LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - The University of Edinburgh

An Improved Systolic Array for String Correction

by

Davide Sangiorgi
Christian Lengauer

An Improved Systolic Array for String Correction

LFCS Report Series

ECS-LFCS-91-153

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

May 1991

Copyright © 1991, LFCS

Copyright © 1991, Laboratory for Foundations of Computer Science,
University of Edinburgh. All rights reserved.

Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.

To appear in *Proc. Advanced Research Workshop on Correct Hardware Design Methodologies*, Turin, Italy, June 1991.

An Improved Systolic Array for String Correction

Davide Sangiorgi Christian Lengauer

Laboratory for Foundations of Computer Science
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ, U.K.

Abstract

The theory of systolic automata has served in the past for comparative assessments of the computational power of different systolic arrays. Here, it is used in the design of a systolic array. Theorems from the theory of systolic automata guide the transformational improvement of a known systolic array for string correction.

1 Introduction

Mechanical systolic design methods have now evolved into an effective tool for the design of systolic arrays in hardware or software (e.g., [LBH91, RK88]). However, one shortcoming of these methods is their restricted scope, as long as one insists on a fully automated derivation. Efforts of extending the scope are under way [QvD89, Raj89]. At present and for the foreseeable future, adjustments of the source description offered to the method or the target description returned by the method are beneficial in many cases. (See [LX90] for a large example.) Such adjustments should be carried out with formal rigour and justification.

We improve a mechanically derived systolic array for string correction with transformation techniques from the theory of systolic automata.

The remainder of the paper is structured as follows:

Sect. 2 introduces machine models and theorems of the theory of systolic automata.

Sect. 3 reviews mechanical systolic design methods; one of them is being employed in the improvement of the systolic array.

Sect. 4 presents a formal specification of the problem of string correction.

Sect. 5 presents the known systolic array for string correction that is going to be improved.

Sect. 6 is the main section of the paper. It derives the improved systolic array from a different known systolic array in four transformation steps.

Sect. 7 evaluates the improved array and the use of systolic automata theory in its derivation.

2 Systolic Automata

The theorems presented here were introduced in the framework of systolic automata [Gru84, Gru90]. *Systolic automata* represent language acceptors; the set of possible states that a computing element may assume is divided into final states and not final states. For our purpose, the notion of a final state is not important and only complicates matters. Therefore, we shall work in the simpler framework of systolic machines. Systolic machines are systolic automata without final states.

A *systolic machine* is a quadruple (S, A, g, f). S defines the underlying structure S, A the input alphabet, g the input function and f the transition function.

The underlying structure of a systolic machine is typically an array or tree of memoryless identical processors. The processors are connected by unidirectional links in a regular pattern. We call the links that supply information to (from) the processor *in-links* (*out-links*). A processor may also have an extra link that supplies it with external data. The model does not provide for the return of data to the external environment; instead the data on the out-links determine the outcome of a computation.

The alphabet A and the domains and codomains of f and g must be finite. If n is the number of out-links of a processor in S, then we will assume that the possible values returned by g or f are n-tuples whose i-th component $(0 < i \leq n)$ represents the output on the i-th out-link. The special symbol $\sharp$ represents non-significant information.

The computational power of this model does not change if we require that all out-links carry the same output. This requirement is usually made in the theory of systolic automata. Our weaker requirement leads to simpler transformations, especially in the proof of the Thm. 3.

Let $w = w_1 \ldots w_n$, where $(\forall\ i : 0 < i \leq n : w_i \in A)$, be an input word. Each letter of w is input to a processor in the *input row* – one of the rows of processors in S whose length does not exceed the length of the input. The order of the letters in w must be maintained; i.e. if w_i enters processor P, then w_{i+1} enters some processor to the left of P. The processors in the input row that receive no letter of w are supplied with $\sharp$.

Processors in the input row modify their data by applying the input function g; all other processors modify it by applying the transition function f. A processor that receives nothing else but $\sharp$ sends nothing else but $\sharp$.

2.1 A Classification of Systolic Machines

There are several types of systolic machines [CC84, CGS84a, CGS84b, CGS86]:

Cellular Machine (CM). The underlying structure is an infinite linear array of identical processors with unit propagation delay on all interconnections. Each processor is two-way connected with both neighbours and one-way connected with itself. We do not need to distinguish an input row; there is only one row of processors.

One-Way Cellular Machine (OCM). A CM with one-way instead of two-way processor connections.

Trellis Grid Machine (TGM). The connection structure is a trellis grid (Fig. 1(b)). The rows are counted bottom-up starting with 1; the input row is always row 1. Diagonal links have unit delay, vertical links have a delay of 2. The computation proceeds bottom-up and row-wise in parallel. That is, at step i, only the processors in row i are computing. Border processors may have in-links missing. Then the assumption is that they take $\sharp$ for any missing in-link.

