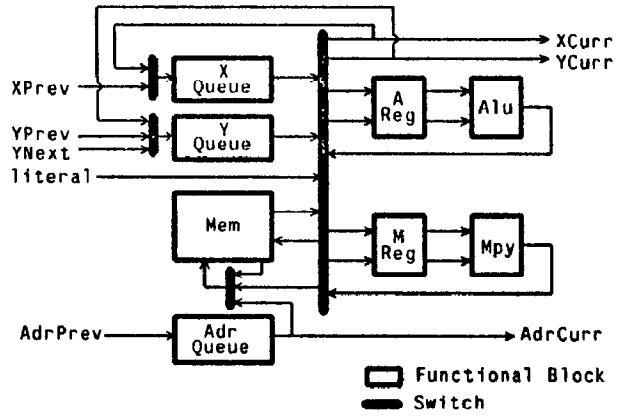

Figure 2-2: Warp cell data path

3. Skewed computation model

After describing the architecture, we now define our computation model of the machine, that is, the compiler's target computation model. In previous systolic machine models, cells latch in data, compute, and output the result for their neighboring cells in one single cycle. In most algorithms, all the cells in the array perform identical functions.

An obvious way to extend this simple model to cover Warp, where each data set may consist of multiple data items and the computation per data set may span multiple clock cycles, is to adopt the SIMD computation model. All cells input a data set in the same cycle, perform an identical sequence of computations, and then output the results for the neighboring cells simultaneously. However, this SIMD model is too restrictive and unattractive to implement. We define a new computation model, which we call the *skewed computation model*. In this model, cells in an array execute the same function, but possibly with a time delay between neighboring cells. To show that the skewed computation model supports an easier and more efficient use of the systolic array than the SIMD model, we first observe that a one-dimensional systolic array is normally used in two modes:

- *Pipeline mode*: each processor constitutes a stage of the pipeline, and data are processed as they flow though the array; repetitive computation can often be decomposed into a number of identical pipeline stages.
- *Parallel mode*: the input data are partitioned among the processors, and each processor performs the same function on data resident in its local memory.

Implementing the pipeline mode on this skewed computation model is natural; the execution of consecutive processors is simply skewed by the amount of time it takes for data to pass through a stage. The data dependency between consecutive stages of the pipeline makes it impossible to initiate useful computation on all the cells simultaneously. To implement this pipeline mode in the SIMD model, one can pretend that input exists for all stages of the pipeline from the very start. The programmer, however, needs to initialize the state of the cells so that no exception is raised on "fictitious" data before the actual data arrive; he also has to provide extra data to flush the pipeline to retrieve the last valid results. Alternatively, the programmer or additional special hardware can mask off the execution or just the exception signalling mechanism on cells during the invalid data period. In either case, the programmer is still left with the tedious task of determining which of the outputs of the array constitutes the desired results.

Implementing a pipeline in the SIMD model is inefficient. The latency of results through each stage in the SIMD model is determined by the computation time of the entire stage. In the skewed model of computation, the latency of each cell is given by the time skew between the cells; this time skew is determined by the minimum lead time a stage needs such that results for the next cell are ready by the time they are used. Consider a simple example program where each stage takes 4 steps and that the fourth step requires the result from the fourth step of the previous stage. The latency through each cell is 4 cycles in the SIMD model, but only one cycle in the skewed model, as illustrated in Figure 3-1. This difference in latency can be significant when a nontrivial amount of computation is involved in each stage.

SIMD computation model			**Skewed computation model**		
CELL 1	CELL 2	CELL 3	CELL 1	CELL 2	CELL 3
$step_1$			$step_1$		
$step_2$			$step_2$	$step_1$	
$step_3$			$step_3$	$step_2$	$step_1$
$step_4$			$step_4$	$step_3$	$step_2$
$step_1$	$step_1$		$step_1$	$step_4$	$step_3$
$step_2$	$step_2$		$step_2$	$step_1$	$step_4$
$step_3$	$step_3$		$step_3$	$step_2$	$step_1$
$step_4$	$step_4$		$step_4$	$step_3$	$step_2$
$step_1$	$step_1$	$step_1$	$step_1$	$step_4$	$step_3$
$step_2$	$step_2$	$step_2$	$step_2$	$step_1$	$step_4$
$step_3$	$step_3$	$step_3$	$step_3$	$step_2$	$step_1$
$step_4$	$step_4$	$step_4$	$step_4$	$step_3$	$step_2$
	$step_1$	$step_1$		$step_4$	$step_3$
	$step_2$	$step_2$			$step_4$
	$step_3$	$step_3$			
	$step_4$	$step_4$			
		$step_1$			
		$step_2$			
		$step_3$			
		$step_4$			

Figure 3-1: Comparing latencies between SIMD and skewed computation models

The skewed computation model is useful for the parallel mode of operation as well. All the processors receive their own distinct set of input data. In the SIMD model computation cannot start until all the data are ready for all the cells. In the skewed model, we can initiate the computation in each cell as soon as its input demand is satisfied, thus reducing the latency of the computation.

