Deadlock Avoidance for Systolic Communication

H. T. Kung
August 28, 1987
CMU-CS-87-163

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

ABSTRACT

Under the systolic communication model, each cell (or processor) in a parallel processing system can operate directly on data residing at the cell's input queues and move computed results directly to the cell's output queues. Incoming and outgoing data need not be stored in the cell's local memory, if not required by the computation. By avoiding these local memory accesses, systolic communication can achieve high efficiency when executing many systolic algorithms. Though efficient, systolic communication may lead to deadlocks at run time if data arriving at a cell's input queues are improperly ordered. This paper describes the nature of this deadlock problem, and provides a deadlock avoidance strategy.

The research was supported in part by Defense Advanced Research Projects Agency (DOD) monitored by the Space and Naval Warfare Systems Command under Contract N00039-85-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.

1. Introduction

In a systolic algorithm, an array of cells perform computations on data flowing through the array [7]. An important characteristic of a systolic array machine is that the program of each cell in the array can operate directly on data residing at the cell's input queues and move computed results directly to the cell's output queues. Incoming and outgoing data need not be stored in the cell's local memory, if not required by the computation. By avoiding these local memory accesses, the machine can achieve high efficiency when executing many systolic algorithms. We call this the *systolic model of communication.* Most of the recent systolic array implementations support this model of communication [1, 2, 3, 4, 5, 8, 9, 10, 12].

It is instructive to contrast the systolic communication with the conventional *memory-to-memory model of communication,* which is often used in message-passing, distributed systems [6, 11, 13]. Under the memory-to-memory model of communication, a cell program does not directly read from or write to the cell's I/O queues. Data residing in an input queue must first be brought in the cell's local memory by the operating system, before they are accessible to the cell program. Similarly, computed results must first be stored in the cell's local memory, before they can be shipped out from the cell via output queues. Figure 1 depicts these two models of communication for a 3-cell linear array. Note that each queue may be both an input queue of a cell and an output queue of a neighboring cell.

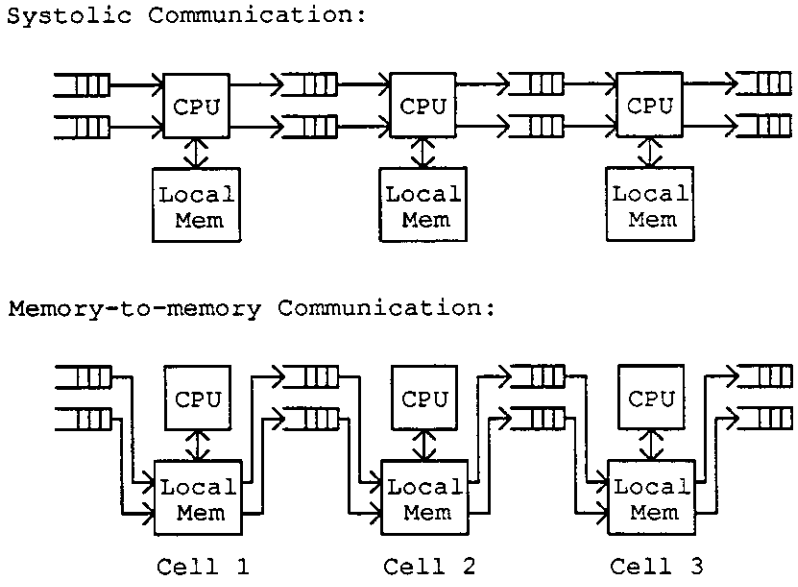


Figure 1. Two models of communication

There is a big difference between these two models with respect to the flexibility of data access by a cell's program. The local memory of a cell can be accessed *randomly*, while the I/O queues of the cell can be accessed only *sequentially*. Therefore, under the systolic communication, one must make sure that

whenever the cell's program reads from an input queue, the right data item will appear at the front of the queue. Also, whenever the program writes to an output queue, it must be safe to insert the data item at the end of the queue in the sense that when the data item emerges from the front of the queue some cycles later, some other cell's program will be ready to read it. If the above is not ensured, then deadlocks may occur.

This paper describes a general deadlock avoidance procedure for systolic communication. Basic definitions and abstractions are given in Section 2. A notion of deadlock-free programs and how to write these programs are described in Section 3. The problem of avoiding queue-induced deadlocks, for deadlock-free programs, at run time is described in Section 4, and a solution to the problem is given in Section 5. Sections 6 and 7 describe schemes for consistent labelling and compatible queue assignment, which the solution calls for. Section 8 discusses how to take advantage of the buffering capability provided by queues. Section 9 contains a summary and some concluding remarks.

The work reported in this paper was motivated by efforts at Carnegie Mellon in developing programmable systolic arrays. The Warp machine, developed during 1984-87 and now produced by GE, has a linear array of high-performance, programmable cells supporting the systolic model of communication [1]. A VLSI version of Warp, named *i*Warp, is currently being developed jointly by Carnegie Mellon and Intel. The *i*Warp system supports more flexible inter-cell communication mechanisms than the Warp machine. The deadlock avoidance scheme of this paper is expected to be implemented on *i*Warp.

2. Definitions, Notations and Abstractions

This section introduces the basic concepts and abstractions needed for presenting the results of this paper. In particular, definitions and notations related to messages, programs and queues are given.

