Compiler Optimizations for Asynchronous Systolic Array Programs

Monica Lam

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

A programmable systolic array of high-performance cells is an attractive computation engine if it attains the same utilization of dedicated arrays of simple cells. However, typical implementation techniques used in high-performance processors, such as pipelining and parallel functional units, further complicate the already difficult task of systolic algorithm design. This paper shows that high-performance systolic arrays can be used effectively by presenting the machine to the user as an array of conventional processors communicating asynchronously. This abstraction allows the user to focus on the higher level problem of partitioning a computation across cells in the array. Efficient fine-grain parallelism can be achieved by code motion of communication operations made possible by the asynchronous communication model. This asynchronous communication model is recommended even for programming algorithms on systolic arrays without dynamic flow control between cells.

The ideas presented in the paper have been validated in the compiler for the Warp machine [4]. The compiler has been in use in various application areas including robot navigation, low-level vision, signal processing and scientific programming. Near-optimal code has been generated for many published systolic algorithms.

The research was supported in part by Defense Advanced Research Projects Agency (DOD) monitored by the Space and Naval Warfare Systems Command under Contract N00039-87-C-0251, and in part by the Office of Naval Research under Contracts N00014-87-K-0385 and N00014-87-K-0533.

1. Introduction

A systolic array of high-performance, programmable processors with floating-point capability is an attractive architecture for numerical processing. Systolic arrays are known for their high utilization; the meticulous synchronization of the flow of data and computation on each cell keeps every cell busy at all times. Many important computation-intensive algorithms in signal processing, image processing, and scientific computation have been mapped to this architecture successfully. At Carnegie Mellon, together with our industrial partner General Electric, we have built a high-performance and programmable systolic array, called Warp [4]. Warp consists of 10 highly pipelined processors, each capable of delivering up to a peak rate of 10 million floating-point operations per second (10 MFLOPS). The peak computation rate of the entire array is therefore 100 MFLOPS. This paper shows that this machine can indeed be programmed to execute fine-grain systolic algorithms with the same utilization typical of special-purpose systolic arrays of simple cells.

The high computation throughput of systolic arrays is derived through the fine-grain cooperation between cells, where computation and the flow of data through the array are tightly coupled. This complexity in designing a systolic algorithm has motivated a lot of research in systolic array synthesis [5, 6, 8, 12, 13, 15, 16]. The target machine model of previous systolic research was a custom hardware implementation of the algorithm in VLSI technology. The main concerns were in mapping specific algorithms onto a regular layout of simple, identical hardware components. The computation performed by each cell must therefore be regular, repetitive and data-independent. The state of the art is that uniform recurrence equations can be mapped onto optimal systolic designs semi-automatically [8, 16].

To fully use the potential of high-performance systolic arrays, previous results must be extended to allow for realistic machine characteristics and a more general application domain. Previous research in automatic synthesis of systolic

Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, San Diego, California (January 1988)

array algorithms focused on the mapping of simple computations onto simple abstract machine models. Cells in a systolic array were assumed to take unit time to process each set of input data; previous systolic array synthesis techniques depend on this simplification. In reality, however, high-performance processors themselves can contain a high degree of pipelining and parallelism. Not only is the internal parallelism of a cell difficult to master, the internal timing of a cell must also be brought to bear in the systolic program design to achieve fine-grain parallelism. Furthermore, the previous assumption that the computation across the array is regular and simple no longer applies. Unlike custom VLSI circuits, a programmable array can be used for a more general problem domain. The cells can execute different programs, with arbitrary control flow and data dependent operations.

Compiling general, processor-oblivious programs to efficient code appears intractable at present. In order that efficiency and generality can be achieved, we choose to expose the configuration of the array to the user, while hiding the internal pipelining and parallelism in each cell in the array. We propose that the array be presented to the user as an array of simple conventional processors that communicate via asynchronous primitives. The proposed abstraction allows the user, or higher level tools, to concentrate on high-level systolic algorithm design; the full programmability of each cell is accessible to the user through high-level language constructs.

In this paper, we concentrate on the effect of the inter-cell communication on code optimization. A characteristic that distinguishes systolic computation from other forms of parallelism is the tight coupling between communication and computation. The high computation rate of a systolic array is matched by an equally high intercell communication rate. Data transfer between cells incur an extremely low overhead. Unlike most other architectures where data are transferred from the memory of a processor to that of the other, results from the data path of one cell can be fed directly to the data path of the neighboring cell. While this coupling between communication and computation makes systolic architectures uniquely supportive of fine-grain parallelism, it poses a difficult problem to code optimization.

We show that the high-level semantics of asynchronous communication, apart from providing a high-level abstraction to the user, is also instrumental in code optimization. The asynchronous communication semantics allows useful code motion be applied without elaborate data dependency analysis across cells. By representing the scheduling constraints of communication operations in a similar fashion as data dependency in the computation of a cell, general code scheduling techniques for high-performance processors can be applied.