4. Programmer's model

There is one central goal in the design of the programmer's model: to abstract out all unnecessary details of the architecture. We want to support a programmer's model as simple as possible; however, programs expressed in this model must be amenable to effective automatic techniques that produce efficient code for our computation model of the machine. For example, as discussed earlier, the general problem of decomposing a computation onto a processor array is difficult and usually benefits from insight of the application domain. Therefore, this problem is left to the user by exposing the processor array configuration in the programmer's model. On the other hand, complexities at the system and the cell level, as discussed in Section 2, are removed from the programmer's model. In fact, not only does this simplified view make programming easier, automatic techniques can produce better code because of the vast amount of machine-level details involved. We have defined a language, *W2*, which allows the user to describe Warp programs in this programmer's model. In the following, we describe the features of our programming model in more detail, and we introduce our specialized language constructs for expressing concepts in this model.

4.1. Asynchronous communication

The most significant deviation of our programmer's model from the conventional view of systolic arrays is asynchronous communication. In previous systolic models, data communicated between cells is identified by the clock cycle the transfer takes place. As Warp's application domain includes far more complicated cell programs, it is impossible for the user to specify the exact timing of the I/O unless he programs at the microcode level. The alternative of allowing user-defined logical clocks has also been investigated. However, this logical clock approach suffices only if cell programs consist of a simple input-compute-output loop; furthermore, it makes code optimization difficult.

Our programmer's model does not include the notion of time. The user specifies the data dependency relationships between cells using asynchronous "receive" and "send" constructs. Connecting each pair of cells are two I/O channels, X and Y, with a dedicated data buffer. To communicate, the sender processor "sends" a data item to a designated channel, and the data item is stored into the buffer of the recipient; when the receiving processor issues a "receive" operation, the oldest data item in the buffer is returned. Therefore, "send" and "receive" operations on a channel are matched by virtue of the ordering; i.e. the nth "receive" operation will get the data item transferred in the nth "send" operation. The semantics of the communication primitives is that the receiver is blocked if the buffer is empty, and similarly, the sender is blocked if the buffer is full. In Section 5, we will describe how we can bridge the semantic gap between the programmer's model and our synchronous machine.

4.2. System abstraction

In the programmer's model, Warp is a regular one-dimensional array; both the host and the IU are hidden from the programmer. Address computation and loop controls are an integral part of a program, it is too tedious if a user must explicitly input into the processor array each and every calculated address. Therefore, addresses and loop controls generation to be performed on the IU are automatically extracted from the user's specification. Similarly, although the processor array, the IU and the host all must participate in transferring data between the host and the array, this partitioning of function is not present in the programmer's model.

4.3. W2 language

W2 is the "machine language" of the Warp machine; the microcode level is not visible to the user, and it is conceivable that higher level languages be built on top of this W2 level. The language is designed to reflect the constructs that can be executed efficiently by the machine. For example, cells can only communicate with their neighbors on either of the two channels between the cells. It lacks generality and high level programming support found in other languages. For example, it differs from CSP (communicating sequential processes) [4] in that it does not have guarded commands and the more basic, unbuffered communication primitives. These features cannot be implemented efficiently, if at all, on the hardware.

The W2 language is a simple block-structured language with assignment, conditional, and loop statements. Communication between cells is made explicit by the use of the send and receive primitives. The receive primitive has four parameters: the direction of the channel, the channel name, an internal variable the data is received into and, lastly, the external (host) variable that is received by this operation. The first cell of the Warp array receives data directly from the host through the IU, and the value is explicitly specified by the external variable; all other cells receive the data transferred in the corresponding send operation of the communicating cell. In other words, the external variable is meaningful only for the first cell. Similarly, the send primitive has the same parameters, except that the internal variable contains the data to be sent and the external variable specifies the location the data is to be stored in the host memory. Again, the external variable only applies to the boundary cell of the array (in this case, the last cell), which sends data directly to the host through the IU.

Figure 4-1 shows a simple example of a Warp program which evaluates a polynomial using an array of ten cells. The program evaluates the polynomial

$$P(z) = c_0 z^9 + c_1 z^8 + \ldots + c_9$$

for a vector of input data $z_0, z_1, z_2, \ldots$. Each cell (starting with cell 0 up to cell 9, the last cell in the system) executes a copy of the program. The first cell receives the values of the host program variables (bound to parameters `c` and `z`), and the results are sent and stored in a host variable bound to parameter `results`.

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

module polynomial (z in, c in, results out)
float z[100], c[10];
float results[100];

cellprogram (cid : 0 : 9)
begin

    function poly
    begin
       float coeff,/* local copy of c[cid] */
             temp,
             xin, yin, ans; /* temporaries */
       int i;

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

         receive (L, X, coeff, c[0]);
         for i := 1 to 9 do begin
             receive (L, X, temp, c[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, z[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
```

Figure 4-1: A sample program: polynomial evaluation

The program of Figure 4-1 is best explained by observing the first couple of iterations on the first two cells. An arrow in Figure 4-2 indicates that the output of the "send" is input for the corresponding "receive" operation.

Figure 4-2 shows the logical sequence of steps only; it does not imply that there is a single statement per cycle. In fact, several of the statements can be executed in parallel, but the compiler will always preserve the data dependencies depicted in Figure 4-2.