There are different ways of introducing the notion of output in CMs, OCMs and TGMs. For TGMs, one of the most studied notions is *diagonal acceptance* [Gru84]. A TGM with diagonal acceptance is called a *trellis.*

Trellis Machine (TM). A trellis is an infinite triangularly shaped trellis grid (Fig. 7 depicts an example that we shall deal with later in the paper). A trellis has the same connection pattern as a trellis grid. The input row for input w is one of the rows with a number of processors greater than or equal to the length of w. The computation proceeds bottom-up and row-wise in parallel. The output of the processors in the top row is the output of the computation. The same border convention as in the case of the TGM applies.

Various modifications of the basic TM model (and its derivatives) help to understand better the computational power of, e.g., different ways of supplying the input or different ways of connecting the processors.

Normalized Trellis Machine (nTM). The vertical connections of the trellis are removed.

Superstable Trellis Machine. The output of the TM does not depend on the way in which the input is provided, as long as the ordering of the letters in the input is maintained. We add the usually not required condition that the left-most letter of the input must be supplied to the left-most processor of the input row; this simplifies one of our theorems (Thm. 3).

Real-time Trellis Machine. The letters of the input word are supplied to consecutive processors of the the single row whose length equals the length of the input.

2.2 Simulation

We say that the systolic machine M_1 *simulates* the systolic machine M_2 iff there exists a transformation which maps each computation in M_1 into an (in some sense) equivalent computation in M_2. For our purposes, this quite informal notion of simulation will suffice. The proofs of the following theorems are based on a more formal notion [Gru84, Gru90].

Theorem 1 [Gru84] *Any CM can be simulated by a TGM.*

Proof Sketch. Unfolding the computation of a CM yields the network shown in Fig. 1(a), where t represents time. This network can be simulated by the network of Fig. 1(b), which is obtained from the former by replacing all intersections by new processors. $\square$

Theorem 2 (Normalization of Trellises) [CGS84a] *Any TM can be simulated by an nTM.*

Proof Sketch. The information that is sent in a TM along a vertical link is transmitted in an nTM (which does not have that link) along the two diagonal links instead. See Fig. 2. The dotted links represent the links involved in the communication; on the left is the TM, on the right the nTM. $\square$

Since the simulation of an nTM by a TM is straight-forward, we can conclude that TMs and nTMs are computationally equivalent. The following theorem states that real-time nTMs and superstable nTM are also computationally equivalent.

Theorem 3 (Superstability of Trellises) [CGS86] *Let M_1 be a real-time nTM with transition function h_1 and M_2 a nTM with transition function h_2 defined by the table below (for two inputs $v, z \neq \sharp$). Then*

1. M_2 is superstable;

2. M_2 simulates M_1.

h_2	v	$\sharp$
z	$h_1(v,z)$	$(\sharp, z)$
$\sharp$	$(v, \sharp)$	$(\sharp, \sharp)$

Proof Sketch. We show (2); (1) is similar. The proof of (2) relies on the following property. Let w be an input, and let w_i be the i-th letter of w, $0 < i \leq |w|$. If w_j is another letter $(i \neq j)$ and P (P') is the processor at the cross between the two diagonals passing from the processors in M_1 (M_2) that receive w_i and w_j, then P and P' perform the same computation step. See Fig. 3. P (P') is the black node.

This property can be proved by induction on the distance between the processors that receive w_i and w_j. Then the definition of function h_2 and the condition in the definition of superstability (that the first letter of the input enters the first processor of the input row) forces the outputs of M_2 and M_1 to be the identical. $\square$

Theorem 4 [CGS84a] *Any nTM can be simulated by an OCM.*

Proof Sketch. Take an nTM (Fig. 4(a)). We obtain the diagonal unrolling of an OCM by shifting the i-th row from the bottom $(i \geq 0)$ $i/2$ places to the right (Fig. 4(b)). $\square$

3 Systolic Design Methods

The starting point of mechanical systolic design methods is, essentially, either an imperative program [HL87, LBH91] or a functional program [Qui84, RK88] without parallel commands or communication directives. We concentrate on the imperative viewpoint.

3.1 The Source Description

Our source specification is of the following form:

```
for $x_0$ from $lb_0$ by $st_0$ to $rb_0$ do
    for $x_1$ from $lb_1$ by $st_1$ to $rb_1$ do
        ⋮
        for $x_{r-1}$ from $lb_{r-1}$ by $st_{r-1}$ to $rb_{r-1}$ do
            $x_0:x_1:\cdots:x_{r-1}$
```

We call $x_0:x_1:\cdots:x_{r-1}$ the *basic statement* of the program. It is, in our case, imperative but could also be functional. It may not contain non-local references other than to variables subscripted by the x_i. The bounds lb_i and rb_i are linear expressions in the loop indices x_0 to x_{i-1} $(0 \leq i < r)$ and in additional variables that specify the problem size. The steps st_i are constants. Certain other restrictions apply [LBH91, RK88]; for an imperative source:

1. each non-local variable is subscripted by exactly $r-1$ linear expressions in the loop indices; their coefficient matrix has rank $r-1$;

2. non-local variables that agree in name must agree in subscripts (in order to ensure the well-definedness of flow; see Sect. 3.2).

