TOWARDS SYSTOLIZING COMPILATION: AN OVERVIEW

Christian Lengauer

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188, U.S.A.
Net: chris@cs.utexas.edu

TR-88-37 October 1988

Abstract

A scheme for the compilation of imperative programs into systolic programs is demonstrated on matrix composition and decomposition. The same scheme can be applied if the input is a functional rather than an imperative program. Using this scheme, programs for the processor network Warp and for an Inmos Transputer network have been generated.

This research was supported in part by the following funding agencies: through Carnegie-Mellon University by the Defense Advanced Research Projects Agency monitored by the Space and Naval Warfare Systems Command under Contract N00039-87-C-0251 and by the Office of Naval Research under Contracts N00014-87-K-0385 and N00014-87-K-0533; through Oxford University by the Science and Engineering Research Council under Contract GR/E 63902; through the University of Texas at Austin by the Office of Naval Research under Contracts N00014-86-K-0763, and by the National Science Foundation under Contract DCR-8610427.

1 Introduction

The concept of a *systolic array* [13] has received a lot of attention in the past decade. Systolic arrays are distributed networks of sequential processors that are linked together by channels in a particularly regular structure. Such networks can process large amounts of data quickly by accepting streams of inputs and producing streams of outputs. Many highly repetitive algorithms are candidates for a systolic implementation. Typical applications are image or signal processing.

More recently, mechanical methods for the design of systolic arrays have been developed (see [8,25] for bibliographies). The starting point is, essentially, either an imperative program [8] or a functional program [25]. The result is a formal description of a systolic array that can be pictorially represented. The description is then usually refined for an implementation in hardware – for the production of a systolic chip.

Just as one can realize systolic arrays in hardware, one can also realize them in software. Programmable processor networks that can emulate systolic arrays are becoming increasingly available. Examples are the Ametek [1] and the Transputer [12]. A distributed computer emulates a systolic array by running a systolic program. The program is in a distributed programming language that provides constructs for process definition and communication. For example, Transputer networks are programmed in occam [11]. The process and channel structure of the program must match the processor and communication structure of the systolic array.[1]

There are also more sophisticated processor networks that provide features additional to the ones prescribed by the systolic paradigm. One example is Warp [14] which provides, e.g., for arithmetic pipelining in each processor (i.e., processors are not strictly sequential). Such networks need more specialized methods of compilation than we are offering here [15]. Our techniques can be applied to Warp and its programming language W2 [5], but they are not particularly well-suited for it.

We propose a compilation scheme by which imperative or functional programs can be transformed mechanically into systolic programs. Essentially, the transformation adds a process and communication structure. The source program does not refer to concurrency or communication. We illustrate our compilation scheme with the example of matrix composition/decomposition.

Let us be more precise about the format of the programs that we shall accept:

$$
\begin{array}{l}
\textbf{for } x_0 \textbf{ from } lb_0 \textbf{ by } st_0 \textbf{ to } rb_0 \textbf{ do} \\
\quad \textbf{for } x_1 \textbf{ from } lb_1 \textbf{ by } st_1 \textbf{ to } rb_1 \textbf{ do} \\
\qquad \vdots \\
\qquad \textbf{for } x_{n-1} \textbf{ from } lb_{n-1} \textbf{ by } st_{n-1} \textbf{ to } rb_{n-1} \textbf{ do} \\
\qquad\quad x_0 : x_1 : \cdots : x_{n-1}
\end{array}
$$

[1] The occam programming environment permits the specification of a mapping from software processes to hardware processors. We require this mapping to be the identity in order to avoid inefficiencies caused by the software simulation of channel communication.

with a basic operation of the form:

$$
\begin{array}{lll}
x_0{:}x_1{:}\cdots{:}x_{n-1} & : & \textbf{if } B_0(x_0, x_1, \cdots, x_{n-1}) \rightarrow S_0 \\
& & [\!] \; B_1(x_0, x_1, \cdots, x_{n-1}) \rightarrow S_1 \\
& & \vdots \\
& & [\!] \; B_{m-1}(x_0, x_1, \cdots, x_{n-1}) \rightarrow S_{m-1} \\
& & \textbf{fi}
\end{array}
$$

The bounds rb_i and lb_i are integer expressions in the loop indices x_0 to x_{i-1} $(0 \leq i < n)$; the steps st_i are constants. The conditions B_j $(0 \leq j < m)$ must be side-effect-free. The S_j $(0 \leq j < m)$ are functional or imperative programs, possibly, with composition, alternation, or iteration but without non-local references. Of a subscripted variable in S_j, each subscript must be a distinct argument of the basic operation, and there must be either $n-1$ or n subscripts.[2]

A systolizing compilation consists of two phases. First there is the design of a systolic array, e.g., by one of the methods mentioned previously. Then there is the generation of a distributed program from the description of the systolic array. The latter step shall be the focus of this exposition.

Today's mechanical systolic design methods describe a systolic array by two functions. Let I denote the integers, and let Op be the set of basic operations of the imperative or functional program:[3]

$step : Op \longrightarrow I$ specifies a temporal distribution of the program's operations. Operations that are performed in parallel are mapped to the same step number.

$place : Op \longrightarrow I^d$ specifies a spatial distribution of the program's operations. The dimension d of the layout space is one less than the maximum number of arguments of the operations.

The systolic design is successful if *step* and *place* are linear. One can pose additional restrictions on the program format that ensure the linearity of *step* and *place* and a constant flow direction and speed for each data item [24,27].[4] At present, there is no comprehensive method of systolic arrays that fall outside this class – although there are examples, e.g., dynamic programming [26] and Gauss-Jordan elimination [9,27].

When *step* and *place* are linear, all other information about the systolic array can be determined, notably the flow direction and layout of the data. Let V be the set of variables of the program:

$flow : V \longrightarrow I^d$ specifies the direction and distance that variables travel at each step. It is defined as follows: if variable v is accessed by distinct basic operations s_0 and s_1 and by no basic operations in the steps between s_0 and s_1, then

$$flow(v) = (place(s_1) - place(s_0))/((step(s_1) - step(s_0))$$

Flow is only well-defined if the choice of the pair $\langle s_0, s_1 \rangle$ is immaterial. In other words, the variable may not change its flow direction or speed during the computation. $Flow(v)$ is well-defined if the subscripts of v comprise $n-1$ of the basic operation's n arguments [8].

[2]The proofs of some theorems become more complex if the format of subscripted variables is relaxed as follows: the subscripts of variables in the S_j must be linear expressions in the x_i $(0 \leq i < n)$, and their coefficient matrix must be of rank $n-1$ or n [7]. This extended format covers, for example, convolution [24].

[3]In a functional program, Op is a set of assignments; in an imperative program Op may contain re-assignments.

[4]Formulated in [24,27] in the functional style, these restrictions are easily translated to the imperative style.

pattern : $V \longrightarrow I^d$ specifies the location of variables in the layout space at the first step. It is defined as follows: if variable v is accessed by basic operation s and fs is the number of the first step, then

$$pattern(v) = place(s) - (step(s) - fs) * flow(v)$$

If $flow$ is well-defined, so is *pattern* [8]. Note that *pattern* is a linear combination of linear functions and, therefore, also linear.