```
Cell 0                          Cell 1
-------------------------------------------------
Receive coeff   c[0]
Receive temp    c[1]
Send temp       c[1]  →  Receive coeff  c[1]
Receive temp    c[2]
Send temp       c[2]  →  Receive temp   c[2]
Receive temp    c[3]     Send temp      c[2]
.
.
.
Receive temp    c[9]     Send temp      c[8]
Send temp       c[9]  →  Receive temp   c[9]
                         Send temp      c[9]
Send 0.0         0.0  →  Receive temp    0.0
                         Send temp       0.0
                         Send 0.0        0.0
```

Figure 4-2: Details for program of Figure 4-1

5. Mapping programmer's model to skewed computation

While the programmer's model presents a simple view to the user, it is quite remote from our skewed computation model of the machine. In this section, we show that only a subset of the computation in the programmer's model can be mapped onto our machine model.

5.1. Restrictions on programmer's model

An obvious restriction for all W2 programs is that it must be possible for the compiler to determine a lower bound on when data is received, and an upper bound on when data is output. The hardware does not support dynamic flow control, and the compiler must guarantee that an input data is not used until the data is output. This implies, for example, that "while" statements, or "for" statements with dynamic loop bounds cannot be supported. This restriction is tolerable in the application domain of Warp.

Since the cells in our skewed computation model all execute the same code, it necessarily follows that all the cells in the array must execute the same W2 program. We call programs that require all cells to perform the same function *homogeneous*. It is, however, not true that all homogeneous W2 programs can be mapped onto our skewed computation model. Specifically, some bidirectional homogeneous programs (programs that send data from left to right as well as from right to left) cannot be mapped onto this model.

5.1.1. Bidirectional W2 programs

We represent the computation of the array as a graph. Since all the cells perform the same function, we can succinctly specify the computation with only one set of nodes representing the operations on a single cell. There are, however, two types of edges representing two kinds of data dependency relationships: intra-cell computation dependencies and inter-cell communication dependencies. The communication edges are labelled by the direction the data is sent. A "right" edge connects a "send-to-right" node to a "receive-from-left" node, and a "left" edge connects a "send-to-left" node to a "receive-from-right" node.

The graph represents the execution of the array; each execution of a statement is represented by a separate node. The computation edges form no cycles, otherwise, the computation is not well defined. The communication edges, however, may or may not introduce cycles in the graph. In both programs A and B in Figure 5-1, each cell receives and sends a data item from and to its left and right neighbor, respectively. In program A, the two data items are unrelated and the communication edge introduces no cycles. However, in program B, each cell sends the data it receives from its neighbor; the communication edge introduces a cycle. We say that the cycle is a *right cycle* if the communication edge completing the cycle is labelled "right"; similarly, it is a *left cycle* if the communication edge is labelled "left".

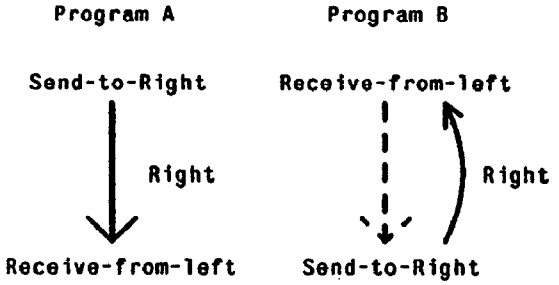

Figure 5-1: Programs with and without communication cycles

A right cycle dictates that the execution of a cell be skewed with respect to the right. Since the receive operation precedes the send operation, in order that the cell does not perform the receive operation before its right neighbor has sent the data, the cell should be delayed with respect to the cell on the right. By the same argument, a left cycle dictates that the execution of a cell be skewed with respect to the left. In other words, it is not possible to map a W2 program containing both right and left cycles onto our skewed computation model.

Fortunately, bidirectional data flow is not necessary for systolic arrays operating in either pipeline or parallel mode. Our current compiler only handles the unidirectional flow programs.

6. Compiler design

The principal problem that is to be solved by the compiler is the management of the parallelism of the Warp machine. The I/O processors of the host, the IU, and the cells in the array cooperate to execute the user's program, and the compiler must generate code for each of these processors. The input to the compiler is a W2 program; the output consists of a program for each host I/O processor, microcode for the IU, and microcode for the Warp array.

6.1. Overview

We now describe how we structure the compiler to manage the parallelism of the Warp machine. The compiler consists of five modules: dataflow analysis, the computation decomposition module, and three code generators (for the Warp array, the IU and the host I/O processors). Figure 6-1 shows the dependencies between these major modules.

The flow analyzer is reasonably machine independent. The flow analyzer builds the central data structure, which is shared between all other modules of the compiler. This data structure is a flowgraph of the program; each node of the flowgraph represents a basic block of the program (see [1] for principal concepts and terminology). The computation of each basic block is represented as a directed acyclic graph (dag).

Each node in a dag corresponds to an abstract operation of the Warp cell. This level models the Warp cell as a simple processor with memory to memory operations and no registers. Later, the code generator translates these abstract instructions into micro-operations, allocates registers, and schedules the code.

The flow analysis module consists of two steps: local and global analysis. The local phase constructs the dag for each basic block and is fairly straightforward. Many local optimizations have been implemented, including common sub-expression elimination, constant folding, height reduction and idempotent operation removal [2].