The machine characteristic assumed by the compilation technique is that cells in the array communicate via data queues. By the semantics of asynchronous communication, cells are blocked on sending to a full queue or receiving from an empty queue. However, if the programs have data independent control flow, as in the case of previously published systolic array designs, flow control need not be provided in hardware. The asynchronous communication model is still recommended in this case because code optimization is simplified by retaining the semantics of asynchronous communication in the code generation phase. The optimized code is then *skewed* to implement static flow control.

For the Warp machine, we have developed a simple Algol-like language, called W2, that captures the proposed asynchronous communication model [9]. In addition to conventional control constructs, the language has two communication primitives: receives and sends. A cell is blocked whenever it tries to send to a full queue or receive from an empty queue until data or space becomes available. We have implemented an optimizing compiler for the Warp machine, and the compiler has been in use for two years in applications in robot navigation, low-level vision, signal processing and scientific programming [2, 3]. Near-optimal code has been derived for a large number of well-known systolic algorithms [14].

The organization of the paper is as follows: We first present the architecture of Warp as an example of a high-performance systolic array. We then illustrate the effect of the internal timing of the cells on the design of a systolic array program. Next, we describe the asynchronous communication model and the code motions permitted by its semantics. We then show how static flow control can be implemented. Lastly, the paper concludes with a summary and some closing remarks.

2. The Warp architecture

Warp is a linear array of ten high-performance cells, and it is integrated as an attached processor into a UNIX host system. An overview of the entire system is depicted in Figure 2-1. Data flow through the array on two data paths (X and Y), while addresses for local cell memories and systolic control signals travel on the Adr path, as shown in Figure 2-1. Each cell can transfer up to 20 million 32-bit words (80 Mbytes) per second to and from its neighboring cells, in addition to 10 million 16-bit addresses. The Y data path is bidirectional, with the direction configurable statically.

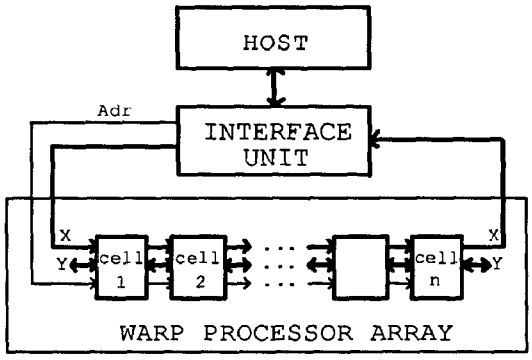

Figure 2-1: Warp system overview

Each processor in the Warp array is capable of a peak computation rate of 10 MFLOPS. It is implemented as a programmable horizontal micro-engine, with its own microsequencer and program memory for 8K instructions. The Warp cell data path, as depicted in Figure 2-2, consists of a 32-bit floating-point multiplier (Mpy), a 32-bit floating-point adder (Add), two local memory banks for resident and temporary data (Mem), a queue for each inter-cell communication channel (XQ, YQ, and AdrQ), and a register file to buffer data for each floating-point unit (AReg and MReg). All these components are connected through a crossbar. Addresses for memory access can be computed locally by the address generation unit (AGU), or taken from the address queue (AdrQ). Both the adder and multiplier have 5-stage pipelines. All the units can be programmed to operate concurrently via wide micro-instructions of over 200 bits.

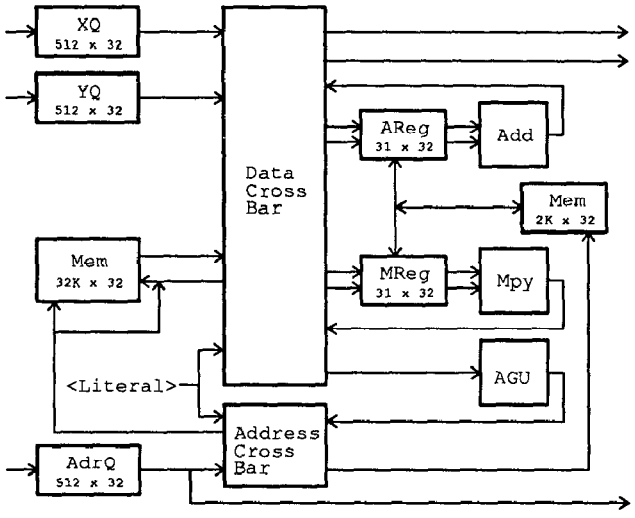

Figure 2-2: Warp cell data path

Because of the heavy pipelining and parallelism in the data path of the cells, efficient code generation for cells is itself a difficult problem. Parallelism in the code must be found across basic blocks to use the hardware efficiently. This problem is known as global microcode compaction [7]. We have extended the scheduling technique of software pipelining [17, 18] and developed a unified approach to global compaction called hierarchical reduction [14]. These two techniques enable us to obtain near-optimal and sometimes optimal schedules for innermost loops in a large sample of programs.