To shorten our exposition, we shall add one more requirement: the coefficients of *place* must be from the set $\{-1, 0, +1\}$. That is, we consider systolic arrays in which only neighbouring processors may communicate. Data streams have nine possible flow directions: right or left, up or down, along the two diagonals in both directions, and not at all.

2 The Specification

We are given three matrices: A, B, and C. To be able to derive a systolic solution, we must assume that the matrices are distinct program objects, i.e., they do not share elements. Our goal is to establish the relation:

$$\forall i, j \in \{0, ..., n-1\} : \quad c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} \cdot b_{k,j}$$

That is, C is the matrix product of A and B.

We may establish this relation in two different ways.

2.1 Matrix Composition

A and B are input and C is output. A and B uniquely determine C.

2.2 Matrix Decomposition

C is input and A and B are output. To determine A and B uniquely, we require them to be triangular matrices: A must be zero in the upper triangle and one on the diagonal, B must be zero on the lower triangle.

3 The Input Program

The following program scheme satisfies either of the two alternative specifications:

for i **from** 0 **to** $n-1$ **do**
  for j **from** 0 **to** $n-1$ **do**
    for k **from** 0 **to** $n-1$ **do**
      $i{:}j{:}k$

The imperative looking for-loop syntax can also be used with the functional programming paradigm (see, for example, SISAL [21]). The definition of the basic operation $i{:}j{:}k$ and he initialization of the result matrices differ for matrix composition and decomposition.

3.1 Matrix Composition

For matrix composition, the basic operation $i{:}j{:}k$ is refined as follows:

Functional Style	Imperative Style
$c_{i,j,k} = c_{i,j,k-1} + a_{i,k} * b_{k,j}$	$c_{i,j} := c_{i,j} + a_{i,k} * b_{k,j}$

Matrix elements $a_{i,k}$ and $b_{k,j}$ are initialized with the input data. Matrix elements $c_{i,j,0}$ in the functional and $c_{i,j}$ in the imperative style are initialized to zero.

3.2 Matrix Decomposition

For matrix decomposition, the basic operation $i{:}j{:}k$ is refined as follows:

Functional Style	Imperative Style
if $i \leq j \wedge i = k \rightarrow b_{k,j} = c_{i,j,k-1}$	**if** $i \leq j \wedge i = k \rightarrow b_{k,j} := c_{i,j}$
$[]\ i > j \wedge j = k \rightarrow a_{i,k} = c_{i,j,k-1} * b_{k,j}^{-1}$	$[]\ i > j \wedge j = k \rightarrow a_{i,k} := c_{i,j} * b_{k,j}^{-1}$
$[]\ i > k \wedge j > k \rightarrow c_{i,j,k} = c_{i,j,k-1} - a_{i,k} * b_{k,j}$	$[]\ i > k \wedge j > k \rightarrow c_{i,j} := c_{i,j} - a_{i,k} * b_{k,j}$
fi	**fi**

Matrix elements $a_{i,k}$ and $b_{k,j}$ are initialized to the identity and the zero matrix, respectively. Matrix elements $c_{i,j,0}$ in the functional and $c_{i,j}$ in the imperative style are initialized with the input data.

3.3 The Independence Criterion

We would like to derive a fully pipelined systolic array, i.e., an array in which no shared access occurs: every individual matrix element is accessed strictly in sequence. We forbid shared reading as well as shared writing.

In the functional style, so-called "dependence vectors" make the dependences in the equations explicit [23,27,28]. In order to enforce full pipelining, the basic operation must be enriched with additional equations that add dependences for variables a and b in a third index. Then, indexing a, b and c each by i, j and k, consecutively, a has a dependence in the second, b in the first and c in the third parameter.[5]

In the imperative style, so-called "independence relations" are proved and declared with the program. In this style, we enforce full pipelining by not declaring the weakest independence relation that is provable for the program but adding exclusion requirements for variables a and b [8].

Functional Style		Imperative Style
(0,1,0)	pipelining of a-elements	$(i_0 \neq i_1 \vee k_0 \neq k_1) \wedge$
(1,0,0)	pipelining of b-elements	$(j_0 \neq j_1 \vee k_0 \neq k_1) \wedge$
(0,0,1)	pipelining of c-elements	$(i_0 \neq i_1 \vee j_0 \neq j_1)$

[5] We do not reproduce the modified basic operations here. See, for example, [25] for matrix composition and [3] for matrix decomposition.

4 The Systolic Array

Remember that the systolic array is completely described by functions *step* and *place*. The challenge is the determination of optimal parallelism, i.e., of a step function with the fewest number of steps possible. Here, the functional and the imperative method proceed completely differently. In the functional method, one employs techniques of integer programming [28]; in the imperative method one uses techniques of program transformation [8]. Both derivations are completely mechanical. After the derivation of *step*, the distribution in time, one chooses a compatible distribution in space by a search. The combination of *step* and *place* is consistent if *step* and every dimension of *place* are linearly independent [8]. In particular, neither *step* nor any dimension of *place* may be constant.

Since this is not the focus of our paper, we simply present the step function derived for our example:

$$step(i{:}j{:}k) = i + j + k$$

That is, basic operation $i{:}j{:}k$ is performed at execution step $i + j + k$. The first execution step is 0. As place function, we select:

$$place(i{:}j{:}k) = (i, j)$$

That is, basic operation $i{:}j{:}k$ is performed by the processor at the coordinates (i, j) in the two-dimensional systolic array.

The combination of our *step* and *place* leads to the systolic array depicted in Figure 1 for a 4×4 input. The array consists of 16 processors that are connected by horizontal and vertical channels (as a consequence of the data flow). The elements of matrix A move up, i.e., have flow (0,1), those of matrix B move to the right, i.e., have flow (1,0), and the elements of matrix C remain stationary, each with one processor, i.e., they have flow (0,0). We call the collection of elements of one matrix a *stream*. A and B are moving streams; C is a stationary stream. Figure 1, generated by our implementation of the imperative method, depicts the data layout at the first step.

This completes the first phase of systolizing compilation: the derivation of a systolic array. The data structures on which Figure 1 is based – they encode the functions *step*, *place*, *flow* and *pattern* – are the starting point of the second phase: the code generation. The expansion of *step* is given in Table 1.

5 From Synchrony to Asynchrony

Step specifies a synchronous execution. We may imagine that each parallel step is initiated by a global clock tick. The assumption of synchrony permits the concept and mechanical derivation of a step function. It is also a standard requirement of systolic circuits.

However, when it comes to programming distributed computers, synchrony is a considerable hinderance. High parallelism is obtained much more easily by synchronization at run time than by sorting operations into steps before run time. For example, Warp [14], was made asynchronous after experience with a synchronous prototype [20].