The global flow analysis phase collects detailed intra-block information for all variables of the program. For regular accessing patterns, the analysis is powerful enough to distinguish between individual array elements and different iterations of a loop [8]. This global data dependency information is incorporated into the dag as arcs between nodes in different basic blocks. There are two types of arcs. If the global analyzer can deduce a strict dependency, we say that node n uses the value of node j (for example, iteration i of a loop uses always the result of iteration $i-1$ of another loop). If a strict dependency cannot be established, the global flow analyser inserts sequencing arcs that enforce a conservative order of evaluation. This information makes it possible for the code generator and scheduler to overlap the execution of different basic blocks, and this is important for heavily pipelined processors like the Warp cells.

The computation decomposition phase partitions the flowgraph generated by the flow analyzer into subgraphs for the different code generators. As it is most important that the Warp array delivers the desired computation bandwidth, code is generated for the Warp cells first. The additional scheduling constraints resulting from the array code are then input to the IU code generation phase; the constraints resulting from the IU code generating phase are input to the host code generation phase.

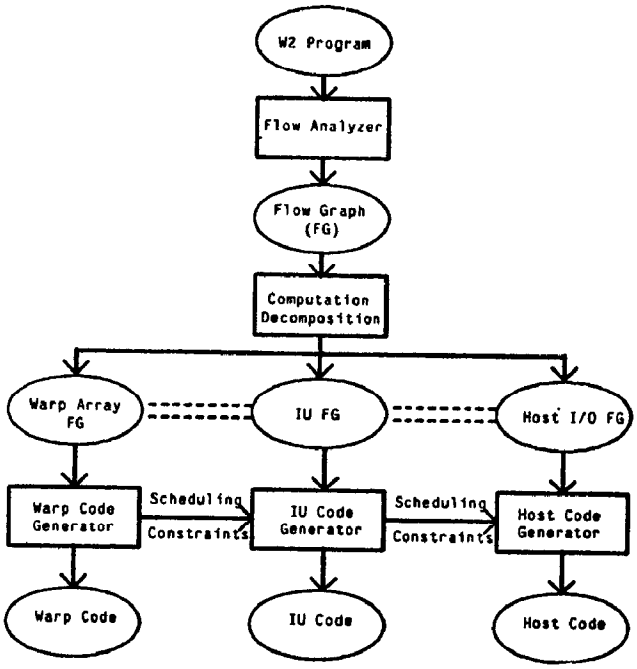

Figure 6-1: Structure of the compiler

The computation decomposition phase also analyses the address computation nodes in the dag to determine if they depend on any input data. Addresses that only depend on loop counters are considered data independent, and these addresses can be computed on the IU. The subgraph containing all data independent addresses is separated from the rest of the dag and is only used by the IU code generator. Calculations of data independent addresses in the original flowgraph are replaced by "receive-address" operations. This modified flowgraph is the input for the Warp array code generator. Addresses moved to the interface unit must be common to all cells in the Warp array. This requirement is always fulfilled for data independent addresses since all cells execute a copy of the same program.

6.2. Warp array code generation

The techniques used in the scheduling of the cell computation is based on those designed originally for increasing the throughput of hardware pipelines [6, 7]. Details on scheduling of individual cells will not be discussed here. In this section, we concentrate on the inter-cell scheduling aspect of the code generation.

The asynchronous communication primitives present in our programming language are not supported by hardware. The semantics of the receive and send statements in the language is that receiving data from an empty queue or sending data to a full queue will cause the execution of the cell to be suspended. Direct hardware support of this semantics would have complicated the design, implementation and debugging of the machine. It is the compiler's responsibility to ensure that queue overflow or underflow does not occur. That is, an input operation cannot be executed until its corresponding output operation has been executed. Also, the number of data items buffered in the queue cannot exceed the size of the queue.

6.2.1. Queue underflow

Let us first consider the issue of queue underflow, by assuming that the buffer is infinite. Because W2 programs accepted by the compiler have unidirectional communication, inter-cell timing constraints can be handled separately from the code generation of the individual cells. To ensure that no underflow occurs, the initiation of the execution of a cell is simply delayed with respect to the preceding cell until no receive operations executed precede the corresponding send operations. This approach suffices because there is no circularity in the data dependency between cells; in the worst case, a cell is not allowed to start executing until the preceding cell has finished completely. If the data flow is bidirectional, however, it may be necessary to insert delays in the middle of the code, which can be difficult in highly optimized horizontal microcode for a deeply pipelined machine.

Ignoring inter-cell timing constraints in the code generation phase simplifies the problem without compromising efficiency. This strategy may only increase the latency through the array, due to the possible increase in the skew between cells. However, optimizing the code of the individual cells optimizes the throughput of the array, the most important performance measure of systolic arrays. (Pathological cases, in which the skew is of the same order as the length of the execution time, are rare in systolic arrays, where data typically stream through the cells at high rates.)

To calculate the minimum skew between cells, we need to identify all matching pairs of input and output operations, and the time the operations are executed with respect to the beginning of the program. The input program must be skewed with respect to the output program by an amount such that all outputs precede their corresponding inputs. Therefore, the minimum skew is the maximum time difference between all matching pairs of inputs and outputs.

Let $\tau_I(n)$ and $\tau_O(n)$ be the functions that map the ordinal number of input and output n to the clock cycle the operation is performed, relative to the beginning of the input and output programs. Then, the minimum skew is given by:

$$\max(\tau_O(n) - \tau_I(n)), \quad 0 \le n < \text{number of input/outputs}$$

Example: straight line code

Consider the simple straight line program in Figure 6-2. Suppose the same program is executed by two cells. Table 6-1 shows the input and output timing, and Figure 6-3 shows how none of the input operations of the second cell precedes the corresponding output operation of the first cell. They are separated by the minimum skew of three cycles.

