A Transformational Approach to the Derivation of Hardware Algorithms from Recurrence Equations

Norihiko Yoshida

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

Abstract

Hardware algorithms have a significant impact on the supercomputing of matrix computation and signal processing. In this paper, we propose a new approach to the derivation of hardware algorithms from recurrence equations, which is based on program transformation, and we also introduce a new representation for hardware algorithms, which we call Relational Representation. In our approach, we transform one relational program corresponding to a given recurrence equation (namely a specification) to another relational program corresponding to a hardware algorithm (namely an implementation). Based on the unfold/fold transformation method of logic programs, we have formalized several transformation tactics. We have succeeded in deriving several implementations of hardware algorithms, such as pipelines, orthogonal grids and trees, from their respective specifications in recurrence equations.

1. Introduction

With the advances in VLSI technology and the increasing demands for higher performance, highly parallel architectures are now pervasively researched and developed. In particular, many computations which were previously implemented in software are going to be implemented in hardware. Such computations in special-purpose hardware are often called hardware algorithms. They have a significant impact on the supercomputing of matrix computation and signal processing. But designing 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.

As for sequential algorithms, several formal techniques have been proposed for designing them systematically. Among them, program transformation in particular is bearing fruitful results. It is, in its essence, a technique to transform one program to another equivalent one. A set of correct transformation rules has been established, and many transformation tactics have been formalized.

For our aim at designing hardware algorithms systematically, we applied a formal technique such as the above. In this paper, we propose a new approach to the derivation of hardware algorithms from recurrence equations, which is based on program transformation.

In order to apply program transformation, we should have a formal representation of hardware algorithms which can express both inner-cell operations and cell configurations in an integrated form. We, therefore, introduce a restricted subset of concurrent logic language, which we call Relational Representation. It also serves as an executable hardware description.

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

2. Relational Representation of Hardware Algorithms

2.1 Hardware Algorithms

First, we specify what we call a hardware algorithm here. It is a class of highly parallel 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 whole system ;
D) deterministic operations of cells.

For example, we consider a finite impulse response (FIR) filter for signal processing defined by the following recurrence equation :

$$y_i = w_0 x_i + w_1 x_{i+1} + \ldots + w_{k-1} x_{i+k-1}$$

where w: weight,
x: source sequence, y: drain sequence.

Figure 1 shows its corresponding hardware algorithm in the case where k is 4, in which 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 pipeline has streams of x's and y's flowing in the same direction along the channels. Y's stream has one more beat of delay per cell, so as to make it flow at half of x's rate.

A hardware algorithm is usually described 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 hardware algorithms, especially systolic arrays, have been proposed [1, 2, 3, 4], but few are widely used.

2.2 Relational Representation

In order to apply program transformation to the derivation of hardware algorithms, we should have their formal representation. It should be able to express both inner-cell operations and cell configurations in an integrated form, and be of an inductive nature [5]. So-called concurrent logic languages such as Guarded Horn Clauses [6] and Concurrent Prolog [7] meet these requirements.

In these languages, a program for a hardware algorithm could be composed of some uniform predicates with different interpretations. A predicate with procedure interpretation would express an inner-cell operation, while a predicate with process interpretation would express a cell configuration with shared variables for channels to connect cells [8].

As mentioned in Chapter 3, a set of correct 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 languages with the following restrictions, and hence call it *Relational Representation* (or *RR* for short) :

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 hardware algorithms.

As for notation, we basically follow a logic language Prolog [9]. A capitalized symbol denotes a variable, and "[A|L]" denotes a list, where A and L are its *car* and *cdr* parts respectively. We only modify a notation of clauses, in order to specify the mode of arguments, as follows :

$Q :: P_1, P_2, .., P_l.$
where P_i, $Q \equiv (I_1, .., I_m)P(O_1, .., O_n)$

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

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 in RR expresses a cell when satisfying the following set of conditions (C1) :

A) It is tail-recursive ;
B) Its every input is a list constructor or a variable ;
C) Its 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

A relational program has no concept of delay in its essence, while a hardware algorithm utilizes delay in order to control the flow rates of its streams.

