PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT. © ICOT, 1988

PROGRAM TRANSFORMATION APPLIED TO THE DERIVATION OF SYSTOLIC ARRAYS

Norihiko Yoshida

Department of Computer Science and Communication Engineering
Kyushu University
Hakozaki, Fukuoka 812, JAPAN

ABSTRACT

In this paper, we propose a new approach to the derivation of systolic arrays from recurrence equations, which is based on program transformation, and we also introduce a new representation for systolic arrays, which we call Relational Representation.

Relational Representation is a subset of concurrent logic language, and it has two eminent characteristics. One is that it can express recurrence equations, inner-cell operations and cell configurations, all in an integrated form. The other is that a program in it can be easily translated into a concurrent logic program, and we can simulate the behavior of a systolic array by executing the translated one. This is an approach to executable hardware description.

Transformational derivation of a systolic array is to transform one relational program corresponding to a given recurrence equation (namely a specification) to another relational program corresponding to a systolic array (namely an implementation). Based on the unfold/fold transformation of logic programs, we have formalized several transformation tactics, so as to promote well-formed organization and easy augmentation of the derivation. We have succeeded in deriving several implementations of systolic arrays, such as pipelines, orthogonal grids and trees, from their respective specifications in recurrence equations.

1. INTRODUCTION

For deriving sequential algorithms systematically, several formal techniques have been proposed. Among them, program transformation in particular is bearing fruitful results. It is, in its essence, a technique to transform one program into another equivalent one. Using it, program implementations is derived from their respective specifications. A set of transformation rules has been established, and many transformation tactics have been formalized.

On the other hand, highly parallel architectures are now pervasively researched and developed. But designing highly parallel algorithms or hardware algorithms is still a very hard job. Some systematic methodology is strongly required for designing both a large collection of fine-grained process cells connected to each other and the various operations of each cell.

Our research aims at developing a systematic technique for designing hardware algorithms, and for this aim, we applied program transformation. In this paper, we propose a new approach to the derivation of systolic array implementations from their respective specifications in recurrence equations.

In order to apply program transformation to deriving systolic arrays, we should have a formal representation of them which can express recurrence equations, inner-cell operations and cell configurations, all in an integrated form. We, therefore, introduce a restricted subset of concurrent logic language, which we call Relational Representation.

Chapter 2 introduces Relational Representation after briefly reviewing systolic arrays, and also notes its translations. Chapter 3 describes program transformation applied to relational programs, and formalizes several transformation tactics for deriving systolic arrays. Chapter 4 gives an example of the transformational derivation of systolic arrays. Chapter 5 contains concluding remarks.

2. RELATIONAL REPRESENTATION OF SYSTOLIC ARRAYS

2.1 Systolic Arrays

First, we review systolic arrays. There is, in fact, no strict definition of them. They are a class of processor arrays with the following characteristics :

A) a regular (recursively-defined) configuration ;
B) local connections of cells by channels ;
C) lock-step synchronization of the system ;
D) deterministic operations of cells.

Here is an example of a matrix-matrix multiplication. Figure 1 shows a systolic array multiplying two matrices in the case sizes are 4 by 3, where a box denotes a cell, and an arrow denotes a channel. When receiving data, each cell sends data out after a small duration, which is called a *beat*. This array has mutually orthogonal input and output streams, and each set of input streams is *skewed* with incremental delay cells in oreder to control data synchronization.

A systolic array is usually specified in such an informal manner. Understanding its behavior intuitively or formally is difficult, and simulation by hand is often required. Some systematic techniques for designing systolic arrays have been proposed (Moldovan 1983, Li and Wah 1985, Lam and Mostow 1985, Huang and Lengauer 1987), but few are widely used. Many systolic arrays were just invented, not systematically designed.