```
output0
input0
input1
nop
nop
output1
```

Figure 6-2: Input and output of a straight-line program example

Number	τ_O	τ_I	$\tau_O - \tau_I$
0	0	1	-1
1	5	2	3
max			3

Table 6-1: Input/output timing functions and minimum skew

Time	Cell 1	Cell 2
0	$output_0$	
1	$input_0$	
2	$input_1$	
3		$output_0$
4		$input_0$
5	$output_1$	$input_1$
6		
7		
8		$output_1$

Figure 6-3: Two cells executing with minimum skew

Example: loop constructs

Programs with loops are more complicated. The difficulty of determining the minimum skew depends on whether the matching input and output statements are nested in similar or different control structures. Figure 6-4 is an example program that illustrates these two cases.

```
nop
Loop 5 times:  input0
               input1
               nop
nop
nop
Loop 2 times:  output0
               output1
nop
nop
Loop 2 times:  output2
               output3
               output4
               nop
               nop
nop
```

Figure 6-4: A program with loops

Table 6-2 gives the input/output timing information. The control structure of the input loop is similar to that of the first, but not the second, output loop. Both the input and the first output loop contain two input/output operations in each iteration. As a result, the first and second input statements are always matched with the first and second output statements, respectively. Since the rate of input (2 every 3 cycles) is lower than that of output (1 every cycle), the maximum skew can be determined by considering only the first iteration. Conversely, if the input rate were higher, only the time difference between the input/output operations of the last iteration needs to be considered. In the second output loop, the number of outputs per iteration differs from that of the input. An input statement is matched to different output statements in different iterations. All combinations of matches need to be considered. Furthermore, the complexity of the analysis increases significantly with the nesting levels of iterations.

In most programs, the input and output control constructs are usually similar, since they are operating on similar data structures. Furthermore, it is not necessary to derive the exact minimum, a close upper bound will be sufficient. The following mathematical formulation of the problem allows us to cheaply calculate the minimum skew in the simple cases and its upper bound in the complex ones.

The key observation that leads to our solution is that it is not necessary to identify all the matching pairs of inputs and outputs, the most difficult step in the calculation of the minimum skew. Each input/output statement may be executed many times, if it is in a loop. We define a timing function $\tau_{I(m)}$ or $\tau_{O(m)}$ for the mth input/output statement in the program. This function maps the ordinal number of the input or output operation n to the clock cycle it is executed; it is applicable only for certain values of n. We then determine a bound on the maximum time difference for each pair of input/output functions, for those values of n that are in both functions' domains. Finding the exact intersection of both domains may be difficult for timing functions that correspond to statements in dissimilar control structures. For these cases, instead of using the constraints to solve for the intersection completely, we simply use the constraints to bound the time difference between the input and output. The value thus obtained bounds the time difference between all possible matches, if any, between these two input and output statements. By taking the maximum of such bounds between every pair of input and output functions, we obtain the minimum skew. Although every pair of input and output statements needs to be considered, functions corresponding to statements in the same loop share many common terms which need to be computed only once. Also, the branch and bound technique is applicable here: bounds on the timing of all the input/output operations in the same loop can be cheaply obtained to reduce the number of pairs of functions that needs to be evaluated.

number	τ_O	τ_I	$\tau_O - \tau_I$
0	18	1	17
1	19	2	17
2	20	4	16
3	21	5	16
4	24	7	17
5	25	8	17
6	26	10	16
7	29	11	18
8	30	13	17
9	31	14	17
max			18

Table 6-2: Input and output timing for program in Figure 6-4

We characterize each input/output statement by five vectors of k elements, where k is the number of enclosing loops. Each

element of the vector characterizes an enclosing loop, with the first representing the outermost loop. The five vectors are:

$R=[r_1,\ldots,r_k]$: Number of iterations

$N=[n_1,\ldots,n_k]$: Number of inputs/outputs in one iteration of the loop.

$S=[s_1,\ldots,s_k]$: Ordinal number of the first input/output in the loop with respect to the enclosing loop.

$L=[l_1,\ldots,s_k]$: Time of execution of one iteration of the loop

$T=[t_1,\ldots,t_k]$: Time to start the first iteration of the loop with respect to the enclosing loop.

For each loop, we calculate the starting time of the iteration the nth input/output is in, relative to that of the enclosing loop. By summing all these starting times, we get the time the nth operation is executed:

$$\tau(n)=t_1+\left\lfloor\frac{n-s_1}{n_1}\right\rfloor l_1+t_2+\left\lfloor\frac{(n-s_1)\bmod n_1-s_2}{n_2}\right\rfloor l_2+\ldots$$

By defining

$$g(j)=\begin{cases}n & \text{if } j=0\\ (g(j-1)-s_{j-1})\bmod n_{j-1} & \text{otherwise}\end{cases}$$

we get