In a hardware algorithm, 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. As shown in Figure 2, this consideration proves that a delay along a channel would be the equivalent of a delay in a cell. This means that every chan-

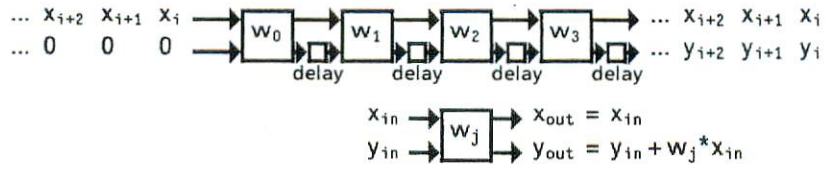


Figure 1. A Hardware Algorithm of an FIR Filter.

nel, 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:

$$+XX \equiv [\bot|XX]$$

2.4 Translations

We can translate a recurrence equation into a relational program in a straightforward manner, since both are of an inductive nature. For example, the recurrence equation defining an FIR filter shown in Section 2.1 has another form as follows:

$$y_i^0 = 0$$
$$y_i^{j+1} = y_i^j + w_j x_{i+j}$$

A relational program corresponding to it is as follows:

```
(WW,XXi)fir(YYo) :: (WW,XXi)pp(YYo).                    (P1)
(WW,[Xi|XXi])pp([Yo|YYo]) ::
    (WW,[Xi|XXi])p(Yo), (WW,XXi)pp(YYo).
([],XXi)p(0).
([W|WW],[Xi|XXi])p(Yo) ::
```

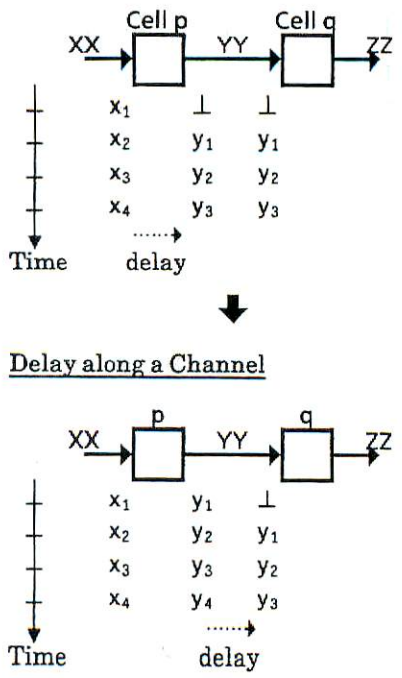


Figure 2. Delay along a Channel instead of in a Cell.

```
    (WW,XXi)p(Ya), (W,Xi,Ya)f(Yo).
(W,⊥,Yi)f(Yi).
(W,Xi,⊥)f(⊥).
(W,Xi,Yi)f(Yo) :: (W,Xi)*(Yb), (Yi,Yb) + (Yo).
```

In this, WW, XX and YY express the sequences of w's, x's and y's respectively. P expresses the recurrence equation itself, while pp expresses x's and y's progression according to the suffix i.

We can also translate a hardware algorithm into a relational program in a straightforward manner, as shown in the previous sections. A relational program corresponding to the hardware algorithm of an FIR filter is as follows:

```
(WW,XXi)fir(YYo) ::                                     (P2)
    (WW,XXi,[0,0,..])pp(_,YYo).
([],XXi,YYi)pp(XXi,YYi).
([W|WW],XXi,YYi)pp(XXo,YYo) ::
    (W,XXi,YYi)p(XXa,YYa),
    (WW, + XXa, + + YYa)pp(XXo,YYo).
(W,[Xi|XXi],[Yi|YYi])p([Xi|XXo],[Yo|YYo]) ::
    (W,Xi,Yi)f(Yo), (W,XXi,YYi)p(XXo,YYo).
(W,⊥,Yi)f(Yi).
(W,Xi,⊥)f(⊥).
(W,Xi,Yi)f(Yo) :: (W,Xi)*(Yb), (Yi,Yb) + (Yo).
```

In this, p expresses each cell, while pp expresses the cell configuration.

Lastly, we can translate a relational program easily into a concurrent logic program, since RR is a restricted subset of concurrent logic language. By executing a translated program, we could simulate the behavior of a hardware algorithm.

3. Transformation of Relational Programs

3.1 Program Transformation

Program transformation is a technique to transform one program to another equivalent one [10, 11]. For logic programs, the unfold/fold transformation method has been established [12]. It transforms a program by combining very primitive rules including *unfold* and *fold*:

```
Unfold: A :: B, C, D. C :: E, F.   → A :: B, E, F, D.
Fold:   A :: B, E, F, D. C :: E, F. → A :: B, C, D.
```

The unfold/fold transformation is correct (semantics-preserving) for concurrent logic programs which imply well-formed causalities and do not have so-called *don't care* nondeterminacy [13]. 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 relational programs

with no modification, if we are careful with their causalities.