3.2 The Target Description

Both for a functional and an imperative source, the resulting systolic array is described by two functions. Let $\mathbb{Z}$ denote the integers, $\mathbb{Q}$ the rationals and $Op \subset \mathbb{Z}^r$ the *index space* over which the basic operations range:

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

place : $Op \longrightarrow \mathbb{Z}^{r-1}$ specifies a spatial distribution of the program's operations. The dimension of the layout space is one less than the number of arguments of the operations.

After the mechanical derivation of a shortest distribution in time, one chooses a compatible distribution in space by a search [HL87, Mol83]. With this choice, one can optimize other aspects of the array, e.g., its throughput, number of processors or channels or storage registers per processor, and so on. If step and place are linear, they determine the direction and distance at which data travel through the array. Let V be the set of variables of the program:

flow : $V \longrightarrow \mathbb{Q}^{r-1}$ 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 then

$$\mathsf{flow}(v) = (\mathsf{place}(s_1) - \mathsf{place}(s_0))/(\mathsf{step}(s_1) - \mathsf{step}(s_0))$$

The restrictions on the source guarantee that flow is well-defined, i.e., independent of the choice of s_0 and s_1 [HL87, LBH91, Qui84, RK88].

4 The Problem: String Correction

The problem is to compare two strings of characters by computing a distance, the so-called *edit distance*, which represents the minimal cost of modifying one of the strings to match the other, one character at a time [WF76]. Modifications are substitution, deletion and insertion. With these three operations, any string may be modified into any other string. We need to identify the shortest sequence of such modifications.

Let a string be a finite sequence of characters. If the length of string $\mathbf{X}$ is n, denote the prefix of $\mathbf{X}$ of length i $(0 \le i \le n)$ by $\mathbf{X}_i$. Denote the cost of substituting character y for character x by $ed(x,y)$; a possible example is:

$$ed(x,y) = \textbf{if } x = y \rightarrow 0 \ [] \ x \neq y \rightarrow 1 \textbf{ fi}.$$

Denote the null character by $\perp$, i.e., $ed(x,\perp)$ is the cost of the deletion of character x and $ed(\perp,y)$ the cost of the insertion of character y. Let $\mathbf{X} = (\underline{\text{seq}}\ i : 0 < i \le n : x_i)$ and $\mathbf{Y} = (\underline{\text{seq}}\ j : 0 < j \le m : y_j)$ be strings. The edit distance of $\mathbf{X}$ and $\mathbf{Y}$, denoted $\mathcal{D}(\mathbf{X}_n, \mathbf{Y}_m)$, is described by the following recurrence relation on the length of $\mathbf{X}$ and $\mathbf{Y}$:

$$\mathcal{D}(\mathbf{X}_0,\mathbf{Y}_0) = 0, \quad (\forall\, i : 0<i\le n : \mathcal{D}(\mathbf{X}_i,\mathbf{Y}_0) = +\infty), \quad (\forall\, j : 0<j\le m : \mathcal{D}(\mathbf{X}_0,\mathbf{Y}_j) = +\infty),$$

$$(\forall\, i : 0<i\le n : (\forall\, j : 0<j\le m : \mathcal{D}(\mathbf{X}_i,\mathbf{Y}_j) \ = \ \min\begin{pmatrix} \mathcal{D}(\mathbf{X}_{i-1},\mathbf{Y}_{j-1}) & + \ ed(x_i,y_j) \\ \mathcal{D}(\mathbf{X}_{i-1},\mathbf{Y}_j) & + \ ed(x_i,\perp) \\ \mathcal{D}(\mathbf{X}_i,\mathbf{Y}_{j-1}) & + \ ed(\perp,y_j) \end{pmatrix}))$$