Let us propose a transformation of the synchronous parallel execution to an asynchronous distributed program. As a programming language, we aim at something imperative like CSP [6]; actual languages that we have used are occam [11] and W2 [14]. That is, from now on, take the basic operations as imperatively refined. A conversion from the functional to the imperative representation is straight-forward.

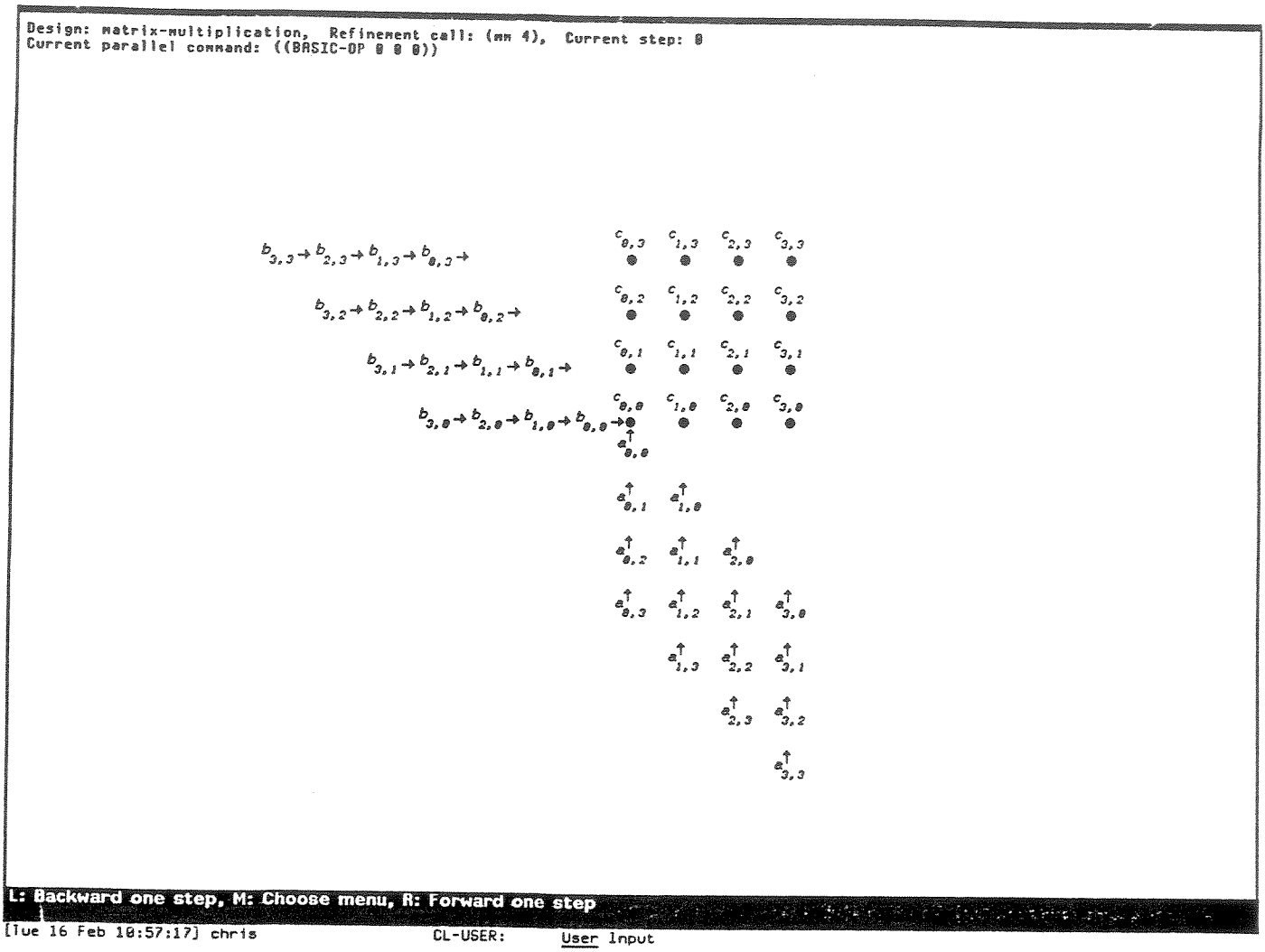

Figure 1: 4×4 Matrix Composition/Decomposition – The Two-Dimensional Systolic Array

0	0:0:0														
1	0:0:1	0:1:0			1:0:0										
2	0:0:2	0:1:1	0:2:0		1:0:1	1:1:0			2:0:0						
3	0:0:3	0:1:2	0:2:1	0:3:0	1:0:2	1:1:1	1:2:0		2:0:1	2:1:0			3:0:0		
4		0:1:3	0:2:2	0:3:1	1:0:3	1:1:2	1:2:1	1:3:0	2:0:2	2:1:1	2:2:0		3:0:1	3:1:0	
5			0:2:3	0:3:2		1:1:3	1:2:2	1:3:1	2:0:3	2:1:2	2:2:1	2:3:0	3:0:2	3:1:1	3:2:0
6				0:3:3			1:2:3	1:3:2		2:1:3	2:2:2	2:3:1	3:0:3	3:1:2	3:2:1	3:3:0
7								1:3:3			2:2:3	2:3:2		3:1:3	3:2:2	3:3:1
8												2:3:3			3:2:3	3:3:2
9																3:3:3

Table 1: 4×4 Matrix Composition/Decomposition – The Synchronous Parallel Execution

Consider Table 1. Each column specifies, from top to bottom, the execution sequence of one process. There is one process per processor, or cell, of the array. In Figure 1, we count rows up and columns to the left. The x-coordinate of a cell specifies the column, the y-coordinate the row to which the cell belongs. The process for the cell at point (col, row) is:

$cell(col, row)$: $col{:}row{:}0$; $col{:}row{:}1$; $col{:}row{:}2$; $col{:}row{:}3$

The flow directions of the streams tell us how cells must be connected and how processes must communicate. For the purpose of the following theorem, we shall think of stationary flows as implemented by looping channels. We shall actually implement them later as assignments. If we know that only neighbouring cells may communicate, only the border cells must be connected by channels to the i/o units of the systolic array. Otherwise, special arrangements have to be made – either in hardware or in software. Figure 1 does not display the i/o units. We shall define their processes in the following section. For now, let us assume that the moving streams are properly injected and extracted by the i/o units and that the systolic array is properly initialized with the stationary stream, and let us deal only with the computation.

Consider a basic operation that is part of process cell(col, row) and that accesses a variable v with flow (α, β). We prefix and suffix the basic operation with the following communications:

Prefix: Input v from cell$(col - \alpha, row - \beta)$.

Suffix: Output v to cell$(col + \alpha, row + \beta)$.

If there are multiple inputs or outputs, they must be combined in parallel.

To support this augmentation, we add the following channels: if there is a variable with flow (α, β), then there is a channel from cell$(col - \alpha, row - \beta)$ to cell(col, row) and a channel from cell(col, row) to cell$(col + \alpha, row + \beta)$, for every point (col, row) that is occupied by a cell.