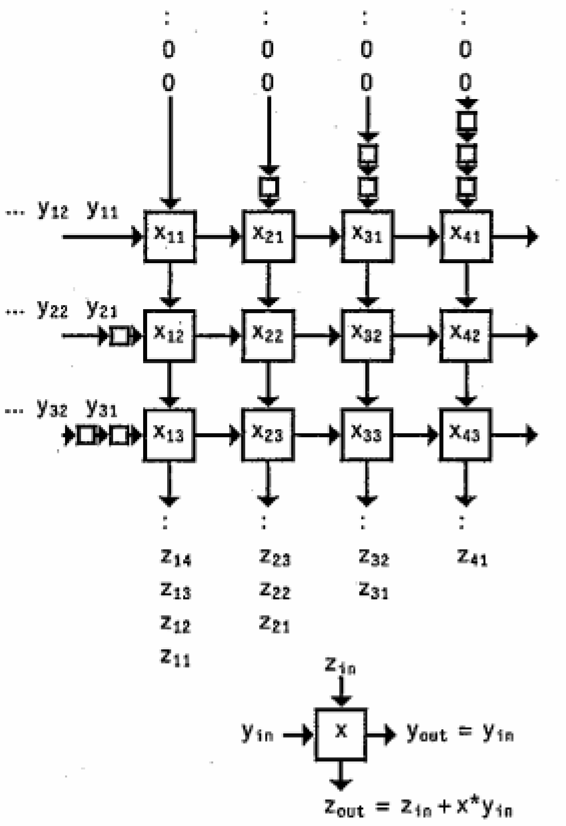


Figure 1. A Systolic Array of Matrix-Matrix Multiplication.

2.2 Relational Representation

In order to apply program transformation to the derivation of systolic arrays, we should have a formal representation of them. There are some requirements for it.

Firstly, it should be able to express both inner-cell operations and cell configurations of a systolic array in an integrated form. They together should make a uniform program to be transformed. If a program would be composed of disjoint parts in two different forms, applying transformation to it would be difficult. Secondly, it should have an inductive nature. A recurrence equation as a specification of a systolic array can be expressed in a recursive form. A perpetual process cell can be expressed in a tail-recursive form (Hoare 85), and a regular configuration of cells can also be expressed in a recursive form.

One form which meets all these requirements is a functional language with stream programming (Ida and Tanaka 1984, 1985). Another is a so-called concurrent logic language such as Concurrent Prolog and Guarded Horn Clauses. Both use, as their bases, stream interpretation of lists with lazy evaluation. As the representation, we chose the latter, since it can express multiple outputs more simply.

In concurrent logic languages, a program for a systolic array is composed of some uniform predicates with different interpretations. A predicate may be interpreted as an inner-cell operation. A tail-recursive predicate may be interpreted as a cell, possibly with its local variables as its internal states (Shapiro 1983). With its body goals interpreted as cells, a predicate may be interpreted as a cell configuration, with shared variables as channels to connect cells.

As mentioned in Chapter 3, a set of transformation rules has only been established so far for a restricted class of concurrent logic programs, not for the general class yet. We, therefore, introduce a subset of concurrent logic language with the following restrictions :

A) Specify the input/output mode of arguments ;
B) Allow no nondeterminacy (namely no guard).

These restrictions together mean that a clause to execute in a predicate is fixed as soon as all its input arguments are instantiated. They do not spoil the ability of concurrent logic languages to express systolic arrays.

We call this subset *Relational Representation*, or *RR* for short. The word *relational* is preferable to *logic* here, since relations between inputs and outputs are of concern, while logic features such as backtracking are not.

As for notation, we basically follow Prolog. We only modify a notation of clauses, in order to distinguish the mode of arguments, as follows :

$$Q :: P_1, P_2, .., P_l.$$
$$\text{where } P_i, Q = (I_1, ..., I_m)P(O_1, ..., O_n)$$

In this, P_i's ($i = 1..l$) and Q denote terms, P is a predicate symbol, I_j's ($j = 1..m$) are input arguments and O_k's ($k = 1..n$) are output arguments.

The followings are two programs of the same meaning, each in RR and in GHC respectively, for comparison :

```
<RR program>
(S,[X|XX])p([Y|YY]) ::
     (S,X)f(T,Y), (T,XX)p(YY).
<GHC program>
p(S,[X|XX], YYO) :- true | YYO = [Y|YY],
     f(S,X, T,Y), p(T,XX, YY).
```

As in concurrent logic languages, a cell is expressed by a tail-recursive predicate in RR. For example, a cell p performing an operation f with an input channel XX, an output channel YY and an internal state transition $S \rightarrow T$ (without delay considered here) is expressed as :

```
(S,[X|XX])p([Y|YY]) :: (S,X)f(T,Y), (T,XX)p(YY).
```

Conversely, a predicate expresses a cell when satisfying the following set of conditions (C1) :

A) It is tail-recursive ;
B) Every input is a list constructor or a variable ;
C) Every output is a list constructor.

A configuration of cells is also expressed by a predicate. For example, a pipeline pp of the same cells p (without delay considered here) is recursively expressed as :

```
(XX)pp(ZZ) :: (XX)p(YY), (YY)pp(ZZ).
```

2.3 Delay

An RR program has no concept of delay in its essence, while a systolic array utilizes delay in order to control the flow rates of its streams.