5 A Known Systolic Array

The following imperative program satisfies the specification; it can be derived with mechanical systolic design methods. $\mathbf{X} = (\underline{\text{seq}}\ i : 0<i\le n : x_i)$ and $\mathbf{Y} = (\underline{\text{seq}}\ j : 0<j\le m : y_j)$ are represented as vectors x and y of characters. With the following initializations:

$$(\forall\, i,j : 0<i\le n \wedge 0<j\le m : a_i := +\infty;\ \ b_{i-j} := \textbf{if } i=j \rightarrow 0 \ [] \ i\neq j \rightarrow +\infty \textbf{ fi};\ \ d_j := +\infty)$$

and with a given function *ed*, the following program computes the edit distance of the two strings – $\mathcal{D}(\mathbf{X}_n, \mathbf{Y}_m)$ in the specification – and returns it in a_n, b_{n-m} and d_m:

```
for $i$ from 1 to $n$ do
for $j$ from 1 to $m$ do
  $i{:}j$
```

$$i{:}j :: a_i, b_{i-j}, d_j := h(a_i, b_{i-j}, d_j, x_i, y_j)$$

$$h(arg1, arg2, arg3, arg4, arg5) \quad = \quad \min \begin{pmatrix} arg1 + ed(arg4, arg5) \\ arg2 + ed(arg4, \perp) \\ arg3 + ed(\perp, arg5) \end{pmatrix}$$

The multiple assignment assigns all three target variables the same value [Gri78]. There are two subtleties in the derivation of this program:

1. The introduction of variables that are not in the specification. This is necessary to keep flow well-defined (Sect. 3.2). It can be automated with available systolic design technology [QvD89, Raj89].
2. The switch from double indexing in the specification to single indexing in the source program. This is advisable for an imperative source, in order to comply with the syntactic restrictions laid down in Sect. 3.1. It is possible, since value $\mathcal{D}(\mathbf{X}_n, \mathbf{Y}_m)$ is the only output of interest.

We derive the optimal step function and choose a compatible place function:

$$\mathsf{step}(i, j) = i+j-1 \qquad\qquad \mathsf{place}(i, j) = i$$

These definitions of step and place lead to the following data flows:

$$\mathsf{flow}(a) = 0 \quad \mathsf{flow}(b) = +1/2 \quad \mathsf{flow}(d) = +1 \quad \mathsf{flow}(x) = 0 \quad \mathsf{flow}(y) = +1$$

This array has been proposed before and built [FL88]. The derivation in [FL88] is based on a doubly-indexed functional source program that reflects the specification directly but, when switching to imperative target code, similar modifications are made as we made in the source here.

Fig. 5 depicts the array for input size $n=3$ and $m=2$ at the first step. Circles represent processors, filled-in circles represent active processors. That is, the left processor is active in the first step. Arrows represent data movement. As indicated by flow, streams a and x are stationary and streams b, d and y move to the right. The array is an OCM: all moving data travel in the same direction. Its shortcoming is that stream b travels to slowly to encounter a processor at each step; one delay cell is required between each two neighbouring processors.

6 The New Systolic Array

Our goal is to eliminate the delays in the previous OCM at no cost of execution time. For reasons to be explained later (Sect. 6.5), we shall do so by starting with a mechanically derived CM (that has also been proposed before [LL87]) and applying a series of transformations that leads to an OCM without delays.

Each transformation is justified by one of the simulation theorems. However, we shall recast the transformations in terms of the imperative method. We apply four transformations. Each modifies step, place or the basic operation. T_i is the machine, S_i the operation, step_i the parallel execution and place_i the processor layout of transformation i. In illustrating figures, we always depict the input size $n=3$ and $m=2$.

T_0 is the initial CM. In the imperative method, it is represented by the following basic statement and distribution functions:

$$S_0(i,j) :: \; i{:}j \qquad \mathsf{step}_0(i,j) = i+j-1 \qquad \mathsf{place}_0(i,j) = i-j$$

with the same initializations as in the previous section. That is, it can be derived mechanically from the source program of the previous section by choosing a different place function. Fig. 6 depicts the array at the first step. The flows are:

$$\mathsf{flow}_0(a) = -1 \qquad \mathsf{flow}_0(b) = 0 \qquad \mathsf{flow}_0(d) = +1 \qquad \mathsf{flow}_0(x) = -1 \qquad \mathsf{flow}_0(y) = +1$$

Processors compute only at every other step in this array; at the rest of the steps they only propagate data without using or modifying them. (One of our transformations exploits this fact.)

6.1 Transformation 1

By applying Thm. 1, we first transform the CM T_0 into a TGM T; the two-dimensional layout simplifies the successive transformations. The sequences of computations that are performed by one processor of T_0 are unrolled into space. (This yields the index space of the problem [Qui84, RK88].) Then extra rows are added to obtain T. Each processor of T_0 corresponds to a column of processors of T. Note that the data flows of T_0 and T differ. Stream b, stationary in T_0, moves vertically in T. Streams a and x (d and y) move along the south-east to north-west (south-west to north-east) diagonal in T.

Since we are interested in a final value, we need only be concerned with a finite part of the infinite trellis grid. The result is the TM T_1. It is depicted in Fig. 7. The dark processors are the ones that do the computing.