2.1. Messages

We consider an array of processing elements, called cells, which is attached to a host. (In the context of this paper, the host is treated as a cell.) The array can be of any dimensionality. To simplify the presentation, all the examples in the paper use 1-dimensional arrays. It is straightforward to see that results of the paper apply to arrays of higher dimensionalities, and other distributed computing systems using any interconnection topology.

A cell can send a *message*, that is, a sequence of words, to any other cell. The cell at which a message originates or terminates is called the *sender* or *receiver* of the message, respectively. We assume that all the messages are declared prior to program execution. The declaration will identify the sender and receiver of every message that the program will ever use during program execution. Throughout the

paper, message names use upper-case letters, and in all the diagrams, messages are represented as arrows from senders to receivers.

2.2. Programs

A *program* for the array is composed of a number of cell programs, one for each cell. Figure 2 is an example program corresponding to a typical systolic algorithm for computing the first two results of a 3-tap FIR filter [7]. Each cell program is listed directly below the corresponding cell. The portion of the program for preloading the weights w_1, w_2 and w_3 into cells C3, C2 and C1, respectively, before program execution, is not shown. The host provides four input words, x_1, x_2, x_3 and x_4, and receives two output words, y_1 and y_2, computed by the cells, where

$$y_1 = w_1x_1 + w_2x_2 + w_3x_3,$$

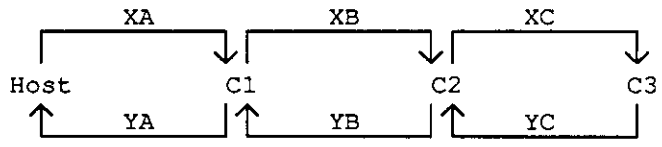
and

$$y_2 = w_1x_2 + w_2x_3 + w_3x_4.$$

The program uses the following messages:

$$\text{XA} = \{x_1, x_2, x_3, x_4\},$$
$$\text{XB} = \{x_1, x_2, x_3\},$$
$$\text{XC} = \{x_1, x_2\},$$
$$\text{YA,YB,YC} = \{y_1, y_2\}.$$

In general there can be a large number of x_i and y_i. Thus these messages can be arbitrarily long.



Host	C1	C2	C3
W(XA) = x_1	R(XA) = x_1	R(XB) = x_1	R(XC) = x_1
W(XA) = x_2	W(XB) = x_1	W(XC) = x_1	$y_1 = w_1x_1$
W(XA) = x_3	R(XA) = x_2	R(XB) = x_2	W(YC) = y_1
R(YA) = y_1	W(XB) = x_2	R(YC) = y_1	R(XC) = x_2
W(XA) = x_4	R(XA) = x_3	W(XC) = x_2	$y_2 = w_1x_2$
R(YA) = y_2	R(YB) = y_1	$y_1 = y_1 + w_2x_2$	W(YC) = y_2
	$y_1 = y_1 + w_3x_3$	W(YB) = y_1	
	W(XB) = x_3	R(XB) = x_3	
	W(YA) = y_1	R(YC) = y_2	
	R(XA) = x_4	$y_2 = y_2 + w_2x_3$	
	R(YB) = y_2	W(YB) = y_2	
	$y_2 = y_2 + w_3x_4$		
	W(YA) = y_2		

Figure 2. Program for filtering

In a program, R(X) or W(X) means reading or writing a word from or to message X, respectively. For

the purpose of this paper, only read (R) and write (W) operations to messages are of interest. That is, the deadlock avoidance strategy to be presented uses only syntactic information in a program given by the write and read operations to messages. We assume that *all* the write and read operations in a program are known at compile time, when the deadlock avoidance procedure is performed. Thus, these operations are assumed to be data-independent. From now on only statements involving write and read operations will be present in a program.

2.3. Queues

There are a fixed number of *queues* between adjacent cells. In the example of Figure 3 (b), this number is four. As shown in the figure, during program execution every message is assigned to a sequence of queues, through which words in the message are transferred from the sender of the message to the receiver of the message. For example, the sequence assigned to message A consists of the 4th queue between C1 and C2, the 3rd queue between C2 and C3, and the 3rd queue between C3 and C4. We assume that transferring words through queues is transparent to cell programs, that is, it is controlled instead by separate I/O processes running on the cells. For example, neither the C2 nor C3 program is involved in transforming words in message A through the 3rd queue between C2 and C3.

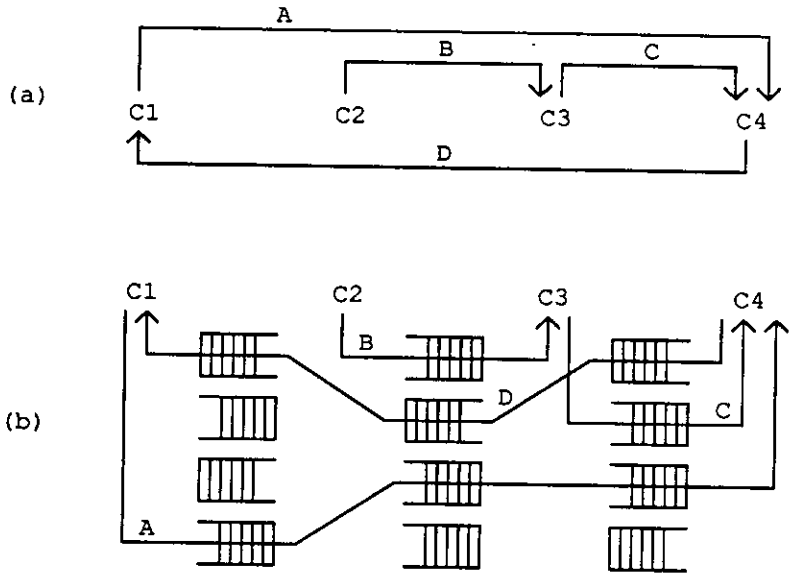


Figure 3. Messages in program (a) assigned to queues shown in (b) during program execution

A message uses the same sequence of queues throughout its lifetime, and a queue in the sequence can be assigned to another message only after the last word in the current message has passed the queue. At the time when a queue is being assigned to a new message, the direction of the queue can be reset.