We transform a program with a sequential (stepwise) combination of rules. In order to apply program transformation to practical problems, we should structure transformation sequences. This is done by formalizing transformation tactics [14]. Each tactic is a specific combination of primitive rules like a *macro*. By formalizing tactics, 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 hardware algorithms 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 correctness proofs based on the primitive unfold/fold rules are found elsewhere [15].

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. Practically, this mapping is 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).
        ↓                                   (T1)
(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 input and output channels respectively. In the initial program, PP expresses a cell, since it satisfies the condition set C1 shown in Section 2.2. FF expresses an inner-cell operation composed of F1 and F2. In the final program, P1 and P2 express cells, since they satisfy C1. PP expresses a cell configuration as a pipeline of P1 and P2 with an intermediate channel YY. The sequence of F1 and F2 in the initial program is mapped onto the pipeline of P1 and P2 in the final program. Figure 3 shows this transformation. T1 is the converse of the *loop fusion* (or the *combining*) tactic [14] or the *process fusion* [13].

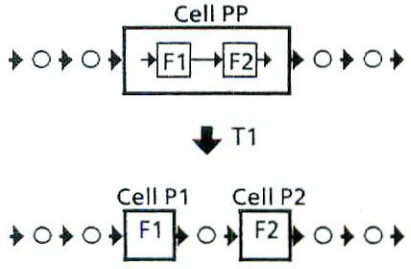


Figure 3. Derivation of a Simple Pipeline.

We can easily generalize T1 to the derivation of pipelines of more than two cells, and moreover recursive pipelines as follows :

```
([X|XX])PP([Z|ZZ]) :: (X)FF(Z), (XX)PP(ZZ).
(X)FF(Z) :: (X)F(Y), (Y)FF(Z).
        ↓                                   (T2)
(XX)PP(ZZ) :: (XX)P(YY), (YY)PP(ZZ).
([X|XX])P([Y|YY]) :: (X)F(Y), (XX)P(YY).
```

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).
        ↓                                   (T3)
(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 is to transpose 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 the configuration as their parallelism. Figure 4 shows this transformation.

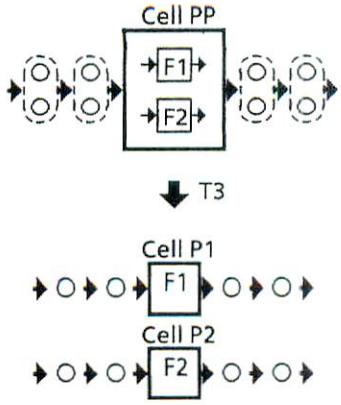


Figure 4. Derivation of a Simple Parallelism.

We can easily generalize T3 to the derivation of parallelisms of more than two cells, and moreover recursive parallelisms as follows :

```
([Xs|XX])PP([Zs|ZZ]) :: (Xs)FF(Zs), (XX)PP(ZZ).
([X|Xs])FF([Z|Zs]) :: (X)F(Z), (Xs)FF(Zs).
                ↓                                   (T4)
(XX)PP(ZZ) ::
    (XX)t(XXs), (XXs)PP'(ZZs), (ZZs)t(ZZ).
([XX|XXs])PP'([ZZ|ZZs]) ::
    (XX)P(ZZ), (XXs)PP'(ZZs).
([X|XX])P([Z|ZZ]) :: (X)F(Z), (XX)P(ZZ).
```

3) Tactics for Deriving Trees

The base method for deriving a tree of cells is called the *recursive doubling* [16, 10], which introduce a bilinear 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).
                ↓                                   (T5)
(XX)PP(ZZ) :: (XX)t(XXs), (XXs)PP'(ZZ).
([XX])PP'(ZZ) :: (XX)P'(ZZ).
([XXs1@XXs2])PP'(ZZ) ::
    (XXs1)PP'(YY1), (XXs2)PP'(YY2),
    (YY1,YY2)P(ZZ).
([X|XX])P'([Z|ZZ]) :: (X)G(Z), (XX)P'(ZZ).
([Y1|YY1],[Y2|YY2])P([Z|ZZ]) ::
    (Y1,Y2)F(Z), (YY1,YY2)P(ZZ).
```

where the operator "[@]" is to divide a list into two as :

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

T5 is applicable only if F is associative. Figure 5 shows this transformation.

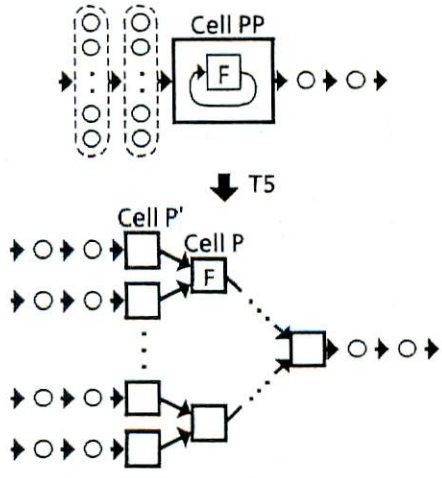


Figure 5. Derivation of a Tree.

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* [17] for sequential programs. It is as follows :