Let us port this transformation to the target end of the imperative method. Because new rows of processors have been introduced in the transformation, an operation which is executed at step t by T_0 is executed at step $2{\cdot}t-1$ by T_1. Of the processor layout of T_1, the column coordinate is the place function of T_0, and the row coordinate is (always in TMs) the step function of T_1 itself. This packs processors more tightly than in T_0: the rows added in the transformation have a fractional column coordinate: $1/2$. (In Fig. 7, a unit in columns is half as wide as a unit in rows.)

$S_1(i,j) :: \; S_0(i,j) \qquad \mathsf{step}_1(i,j) = 2\cdot\mathsf{step}_0(i,j)-1 \qquad \mathsf{place}_1(i,j) = (\mathsf{place}_0(i,j), \mathsf{step}_1(i,j))$

6.2 Transformation 2

We eliminate the vertical links following Thm. 2. Remember that a processor computes only at every other step. At the remaining steps, it only passes on variables. In T_1, this is reflected by the fact that, per column, only every other processor computes. The processors inbetween pass but do not otherwise use variables. See Fig. 8. Consider the two adjacent computing processors labelled P and Q. We can redirect their vertical communications diagonally, applying the proof of Thm. 2 in a geometry scaled by a factor of 2. This yields simpler successive transformations than applying the theorem directly. The dotted lines represent the links involved in the communication. To depict T_2, remove the vertical links in Fig. 7.

In the imperative method, this transformation is represented by the introduction of new variables c_j $(0<j\leq m)$ moving left to right, and e_i $(0<i\leq n)$ moving right to left. Consider Fig. 8. The information that is sent by T_1 from P to Q is sent by T_2 first from P to R, applying c_j, and then from R to Q, applying e_{i+1}. The basic statement changes to:

$$\begin{pmatrix} a_i, c_j, d_j := h(a_i, e_i, d_j, x_i, y_j) \\ e_i := c_j \end{pmatrix}$$

Again, all assignments are unordered, but the two right-hand sides must be evaluated before any assignment is made [Gri78].

The flow of c_j (e_i) is as the flow of d_j (a_i). Moreover, the initial value of c_j is useless; therefore, we can make it the initial value of d_j. Then d_j and c_j must also be assigned the same final value by the basic statement. Hence, we can eliminate one of the two variables – say c_j. To preserve the naming conventions in the OCM that we are trying to improve, we rename e to b.

$$S_2(i,j) :: \begin{pmatrix} a_i, d_j := h(a_i, b_i, d_j, x_i, y_j) \\ b_i := a_i \end{pmatrix}$$

$$\mathsf{step}_2(i,j) = \mathsf{step}_1(i,j) \qquad \mathsf{place}_2(i,j) = \mathsf{place}_1(i,j)$$

The initialization of b is: $(\forall\, i,j : 0<i\leq n : b_i :=$ **if** $i=0 \rightarrow 0 \;[]\; i\neq 0 \rightarrow +\infty$ **fi**$)$.

6.3 Transformation 3

We improve the bad utilization of processors in T_2 by applying Thm. 3 with transition function $h_1 = (S_2(v,z), S_2(v,z))$. The transition function of T_2 matches the requirements of h_2: a processor performs a basic operation iff both its inputs are not $\sharp$; if one of the inputs is $\sharp$, the processor sends the other input unchanged along the diagonal along which it was received. From Thm. 3, we can conclude that T_2 represents a superstable trellis;

therefore, we do not modify the output produced if we feed the input into consecutive processors. The result is depicted in Fig. 9. Again, the dark processors do the computing.

We can rephrase this transformation in terms of the imperative method by directly following the proof of the Thm. 3. Let P_{x_i} (P_{y_j}) be the processor that receives x_i (y_j) from the external environment. In T_2, the processor at the intersection of the diagonals along which x_i and y_j travel executes $S_2(i,j)$. T_3 must have the same property; this determines the placement of $S_3(i,j)$. The time at which $S_3(i,j)$ is executed is determined by the distance between P_{x_i} and P_{y_j}:

$$S_3(i,j) :: \; S_2(i,j) \qquad \mathsf{step}_3(i,j) = i+j-1 \qquad \mathsf{place}_3(i,j) = ((j-i)/2, i+j-1)$$

6.4 Transformation 4

Applying Thm. 4, we interpret TM T_3 as the diagonal unrolling of an OCM. In terms of the imperative method, this changes only the place function; an operation which T_3 places at processor $((j-i)/2, i+j-1)$, for any j, is in T_4 at processor i. We have arrived at our target OCM:

$$S_4(i,j) :: \; S_3(i,j) \qquad \mathsf{step}_4(i,j) = i+j-1 \qquad \mathsf{place}_4(i,j) = i$$

Fig. 10 depicts the array at the first step. The flows are:

$$\mathsf{flow}_4(a) = 0 \qquad \mathsf{flow}_4(b) = 0 \qquad \mathsf{flow}_4(d) = +1 \qquad \mathsf{flow}_4(x) = 0 \qquad \mathsf{flow}_4(y) = +1$$

6.5 Summary

Let us briefly review our transformations:

- Transformation 1 obtains a two-dimensional structure, resembling the index space of the problem. We made this structure a trellis so that we could use the normalization theorem for trellises to eliminate the links of the troublesome stream b. For that, it was easiest to choose as our starting point a CM in which b is stationary.