A message is said to *cross* the interval between two adjacent cells if it will be assigned to queues

between the two cells during program execution. Suppose that a minimum-length route is always taken. Then for a 1-dimensional array, intervals that a message will cross are completely determined by its sender and receiver. However, for a 2-dimensional array, intervals that a message crosses will also depend on the routing scheme.

The number of queues between any two adjacent cells is fixed, whereas the number of messages crossing the interval between the two cells can be arbitrarily large depending on the program. Messages that cross the same interval in the same direction are called *competing* messages. Competing messages may have to share queues if there are not enough queues to allow a separate queue to be assigned to each message. In this case, after a message has finished using a queue, the queue may be assigned to another message. Of course, for efficiency reasons it is a useful practice to place a computation on an array of cells so that the number of competing messages crossing an interval is minimized for most intervals. This placement problem is beyond the scope of this paper, so we will not dwell on it here.

3. The Crossing-off Procedure and Deadlock-free Programs

3.1. The Crossing-off Procedure

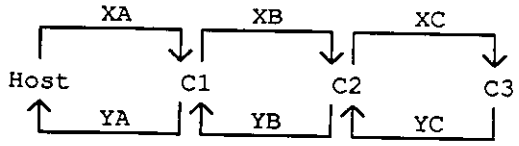
In the following we describe a procedure called "crossing-off", which can be performed on any program. This procedure is for program analysis rather than any actual computation. Whether or not the procedure can be completed on a program will determine if the program is deadlock-free. (The procedure will also be used in Section 6 to define a consistent labelling scheme.)

Suppose that for some message X, both W(X) and R(X) appear as the first statements in two cell programs. Then W(X) and R(X) form an *executable pair* of write and read operations, and X is called the message corresponding to the pair. Sometimes there can be multiple executable pairs corresponding to distinct messages at a given time. Consider, for example, the program of Figure 2. At the beginning of execution, the host and C1 are ready to write and read to and from XA, respectively. Thus the first W(XA) and R(XA) in the host and C1 programs, respectively, form an executable pair.

The crossing-off procedure works as follows: when an executable pair is identified, cross off its write and read operations from the program. The process repeats on the remaining portion of the program until there are no more executable pairs.

Consider for example the execution of the crossing-off procedure on the program of Figure 2. After the first W(XA) and R(XA) in the host and C1 programs, respectively, are crossed off, the pair consisting of the first W(XB) and R(XB) in C1 and C2 programs, respectively, become executable, and will be crossed

off. The process continues until there are no more executable pairs. This is depicted in Figure 4, where the pairs that are crossed off in each step are shown. Note that steps 3, 5 and 9 each have two pairs of write and read operations that can be crossed off.



Step 1:	W(XA)	R(XA)		
2:		W(XB)	R(XB)	
3:	W(XA)	R(XA)	W(XC)	R(XC)
4:		W(XB)	R(XB)	
5:	W(XA)	R(XA)	R(YC)	W(YC)
6:			W(XC)	R(XC)
7:		R(YB)	W(YB)	
8:		W(XB)	R(XB)	
9:	R(YA)	W(YA)	R(YC)	W(YC)
10:	W(XA)	R(XA)		
11:		R(YB)	W(YB)	
12:	R(YA)	W(YA)		

Figure 4. Crossing-off procedure performed on the program of Figure 2.

3.2. Deadlock-free and Deadlocked Programs

A program is said to be *deadlock-free*, if the crossing-off procedure can cross off *all* its read and write operations. That is, when performing the procedure, there always exists at least one executable pair unless all the read and write operations have been crossed off. Figure 4 shows that the program of Figure 2 is deadlock-free. However, if the first two statements in the C3 program are reversed so that R(XC) follows W(YC), then the program is no longer deadlock-free. A program that is not deadlock-free is called a *deadlocked* program.

Deadlocked programs should be avoided, because a deadlocked program will lead to a deadlock at run time, unless it happens to be the case that queues have a sufficient amount of buffering capability and there are enough queues. This is illustrated by the three example programs in Figure 5. When performing the crossing-off procedure on any of these programs, we see that there is no executable pair even at the beginning of program execution. Thus these programs are deadlocked programs. Suppose that queues have no buffering capability. Then the execution of these programs will be deadlocked. For program P1, cell C1 can't finish writing the first word in A, because cell C2 is not ready to read any word in A. For programs P2 and P3, neither C1 nor C2 can finish writing the first word in its output message.