In a systolic array, a cell sends outputs at the next beat after receiving inputs, and the next cell receives them almost immediately. Now, imagine the situation where a cell sends outputs immediately upon receiving inputs, and the next cell receives them at the next beat. This consideration proves that a delay along a channel would be the equivalent of a delay in a cell. This means that every channel, instead of every cell, must have one or more beat(s) of delay.

A beat of channel delay can be expressed by a shift of a list. For example, a pipeline of cells p and q connected by a channel YY is expressed, with delay considered, as :

```
(XX)pq(ZZ) :: (XX)p(YY), ([⊥|YY])q(ZZ).
```

where "⊥" denotes the so-called bottom. If any input is the bottom, a cell bypasses all its inputs to outputs with no operation. We introduce an operator " + " to denote this shift of a list :

```
+XX = [⊥|XX] ;
+[XX1,XX2,...] = [+XX1, +XX2,...]
   if XXi is a stream.
```

2.4 Translations

We can translate a recurrence equation into an RR program in a straightforward manner, since both are of an inductive nature. We can also translate a systolic array into an RR program, as shown in the previous sections. Every systolic array can, in principle, be translated into an RR program, while an RR program can be translated into a systolic array, only if it satisfies the condition set C1 shown in Section 2.2. Lastly, we can translate an RR program directly into a concurrent logic program. By executing a translated program, we can simulate the behavior of a systolic array. RR can, therefore, be considered as an executable hardware description.

3. TRANSFORMATION OF PROGRAMS IN RELATIONAL REPRESENTATION

3.1 Program Transformation

For transformation of logic programs, the unfold/fold transformation has been established (Tamaki and Sato 1984). It is correct (semantics-preserving) for concurrent logic programs which imply well-formed causalities and do not have so-called *don't care* nondeterminacy (Furukawa and Ueda 1985). RR is a restricted subset of concurrent logic language, so as to specify causalities (by the mode of arguments) and not to allow nondeterminacy. We, therefore, can apply the unfold/fold transformation for logic programs to RR programs with no modification, if we are careful with their causalities.

In order to apply program transformation to practical problems, we should structure transformation sequences. This is done by formalizing transformation tactics, each of which is a specific combination of primitive rules like a *macro*. By doing it, we can transform a program with more specific and abstract tactics, instead of primitive minute rules. In this case, primitive rules serve as axioms for proving the correctness of the tactics.

The assorted tactics for the transformational derivation of systolic arays are of three types : for deriving cell configurations, for cascading channels, and for introducing delay. Here, we show some typical tactics, using a form of "initial program schema → final program schema". The outline of their proofs based on the primitive unfold/fold rules are found elsewhere (Yoshida 1988).

3.2 Tactics for Deriving Cell Configurations

The essence of a tactic for deriving a cell configuration is mapping an inner-cell operation in the initial program onto a cell configuration in the final program. This mapping is practically done by making a more inner (or lower) predicate express a cell, as shown below :

1) Tactics for Deriving Pipelines

The simplest pipeline is composed of two consecutive cells. A tactic for deriving it is as follows :

```
([X|XX])PP([Z|ZZ]) :: (X)FF(Z), (XX)PP(ZZ).
(X)FF(Z) :: (X)F1(Y), (Y)F2(Z).
           ↓
(XX)PP(ZZ) :: (XX)P1(YY), (YY)P2(ZZ).
([X|XX])P1([Y|YY]) :: (X)F1(Y), (XX)P1(YY).
([Y|YY])P2([Z|ZZ]) :: (Y)F2(Z), (YY)P2(ZZ).
```

In this, XX and ZZ are the input and output channels respectively. In the initial program, PP expresses a cell, since it satisfies the condition set C1, and FF expresses an inner-cell operation as a sequence of F1 and F2. In the final program, P1 and P2 express cells respectively, and PP expresses a cell configuration as a pipeline of P1 and P2 with an intermediate channel YY. The sequence of F1 and F2 is mapped onto the pipeline of P1 and P2. Figure 2 shows this transformation. It is the converse of the *loop fusion* (or the *combining*) tactic (Feather 1982) and the *process fusion* (Furukawa and Ueda 1985).

We can easily generalize this tactic to the derivation of pipelines of more than two cells, and moreover, recursive pipelines.