- Transformation 2 exploits the normalization of trellises and leads to a modification of the basic statement.

- Transformation 3 compresses the resulting trellis to optimize the distributions in space and time.

- Transformation 4 rolls the compressed trellis up into our target OCM.

Transformation 2 is the key; it alters the semantics of the systolic array. The other transformations correspond to manipulations of the index space.

7 Conclusions

Our goals in this work were two-fold.

On the one hand, we were looking for a more efficient linear array for the string correction problem. We consider the OCM T_4 an improvement over the OCM presented initially (Sect. 5). There are no delay cells and the length of stream b has been halved. Both have the same execution time, number of processors and complexity of the basic operation: one minimization and three unordered assignments. The fact that stream b has become stationary, is not detrimental: since it holds intermediate values only, it need neither be loaded into nor recovered from the array. Indexing stream c with i and stream e with j in Transformation 2 (ditto for d and a) gives stream b a flow of $+1$ instead, should that be preferred.

On the other hand, we wanted to explore whether the theory of systolic automata can support mechanical systolic design methods. One of the benefits of using systolic automata theory is its formal basis. The transformations that we apply might have been obtained with a different methodology; but we would insist on an equal level of formality, especially in semantic modifications like Transformation 2. The future will have to reveal whether systolic automata theory can make practical contributions to the development of systolic arrays, in general. At present, no automatic tools for the kind of manipulations that we performed exists – neither based on the theory of systolic automata nor based on any other theory.

Our improved array happens to be derivable with the imperative method alone, but the source has to be chosen cleverly. We discovered this choice with the use of systolic automata theory. We are not aware that our improvement has been noted before.

8 Acknowledgement

We are grateful to Jingling Xue for insightful comments.

9 References

[CC84] K. Culik II and C. Choffrut. On real-time cellular automata and trellis automata. *Acta Informatica*, 21:393–407, 1984.

[CGS84a] K. Culik II, J. Gruska, and A. Salomaa. Systolic trellis automata, Part I. *Int. J. Computer Math.*, 15:195–212, 1984.

[CGS84b] K. Culik II, J. Gruska, and A. Salomaa. Systolic trellis automata, Part II. *Int. J. Computer Math.*, 16:3–22, 1984.

[CGS86] K. Culik II, J. Gruska, and A. Salomaa. Systolic trellis automata: Stability, decidability and complexity. *Information and Control*, 71:218–230, 1986.

[FL88] P. Frison and D. Lavenier. A VLSI systolic machine for string correction. In L. P. and S. I. Kartashev, editors, *Proc. Third Int. Conf. on Supercomputing (ICS 88), Vol. III: Supercomputer Design: Hardware & Software*, pages 19–24. Int. Supercomputing Institute, Inc., 1988.

[Gri78] D. Gries. The multiple assignment statement. *IEEE Trans. on Software Engineering*, SE-4(2):89–93, Mar. 1978.

[Gru84] J. Gruska. Systolic automata: Power, characterization, nonhomogeneity. In M. P. Chytil and V. Koubek, editors, *Proc. Mathematical Foundations of Computer Science (MFCS '84)*, Lecture Notes in Computer Science 176, pages 32–49. Springer-Verlag, 1984.

[Gru90] J. Gruska. Synthesis, structure and power of systolic computations. *Theoretical Computer Science*, 71:47–77, 1990.

[HL87] C.-H. Huang and C. Lengauer. The derivation of systolic implementations of programs. *Acta Informatica*, 24(6):595–632, Nov. 1987.

[LBH91] C. Lengauer, M. Barnett, and D. G. Hudson. Towards systolizing compilation. *Distributed Computing*, 5(1), 1991. To appear.

[LL87] R. Lipton and D. Lopresti. Comparing long strings on a short systolic array. In W. Moore, A. McCabe, and R. Urquart, editors, *Systolic Arrays*, pages 181–190. Adam Hilger, 1987.

[LX90] C. Lengauer and J. Xue. A systolic array for pyramidal algorithms. Technical Report ECS-LFCS-90-114, Department of Computer Science, University of Edinburgh, June 1990. To appear in *J. VLSI Signal Processing.*

[Mol83] D. I. Moldovan. On the design of algorithms for VLSI systolic arrays. *Proc. IEEE*, 71(1):113–120, Jan. 1983.