Code Generation Theorem (Proof in Appendix A):

Let *step* be linear and *place* be linear with coefficients in $\{-1, 0, +1\}$. Let the basic operation have n arguments: $x_0{:}\cdots{:}x_{n-1}$. Let the independence criterion be "no shared access of variables" (i.e., full pipelining). Then the synchronous parallel execution specifies the same partial order of basic operations as the asynchronous process and communication structure derived from it with the previous augmentation.

6 The Generation of Loops

The code generation scheme requires the description of the systolic array (by *step*, *place*, *flow*, and *pattern*) in an expanded form. We prefer the systolic program to contain loops – just as the input program does. We need to rediscover loops, expansion or not, since the definitions of *step*, *place*, *flow*, and *pattern* are not recursive. We represent loops by repeaters.

A *repeater* is a triple $\{fst, cnt, inc\}$, where fst is the first element in the sequence, cnt is the number of elements in the sequence, and inc is the increment by which an element is derived from its predecessor. Repeater $\{fst, cnt, inc\}$ is easily translated into a target language, e.g.,

occam

```
SEQ i = [0 FOR cnt]
  fst+(i*inc)
```

W2

```
for i:=0 to cnt-1 do
  fst+i*inc
```

Take the sequence of pairs $\langle(0,6),(1,4),(2,2),(3,0)\rangle$. Its repeater is {(0,6) 4 $(+1,-2)$}. Its respective occam and W2 loops are:

occam

```
SEQ i = [0 FOR 4]
  (i,6-(2*i))
```

W2

```
for i:=0 to 3 do
  (i,6-2*i)
```

Just as for-loops can be nested, so can repeaters [18]. However, we shall not need nested repeaters in this paper.

Any finite sequence can be represented by a sequence of repeaters. Let us require that this representation contain only maximal repeaters. A sequence of repeaters is *maximal* if it does not contain successive repeaters with an identical increment. For example, $\langle$ {(0,6) 2 $(+1,-2)$}, {(2,2) 2 $(+1,-2)$}$\rangle$ is a non-maximal repeater representation of $\langle(0,6),(1,4),(2,2),(3,0)\rangle$. Sequences without non-nested recurrences have a unique maximal repeater representation. For sequences with non-nested recurrences, several maximal repeater representations exist. For instance, the sequence $\langle(0,6),(1,4),(2,2),(3,0),(3,1),(3,2)\rangle$ is equally represented by $\langle$ {(0,6) 4 $(+1,-2)$}, {(3,1) 2 $(0,+1)$}$\rangle$ and by $\langle$ {(0,6) 3 $(+1,-2)$}, {(3,0) 3 $(0,+1)$}$\rangle$. A maximal repeater representation can be generated with a single traversal of the sequence.

The unique maximal repeaters for 4×4 matrix composition/decomposition are displayed in Table 2. We must deal with three separate cases: the computation, the injection and extraction of the moving streams, and the loading and recovery of the stationary stream. The layout of the repeaters matches the layout of cells in Figure 1. For example, the lower left computation repeater in Table 2 is for the lower left cell in Figure 1.

The computation repeaters are derived from Table 1.

The repeaters for the injection and extraction of moving streams are derived from the data layout in Figure 1: for each channel, trace along the stream that it propagates, starting at the array boundary and moving away from the array. In systolic arrays, elements retain their relative position in the stream on their way through the array. Therefore, the injection and extraction repeaters are identical.

A stationary stream needs to be loaded into the array before the computation begins and recovered from the array after the computation finishes. Both of these phases require the cooperation of the i/o processes and the computation processes. Let us agree – arbitrarily – to load and recover stationary streams along the horizontal channels; we load from the left and recover from the right. On loading, a computation cell first accepts a stationary element and then passes on successive elements; on recovery it first passes on elements and then ejects its own element.[6] The repeaters

[6] With the appropriate convention, we can minimize either the number of channels or the number of repeaters. We have chosen to minimize the number of channels.

Computation:

$$\begin{array}{cccc}
\{0{:}3{:}0\ 4\ (0,0,+1)\} & \{1{:}3{:}0\ 4\ (0,0,+1)\} & \{2{:}3{:}0\ 4\ (0,0,+1)\} & \{3{:}3{:}0\ 4\ (0,0,+1)\} \\
\{0{:}2{:}0\ 4\ (0,0,+1)\} & \{1{:}2{:}0\ 4\ (0,0,+1)\} & \{2{:}2{:}0\ 4\ (0,0,+1)\} & \{3{:}2{:}0\ 4\ (0,0,+1)\} \\
\{0{:}1{:}0\ 4\ (0,0,+1)\} & \{1{:}1{:}0\ 4\ (0,0,+1)\} & \{2{:}1{:}0\ 4\ (0,0,+1)\} & \{3{:}1{:}0\ 4\ (0,0,+1)\} \\
\{0{:}0{:}0\ 4\ (0,0,+1)\} & \{1{:}0{:}0\ 4\ (0,0,+1)\} & \{2{:}0{:}0\ 4\ (0,0,+1)\} & \{3{:}0{:}0\ 4\ (0,0,+1)\}
\end{array}$$

Stream A (up):

$$\{a_{0,0}\ 4\ (0,+1)\} \qquad \{a_{1,0}\ 4\ (0,+1)\} \qquad \{a_{2,0}\ 4\ (0,+1)\} \qquad \{a_{3,0}\ 4\ (0,+1)\}$$

Stream B (right):

$\{b_{0,3}\ 4\ (+1,0)\}$
$\{b_{0,2}\ 4\ (+1,0)\}$
$\{b_{0,1}\ 4\ (+1,0)\}$
$\{b_{0,0}\ 4\ (+1,0)\}$

Stream C (stationary):

Loading:

$$\begin{array}{c|cccc|}
\{c_{0,3}\ 4\ (+1,0)\} & \{c_{0,3},\ prop\ 3\} & \{c_{1,3},\ prop\ 2\} & \{c_{2,3},\ prop\ 1\} & \{c_{3,3},\ prop\ 0\} \\
\{c_{0,2}\ 4\ (+1,0)\} & \{c_{0,2},\ prop\ 3\} & \{c_{1,2},\ prop\ 2\} & \{c_{2,2},\ prop\ 1\} & \{c_{3,2},\ prop\ 0\} \\
\{c_{0,1}\ 4\ (+1,0)\} & \{c_{0,1},\ prop\ 3\} & \{c_{1,1},\ prop\ 2\} & \{c_{2,1},\ prop\ 1\} & \{c_{3,1},\ prop\ 0\} \\
\{c_{0,0}\ 4\ (+1,0)\} & \{c_{0,0},\ prop\ 3\} & \{c_{1,0},\ prop\ 2\} & \{c_{2,0},\ prop\ 1\} & \{c_{3,0},\ prop\ 0\}
\end{array}$$