There have been two versions of the Warp cell architecture: a prototype and a production version [1]. One of the major additions to the final version is dynamic flow control hardware. In the prototype machine, it is the responsibiility of the software that a cell does not try to receive data from an empty queue, or send data to a full queue. In the production version, a cell is stalled at run-time by hardware to prevent overflowing or underflowing the data queues. Only a subset of programs executable on the final machine can be executed on the prototype; nonetheless, the applicable domain of the prototype is still quite large, as evident from the large set of applications developed for the machine [2, 3]. The W2 compiler can generate code for either machine. In either case, code is first generated for the individual cells, assuming that there is dynamic flow control hardware. If the prototype machine is the target, additional code to implement compile-time flow control is inserted after the code has been optimized.

3. Effect of cell timing on systolic designs

The use of parallel and pipelined cells was first studied in the context of specific algorithms on custom hardware [10, 11]. The *cut theorem* introduced by Kung and myself is an attempt of a general solution [12]. The theorem states that if the data flow through the array is acyclic, pipelining can be introduced into the cells while maintaining the throughput rate in terms of results per clock. This is achieved by adding delays on selected communication paths between the cells. In the case of cyclic data flow, however, the resources in a cell can be fully utilized only by interleaving multiple independent problems.

Let us use polynomial evaluation as an example of fine-grain parallelism. Suppose we wish to evaluate the polynomial

$$P(x)=C_m x^m+C_{m-1}x^{m-1}+\ldots+C_0$$

for $x_0,x_1,\ldots,x_{n-1}$. By Horner's rule, the polynomial can be reformulated from a sum of product terms into an alternating sequence of multiplications and additions:

$$P(x)=((C_m x+C_{m-1})x+\ldots+C_1)x+C_0$$

The computation can be partitioned among $m+1$ cells by allocating each pair of multiplication and addition in Horner's rule to a cell. A systolic array for solving a polynomial of degree 9 is illustrated in Figure 3-1.

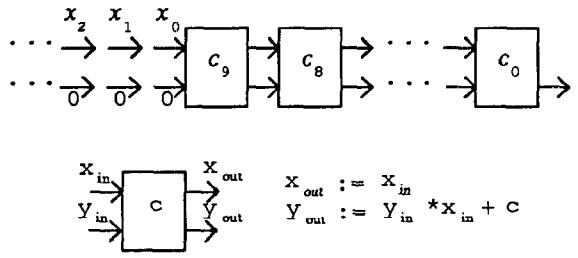

Figure 3-1: Systolic array for polynomial evaluation

The steady state of the computation is straightforward: In each clock cycle, each cell receives a pair of data, performs a multiplication and an addition, and outputs the results to the next cell. However, the boundary conditions are more complex: only the first cell is supplied with valid data in the first clock cycle, the rest of the cells must wait or compute with invalid data until the first valid one arrives. Since the first result does not emerge from the last cell until the end of the m+1st cycle, the computation must be iterated n+m times to calculate the polynomial for n sets of data, and m sets of fictitious data must be tagged on at the end of the input data.

Let us now consider implementing the algorithm in Figure 3-1 on processors with a 3-stage multiplier and a 3-stage adder. The optimal throughput of an array with such processors is one result every clock cycle. This can be achieved by pipelining the computation, and inserting a 6-word buffer into the x data path between each pair of cells. A snapshot of the computation is shown in Figure 3-2. In the original algorithm, consecutive cells process consecutive data items concurrently; in the pipelined implementation, as the second cell starts to process the ith data item, the first is ready to start the i+6th item.

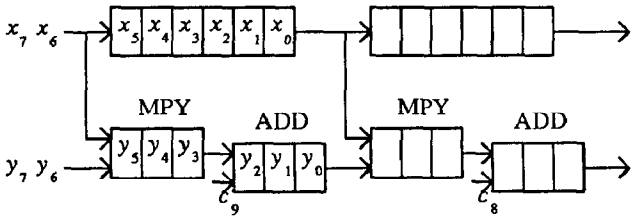

Figure 3-2: Polynomial evaluation on pipelined processors

Although the cut theorem can be used to transform unpipelined designs to pipelined arrays such as the one above, it applies only to simple systolic algorithms, where all cells repeat the same operation all the time. Any complication such as time-variant computation, heterogeneous cell programs, or conditional statements would render this technique inapplicable.

4. Asynchronous communication model

As shown by the example above, timing information on the internal cell behavior must be used to decide which operations should be executed concurrently on different cells. That is, if the machine abstraction hides the internal complexity of the cells from the user, it must also be responsible to synchronize the computations across the cells. Therefore we propose that the user programs the interaction between cells using asynchronous communication operations: Cells send and receive data to and from their neighbors through dedicated buffers. Only when a cell tries to send data to a full queue or receive from an empty queue will a cell wait for other cells.

Systolic array algorithms can be easily expressed in this asynchronous communication model. It is no longer necessary to pad the computation with fictitious data to obtain a simple steady state specification for the array; we can write arbitrarily complex cell programs with irregular communication and computation patterns.