2) Tactics for Deriving Parallelisms

The simplest parallelism is composed of two adjacent cells. A tactic for deriving it is as follows :

```
([Xs|XX])PP([Zs|ZZ]) :: (Xs)FF(Zs), (XX)PP(ZZ).
([X1,X2])FF([Z1,Z2]) :: (X1)F1(Z1), (X2)F2(Z2).
                  ↓
(XX)PP(ZZ) ::
    (XX)t(XXs), (XXs)PP'(ZZs), (ZZs)t(ZZ).
([XX1,XX2])PP'([ZZ1,ZZ2]) ::
    (XX1)P1(ZZ1), (XX2)P2(ZZ2).
([X|XX])P1([Z|ZZ]) :: (X)F1(Z), (XX)P1(ZZ).
([X|XX])P2([Z|ZZ]) :: (X)F2(Z), (XX)P2(ZZ).
```

where the predicate t transposes a list of lists as :

```
([[1,2],[3,4],[5,6],..])t([[1,3,5,..],[2,4,6,..]])
```

In the initial program, the cell which PP expresses operates on a stream of paired items. In the final program, the stream is separated into two, and the cells P1 and P2 operate on each of them. PP' expresses a configuration as their parallelism. Figure 3 shows this transformation.

We can easily generalize this tactic to the case of more than two cells, and moreover, recursive parallelisms.

3) Tactics for Deriving Trees

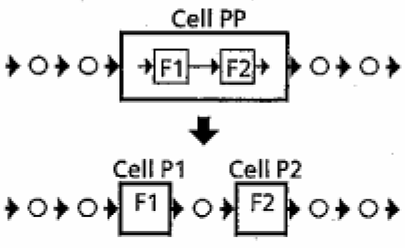


Figure 2. Derivation of a Simple Pipeline.

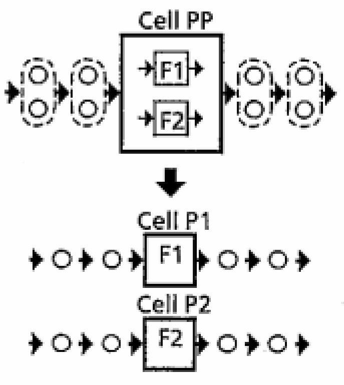


Figure 3. Derivation of a Simple Parallelism.

The base method for deriving a tree of cells is called the *recursive doubling* (Kogge and Stone 1973), which introduces a bi-linear recursion as follows :

```
([Xs|XX])PP([Z|ZZ]) :: (Xs)FF(Z), (XX)PP(ZZ).
([])FF(E).                    % E is the identity of F.
([X|Xs])FF(Z) :: (Xs)FF(Y), (X)G(X'), (X',Y)F(Z).
          ↓
(XX)PP(ZZ) :: (XX)t(XXs), (XXs)PP'(ZZ).
([XX])PP'(ZZ) :: (XX)P1(ZZ).
([XXs1@XXs2])PP'(ZZ) ::
     (XXs1)PP'(YY1), (XXs2)PP'(YY2),
     (YY1,YY2)P2(ZZ).
([X|XX])P1([Z|ZZ]) :: (X)G(Z), (XX)P1(ZZ).
([Y1|YY1],[Y2|YY2])P2([Z|ZZ]) ::
     (Y1,Y2)F(Z), (YY1,YY2)P2(ZZ).
```

where the operator "[@]" divides a list into two as :

```
[1,2,3,4,5,6] = [[1,2,3]@[4,5,6]]
```

This tactic is applicable only when F is associative. Figure 4 shows this transformation.

3.3 Tactics for Cascading Channels

One of the tactics for cascading channels transforms outflow channels to cascade ones. This is exactly the same as the *recursion removal* (Huet and Lang 1978) for sequential programs, and transforms recursion into tail-recursion prior to the configuration-deriving tactics. It is as follows :

```
([])P(E).
([X|XX])P(Z) :: (XX)P(Z'), (X,Z')F(Z).
          ↓
(X)P(Z) :: (X,E)P'(Z).
([],Z)P'(Z).
([X|XX],Z)P'(Z'') :: (X,Z)F(Z'), (XX,Z')P'(Z'').
```

This is applicable only when F is associative.

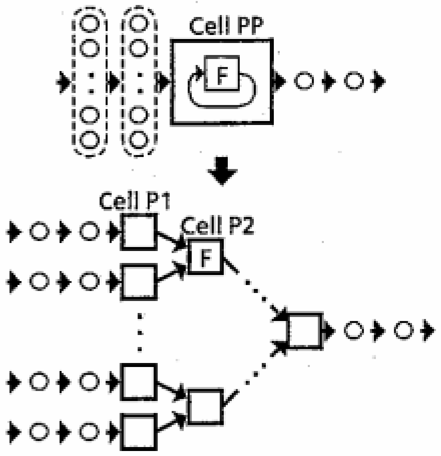


Figure 4. Derivation of a Tree.