Recovery:

$$\begin{array}{|cccc|c}
\{prop\ 0,\ c_{0,3}\} & \{prop\ 1,\ c_{1,3}\} & \{prop\ 2,\ c_{2,3}\} & \{prop\ 3,\ c_{3,3}\} & \{c_{0,3}\ 4\ (+1,0)\} \\
\{prop\ 0,\ c_{0,2}\} & \{prop\ 1,\ c_{1,2}\} & \{prop\ 2,\ c_{2,2}\} & \{prop\ 3,\ c_{3,2}\} & \{c_{0,2}\ 4\ (+1,0)\} \\
\{prop\ 0,\ c_{0,1}\} & \{prop\ 1,\ c_{1,1}\} & \{prop\ 2,\ c_{2,1}\} & \{prop\ 3,\ c_{3,1}\} & \{c_{0,1}\ 4\ (+1,0)\} \\
\{prop\ 0,\ c_{0,0}\} & \{prop\ 1,\ c_{1,0}\} & \{prop\ 2,\ c_{2,0}\} & \{prop\ 3,\ c_{3,0}\} & \{c_{0,0}\ 4\ (+1,0)\}
\end{array}$$

Table 2: 4×4 Matrix Composition/Decomposition – The Repeaters

for the i/o cells are derived by tracing through the array along the horizontal channels – right to left for loading and left to right for recovery. For the computation cells, two pieces of information are derived:

1. the identity of the element that the cell must accept on loading and eject on recovery, and
2. a number, *prop*, that specifies how many elements the cell must propagate to the cells on its right on loading and from cells on their left on recovery.

It is possible that a stationary stream does not cover the entire systolic array.[7] A cell that does not hold an element will not be represented in the cell table for loading or recovery. Its *prop* number is the same as that of the cell to its left on loading and to its right on recovery. The following theorem states that a cell cannot hold several values of a stationary stream.

Data Distribution Theorem (Proof in Appendix A):

Assume that *step* and *place* are linear and that *flow* is well-defined, that is, for any particular stream, *flow* is constant. Then, for any particular stream, the data layout function *pattern* is injective. That is, no two elements within a stream are laid out at the same point.

7 Homogenization

We have derived *heterogeneous* systolic program code: there is a separate process definition for each cell. The size of the program and the effort of compiling it grow if we increase the size of the systolic array that the program emulates. If the process definitions follow a regular pattern, we can eliminate this growth by homogenizing the program. A *homogeneous* systolic program is one in which there is only one process definition – one that is parameterized with the cell identifiers.

We achieve homogenization by generating and solving a system of equations for every component of the program. Let us demonstrate with the propagation information for the loading of the stationary stream:

$$\begin{array}{llll} \{c_{0,3},\ prop\ 3\} & \{c_{1,3},\ prop\ 2\} & \{c_{2,3},\ prop\ 1\} & \{c_{3,3},\ prop\ 0\} \\ \{c_{0,2},\ prop\ 3\} & \{c_{1,2},\ prop\ 2\} & \{c_{2,2},\ prop\ 1\} & \{c_{3,2},\ prop\ 0\} \\ \{c_{0,1},\ prop\ 3\} & \{c_{1,1},\ prop\ 2\} & \{c_{2,1},\ prop\ 1\} & \{c_{3,1},\ prop\ 0\} \\ \{c_{0,0},\ prop\ 3\} & \{c_{1,0},\ prop\ 2\} & \{c_{2,0},\ prop\ 1\} & \{c_{3,0},\ prop\ 0\} \end{array}$$

In order to decide how many equations to lay down, we must put a bound on the degree of the progression. A linear progression requires two equations, a quadratic progression three, and so on. With a linear temporal distribution of operations (*step*), spatial distribution of operations (*place*), and spatial distribution of data (*pattern*), the progressions we are considering can be at most linear. Hence, if we name the progression of the *prop* numbers for loading *count*, for the moment, we know for columns, say:

$$count(col) = \alpha \cdot col + \beta$$

For every row, we now obtain two linear equations by filling in the information for two distinct columns, and solve for α and β, e.g.:

[7] An example is a more efficient dedicated array for matrix decomposition in which only the non-zero triangles of matrices A and B are processed [8]: with place function (i, k), the triangle of A is the stationary stream; the processor layout includes the diagonal, whereas the triangle of A does not.

$$\begin{array}{lrcl} count(0) = & \beta & = & 3 \\ count(1) = & \alpha + \beta & = & 2 \\ \hline & \alpha & = & -1 \end{array} - \qquad \Longrightarrow \qquad count(col) = 3 - col$$

Remember that the identified pattern of a stationary stream may be broken by cells that do not hold an element. To make sure that we can homogenize all the way, we must check that the progression we have derived is adhered to in every instance. If it is not, the array is heterogeneous. Partial homogenization may still be possible, though.

We have collected *prop* horizontally, across columns, and have derived one expression for each row. Next, we collect vertically, across rows. In this case, the expression remains constant across rows, and we obtain:

$$count(col, row) = 3 - col$$

8 Generalization to an Arbitrary Problem Size

We can employ the same trick for a generalization from the 4×4 to the $n \times n$ problem. This procedure works best, if we can be sure that *step* remains the same linear function in the parameters of the basic operation with varying problem size n. If that is so, each dimension of *place* must be linear and the same for all n as well – or it would not be consistent with *step* – and then, if *flow* is well-defined, *pattern* is linear and the same for all n, too. We can ensure that *step* is linear and the same for all n by enforcing additional restrictions on the format of the input program [24].

Let us continue to demonstrate with the propagation information for the loading of the stationary stream. Because of the linearity of the layout of the operations in time and space and the data in space, all components of the repeater representation (indices, *cnt* numbers, increments, and *prop* numbers) can change at most linearly. That is, we may assume:

$$count(n) = \alpha \cdot n + \beta$$

We solve, again, a system of two equations – say, for sizes 4 and 5:

$$\begin{array}{lrcl} count(4) = & \alpha \cdot 4 + \beta & = & 3 - col \\ count(5) = & \alpha \cdot 5 + \beta & = & 4 - col \\ \hline & \alpha & = & 1 \end{array} - \qquad \Longrightarrow \qquad count(n) = n - 1 - col$$

When building this system of equations, we must be careful in the selection of values for n. We must pick a succession that captures the recurrence. Any choice will do for a direct recurrence with one base case. Multiple base cases or an indirect recurrence require some care. If *step* is the same linear function for all n, the recurrence is direct with one base case.

We employed this technique successfully to derive mechanically other simple expressions in the input size – for example, the number of processors of a two-dimensional systolic array [9]. In this case, we had to solve three equations because the growth of processors in a two-dimensional systolic array is quadratic.

The mathematically most satisfying way of generalizing is by induction, but our overriding concern is mechanical autonomy. At present, the techniques for solving systems of equations are more autonomous than those for recognizing and proving induction schemes.