[Qui84] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proc. 11th Ann. Int. Symp. on Computer Architecture*, pages 208–214. IEEE Computer Society Press, 1984.

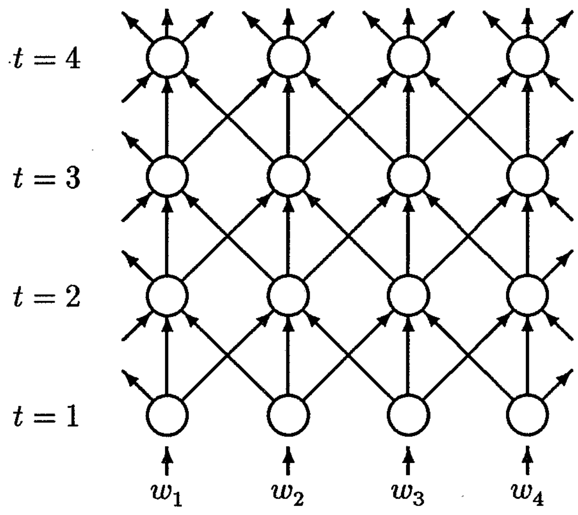
[QvD89] P. Quinton and V. van Dongen. The mapping of linear recurrence equations on regular arrays. *J. VLSI Signal Processing*, 1(2):95–113, Oct. 1989.

[Raj89] S. V. Rajopadhye. Synthesizing systolic arrays with control signals from recurrence equations. *Distributed Computing*, 3:88–105, 1989.

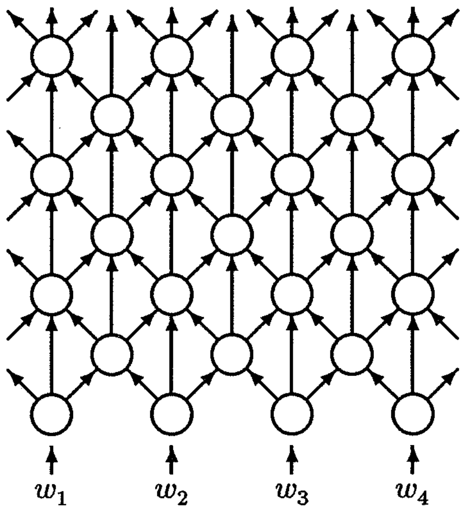
[RK88] S. K. Rao and T. Kailath. Regular iterative algorithms and their implementations on processor arrays. *Proc. IEEE*, 76(3):259–282, Mar. 1988.

[WF76] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, Jan. 1976.

A Figures



(a) A CM, Vertically Unfolded



(b) A Trellis Grid

Figure 1: From a CM to a TGM

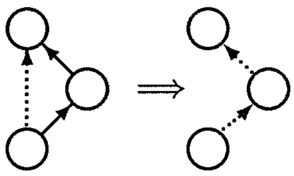


Figure 2: From a TM to an nTM

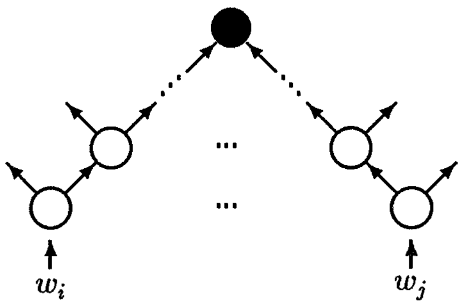
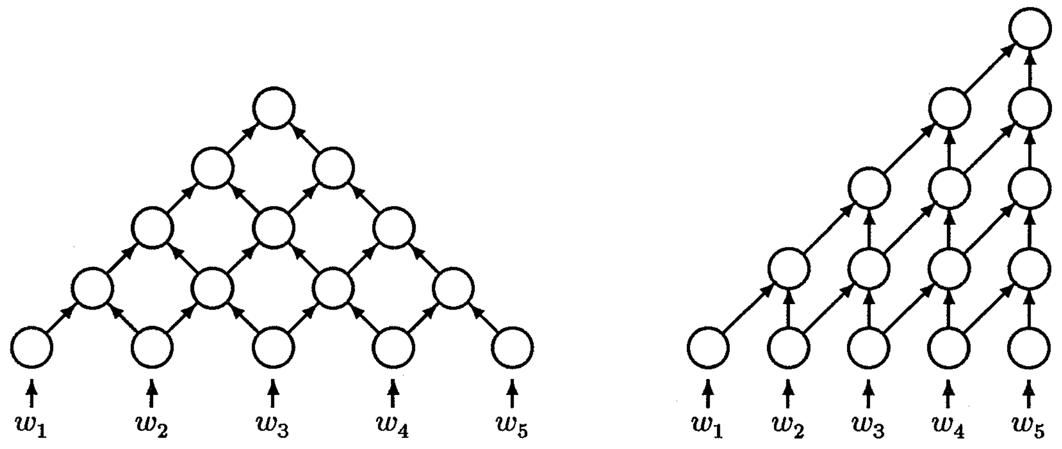