The other transforms branch channels to cascade ones. It is as follows :

```
..., (XX)P1(), (XX)P2(), ...
([X|XX])P1() :: (X)F1(), (XX)P1().
          ↓
.., (XX)P1'(XX'), (XX')P2'(XX''), ..
([X|XX])P1'([X|XX']) :: (X)F1(), (XX)P1'(XX').
```

Figure 5 shows these two transformations.

3.4 Tactics for Introducing Delay

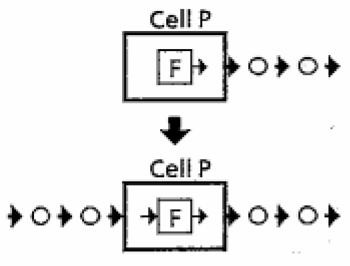
A cell with the same amount of additional delay along every input and output channel is equivalent to the original cell. Namely, the following transformation is correct :

$$(S,I_1,\ldots,I_m)P(O_1,\ldots,O_n)$$
$$\downarrow$$
$$(S,+I_1,\ldots,+I_m)P(+O_1,\ldots,+O_n)$$

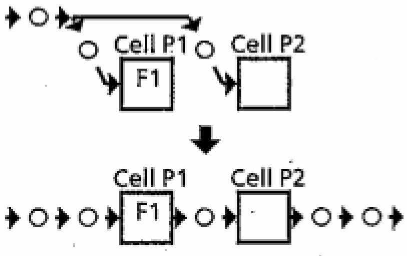
We, hereafter, make an assumption that we may ignore delay (or "+" operators) on the last output channel. Then, the following tactics for recursive cell pipelines, for example, become correct :

```
(XX)PP(ZZ) :: (XX)P(YY), (YY)PP(ZZ).
          ↓
(XX)PP(ZZ) :: (XX)P(YY), (+YY)PP(ZZ).
(XX)PP(ZZ) :: (YY)P(ZZ), (XX)PP(YY).
          ↓
(XX)PP(ZZ) :: (YY)P(ZZ), (+XX)PP(+YY).
```

In these, a beat of additional delay is put along the arguments of PP. In the former, a "+" on ZZ, which is the last output channel, is omitted. Figure 6



(a) Cascading an Outflow Channel.



(b) Cascading a Branch Channel.

Figure 5. Tactics for Cascading Channels.

shows these two transformations, where "–" denotes the inverse of "+". A transformation sequence should end with these tactics so that every channel is arranged to have one or more beat(s) of delay.

3.5 Transformation Strategy

We should have a transformation strategy to decide how to combine transformation tactics. Some channel-cascading tactics should be applied before configuration-deriving ones, the others should be applied afterwards, and delay-introducing ones should be applied last. In the case where several configuration-deriving tactics are applicable to a given program, we should transform its predicates in the order "from outer to inner". In the case where several transformation sequences are applicable to a given program, we should select one, following a certain criterion.

4. DERIVATION EXAMPLE OF SYSTOLIC ARRAYS

Transformational derivation of a systolic array is to transform one RR program corresponding to a given recurrence equation (namely a specification) to another RR program corresponding to a systolic array (namely an implementation). Here is an example of the concrete derivation of a systolic array, especially to show how a transformation sequence is composed of a combination of the tactics shown in Chapter 3.

We usually express a matrix-matrix multiplication in an abstract form as :

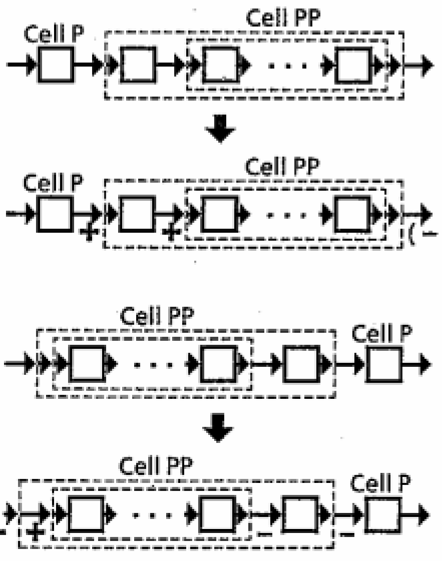


Figure 6. Tactics for Introducing Delay.