Computation:

$$cell(col, row) \quad = \quad \{col{:}row{:}0 \ n \ (0, 0, +1)\}$$

Stream A (up):

$$i/o\text{-}unit(col) \quad = \quad \{a_{col,0} \ n \ (0, +1)\}$$

Stream B (right):

$$i/o\text{-}unit(row) \quad = \quad \{b_{row,0} \ n \ (+1, 0)\}$$

Stream C (stationary):

Loading:

$$\begin{array}{rcl} i\text{-}unit(row) & = & \{c_{0,row} \ n \ (+1, 0)\} \\ cell(col, row) & = & \{c_{col,row}, \ n - 1 - col\} \end{array}$$

Recovery:

$$\begin{array}{rcl} cell(col, row) & = & \{col, \ c_{col,row}\} \\ o\text{-}unit(row) & = & \{c_{0,row} \ n \ (+1, 0)\} \end{array}$$

Table 3: $n \times n$ Matrix Composition/Decomposition – The Repeaters

9 Translation to a Target Language

This completes the derivation of machine-independent program code (repeaters) from *step* and *place*. Table 3 displays the final result. We have gone on and produced concrete distributed programs in W2 for the processor array Warp and in occam for several different Transputer networks. The W2 program was derived from a different set of repeaters: we projected the two-dimensional array (Fig. 1) to one dimension before we generated the repeaters [18]. (Warp has a one-dimensional processor layout.) occam programs for a two- and a one-dimensional Transputer network were derived from the the repeaters in Table 3. The two-dimensional program can be found in Appendix B; for the one-dimensional program see [19]. Here we performed the projection after the generation of repeaters: the one-dimensional program was derived from the two-dimensional one by techniques of program transformation.

The mechanical translation of repeaters to W2 or occam for-loops is trivial. The rest – mainly variable declarations – has been coded by hand. We do not anticipate any problems with its mechanization.

10 Other Systems

There are a number of systems that implement the functional method of systolic design [2,23,25]; at present, none of them incorporates a mechanical code generation.

We know of only one other system, SDEF [4], that performs code generation. SDEF is a programming system that serves as a back-end for the functional method of systolic design. Beside an automatic generation of C code augmented with communication directives it provides simulation, tracing and debugging facilities. The code, generated by a "translator", may be tested with a "simulator" and is ultimately executed on a programmable processor array that provides a run-time system for C programs with communication. SDEF requires the use of a rectangular processor grid of sufficient size; streams that are not horizontal or vertical are appropriately rerouted.[8]

Our work has a different focus. We consider only the translation step but make it more independent than it is in SDEF. Our translation scheme applies to both the functional and the imperative method of systolic design (in our own work, we emphasize the imperative method). Our goal is a set of simple, precise and generally applicable techniques for the generation of systolic programs in any distributed programming language.[9] To that end we have defined a machine-independent code form (repeaters) and have addressed issues like homogenization and generalization in terms of it. It will be interesting to compare the quality and nature of the C code generated by SDEF's translator with equivalent C code generated from repeaters.

11 Conclusions

We have also generated repeaters for other systolic arrays that perform, e.g., matrix composition, matrix decomposition, convolution, polynomial evaluation, and Gauss-Jordan elimination. We do not offer a systolizing compiler at this stage. But we offer a view of how a systolizing compiler would function after some further development.

The programs that the compiler would accept are of the nested loop format that we specified in the introduction. In the imperative style, an optimal step function can be found for any program of this form [8], but only if *step* is linear may the systolizing compilation proceed. *Step* is guaranteed to be optimal and linear if additional restrictions on the input program are made [24]. The place function can either be supplied externally or chosen by the compiler after a search. Ill-defined flows can be dealt with by a technique of adding variables and providing reflection operations [9]. Once a systolic array has been identified, heterogeneous fixed-size repeaters can always be generated. "Soaking" or "draining", i.e., the case where the first or last use of a moving stream element is not at the border of the systolic array can also be dealt with [18]. Complete homogenization may not be possible but, with the additional restrictions on the input program, the generalization of the repeaters to an arbitrary problem size is automatic.

We are working on extending the scope of the compilation scheme to piecewise linearity as in the case of the algebraic path problem [9]. Then, the compiler would accept non-nested compositions of the previous input format. We have already obtained mechanically generated repeaters and an occam program for the algebraic path problem [10]. For piecewise linearity, the instantiation of the problem size helps considerably in the development of a systolic array. The challenge lies in the mechanical generalization to an arbitrary problem size.

[8] In our work, we have addressed a different hardware adaptation technique: projection [18,19].

[9] SDEF goes straight to C.

A systolizing compiler will also have to address the problem of how to cope with the physical limitations of a systolic computer. For example:

- Given a dimension, the range (i.e., the number of processors) of the systolic array exceeds that of the systolic computer. This problem can be addressed by partitioning the design or converting it into a ring or toroid (e.g., [16,22]).
- The number of channel connections per processor in the optimal systolic design exceeds that of the systolic computer. This problem can be addressed by rerouting techniques (e.g., [4]).
- The number of dimensions (of the processor layout) of the systolic array exceeds that of the systolic computer. This problem can be addressed by projecting the systolic array (e.g., [16,18,19]). Essentially, a projection takes one dimension away from the place function and adds it to the step function.

Already our present implementation has been a big help in the derivation of systolic programs – although it does not include homogenization and generalization, and performs only a partial translation (namely that of repeaters) into the target language. With it, we developed one of the largest Warp programs at the time of its conception. This program performs matrix decomposition with triangular streams A and B. It is heterogeneous and its coding by hand would have been extremely difficult [18].

12 Acknowledgements

This work took shape during consecutive visits at Carnegie-Mellon and Oxford University. I am grateful to H. T. Kung and C. A. R. Hoare for their invitations. I would also like to thank G. Jones, C.-H. Huang, W. Luk, P. Quinton, H. Ribas, and J. Sanders for discussions.

13 References

[1] Ametek Computer Research Division, "Series 2010 System, General Description", Issue 3, Ametek, Inc., Apr. 1988.

[2] M. C. Chen, "A Parallel Language and Its Compilation to Multiprocessor Machines", *J. Parallel and Distributed Computing 3*, 4 (Dec. 1986), 461–491.

[3] M. C. Chen, "Placement and Interconnection of Systolic Processing Elements: A New LU-Decomposition Algorithm", Research Report YALEU/DCS/RR-498, Department of Computer Science, Yale University, Oct. 1986.

[4] B. R. Engstrom and P. R. Cappello, "The SDEF Systolic Programming System", *Proc. 1987 Int. Conf. on Parallel Processing*, The Pennsylvania State University Press, 1987, 645-652; full paper: TRCS87-15, Department of Computer Science, UC Santa Barbara, Aug. 1987.

[5] T. Gross, M. Lam and J. Reinders, "Programming Warp in W2", Department of Computer Science, Carnegie-Mellon University.

[6] C. A. R. Hoare, "Communicating Sequential Processes" *Comm. ACM 21*, 8 (Aug. 1978), 666–677.

[7] C.-H. Huang, "The Mechanically Certified Derivation of Concurrency and its Application to Systolic Design", Ph. D. Thesis, Department of Computer Sciences, The University of Texas at Austin, Aug. 1987.