In Section 8, we will take advantage of the fact that queues can buffer a number of words. However

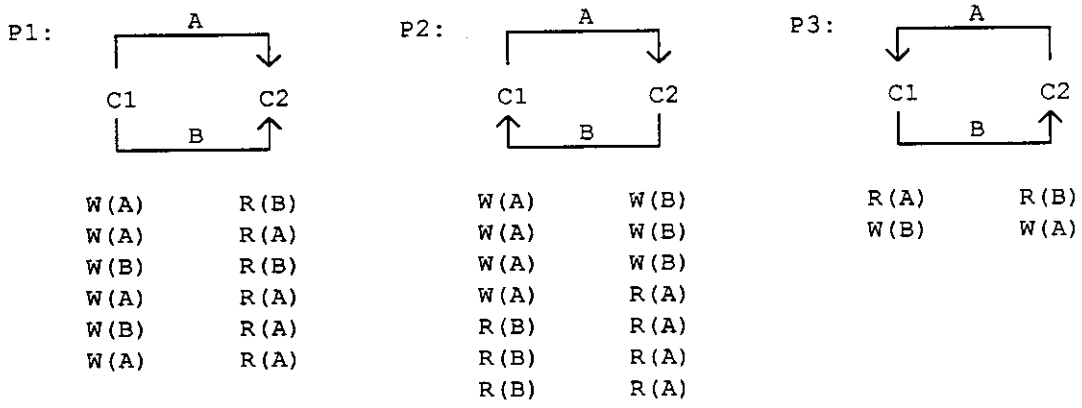


Figure 5. Deadlocked program examples

before that section we will assume that queues are just latches without buffering capability to simplify the presentation.

3.3. How to Write Deadlock-free Programs

From the preceding discussions, we see that deadlocked programs should be avoided as they will lead to deadlocks during program execution. Fortunately, deadlocked programs can be avoided at programming time.

A general strategy is to write the cell programs as if only one word in one message would be transferred in a given step. A program written under this strategy can still allow simultaneous transfers of multiple words during program execution. Consider for example the deadlock-free program of Figure 2, which was written under this strategy. After the first words in both XA and XB have been transferred, the transfers of the second word in XA and the first word in XC can proceed simultaneously, as shown in Figure 4.

Note that to determine if a program is deadlock-free, it is insufficient just to check whether the messages form a cycle by their senders and receivers. Messages in the program of Figure 6 form such a cycle, but the program is deadlock-free.

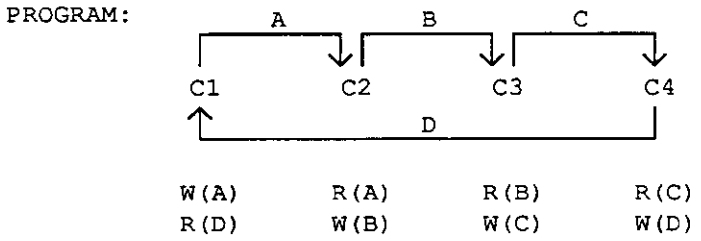


Figure 6. Deadlock-free program example

4. Queue-induced Deadlock

From now on we assume that all programs are deadlock-free. As pointed out above, this assumption can be made true at programming time.

However, *when executing a deadlock-free program using a given set of queues, the execution may still be deadlocked due to the limited number of queues.* This is called a *queue-induced deadlock.* The rest of the paper addresses how to avoid queue-induced deadlocks at run time, for deadlock-free programs.

Figures 7, 8 and 9 each give a deadlock-free program example. Because only one queue is assumed between each pair of adjacent cells, queue-induced deadlocks can occur at run time as shown by the figures.

In Figure 7, W(X) ... or R(X) ... denote a sequence of W(X)s or R(X)s, respectively. In the program the W(C) ... and R(C) ... sequences are assumed to have the same length, and so are the W(B) ... and R(B) ... sequences. When executing the program, the arriving order of B and C at cell C4 depends on the order in which B and C are assigned to the queue between C3 and C4. The lower half of Figure 7 depicts the case that B is assigned to the queue before C. Therefore B arrives at cell C4 first. Since the cell is only ready to read a word in C rather than B, a deadlock occurs. This example illustrates the need of assigning messages to queues according to the order in which the messages are received by the receiver cell.

However, suppose that a receiver cell reads words from multiple messages in an interleaved order, as illustrated by the C3 program in Figure 8. Then merely assigning the queue between C2 and C3 to one of these messages ahead of the others will not solve the queue-induced deadlock problem. In this case separate queues must be used, one for each message. Figure 9 depicts the symmetric case where a sender cell writes words to multiple messages in an interleaved order.

5. A General Procedure for Avoiding Queue-induced Deadlocks

Motivated by the three examples in the preceding section, we present a general procedure for avoiding queue-induced deadlocks:

1. *Consistent message labelling* before program execution. Label all messages with positive numbers, allowing multiple messages to receive the same label. The labelling must be *consistent* in the sense that each cell program will write to or read from messages with *non-decreasing* labels.