$$\begin{aligned}\tau(n)&=\sum_{j=1}^{k}\left(t_j+\left\lfloor\frac{g(j)-s_j}{n_j}\right\rfloor l_j\right)\\&=\sum_{j=1}^{k}t_j+\sum_{j=1}^{k}\left(\frac{g(j)-s_j}{n_j}l_j-\frac{(g(j)-s_j)\bmod n_j}{n_j}l_j\right)\\&=\sum_{j=1}^{k}t_j-\sum_{j=1}^{k}\frac{l_j}{n_j}s_j+\sum_{j=1}^{k}\frac{l_j}{n_j}(g(j)-g(j+1))\\&=\sum_{j=1}^{k}t_j-\sum_{j=1}^{k}\frac{l_j}{n_j}s_j+\frac{l_1}{n_1}g(1)+\sum_{j=2}^{k}\left(\frac{l_j}{n_j}-\frac{l_{j-1}}{n_{j-1}}\right)g(j)-\frac{l_k}{n_k}g(k+1)\end{aligned}$$

for the values of n such that

$$\sum_{m=j}^{k}s_m\le g(j)\le(r_j-1)n_j+\sum_{m=j}^{k}s_m$$

For uniformity in notation, the input/output operations themselves are considered a single-iteration loop. For example, in the program in Fig. 6-4, all the vectors describing the operations contain two elements; the first gives information on the enclosing loop, and the second gives information on the statement itself. Therefore, the vector R characterizing $I(0)$ is [5,1], because it is in a 5-iteration loop, and the operation is treated as a single iteration loop. Timing information on all the input and output operations is tabulated in Table 6-3 and the timing functions are given in Table 6-4.

	$I(0)$	$I(1)$	$O(0)$	$O(1)$	$O(2)$	$O(3)$	$O(4)$
R	[5,1]	[5,1]	[2,1]	[2,1]	[2,1]	[2,1]	[2,1]
N	[2,1]	[2,1]	[2,1]	[2,1]	[3,1]	[3,1]	[3,1]
S	[0,0]	[0,1]	[0,0]	[0,1]	[4,0]	[4,1]	[4,2]
L	[3,1]	[3,1]	[2,1]	[2,1]	[5,1]	[5,1]	[5,1]
T	[1,0]	[1,1]	[18,0]	[18,1]	[24,0]	[24,1]	[24,2]

Table 6-3: Vectors characterizing input/outputs in Figure 6-4

$\tau(n)$	function	domain constraints
$I(0)$	$1+3/2n-1/2n\bmod 2$	$0\le n\le 8$ and $n\bmod 2=0$
$I(1)$	$1+3/2n-1/2n\bmod 2$	$1\le n\le 9$ and $n\bmod 2=1$
$O(0)$	$18+n+0n\bmod 2$	$0\le n\le 2$ and $n\bmod 2=0$
$O(1)$	$18+n+0n\bmod 2$	$1\le n\le 3$ and $n\bmod 2=1$
$O(2)$	$52/3+5/3n-2/3(n-4)\bmod 3$	$4\le n\le 7$ and $(n-4)\bmod 3=0$
$O(3)$	$52/3+5/3n-2/3(n-4)\bmod 3$	$5\le n\le 8$ and $(n-4)\bmod 3=1$
$O(4)$	$52/3+5/3n-2/3(n-4)\bmod 3$	$6\le n\le 9$ and $(n-4)\bmod 3=2$

Table 6-4: Timing functions for program in Figure 6-4

The domains of a pair of input and output can be disjoint, completely overlapped and partially overlapped. We give an example for each category:

- *Disjoint*: the domains to which the functions $\tau_{I(0)}(n)$ and $\tau_{O(1)}(n)$ are applicable do not intersect, since $n\bmod m=0$ and $n\bmod m=1$ cannot be satisfied simultaneously. That is, no instance of data items produced by $O(1)$ is read by $I(o)$.
- *Completely overlapped*: the domain of $\tau_{O(0)}(n)$ is completely contained in that of $\tau_{I(0)}(n)$. That is, all the data items produced by $O(0)$ are read by $I(0)$. The time difference is given by

$$\begin{aligned}\max\tau_{O(0)}(n)-\tau_{I(0)}(n)&=17-1/2n+1/2n\bmod 2,\\&\quad\text{where } 0\le n\le 2 \text{ and } n\bmod 2=0\\&\le 17\end{aligned}$$

- *Partially overlapped*: the domains of $\tau_{I(0)}(n)$ and $\tau_{O(4)}(n)$ intersect, but are not completely overlapped. That is, some, but not all, of the data produced by $O(4)$ is read by $I(0)$. Here, we don't find the intersection exactly, but just use the constraint domains to bound the value of their time differences:

$$\begin{aligned}&\max\tau_{O(4)}(n)-\tau_{I(0)}(n)\\&\quad=52/3-1+(5/3-3/2)n\\&\qquad-2/3(n-4)\bmod 3+1/2n\bmod 2,\\&\qquad\text{where}\\&\qquad\quad 6\le n\le 8, n\bmod 2=1 \text{ and } (n-4)\bmod 3=2\\&\quad\le 49/3+1/6\times 8-2/3\times 0+1/2\times 0\\&\quad=17+2/3\end{aligned}$$

Compile-time synchronization is possible only if the loop bounds are compile-time constants, or if the loop bounds for different loops satisfy certain relationships (e.g. if the loop bounds for the input and output loop are always the same). The compiler currently can only handle the former case; powerful symbolic manipulation routines would be necessarily to handle the latter.

The initiation of the input program is skewed with respect to the output program by the minimum skew. It is possible to vary the skew in the course of the computation. This alternative of inserting the necessary delays before each input operation may lower the demand on the size of the buffers. However, it does not lead to higher utilization of the machine; the latency of the computation remains the same, since it is limited by the same minimum skew between cells.

6.2.2. Queue overflow

The problem of determining the minimum buffer size for the queues is similar to determining the minimum skew. In the minimum skew problem, we define a function for each input/output statement that maps the ordinal number of the I/O operation to time. In the minimum buffer size problem, we define a function for each input/output statement that maps time to the number of input and output data received/sent.

The queue overflow problem is currently only detected and reported. Eventually, overflow data needs to be buffered in local data memory, and overflow addresses need to be generated on the cells directly.

6.3. Coupling between IU and Warp array

The program for the IU has two components: the generation of the loop termination signals and the generation of addresses. The IU program is not specified by the user, the compiler must generate it from the original user program and the schedule of cell instructions. The IU and the cells of the Warp array logically operate in lock step. The IU could get ahead of the cells (i.e. send an address required in cycle 3 already in cycle 2), but the compiler utilizes this freedom only inside a basic block. There are two reasons for this restraint. It makes debugging the compiler easier, and the finite length of the input queues in the cells would impose limits in any case.