[8] C.-H. Huang and C. Lengauer, "The Derivation of Systolic Implementations of Programs", *Acta Informatica 24*, 6 (Nov. 1987), 595–632.

[9] C.-H. Huang and C. Lengauer, "Mechanically Derived Systolic Solutions to the Algebraic Path Problem", in *VLSI and Computers (CompEuro 87)*, W. E. Proebster and H. Reiner (eds.), IEEE Computer Society Press, 1987, 307–310; full paper: TR-86-28, Department of Computer Sciences, The University of Texas at Austin, Dec. 1986.

[10] D. G. Hudson III, Graduate Class Project, Fall 1988. A tech. report is forth-coming.

[11] Inmos, Ltd., occam *Programming Manual*, Prentice/Hall Int., Series in Computer Science, 1984.

[12] Inmos, Ltd., *Transputer Reference Manual*, Prentice Hall, 1988.

[13] H. T. Kung and C. E. Leiserson, "Algorithms for VLSI Processor Arrays", in *Introduction to VLSI Systems*, C. Mead and L. Conway (eds.), Addison-Wesley, 1980, Sect. 8.3.

[14] H. T. Kung et al., "The Warp Computer: Architecture, Implementation, and Performance", *IEEE Trans. on Computers C-36*, 12 (Dec. 1987), 1523–1538.

[15] M. Lam, "A Systolic Array Optimizing Compiler", Ph. D. Thesis, Department of Computer Science, Carnegie-Mellon University, May 1987.

[16] P. Lee, Z. Kedem, "Synthesizing Linear Array Algorithms from Nested for Loop Algorithms", Tech. Report 355, Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, Mar. 1988

[17] C. Lengauer, "A Methodology for Programming with Concurrency: The Formalism", *Science of Computer Programming 2*, 1 (Oct. 1982), 19–52.

[18] C. Lengauer, "On the Projection Problem in Systolic Design", Tech. Report CMU-CS-88-102, Computer Science Department, Carnegie-Mellon University, Feb. 1988.

[19] C. Lengauer and J. Sanders, "The Projection of Systolic Programs", Extended Abstract, Dec. 1988. A tech. report is forth-coming.

[20] P. J. Lieu, Personal communication, Department of Computer Science, Carnegie-Mellon University, Nov. 1987.

[21] J. R. McGraw et al., "SISAL Language Reference Manual, Version 1.2", Manual M-146, Lawrence Livermore National Laboratory, University of California at Davis, Mar. 1985.

[22] D. I. Moldovan and J. A. B. Fortes, "Partitioning and Mapping Algorithms into Fixed-Size Systolic Arrays", *IEEE Trans. on Computers C-35*, 1 (Jan. 1986), 1–12.

[23] D.I. Moldovan, "ADVIS: A Software Package for the Design of Systolic Arrays", *IEEE Trans. on Computer-Aided Design CAD-6*, 1 (Jan. 1987), 33–40.

[24] P. Quinton, "The Systematic Design of Systolic Arrays", Tech. Report 193, Publication Interne IRISA, Apr. 1983; also: TR84-11, The Microelectronics Center of North Carolina, May 1984.

[25] P. Quinton et al., "Designing Systolic Arrays with DIASTOL", in *VLSI Signal Processing II*, S.-Y. Kung, R. E. Owen, and J. G. Nash (eds.), IEEE Press, 1986, 93–105.

[26] P. Quinton et al., "Synthesizing Systolic Arrays Using DIASTOL", in *Systolic Arrays*, W. Moore, A. McCabe, and R. Urquart (eds.), Adam Hilger, 1987, 25–36.

[27] P. Quinton, "Mapping Recurrences on Parallel Architectures", in *Supercomputing '88 (ICS 88)*, Vol. III: *Supercomputer Design: Hardware & Software*, L. P. Kartashev and S. I. Kartashev (eds.), Int. Supercomputing Institute, Inc., 1988, 1–8.

[28] S. K. Rao, "Regular Iterative Algorithms and their Implementations on Processor Arrays", Ph. D. Thesis, Department of Electrical Engineering, Stanford University, Oct. 1985.

A Proofs

Code Generation Theorem:

Let *step* be linear and *place* be linear with coefficients in $\{-1, 0, +1\}$. Let the basic operation have n arguments: $x_0{:}\cdots{:}x_{n-1}$. Let the independence criterion be "no shared access of variables". Then the synchronous parallel execution specifies the same partial order of basic operations as the asynchronous process and communication structure derived from it with the previous augmentation.

Proof:

We construct the two partial orders and show that they are equal.

(a) The synchronous partial order is derived from the independence of basic operations and from the step function. Let us denote the independence of $x_0{:}\cdots{:}x_{n-1}$ and $y_0{:}\cdots{:}y_{n-1}$ by

$$x_0{:}\cdots{:}x_{n-1} \ \textbf{ind} \ y_0{:}\cdots{:}y_{n-1}$$

We write $x_0{:}\cdots{:}x_{n-1} \prec_s y_0{:}\cdots{:}y_{n-1}$ and say $x_0{:}\cdots{:}x_{n-1}$ *synchronously precedes* $y_0{:}\cdots{:}y_{n-1}$ iff

$$\neg(x_0{:}\cdots{:}x_{n-1} \ \textbf{ind} \ y_0{:}\cdots{:}y_{n-1}) \ \wedge \ \mathit{step}(x_0{:}\cdots{:}x_{n-1}) < \mathit{step}(y_0{:}\cdots{:}y_{n-1})$$

Following our program format, a variable's subscripts must comprise at least $n-1$ out of the n arguments of the basic operation. For the purpose of this proof, the order of the subscripts in the subscript list is irrelevant. Let us assume it matches their order in the argument list of the basic operation. A variable that is subscripted by all n arguments is accessed only once during the computation and, therefore, cannot be the cause of a dependence. We need not consider it. Let $Vars$ be the set of the names of all variables that are accessed by $n-1$ arguments. Denote the position of the argument that does not appear in the index list of the variable by i_v. Our criterion for independence is "no shared access of variables":

$$(\forall v \in Vars : \langle x_0, ..., x_{i_v-1}, x_{i_v+1}, ..., x_{n-1}\rangle \neq \langle y_0, ..., y_{i_v-1}, y_{i_v+1}, ..., y_{n-1}\rangle)$$
$$\Longrightarrow$$
$$x_0{:}\cdots{:}x_{n-1} \ \textbf{ind} \ y_0{:}\cdots{:}y_{n-1}$$

Applying de Morgan's laws and plugging in the independence criterion, we obtain:

$$(\exists v \in Vars : \langle x_0, ..., x_{i_v-1}, x_{i_v+1}, ..., x_{n-1}\rangle = \langle y_0, ..., y_{i_v-1}, y_{i_v+1}, ..., y_{n-1}\rangle \\ \wedge\ step(x_0{:}\cdots{:}x_{n-1}) < step(y_0{:}\cdots{:}y_{n-1}))$$