   Consider, for example, the program in the upper half of Figure 7. Suppose that messages A, B and C are labelled by 1, 3 and 2, respectively. Then we can check that the labelling is consistent. Note that the C3 program first reads from A, which has label 1, and then writes to B, which has a larger label, namely, 3, whereas the C4 program first reads from C, which

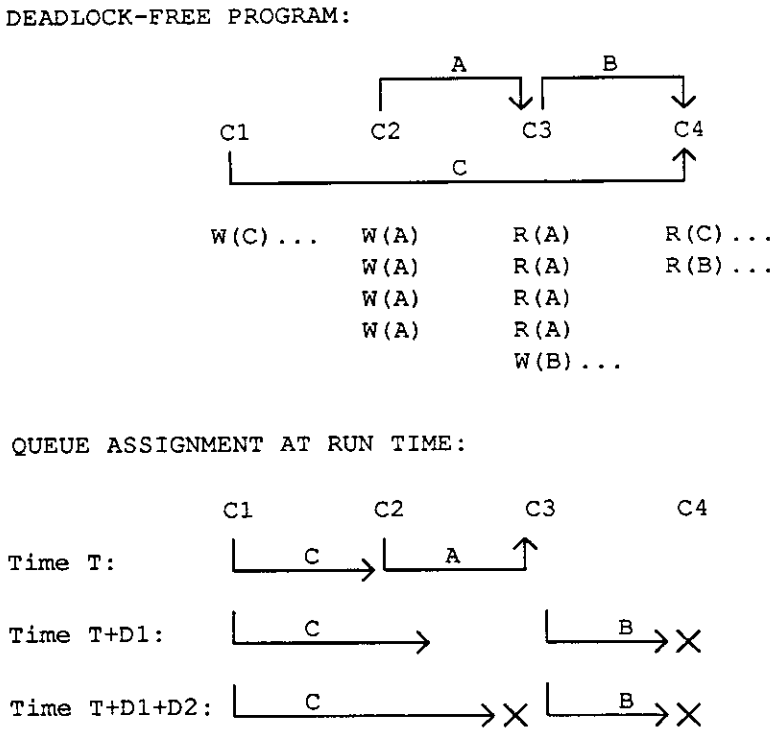


Figure 7. Queue-induced deadlock example 1, with T, D1 and D2 being some positive constants

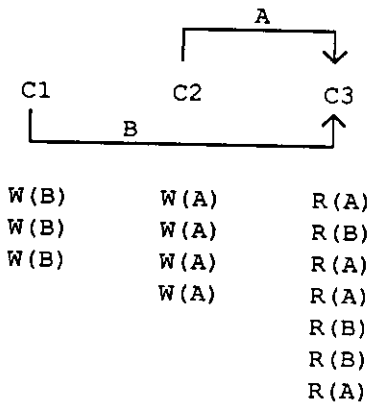
has label 2, and then reads from B, which has a larger label, namely, 3.

2. *Compatible queue assignment* during program execution. Assignment of queues to competing messages must be *compatible* with the labels of these messages in the following sense: a message can be assigned to a queue only if any other competing message whose label is smaller than or equal to the label of the current message has been successfully assigned to a queue, or can be successfully assigned to a queue in the future. (Note that a cell can use some reservation scheme to reserve a queue to a message prior to the message's arrival.) Therefore when the header of a message arrives at a cell, the message may not be assigned to an output queue immediately, that is, it may be blocked at the cell until some other competing messages have secured output queues.

   Consider, for example, again the program in the upper half of Figure 7. As above, suppose that messages A, B and C are labelled by 1, 3 and 2, respectively. When the compatible queue assignment is applied at run time with respect to this consistent labelling, it is guaranteed that C is assigned to the queue between C3 and C4 before B. Therefore the deadlock situation depicted in the lower half of Figure 7 can't occur.

Many labelling schemes can be used as long as they produce a consistent labelling. In fact a trivial consistent labelling scheme is to give the same label to all messages. This scheme will not likely yield an efficient use of queues. This is because the compatible queue assignment implies a stringent condition under which a message that shares the same label with many other messages can be assigned to a queue. A more reasonable labelling scheme will be described in Section 6.

DEADLOCK-FREE PROGRAM:



QUEUE ASSIGNMENT AT RUN TIME:

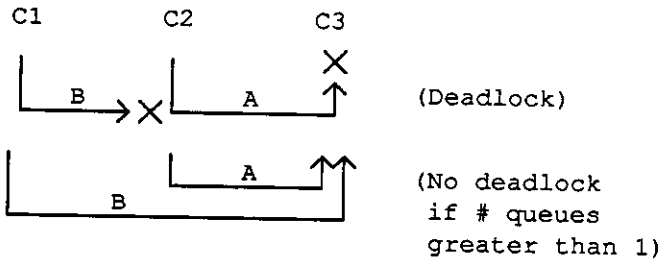


Figure 8. Queue-induced deadlock example 2: interleaved reads from multiple messages by cell C3

Similarly, many schemes can be used to ensure compatible queue assignment. The tradeoff is between the implementation cost of a scheme and the stringency of the scheme's conditions under which a message can be assigned to a queue. Some queue assignment schemes will be presented in Section 7.

The following theorem establishes the validity of the proposed procedure for avoiding queue-induced deadlocks.

Theorem 1: Suppose that the following holds:

C1. The given program is deadlock-free.

C2. There is a consistent labelling for the messages of the program, for which a compatible queue assignment is possible during program execution.

C3. During program execution, the assignment of queues to competing messages is compatible with their labels.

Then the program can run to completion, that is, queue-induced deadlocks will not occur.