Unidirectional cell programs written using the asynchronous communication model can be compiled into efficient array code. Cells in a unidirectional systolic array can be viewed as stages in a pipeline. The strategy used to maximize the throughput of this pipeline is first to minimize the execution time of each cell, then insert necessary buffers between the cells to smooth the flow of data through the pipeline. The use of buffering to improve the throughput has been illustrated by the polynomial evaluation example.

This approach of code optimization is supported by the high-level semantics of the asynchronous communication model. In asynchronous communication, buffering between cells is implicit. This semantics is retained throughout the cell code optimization phase, thus permitting all code motions that do not change the semantics of the computation. The necessary buffering is determined after code optimization.

Using the asynchronous communication primitives in the W2 language, the polynomial evaluation algorithm is specified by the following cell program:

```
/* shift in the coefficients */
c := 0.0;
for i := 0 to m do begin
   Send (R, X, c);
   Receive (L, X, c);
end;

/*  compute the polynomials  */
for i := 0 to n-1 do begin
   Receive (L, X, xdata);
   Receive (L, Y, yin);
   Send (R, X, xdata);
   Send (R, Y, xdata * yin + c);
end;
```

The **receive** and **send** operations take three arguments: direction, channel used, and a variable name. In a **send** operation, the third parameter can also be an expression. The direction, `L` (left) or `R` (right), and the name of the channel, `X` or `Y`, specify the hardware communication link to be used. In a **receive** operation, the third argument is the variable to which the received value is assigned; in a **send** operation, the third argument is the value sent.

The above cell program is executed by all the cells in the array. The first loop shifts in the coefficients; the second loop computes the polynomials. In the second loop, each cell picks up a pair of **xdata** and **yin**, updates **yin**, and forwards both values to the next cell. By the definition of asynchronous communication, the computation of the second cell is blocked until the first cell sends it the first result. Figure 4-1 shows the first few cycles of computation of the two cells. This description is simpler and more intuitive, as the asynchronous communication model relieves the user from the task of specifying the exact operations executed concurrently on the cells.

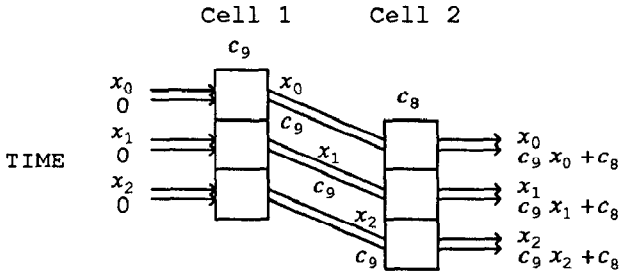

Figure 4-1: Polynomial evaluation using asynchronous communication

Let us consider the compilation of the second loop in the program again for cells with a 3-stage multiplier and a 3-stage adder. In a straightforward implementation of the program, a single iteration of the polynomial evaluation loop takes 8 cycles, as illustrated in Figure 4-2.

The figure contains the microcode for one iteration of the loop, and an illustration of the interaction between cells. The communication activity of each cell is captured by two time lines, one for each neighbor. The data items received or sent are marked on these lines. The solid lines connecting the time lines of neighboring cells represent data transfers on the `X` channel, whereas the dashed lines represent data transfers on the `Y` channel.

As shown in the figure, the second cell cannot start its computation until the first result is deposited into the `Y` queue. However, once a cell starts, it will not stall again, because of the equal and constant input/output rates of each cell. Therefore, the throughput of the array is one polynomial evaluation every eight cycles.

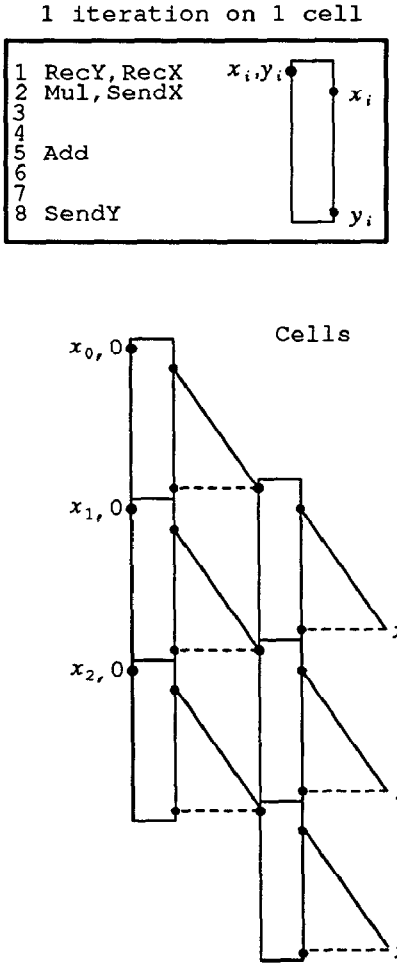

Figure 4-2: Unoptimized polynomial evaluation

However, the hardware is capable of delivering a throughput of one result every cycle. This maximum throughput can be achieved as follows: We notice that the semantics of the computation remains unchanged if we reorder communication operations on different queue buffers. This observation allows us to perform extensive code motion among the communication operations, and hence the computational operations, to achieve the compact schedule of Figure 4-3.