(b) We construct the asynchronous partial order by disjunctively composing the sequencing constraints that are enforced by the semantics of our asynchronous program – one disjunct per variable. Pick any variable v that is communicated from $x_0{:}\cdots{:}x_{n-1}$ to $y_0{:}\cdots{:}y_{n-1}$. According to our construction, v must be accessed by both basic operations. Thus, there must be some i_v such that $\langle x_0, ..., x_{i_v-1}, x_{i_v+1}, ..., x_{n-1}\rangle = \langle y_0, ..., y_{i_v-1}, y_{i_v+1}, ..., y_{n-1}\rangle$. Also, $x_{i_v} \neq y_{i_v}$, since the programs we accept may not contain identical calls of the basic operation. For all pairs of basic operations of which the first communicates v to the second, x_{i_v} and y_{i_v} must be in the same order – let us denote it by $x_{i_v} <_v y_{i_v}$. To understand that, note the following. Our augmentation of basic operations with communications guarantees

$$x_{i_v} <_v y_{i_v} \iff step(x_0{:}\cdots{:}x_{n-1}) < step(y_0{:}\cdots{:}y_{n-1})$$

But the order imposed by *step* on the basic operations that access v is the same for each pair, since *step* is monotone as a non-constant linear function.

We write $x_0{:}\cdots{:}x_{n-1} \prec_a y_0{:}\cdots{:}y_{n-1}$ and say $x_0{:}\cdots{:}x_{n-1}$ *asynchronously precedes* $y_0{:}\cdots{:}y_{n-1}$ iff

$$(\exists v \in Vars : \langle x_0, ..., x_{i_v-1}, x_{i_v+1}, ..., x_{n-1}\rangle = \langle y_0, ..., y_{i_v-1}, y_{i_v+1}, ..., y_{n-1}\rangle \ \wedge\ x_i <_v y_i)$$

Now observe that the relations $\prec_s$ and $\prec_a$ are both disjunctions over the same range, and that the mutual equations in the disjuncts match. And we have just defined the order $x_i <_v y_i$ to match the order $step(x_0{:}\cdots{:}x_{n-1}) < step(y_0{:}\cdots{:}y_{n-1})$.

(*End of Proof*)

Data Distribution Theorem:

Assume that *step* and *place* are linear and that *flow* is well-defined, that is, for any particular stream, *flow* is constant. Then, for any particular stream, the data layout function *pattern* is injective. That is, no two elements within a stream are laid out at the same point.

Proof:

Pattern is linear. A linear function is either constant (then all multiplicative coefficients are zero) or injective (then at least one multiplicative coefficient is not zero). We show that *pattern* cannot be constant. Pick any one of its dimensions arbitrarily:

(a) *Flow* is zero in this dimension. Then *pattern* equals *place* in this dimension. But *place* cannot be constant in any dimension and be consistent.

(b) *Flow* is not zero in this dimension. For *pattern* to be constant, *place* in this dimension and *step* must be linearly dependent, i.e., inconsistent.

(*End of Proof*)

B Systolic $n \times n$ Matrix Composition/Decomposition

The pre- and postprocessing phases are omitted. Other than that, the program is executable. Note the limitations of the original version of occam [11]: only one-dimensional arrays, no floating point arithmetic (we use a software floating point package)[10], full parenthesization of expressions.

B.1 The occam Program

```
VAR AIn[n*n], BIn[n*n], CIn[n*n],          -- input matrices
    AOut[n*n], BOut[n*n], COut[n*n]:       -- output matrices

CHAN Right[n*(n+1)]:                       -- horizontal channels
CHAN Up[(n+1)*n]:                          -- vertical channels

SEQ

  -- PREPROCESSING

  "read in input matrices"
  "initialize output matrices"

  PAR

    -- INPUT CELLS

    -- vertical input: inject stream A

    PAR col = [0 FOR n]
      SEQ row = [0 FOR n]
        Up[((n+1)*col)+0] ! AIn[(n*col)+row]

    -- horizontal input: load stream C and inject stream B

    PAR row = [0 FOR n]
      SEQ

        SEQ col = [0 FOR n]
          Right[(n*0)+row] ! CIn[(n*col)+row]

        SEQ col = [0 FOR n]
          Right[(n*0)+row] ! BIn[(n*col)+row]
```

[10] Read `RealOp(z,x,Op,y)` as `z := x Op y`

```
-- COMPUTATION CELLS

PAR col = [0 FOR n]
  PAR row = [0 FOR n]

    VAR AElement, BElement, CElement:

    SEQ

      -- load stream C

      Right[(n*col)+row] ? CElement

      SEQ unused = [0 FOR (n-1)-col]
        VAR tmp:
        SEQ
          Right[(n*col)+row] ? tmp
          Right[(n*(col+1))+row] ! tmp

      -- do the computation

      SEQ k = [0 FOR n]
        SEQ

          PAR
            Up[((n+1)*col)+row] ? AElement
            Right[(n*col)+row] ? BElement

          BasicOp(col, row, k, AElement, BElement, CElement)

          PAR
            Up[((n+1)*col)+row+1] ! AElement
            Right[(n*(col+1))+row] ! BElement

      -- recover stream C

      SEQ k = [0 FOR col]
        VAR tmp:
        SEQ
          Right[(n*col)+row] ? tmp
          Right[(n*(col+1))+row] ! tmp

      Right[(n*(col+1))+row] ! CElement
```

```
    -- OUTPUT CELLS

    -- horizontal output: extract stream B and recover stream C

    PAR row = [0 FOR n]
      SEQ

        SEQ col = [0 FOR n]
          Right[(n*n)+row] ? BOut[(n*col)+row]

        SEQ col = [0 FOR n]
          Right[(n*n)+row] ? COut[(n*col)+row]

    -- vertical output: extract stream A

    PAR col = [0 FOR n]
      SEQ row = [0 FOR n]
        Up[((n+1)*col)+n] ? AOut[(n*col)+row]

  -- POSTPROCESSING

  "read out output matrices"
```

B.2 The Basic Operation for Matrix Composition in occam

```
PROC BasicOp(VALUE i, j, k, AElement, BElement, VAR CElement) =

  VAR tmp:

  SEQ

    RealOp(tmp, ProjElement, Mul, MoveElement)
    RealOp(StatElement, StatElement, Add, tmp) :
```

B.3 The Basic Operation for Matrix Decomposition in occam

```
PROC BasicOp(VALUE i, j, k, VAR AElement, BElement, CElement) =

  VAR tmp:

  SEQ

    IF

      (i<=j) AND (i=k)
        BElement := CElement
```

```
(i>j) AND (j=k)
  SEQ
    RealOp(tmp, One, Div, BElement)
    RealOp(AElement, CElement, Mul, tmp)

(i>k) AND (j>k)
  SEQ
    RealOp(tmp, AElement, Mul, BElement)
    RealOp(CElement, CElement, Sub, tmp)

TRUE
  SKIP :
```