Proof: Consider the beginning of the program execution. Because of the consistent labelling, there exists a message X with the smallest label such that at least one of W(X) and R(X) must be the first statement in a cell program. Suppose that W(X) and R(X) do not form an executable pair. Then

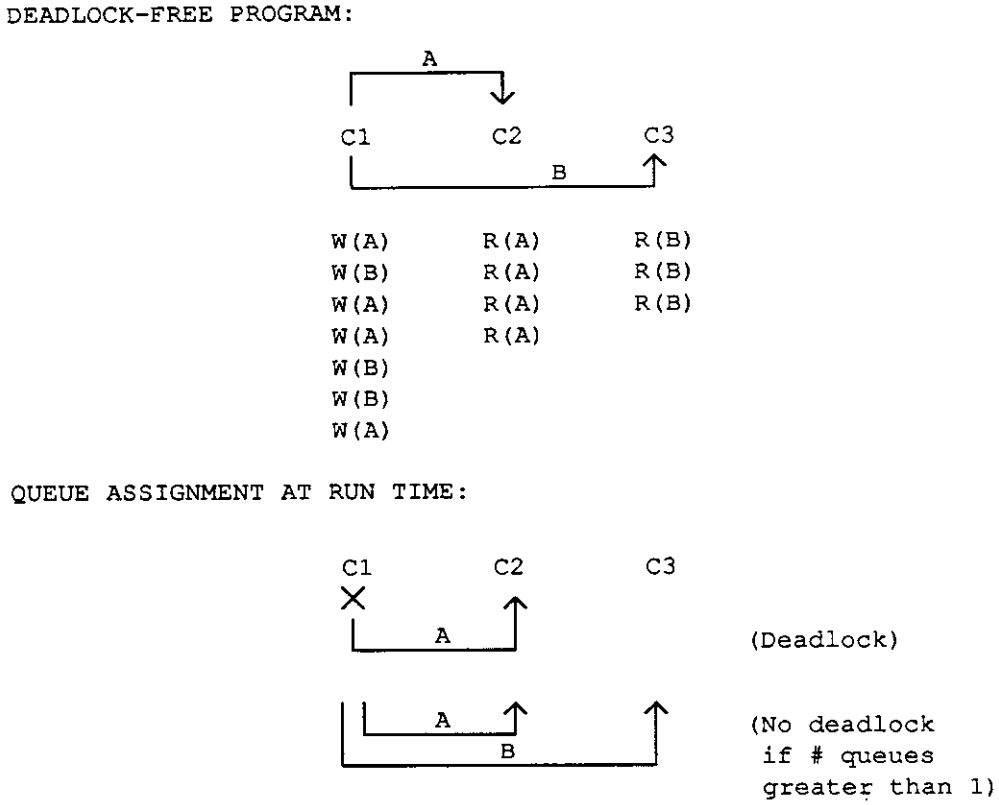


Figure 9. Queue-induced deadlock example 3: interleaved writes to multiple messages by cell C1

one of them is not the first statement in the cell program to which it belongs. The first statement must be one of W(Y) and R(Y), where Y is another message with the smallest label. Now suppose that W(Y) and R(Y) do not form an executable pair. Then one of them is not the first statement in the cell program to which it belongs. The first statement must be one of W(Z) and R(Z), where Z ($\neq Y$) is a message with the smallest label. If $Z=X$ then when the crossing-off procedure is performed on the program, neither the pair W(X) and R(X) nor W(Y) and R(Y) can be crossed off. This contradicts to condition C1 that the program is deadlock free. Therefore $Z \neq X$. Suppose that W(Z) and R(Z) do not form an executable pair. Then yet another message with the smallest label must be found. Continuing with this process, we will eventually come across W(V) and R(V), for some message V with the smallest label, that form an executable pair. This is because the number of messages with the smallest label is finite.

We have shown that one of the messages corresponding to the executable pairs must have the smallest label. The transfer of the first word in the message from its sender to receiver can't be blocked because of the compatible assignment of queues.

Since the transfer of this word is guaranteed to complete, it can't block the transfer of any other word forever. Therefore as far as determining whether or not the transfers of other words can all be

completed, we can assume that the transfer of this word never takes place. That is, the write and read operations of this word can be removed from the sender's and receiver's program, respectively.

Now assume that these operations have been removed from the program. Again, since the remaining of the program is deadlock-free and the message labelling is consistent, one of the messages corresponding to the executable pairs must have the smallest label at this moment. The transfer of the next word in the message from its sender to receiver can't be blocked. In this way, we can show the successful transfer of every word in the program.

6. A Consistent Message Labelling Scheme

The labelling scheme of this section is guaranteed to produce a consistent labelling for the messages of any deadlock-free program. The scheme may not yield the most efficient use of the array for some situations, but it is simple and general. To describe the scheme, we need to introduce the notion of related messages.

Two messages A and B are said to be *related*, if in some cell program, R(A) or W(A) appears between R(B) and R(B) (i.e., after the first R(B) and before the second R(B)), or between W(B) and W(B). The relation is defined to be symmetric and transitive. For example, if a cell writes (or reads) words to (or from, respectively) multiple messages in an interleaved order, then any two of these messages are related. Therefore, messages A and B in Figure 8 or 9 are related.

Given a deadlock-free program, we perform the crossing-off procedure on it. While performing the procedure, we label the messages with positive numbers in the increasing order. In the following procedure, statements inside [...] involve "lookahead" in the crossing-off procedure, which is to be described in Section 8. These statements should be ignored if lookahead is not used.

1. [Perform lookahead.] Pick an executable pair. Let A denote the corresponding message. If A has not been labelled, do the following:

    a. Suppose that neither the sender nor the receiver of A will read from or write to any message that has already been labelled. Then label A with a number larger than all other labels currently in use.

    b. Suppose that either the sender or the receiver of A will read from or write to some messages that have already been labelled. Label A with a number which is smaller than the labels of these messages and larger than the label of the last message to or from which the sender or receiver wrote or read, respectively. (The number may have to be a real number between two consecutive integers.)

    c. If A has other related messages, label them with the label of A.

    d. [If lookahead was used, then label all the messages, to which some write operation

was skipped when using lookahead, with the label of A.]