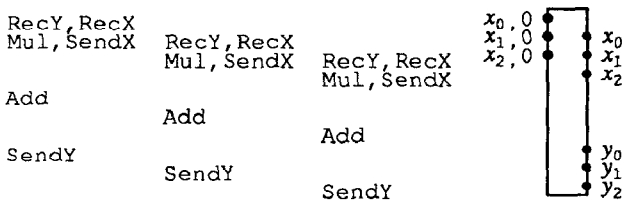

Figure 4-3: Three iterations of polynomial evaluation

The figure shows only three iterations, but this optimal throughput of one iteration per cycle can be kept up for the entire loop using the scheduling technique of software pipelining [14, 17, 18]. The computation of the loop is depicted in Figure 4-4. The only cost of this eight-times speed up is a longer queue between cells. While the original schedule needs a one-word queue between cells, the optimized schedule needs a six-word queue.

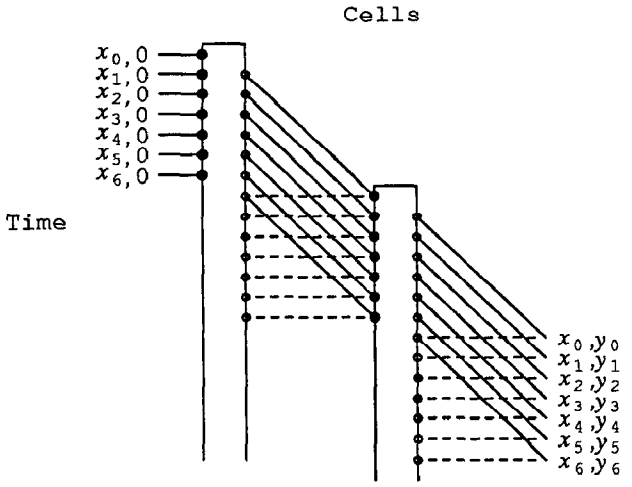

Figure 4-4: Efficient polynomial evaluation

5. Scheduling constraints between communication operations

To use the internal resources of high-performance processors effectively, the original sequential ordering of execution must be relaxed. The approach used in the W2 compiler is to translate the data dependencies within the computation into scheduling constraints, and allow the scheduler to rearrange the code freely so long as the scheduling constraints are satisfied. This approach supports the extensive code motion necessary to use the parallel hardware resources effectively.

As shown in the example above, communication operations in systolic programs must also be reordered to achieve efficient fine-grain parallelism. Fortunately, efficient code can be generated for unidirectional systolic programs by simply analyzing each cell independently and constraining only the ordering of communication operations *within* each cell. These sequencing constraints are represented similarly as data dependency between computational operations. The uniform representation allows general scheduling techniques to be applied to both communication and computational operations.

5.1. The problem

When communication operations are permuted, it is possible to introduce deadlock into a program. Consider the following examples:

(a)
```
First cell                Second cell
Send(R,X,a);              Receive(L,X,c);
Receive(R,X,b);           Send(L,X,d);
```

(b)
```
First cell                Second cell
Receive(R,X,b);           Receive(L,X,c);
Send(R,X,a);              Send(L,X,d);
```

(c)
```
First cell                Second cell
Send(R,X,a);              Send(L,X,d);
Receive(R,X,b);           Receive(L,X,c);
```

The original program (a) is deadlock-free. Reordering the communication operations as in program (b) is illegal, because the two cells will be blocked forever waiting for each other's input. While program (b) deadlocks regardless of the size in the communication buffer, program (c) deadlocks only if there is not enough buffering within the channels. In this particular example, the cells must have at least a word of buffer on each channel.

We say that the semantics of a program is preserved only if an originally deadlock-free program remains deadlock-free, and that the program computes the same results. Here we answer the following question: given that no data dependency analysis is performed across cells, what are the necessary and sufficient scheduling constraints that must be enforced within each cell to preserve the semantics of the program?

Theoretically, it is possible to allow more code motion by analyzing the data dependency across cells and imposing scheduling constraints between computations of different cells. However, since receive and send operations correspond by the order in which they are executed, any conditionally executed receive or send operations would make it impossible to analyze subsequent cell interaction. Furthermore, the scheduling procedure would be greatly complicated if scheduling constraints relate computations from different cells. In the following, we first study the extent by which cell programs can be optimized if compiled independently. Discussions on the scope of the approach are given later.

5.2. The analysis

We separate the preservation of semantics of an asynchronous systolic program into two issues: the correctness of the computation (if the program completes), and the avoidance of introducing deadlock into a program. Here we only concentrate on the interaction between cells; the correctness of the rest of the computation in each cell is assumed.

First, to ensure correctness, the ordering of operations on the same communication channel must be preserved. That is, we cannot change the order in which the data are sent to a queue, or received from a queue. Since receive and send operations correspond by the order in which they are executed, a pair of receives on the same queue can be permuted

only if the corresponding sends on the sender cell are permuted similarly, and vice versa. Therefore, if code motions on the different cells are not coordinated, the data on each queue must be sent or received in the same order as the original. Conversely, if the ordering of operations on each communication channel is preserved, provided that the program completes, the computation is correct.