$$Z = X \cdot Y$$

When specifying the meaning of this, we should express it as :

$$Z = \langle\langle x_1 y_1, \ldots, x_m y_1\rangle^t, \ldots, \langle x_1 y_n, \ldots, x_m y_n\rangle^t\rangle$$
$$\text{where } X = \langle x_1^t, \ldots, x_m^t\rangle^t \text{ and } Y = \langle y_1, \ldots, y_n\rangle$$

or, more precisely, as :

$$z_{ij} = \Sigma x_{ik} y_{kj}$$

In this, "Σ" is also an abstract form. When specifying its meaning, we should express it in an inductive form as :

$$\Sigma^0 \text{foo}(i) \equiv 0$$
$$\Sigma^{n+1} \text{foo}(i) \equiv \text{foo}(n+1) + \Sigma^n \text{foo}(i)$$

In this consequence, an RR program corresponding to the matrix-matrix multiplication is as follows :

```
(XXX,YYY)mtxmlt(ZZZ) ::
      (YYY)t(YYY'), (XXX,YYY')mxm(ZZZ).
([],__)mxm([]).
([XXi|XXXi],YYYi)mxm([ZZo|ZZZo]) ::
      (XXi,YYYi)vxm(ZZo),
      (XXXi,YYYi)mxm(ZZZo).
(__,[])vxm([]).
(XXi,[YYi|YYYi])vxm([Zo|ZZo]) ::
      (XXi,YYi)vxv(Zo), (XXi,YYYi)vxm(ZZo).
([],[])vxv(0).
([Xi|XXi],[Yi|YYi])vxv(Zo) ::
      (XXi,YYi)vxv(Za), (Xi,Yi,Za)f(Zo).
(X,Y,Zi)f(Zo) :: (X,Y)*(Zb), (Zi,Zb)+(Zo).
```

In this, a matrix is expressed by a list of row lists, and predicates mxm, vxm and vxv calculate "matrix · matrix", "vector · matrix" and "vector · vector" respectively. Vxm satisfies the condition set C1, which means that vxm expresses a cell, while mxm expresses a cell configuration, and vxv expresses an inner-cell operation.

① First, apply a channel-cascading tactic to vxv so as to transform it to a tail-recursive predicate (we omit base terms for simplicity) :

```
(XXX,YYY)mtxmlt(ZZZ) ::
      (YYY)t(YYY'),
      (XXX,YYY',[[0,..],..])mxm(ZZZ).
([XXi|XXXi],YYYi,[ZZi|ZZZi])mxm([ZZo|ZZZo]) ::
      (XXi,YYYi,ZZi)vxm(ZZo),
      (XXXi,YYYi,ZZZi)mxm(ZZZo).
(XXi,[YYi|YYYi],[Zi|ZZi])vxm([Zo|ZZo]) ::
      (XXi,YYi,Zi)vxv(Zo),
      (XXi,YYYi,ZZi)vxm(ZZo).
([Xi|XXi],[Yi|YYi],Zi)vxv(Zo) ::
      (Xi,Yi,Zi)f(Za), (XXi,YYi,Za)vxv(Zo).
```

② Then, apply a pipeline-deriving tactic to vxm and vxv :

```
(XXX,YYY)mtxmlt(ZZZ) ::
      (XXX,YYY,[[0,..],..])mxm(ZZZ).
([XXi|XXXi],YYYi,[ZZi|ZZZi])mxm([ZZo|ZZZo]) ::
      (XXi,YYYi,ZZi)vxm(ZZo),
      (XXXi,YYYi,ZZZi)mxm(ZZZo).
([Xi|XXi],[YYi|YYYi],ZZi)vxm(ZZo) ::
      (Xi,YYi,ZZi)vxv(ZZa),
      (XXi,YYYi,ZZa)vxm(ZZo).
(Xi,[Yi|YYi],[Zi|ZZi])vxv([Zo|ZZo]) ::
      (Xi,Yi,Zi)f(Zo), (XXi,YYi,ZZi)vxv(ZZo).
```