2. Delete the write and read operations in the executable pair from the program.

3. If there is any write or read statement left in the program, go to step 1.

When performing step 1 if there are multiple executable pairs available then one of them needs to be selected. How to pick an "optimal" one in some sense is an issue that could be related to the efficient use of queues during program execution.

Suppose that the above labelling is applied to the program of Figure 7. If A is the message corresponding to the first executable pair picked by step 1, then messages A, B and C will receive labels 1, 3 and 2, respectively.

7. Queue Assignment Schemes

Competing messages can be assigned to queues in two ways:

1. *Static queue assignment.* Suppose that there are enough queues so that competing messages need not share queues. Then every message can be assigned to a queue prior to program execution, and this assignment does not change throughout program execution. Since every message is guaranteed to secure a queue at run time, this static queue assignment is automatically compatible for any consistent message labelling. Therefore by Theorem 1 queue-induced deadlocks can't happen at run time. Consider for example the program of Figure 9. If there are two queues between C1 and C2, then messages A and B can each be assigned to a separate queue statically, and no deadlock will occur at run time.

2. *Dynamic queue assignment.* Suppose that there are not enough queues, and as a result multiple, competing messages have to share a queue. Then during program execution a queue may be assigned to different messages at different times. When a dynamic queue assignment is used, one must take steps to ensure that it is compatible.

The following is an example of a dynamic queue assignment which is compatible. It uses two rules at run time:

1. *Ordered assignment*: A message can be assigned to a queue only after all other competing messages with labels smaller than label of the current message have been successfully assigned to queues.

2. *Simultaneous assignment*: messages with the same label are assigned to separate queues, and the assignments are done simultaneously when all the needed queues become available.

It is easy to see that this is a compatible queue assignment scheme. The conditions imposed by the scheme may be more stringent than necessary for some cases, but it is relatively easy to implement. Note that the simultaneous assignment rule implies that between two adjacent cells the number of queues can't be less than the number of competing messages having the same label. This is an instance of condition

C2 in Theorem 1.

8. Taking Advantage of Queue Buffering

Until now we have been assuming that a queue does not buffer any words. But in reality, a queue is able to buffer a number of words. In the following we show how to extend the machinery we have developed so far to take advantage of the queue buffering capability. In particular we will relax the definition of deadlock-free programs.

Consider for example program P1 in Figure 5, which has been regarded as a deadlocked program up to this point. Now suppose that each queue can buffer two words. Then the run time deadlock described in Section 3.2 will not occur, provided that A and B are assigned to separate queues between C1 and C2. The rest of this paragraph explains why this is the case. Although the first two words in A that C1 writes can't be read immediately by C2, they will be buffered in the queue assigned to A. C1 can then write the first word in B. This word will go through the queue assigned to B, and will be read by C2. After reading the word, C2 can start reading the first word in A, which until now has been buffered in the queue assigned to A. We can see that all the read and write operations in the program can be completed in this manner.

Motivated by the above example, we proceed in Section 8.1 to relax the definition of deadlock-free programs. Under the new definition, additional programs, including program P1 in Figure 5, will be classified as deadlock-free programs. The execution of any deadlock-free program under the new definition is guaranteed to run to completion, provided that a modified consistent message labelling, which is described in Section 8.2, and a compatible queue assignment are used.

8.1. Crossing-off Procedure Using Lookahead

Recall that to determine whether or not a program is deadlock-free we perform the crossing-off procedure on it. In performing the procedure we need to find executable pairs of write and read operations. Here we relax the definition of executable pairs so that their write and read operations no long have to be the first statements in the cell programs. That is, in locating the write or read operation of an executable pair, we are allowed to "look ahead" into the middle of a cell program. By using lookahead the crossing-off procedure will be able to cross off all the write and read operations for a larger set of programs, and thus a larger set of programs will be classified as deadlock-free programs.

The following two rules must be obeyed when using lookahead:

R1. *Skipping write operations only.* When looking ahead into the middle of a cell program, only write operations in the program can be skipped.

If skipping read operations were allowed, then program P3 in Figure 5 would be incorrectly classified as a deadlock-free program. Note that since the value associated with the write operation in any of the two cell programs may depend on the preceding read operation, there is no chance that by using queue buffering or other means deadlocks at run time can be avoided.

R2. *Bounded skipping.* The total number of write operations to a message that are skipped should not be greater than the total size of the queues that the message will cross.

Some systems such as *i*Warp provide a mechanism to extend logically a queue into the local memory of the receiving cell of the queue. This "queue extension" mechanism can implement very long queues at the expense of larger queue access time. The number of skipped write operations can be used to determined whether or not the queue extension mechanism should be invoked for a queue assigned to a message. That is, the mechanism needs to be invoked only if the number of skipped write operations to the message is larger than the total size of the queues that the message will cross.