On the second issue, let us first analyze the occurrence of a deadlock under a general systolic array model. (Important special cases, such as linear or unidirectional arrays, are given below.) A systolic array is assumed to consist of locally connected cells, where each cell can only communicate with its neighbors. In a deadlock, there must be two or more connected cells that are involved in a circular wait. Each cell involved is waiting for some action by one of the other cells. If the deadlock does not occur in the original program, then the code scheduler must have moved an operation that could have prevented the deadlock past the operation on which the cell is blocked. In other words, the operation on which the cell is blocked *depends* on the execution of some preceding operation, which must be directed at one of the cells involved in the circular wait. Therefore, the key is to identify all such dependency relationships, and insert necessary scheduling constraints to ensure that operations sharing a dependency relationship are not permuted.

If there is no knowledge on the topology of the array or the direction of data flow, the scheduling constraints are quite strict. If the communication buffers are infinite in length, then cells block only on receiving from an empty queue. The only operation that could have unblocked a neighboring cell is a send operation. Without any further knowledge of the systolic program, the unblocking of a receive operation may depend on any of the send operations that precede it. Therefore, we cannot move a send operation below any receive operations. If the communication buffers are finite in length, however, a cell can block either on a send or a receive operation. Any preceding send or receive operation to or from any neighbor may be necessary to unblock the operation. Therefore, the original sequential ordering of all communication operations must be observed.

The scheduling constraints can be relaxed in a linear systolic array, provided that there is no feedback from the last cell to the first cell. In a linear array, two and only two cells can be involved in a circular wait; since a cell can only wait for a neighbor one at a time, it is impossible to form a cycle with more than two cells in a linear array. Therefore the unblocking of a communication operation can only depend on receive or send operations from or to *the same cell*. The scheduling constraints above can thus be relaxed as follows: if the queues are infinite in length, send operations to a cell must not be moved below receive operations from the same cell. If the queues are finite, all the sends and receives to and from the same cell must be ordered as before. That is, receive and send operations with the right neighbor are not related to the receive and send operations with the left neighbor.

If the data flow through the array is acyclic, and if the queues are infinite, then no scheduling constraints need to be imposed between communication operations on different channels. This is true for any array topology. The reason is that cells cannot be mutually blocked, and thus there is no possibility of a deadlock. However, if the queues are finite, a cyclic dependency can be formed between the cell and any two of its neighbors. Therefore, as in the general systolic array model, all send and receive operations on every cell must be ordered as in the original program.

The constraints that must be inserted between communication operations to preserve the semantics of systolic programs are summarized as follows:

1. The ordering of all sends to the same queue, or receives from the same queue, must be preserved.
2. For arrays of acyclic data flow, if the queues are infinite, then no other scheduling constraints between communication operations are necessary.
3. Additional scheduling constraints necessary to ensure that no deadlock is introduced are:

TOPOLOGY	INFINITE QUEUES	FINITE QUEUES
Linear	Sends before receives from same cell	Receives and sends with same cell
General	Sends before receives from any cell	All receives and sends

5.3. Practical implications

Both the finite and infinite queue models permit extensive code motion in the compilation of unidirectional, linear systolic array programs. Since receive operations are not constrained to execute after any send operations, the scheduler can simply schedule these operations whenever the data are needed, or whenever the results are computed. As in the polynomial evaluation example, the second data set can be received and computed upon in parallel with the first set. It is not necessary to wait for the completion of the first set before starting the second. The send and receive operations are arranged to minimize the computation time on each cell. Provided that sufficient buffering is available between cells, the interaction between cells may only increase the latency for each data set, but not the throughput of the system.

The major difference between the finite and infinite queue models is that in the former, data buffering must be implemented explicitly in memory or in the register file on the cell. In the finite queue model, because we cannot increase the minimum queue size requirement, the receive operations from different queues must be ordered, and so must the send opera-

tions. In the polynomial evaluation example, the result of an iteration must be sent out before passing the data (**xdata**) of the next iteration to the next cell. To overlap different iterations, the values of **xdata** from previous iterations must be buffered internally within the cell. The number of data that needs to be buffered exactly equals the increase in minimum queue size in the infinite queue model. Therefore, while the infinite queue model automatically uses the existing communication channel for buffering, the finite queue model requires the buffering to be implemented explicitly. This buffering may be costly especially when the number of operations executed per data item is small.

The infinite queue model has been used successively in the W2 compiler for the Warp machine, and it is likely to be an important model for other systolic architectures. First, as discussed above, for any non-linear systolic array, the infinite queue model can support more code motions than the finite queue model if the data flow is acyclic.