③ Now, the innermost predicate vxv satisfies C1, which means that no more configuration-deriving tactics can be applied. Therefore, apply a channel-cascading tactic to mxm so as to transform YY to a cascade channel :

```
(XXX,YYY)mtxmlt(ZZZ) ::
      (XXX,YYY,[[0,.],..])mxm(__,ZZZ).
([XXi|XXXi],YYYi,[ZZi|ZZZi])mxm
                         (YYYo,[ZZo|ZZZo]) ::
      (XXi,YYYi,ZZi)vxm(YYYa,ZZo),
      (XXXi,YYYa,ZZZi)mxm(YYYo,ZZZo).
([Xi|XXi],[YYi|YYYi],ZZi)vxm
                         ([YYo|YYYo],ZZo) ::
      (Xi,YYi,ZZi)vxv(YYo,ZZa),
      (XXi,YYYi,ZZa)vxm(YYYo,ZZo).
(Xi,[Yi|YYi],[Zi|ZZi])vxv([Yi|YYo],[Zo|ZZo]) ::
      (Xi,Yi,Zi)f(Zo), (XXi,YYi,ZZi)vxv(YYo,Zo).
```

④ Lastly, apply a delay-introducing tactic to mxm and vxm so as to make channels YYY and ZZZ have one or more beat(s) of delay (YYYo and ZZZo are the last output channels, so "+"'s along them are omitted) :

```
(XXX,YYY)mtxmlt(ZZZ) ::
      (XXX,YYY,[[0,...],...])mxm(__,ZZZ).
([],YYYi,[])mxm(YYYi,[]).
([XXi|XXXi],YYYi,[ZZi|ZZZi])mxm
                         (YYYo,[ZZo|ZZZo]) ::
      (XXi,YYYi,ZZi)vxm(YYYa,ZZo),
      (XXXi,+YYYa,+ZZZi)mxm(YYYo,ZZZo).
([],[],ZZi)vxm([],ZZi).
([Xi|XXi],[YYi|YYYi],ZZi)vxm
                         ([YYo|YYYo],ZZo) ::
      (Xi,YYi,ZZi)vxv(YYo,ZZa),
      (XXi,+YYYi,+ZZa)vxm(YYYo,ZZo).
(Xi,[Yi|YYi],[Zi|ZZi])vxv([Yi|YYo],[Zo|ZZo]) ::
      (Xi,Yi,Zi)f(Zo), (Xi,YYi,ZZi)vxv(YYo,ZZo).
(X,⊥,Zi)f(Zi).
(X,Yi,⊥)f(⊥).
(X,Yi,Zi)f(Zo) :: (X,Yi)*(Zb), (Zi,Zb)+(Zo).
```