This is illustrated in Figure 10, where (a), (b) and (c) show the three executable pairs that are crossed off in the first three times. For this example, we assume that the queues between C1 and C2 each can buffer at least two words. The W(B) in step 3 of the C1 program and R(B) in step 1 of the C2 program form the first executable pair. Note that to locate the W(B), we skipped two W(A)s in steps 1 and 2 of the C1 program. After the first pair are crossed off, W(A) in step 1 of the C1 program and R(A) in step 2 of the C2 program form the second executable pair. After these write and read operations are crossed off, W(B) in step 5 of C1 program and R(B) in step 3 of the C2 program form the third executable pair. To locate the W(B), we skipped the two W(A)s in steps 2 and 4. In this way all the write and read operations in the program can be crossed off. Therefore program P1 in Figure 5 is deadlock-free. Note that in locating the write or read operation for any executable pair we skipped at most two write operations to message A; both rules R1 and R2 are satisfied.

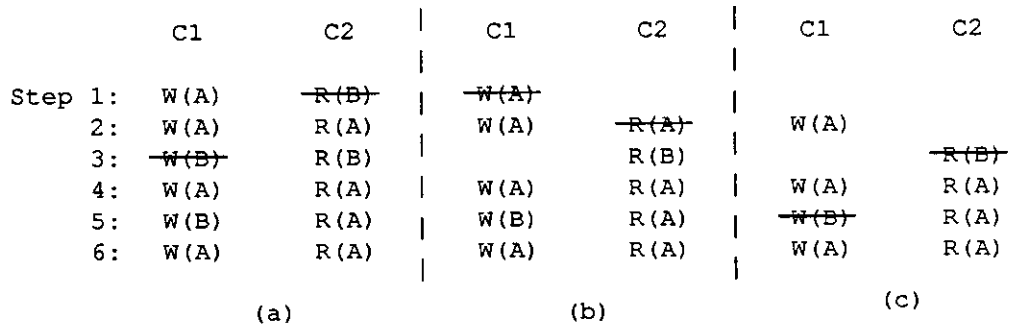


Figure 10. Program P1 in Figure 5 with the (a) first, (b) second, and (c) third executable pair crossed off

8.2. Modified Consistent Message Labelling

Suppose that lookahead is used in locating R(A) or W(A) for some executable pair. If W(B) is one of the write operations skipped, then message B should receive the same label as message A. (See step 1.d in the labelling scheme of Section 6.) By having the same label, these messages are guaranteed by the compatible queue assignment to be assigned to separate queues. This implies that when lookahead is used the number of required queues between two cells may increase.

9. Summary and Concluding Remarks

We discuss the major steps in the deadlock avoidance scheme presented in the paper:

1. Deadlock-free programs: It is the programmer's or compiler's responsibility to make sure that programs are deadlock-free. It is fairly easy to write these programs by following the strategy suggested in Section 3.3. However checking whether or not a random program that has already been written is deadlock-free may not be so easy. It seems that in the worst case it may take as long as running the entire program sequentially. It could be worthwhile to investigate fast checking methods.

2. Consistent labelling of messages before program execution: Any labelling is allowed as long as it is consistent. The labelling scheme of Section 6 is just an example. Further work is needed to devise other consistent labelling schemes that will lead to the efficient use of queues for various kinds of programs. For example, to minimize the number of required queues, only a few of the competing messages should receive the same label.

3. Compatible assignment of queues to messages during program execution: Assignment of queues between two adjacent cells must be compatible with the labels of competing messages. This for example can be enforced by the ordered assignment and simultaneous assignment rules described in Section 7. We need to investigate other schemes which have less restrictions about when a message can be assigned to a queue.

References

1. 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).

2. Avila, J. and Kuekes, P. One-Gigaflop VLSI Systolic Processor. Proceedings of SPIE Symposium, Vol. 431, Real-Time Signal Processing VI, Society of Photo-Optical Instrumentation Engineers, August, 1983, pp. 159-165.

3. Blackmer, J., Frank, G. and Kuekes, P. A 200 Million Operations per Second (MOPS) Systolic Processor. Proceedings of SPIE Symposium, Vol. 298, Real-Time Signal Processing IV, Society of Photo-Optical Instrumentation Engineers, August, 1981, pp. 10-18.

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

5. Foulser, D. E. and Schreiber, R. "The Saxpy Matrix-1: A General Purpose Systolic Computer". *Computer Magazine 20*, 7 (July 1987), 37-43.

6. Hayes, J. P., Mudge, T. N., Stout, Q. F., Colley, S. and Palmer, J. Architecture of a Hypercube Supercomputer. Proceedings of the 1986 International Conference on Parallel Processing, IEEE Computer Society, Aug., 1986, pp. 653-660.

7. Kung, H.T. "Why Systolic Architectures?". *Computer Magazine 15*, 1 (Jan. 1982), 37-46.

8. Lopresti, D. P. "P-NAC: A Systolic Array for Comparing Nucleic Acid Sequences". *Computer Magazine 20*, 7 (July 1987), 98-99.

9. McCanny, J. V, and McWhirter, J. G. "Some Systolic Array Development in the United Kingdom". *Computer Magazine 20*, 7 (July 1987), 51-63.

10. Nash, J. G. and Petrozolin, C. VLSI Implementation of a Linear Systolic Array. Proc. ICASSP '85, Tampa, Florida., March 26-29, 1985, pp. 1392-1395.

11. Seitz, C. L. "The Cosmic Cube". *Comm. ACM 28*, 1 (January 1985), 22-33.

12. Symanski, J.J. Systolic Array Processor Implementation. Proceedings of SPIE Symposium, Vol. 298, Real-Time Signal Processing IV, Society of Photo-Optical Instrumentation, August, 1981, pp. 27-32.

13. Tanenbaum, A. S. "Network Protocols". *Computing Surveys 13*, 4 (1981), 453-489.