Second, data buffering is an important part of fine-grain systolic algorithms. It has been used in systolic algorithms to alter the relative speeds of data streams so that data of an operation arrive at the same cell at the same time. Examples include 1-dimensional and 2-dimensional convolutions. We have also shown that buffering is useful in streamlining the computation on machines with parallel and pipelined units. Moreover, a large queue is useful in minimizing the coupling between computations on different cells; this is especially important if the execution time for each data set is data dependent. Therefore, data buffering is likely to be well supported on systolic processors. For example, hardware queues of 512 words are implemented on the Warp cells. This queue size has not posed any problem in our application experience. The infinite queue model uses the data buffering capability of systolic arrays effectively.

Lastly, the increase in queue size can be controlled. In the current implementation of W2, only communication operations in the same innermost loop are reordered. Code motion is generally performed within a small window of iterations in the loop; the size of the window increases with the degree of internal pipelining and parallelism of the cell. It is possible to further control the increase in buffer size by limiting the code motion of communication operations to within a fixed number of iterations. The analysis and the manipulation of the scheduling constraints are no different from those of computational operations involving references to array variables.

5.4. Discussion

The optimization of systolic algorithms by exploiting the semantics of asynchronous communication is simple and powerful. In this approach, each cell is individually compiled and optimized, and then the necessary buffers are inserted between cells. Results similar to those of the cut theorem can be obtained. One important difference is that while the cut theorem is applicable only to simple, regular computations, the proposed approach applies to general programs.

This approach allows us to obtain efficient code when the following properties are satisfied: large queue size (with respect to the grain size of parallelism), the performance criterion is throughput rather than latency, and unidirectional data flow. If any of these properties is violated, more efficient code can probably be achieved if the internal timing of the cells is considered in the computation partitioning phase, or if all the cells in the array are scheduled together.

The effect of cyclic data flow on efficiency depends on the grain size of parallelism. Cyclic data flow has little negative effect on computation with large-grain parallelism, but it may induce a significant performance loss in computation with fine-grain parallelism. A common example of the former is domain decomposition, where data is exchanged between neighboring cells at the end of some long computation. This communication phase is relatively short compared to the computation phase, and optimality is not crucial. In cyclic arrays of fine-grain parallelism, data sent to other cells are constantly fed back into the same cell. To minimize the time a cell is blocked, we must analyze the dependency across cells to determine the processing time required between each pair of send and receive operations. Moreover, Kung and I showed that the internal pipelining and parallelism within a cell cannot be used effectively for a single problem in arrays of cyclic flow [12]. Multiple problems must be interleaved to use the resources in a cell effectively. In the paper, we also showed that many of the problems, for which cyclic algorithms have been proposed, can be solved by a ring or torus architecture. Rings and tori are amenable to similar optimization techniques as arrays of unidirectional data flow.

6. Compile-time flow control

Many systolic algorithms can be, and have been, implemented without dynamic flow control hardware. Constructs that are supported on such hardware include iterative statements with compile-time loop bounds, and conditional statements for which the execution time of each branch can be bounded and which contain no receive and send operations. The asynchronous communication model does not preclude such programs from being implemented on cells without dynamic flow control support. Static flow control can be implemented after the cells have been optimized.

To implement static flow control, we adopt a simple computation model, called the *skewed* computation model. In this model, the computation on a cell is delayed by the necessary

amount of time to guarantee that it would not execute a receive before the corresponding send is executed. The delay a cell needs to wait after the preceding cell has started its computation is called the skew. Code generated using this model executes as fast, and requires as much buffer, as a program executing full speed on a machine with dynamic flow control. The algorithm for finding the skew has been discussed in another paper [9]. The main ideas are presented here for completeness.

The problem of finding the skew can be formulated as follows. Let $\tau_r(n)$ and $\tau_s(n)$ be the time the nth receive or send operation is executed with respect to the beginning of the program, respectively. The minimum skew is given by:

$$\max(\tau_s(n)-\tau_r(n)),\quad 0 \le n < \text{number of receives/sends}$$

Identifying all the matching pairs of receives and sends in programs containing non-sequential control constructs can be difficult. The key observation is that it is not necessary to match all pairs of receives and sends in the calculation of the minimum skew. A bound can be obtained by the following formulation of the problem: A receive/send statement in a loop corresponds to multiple receive/send operations. Each receive/send statement is characterized by its own timing function, τ_{r_i} or τ_{s_i}, and an execution set E_{r_i} or E_{s_i}. The execution set is the set of ordinal numbers of the receive/send operations for which the statement is responsible for. The timing function maps the ordinal number of a receive or send operation to the cycle it is executed. This is an exact value if the ordinal number of the operation belongs to the execution set; otherwise, it gives a value extrapolated from the values of those that are. The problem of calculating the skew computation is reduced to the following: For each pair of timing functions, τ_{r_i} and τ_{s_j}, we would like to find $\tau_{s_j}-\tau_{r_i}$ for all n that is in the intersection of the execution sets of both functions. The maximum of the differences is the minimum skew. When the intersection of the execution set is difficult to compute, instead of using the constraints defining the sets to solve for the intersection completely, we simply use the constraints to bound the difference between the two timing functions. This formulation of the problem allows us to cheaply calculate the minimum skew in the simple cases and its upper bound in the complex ones.