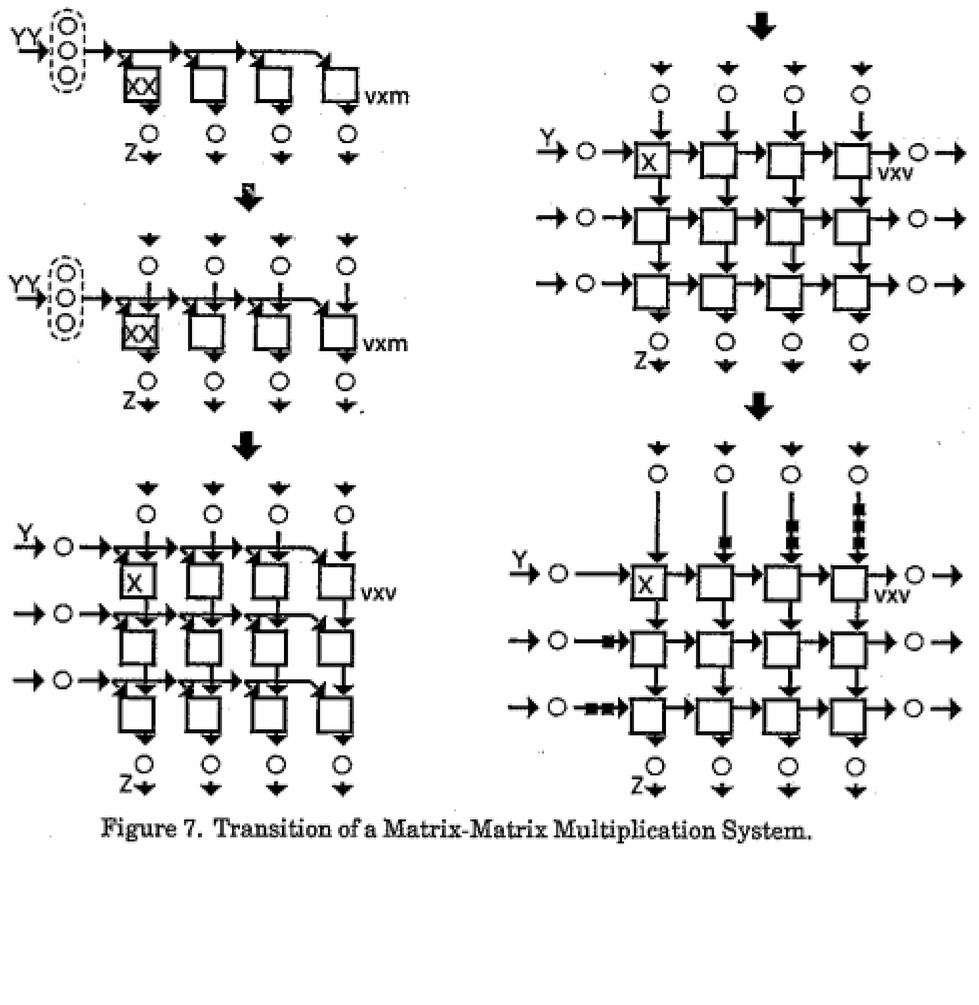


Figure 7. Transition of a Matrix-Matrix Multiplication System.

In this, vxv satisfies the condition set C1, which means that vxv, vxm and mxm express a cell, a column of the cells and the whole array respectively. We get an RR program which can correspond to the systolic array of a matrix-matrix multiplication shown in Section 2.1. Figure 7 shows the transition of the multipication system along the transformational derivation shown above.

5. CONCLUSIONS

In this paper, we proposed a new approach to the derivation of systolic arrays from recurrence equations, which is based on program transformation. We also introduced a new representation for systolic arrays, which we call Relational Representation. We have succeeded in deriving several implementations of systolic arrays from their respective specifications in recurrence equations.

There is one open problem. If an outcome RR program can not correspond to any implementation, either its specification or its transformation sequence is not good, but we can not now say which is the case, since we have not yet completed this transformational derivation technique.

We believe that this work has proved the transformational derivation of systolic arrays is indeed promising.

ACKNOWLEDGMENTS

The author would like to thank Professor Kazuo Ushijima and Professor Shinji Tomita of Kyushu University for their valuable support and encouragement, and also thank Doctor Jiro Tanaka of Fujitsu Ltd. for his advice concerning stream programming.

REFERENCES

(Moldovan 1983)
Moldovan,D.I., "On the Design of Algorithms for VLSI Systolic Arrays", Proc. IEEE 71:1 (1983) 113-120.

(Li and Wah 1985)
Li,G.-J. and Wah,B.W., "The Design of Optimal Systolic Arrays", IEEE Trans. C-34:1 (1985) 66-77.

(Lam and Mostow 1985)
Lam,M.S. and Mostow,J., "A Transformational Model of VLSI Systolic Design", IEEE Comp. 18:2 (1985) 42-52.

(Huang and Lengauer 1987)
Huang,C.-H. and Lengauer,C., "The Derivation of Systolic Implementations of Programs", Acta Inf. 24 (1987) 595-632.

(Hoare 1985)
Hoare,C.A.R., *Communicating Sequential Processes*, Prentice-Hall (1985).

(Ida and Tanaka 1983)
Ida,T. and Tanaka,J., "Functional Programming with Streams", Proc. IFIP '83, Paris (1983) 265-270.

(Ida and Tanaka 1984)
Ida,T. and Tanaka,J., "Functional Programming with Streams – Part II –", New Generation Computing 2:3 (Ohm-sha, JAPAN) (1984) 261-275.

(Shapiro 1983)
Shapiro,E.Y., "A Subset of Concurrent Prolog and Its Interpreter", Tech. Report TR003, ICOT (1983).

(Tamaki and Sato 1984)
Tamaki,H. and Sato,T., "Unfold/Fold Transformation of Logic Programs", Proc. 2nd Logic Programming Conf., Uppsala (1984) 127-138.

(Furukuwa and Ueda 1985)
Furukawa,K. and Ueda,K., "GHC Process Fusion by Program Transformation", Proc. 2nd Japan Soc. Soft. Sci. and Tech. Conf., Tokyo (1985) 89-92.

(Yoshida 1988)
Yoshida,N., "Transformational Derivation of Highly Parallel Programs", Proc. 3rd Int. Conf. on Supercomputing Vol.3, Boston (1988) 445-454.

(Feather 1982)
Feather,M.S., "A System for Assisting Program Transformation", ACM Trans. Prog. Lang. Syst. 4:1 (1982) 1-20.

(Kogge and Stone 1973)
Kogge,P.M. and Stone,H.S., "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations", IEEE Trans. C-22:8 (1973) 786-793.

(Huet and Lang 1978)
Huet,G. and Lang,B., "Proving and Applying Program Transformation Expressed with Second-Order Patterns", Acta Inf. 11 (1978) 31-55.