6.3.1. Loop signals

A control signal is sent from the IU to the Warp array at the end of a loop to indicate whether the cell should continue with or terminate the loop. The flowgraph of the program together with the scheduling constraints produced by the cell code generator make it fairly straightforward to determine the loop signals to be sent. The main problem is that the cell can execute a loop with a loop body of exactly one instruction. The IU on the other hand needs at least three cycles to update and test the loop counter. However, unrolling the last k iterations (with $k = (3 / (\text{length of cell loop})) + 1$) of the IU loop solves this problem.

6.3.2. Address generation

The input to the IU code generator contains the addresses to be generated (from the local dataflow module) and the time at which the cells need the addresses. This timing information is determined by the Warp array code generator and establishes a strict deadline which must be met. These address expressions of the program plus their timing are translated into a labelled flowgraph in the same format as the original flowgraph. Those dag nodes that represent an address to be sent to the Warp array obtain an additional label, their deadline. The dags of the flowgraph capture the computation that must be performed, but no decisions about the code sequences have been made.

The problem of code generation for dags with deadlines and other external constraints has been studied before. However, the hardware idiosyncrasies of the implementation of the IU and the specific features of the Warp array create additional constraints. First, the computational power of the IU is not always sufficient. The IU has been designed to deliver the average performance required, but not peak performance. Together with the inflexibility of the timing constraints demanded by the Warp array, this lack of computation power means that there is sometimes no schedule that satisfies all constraints. If this happens, the IU provides an unorthodox escape mechanism.

This mechanism imposes the second constraint that is unique to the IU. The compiler can pre-store addresses that the IU cannot produce at run-time in a table. This table has 32K elements, and these elements can only be accessed in sequential order. This is a small table for a machine that can demand up to 10 million addresses per second. For this reason, table memory is considered a scarce resource. If at all possible, the compiler computes the address at run time. If addresses must be moved to the table, then complicated address computations with no common sub-expressions are good candidates. Address computations inside nested loops are bad candidates, since they can overflow the table memory easily.

The third problem is the lack of sufficient registers. Most compilers handle this problem by spilling some registers into memory. But there is no memory in the IU, at no time can there be more than 16 live variables, since there are only 16 registers.

The last constraint is that all addresses must be generated by additions or subtractions only. The IU does not contain hardware for multiplication, and the compiler must apply strength reduction to remove multiplications. Strength reduction is essential for multi-dimensional array references. This constraint implies that there must be at least one register for every address expression.

Given these constraints, the scheduling algorithm is quite complex. It consists of two steps: First, based on a set of strategies, the compiler transforms the address computation dags for the IU so that an acceptable sequence of instructions is generated. A sequence of instructions is acceptable if it contains no multiplications, meets the deadline, and uses as few registers as possible to meet the deadline. Then, a second phase combines register allocation and scheduling to produce the final IU code sequence. This second phase determines the exact cycle for each instruction and is based on list scheduling. We describe the first phase, IU code generation, in more detail.

IU code generation

The problem faced by the IU code generator is primarily a problem of operand selection and combination; all arithmetic must be reduced to addition/subtraction or the address must be stored in the table. To appreciate the wealth of choices, consider the generation of addresses for

`a[i,j+1]` *and* `b[i+j,j]`

inside a nested loop (`i` and `j` are indices, `a`, `b` are $N \times N$ arrays). The compiler can keep several sub-expressions in registers. The next table gives possible register allocations, their cost (measured

in number of arithmetic operations needed to form the addresses), and the number of registers that must be updated in the inner loop (index `j`):

Allocated to registers	Number of registers	Arithmetic operations	Update operations
`i*N, j*N, j`	3	6	2
`a[i], b[i], j, j*N`	4	2	2
`a[i], b[i], a[i,j], b[i+j], j`	5	1	3

Table 6-5: Operand allocation to registers

The first option needs the absolute minimum; all allocations need at least three registers. The options in Table 6-5 are not complete, there are also other possibilities. The number of options to the IU code generator increases with the number of address expressions in a basic block. The algorithm to generate code for the IU is given as follows:

1. Find the next basic block (in this loop). Each root in the dag corresponds to an address and has a label, the deadline.