7. Concluding remarks

Asynchronous communication is proposed here as the machine abstraction for systolic arrays, not only for its programmability and, perhaps surprisingly, for its efficiency as well. The high-level semantics of the asynchronous communication model allows us to relax the sequencing constraints between the communication operations within a cell. Representing only those constraints that must be satisfied in a similar manner as data dependency constraints within a computation, general scheduling techniques can be used to minimize the execution time on each cell. This approach allows us to generate highly efficient code from complex systolic programs; the experience with the compiler indicates that the efficiency obtained is comparable to hand-crafted systolic programs.

The asynchronous communication model is also useful for hardware implementations without direct support for dynamic flow control. Computation admissible of compile-time flow control includes all programs with unidirectional data flow and data independent control flow. Programs written using asynchronous communication primitives can be mapped to the skewed computation model where the computation on each cell is delayed with respect to its preceding cell by some predetermined amount of time. The delay, or the skew, can be calculated accurately and quickly for most cases; approximations can be obtained for pathological cases cheaply. The compile-time flow control algorithm may be used in silicon compilers where custom hardware is built for specific applications.

Acknowledgments

The research reported in this paper is part of my Ph.D. thesis. I especially want to thank my thesis advisor, H. T. Kung, for his support and advice in the past many years. I would like to thank all the members in the Warp project, and in particular, Thomas Gross for his effort in the W2 compiler. I also want to thank Mosur Ravishankar for his helpful comments on my thesis drafts, and for the many hours of discussions and arguments.

References

1. Annaratone, M., Arnould, E., Cohn, R., Gross, T., Kung, H. T., Lam, M., Menzilcioglu, O., Sarocky, K., Senko, J., and Webb, J. Warp Architecture: From Prototype to Production. Proc. 1987 National Computer Conference, AFIPS, Chicago, June, 1987.

2. Annaratone, M., Bitz, F., Clune E., Kung H. T., Maulik, P., Ribas, H., Tseng, P., and Webb, J. Applications of Warp. Proc. Compcon Spring 87, San Francisco, Feb., 1987.

3. Annaratone, M., Bitz, F., Deutch, J., Hamey, L., Kung, H. T., Maulik P. C., Tseng, P., and Webb, J. A. Applications Experience on Warp. Proc. 1987 National Computer Conference, AFIPS, Chicago, June, 1987.

4. Annaratone, M., Arnould, E., Gross, T., Kung, H. T., Lam, M., Menzilcioglu, O. and Webb, J. A. "The Warp Computer: Architecture, Implementation and Performance". *IEEE Transactions on Computers C-36*, 12 (December 1987).

5. Chen, Marina C. A Parallel Language and Its Compilation to Multiprocessor Machines or VLSI. Proc. 13th Annual ACM Symposium on Principles of Programming Languages, Jan., 1986.

6. Delosme, J.-M., Ipsen, I.C.F. Design Methodology for Systolic Arrays. Proc. SPIE Symp., 1986, pp. 245-259.

7. Fisher, J. A., Landskov, D. and Shriver, B. D. Microcode Compaction: Looking Backward and Looking Forward. Proc. 1981 National Computer Conference, 1981, pp. 95-102.

8. Fortes, J.A.B., Moldovan, D.I. "Parallelism Detection and Transformation Techniques Useful for VLSI Algorithms". *Journal of Parallel and Distributed Computing 2* (1985), 277-301.

9. Gross, T. and Lam, M. Compilation for a High-performance Systolic Array. Proc. ACM SIGPLAN 86 Symposium on Compiler Construction, June, 1986.

10. Kung, H.T., Ruane, L.M., and Yen, D.W.L. "Two-Level Pipelined Systolic Array for Multidimensional Convolution". *Image and Vision Computing 1*, 1 (Feb. 1983), 30-36. An improved version appears as a CMU Computer Science Department technical report, November 1982.

11. Kung, H.T. Two-Level Pipelined Systolic Arrays for Matrix Multiplication, Polynomial Evaluation and Discrete Fourier Transform. Proc. Workshop on Dynamical Behavior of Automata: Theory and Applications, Sep., 1983.

12. 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 appears in Proc. Conference on Advanced Research in VLSI, MIT, January 1984, pp. 74-83.

13. Lam, Monica and Mostow, Jack. "A Transformational Model of VLSI Systolic Design". *Computer 18*, 2 (Feb. 1985). An earlier version appears in Proc. 6th International Symposium on Computer Hardware Description Languages and their Applications, May, 1983.

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

15. Leiserson, C.E. and Saxe, J.B. "Optimizing Synchronous Systems". *Journal of VLSI and Computer Systems 1*, 1 (1983), 41-68.

16. Quinton, Patrice. Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations. Proc. 11th Annual Symposium on Computer Architecture, 1984.

17. 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.

18. Touzeau, R. F. A Fortran Compiler for the FPS-164 Scientific Computer. Proc. ACM SIGPLAN '84 Symp. on Compiler Construction, June, 1984, pp. 48-57.