Figure 3: On the Superstability of Trellises



(a) An nTM shifted right is... (b) The Space-Time Diagram of an OCM

Figure 4: From an nTM to an OCM

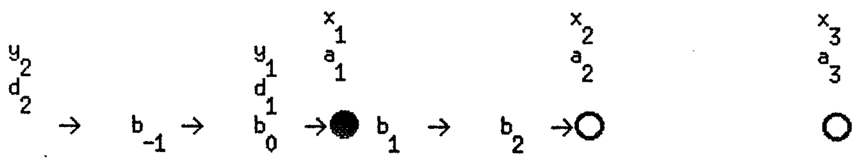


Figure 5: A Known OCM for String Correction

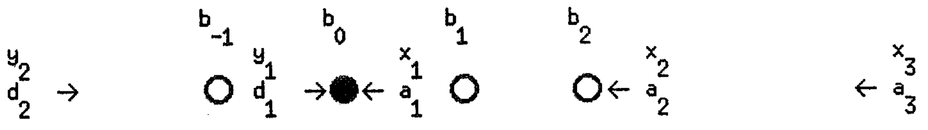


Figure 6: A Known CM for String Correction

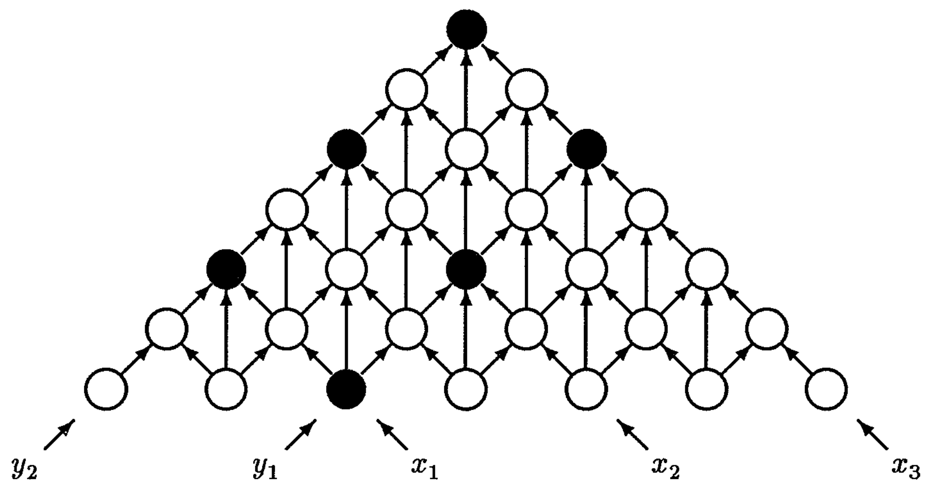


Figure 7: The Result of Transformation 1

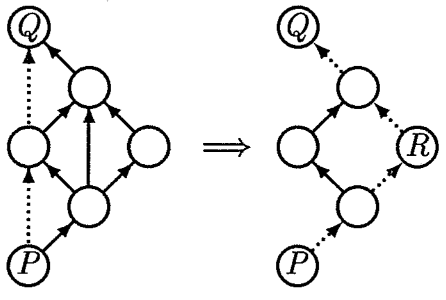


Figure 8: Transformation 2

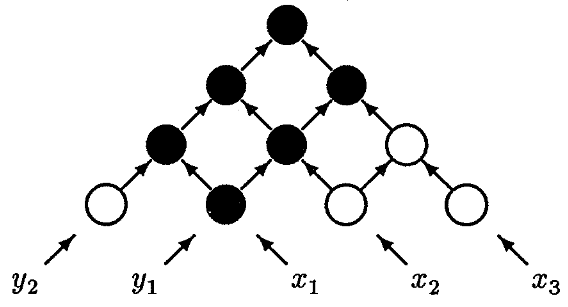


Figure 9: The Result of Transformation 3

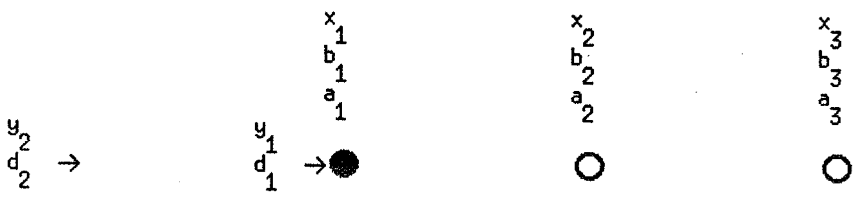


Figure 10: The New OCM for String Correction