2. Order all root nodes according to their label; count the number of operations needed to compute the address on the IU.
3. For all labelled dag nodes *D*, compare the label value with the number of operations needed (the time the result is available at the earliest).
   a. If the deadline cannot be met for *D*, create a new loop counter and bind a sub-expression to this loop counter. For example, in Table 6-5, `a[i]` is a sub-expression that is bound to a new loop counter and updated by *N* for each iteration of the outer loop.

      This counter needs a register, if there is no register left, then mark this address rooted at *D* to be pre-stored in the table.

   b. Insert code to initialize this register in the loop header, and insert code to update the counter at the end of the loop body. If no cycle is available to initialize the register, mark the address.

   c. Check if the deadline can be met; if not, repeat until either the deadline is met or the node is marked for the table.
4. If no basic blocks are left, we are done; otherwise, continue with step 1.

We have only done preliminary performance evaluation of the IU code generator. The most serious problem is the restricted number of registers, and we are investigating hardware changes to solve this problem. The size of the table space has been less critical, but it is too early to tell if the programs compiled so far are representative.

7. Preliminary results

The compiler is implemented in Common Lisp and runs on a Perq, a 16-bit minicomputer with microcode support for Lisp. All modules together account for approximately 25,000 lines of code. We present some preliminary data in Table 7-1. The length of the μcode is measured in micro-instructions for the IU or Warp cell.

1d-Conv	Simple 1-dimensional convolution for kernel of size 9, one kernel element per cell [5]. All the arithmetic units are fully utilized in the innermost loop, giving a throughout of one result per cycle.
Binop	Binary operator on an image with 512×512 elements.
ColorSeg	Feature separation in a 512×512 image based on color values.
Mandelbrot	Implementation of the Mandelbrot program for a 32×32 image and 4 iterations on one cell.
Polynominal	The sample program from Figure 4-1, with one coefficient per cell, for an array of ten cells. The throughput is also one result per cycle.

Name	W2 Lines	Cell μcode	IU μcode	Compile time
1d-Conv	59	69	72	4 min 58 sec
Binop	61	118	130	5 min 1 sec
ColorSeg	157	213	242	8 min 35 sec
Mandelbrot	107	94	101	4 min 51 sec
Polynominal	39	50	51	2 min 30 sec

Table 7-1: Metrics for sample programs

8. Concluding remarks

We designed and implemented a compiler for a systolic processor array, a first version of the compiler has been released to users. We developed two models of computation for systolic arrays, a programmer's model and a machine model. The compiler maps programs written in the programmer's model to the machine computation model.

We identified a useful computation model for the machine, the skewed computation model. This extremely simple model enables the compiler to efficiently generate code that satisfies the stringent synchronous timing constraints across the array. The computation model has been demonstrated to be very powerful in capturing a wide range of applications for the Warp machine. We also showed that the semantic gap between the programmer's model and this machine model can be bridged by restricting the data flow through the array to unidirectional.

The programmer's model partitions the task of managing the immense parallelism into two subtasks. The user performs the high-level task of problem decomposition onto the individual

cells in the array. The compiler manages the low-level parallelism and schedules the code so that the timing constraints of the synchronous hardware are met. This separation of responsibilities works reasonably well in practice. We do not preclude future work on a higher level user model, but we conclude that the current user model and compiler implement a workable system.

Acknowledgments

We appreciate the contributions of all members of the Warp project at Carnegie Mellon. We thank especially C. Chang, R. Cohn, K. Hughes, P. Lieu, R. Mosur, and P. Steenkiste, who all helped with the implementation of this compiler. H. Enderton, B. Siegell, and J. Webb are the first users of the compiler and suffered through the first releases.

References

1. Aho, A.V. and Ullman J.D.. *Principles of Compiler Design.* Addison-Wesley, Menlo Park, 1977.

2. Allen, F.E. and Cocke, J. A Catalogue of Optimizing Transformations. In *Design and Optimization of Compilers*, Rustin, R., Ed., Prentice-Hall, Englewood Cliffs, N.J., 1972, pp. 1-30.

3. Annaratone, M., Arnould, E., Gross, T., Kung, H. T., Lam, M. S., Menzilcioglu, O., Sarocky, K., and Webb, J. A. Warp Architecture and Implementation. Proceedings of the 13 Intl. Symposium on Computer Architecture, ACM, June, 1985.

4. Hoare, C. A. R. "Communicating Sequential Processes". *Communications of the ACM 21*, 8 (August 1978), 666-677.

5. Kung, H.T. Systolic Algorithms for the CMU Warp Processor. Proceedings of the Seventh International Conference on Pattern Recognition, International Association for Pattern Recognition, 1984, pp. 570-577.

6. Patel, Janak H. and Davidson, Edward S. Improving the Throughput of a Pipeline by Insertion of Delays. Proc. 3rd Annual Symposium on Computer Architecture, Jan., 1976, pp. 159-164.

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, October, 1981.

8. Steenkiste, P. Global dataflow for W2. Internal report.