```
([])P(E).
([X|XX])P(Z) :: (XX)P(Z'), (X,Z')F(Z).
                ↓                                   (T6)
(X)P(Z) :: (X,E)P'(Z).
([],Z)P'(Z).
([X|XX],Z)P'(Z'') :: (X,Z)F(Z'), (XX,Z')P'(Z'').
```

T6 is applicable only if F is associative.

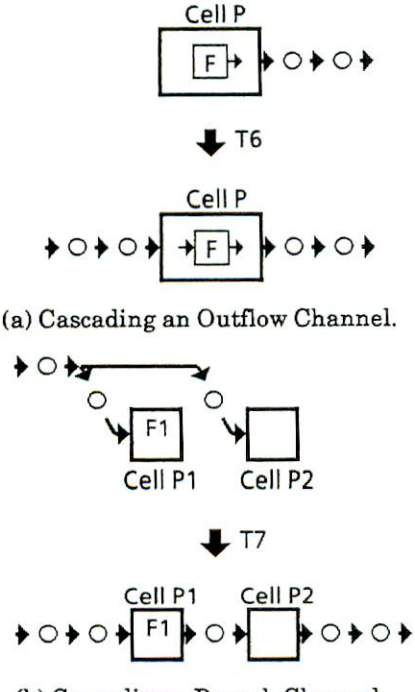
The other transforms branch channels to cascade ones. It is as follows :

```
..., (XX)P1(), (XX)P2(), ...
([X|XX])P1() :: (X)F1(), (XX)P1().
                ↓                                   (T7)
.., (XX)P1'(XX'), (XX')P2'(XX''), ..
([X|XX])P1'([X|XX']) :: (X)F1(), (XX)P1'(XX').
```

Figure 6 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 :



(a) Cascading an Outflow Channel.

(b) Cascading a Branch Channel.

Figure 6. Tactics for Cascading Channels.

$(S,I_1,...,I_m)P(O_1,...,O_n)$

↓

$(S,+I_1,...,+I_m)P(+O_1,...,+O_n)$

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

```
(XX)PP(ZZ) :: (XX)P(YY), (YY)PP(ZZ).
        ↓                                   (T8)
(XX)PP(ZZ) :: (XX)P(YY), (+YY)PP(ZZ).

(XX)PP(ZZ) :: (YY)P(ZZ), (XX)PP(YY).
        ↓                                   (T9)
(XX)PP(ZZ) :: (YY)P(ZZ), (+XX)PP(+YY).
```

In these, a beat of additional delay is put along the arguments of PP. A "+" on ZZ, which is the last output channel in T8, is omitted. Figure 7 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 other channel-cascading ones 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.

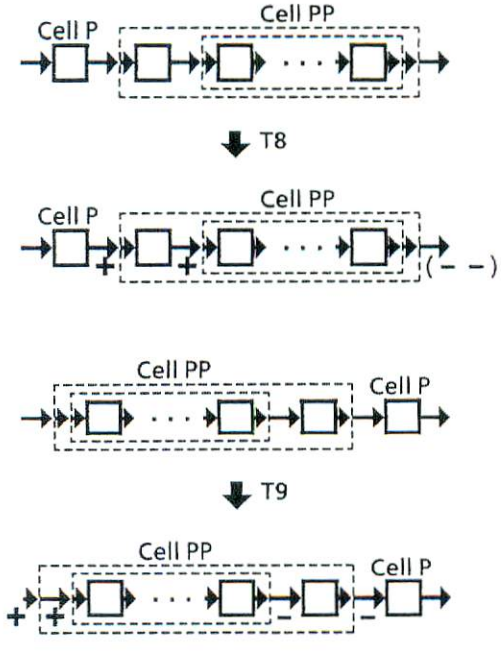


Figure 7. Tactics for Introducing Delay.

4. Derivation Example of Hardware Algorithms

Transformational derivation of a hardware algorithm is to transform one relational program corresponding to a given recurrence equation (namely a specification) to another relational program corresponding to a hardware algorithms (namely an implementation). We describe, using a simple example, the concrete derivation of hardware algorithms, especially to show how transformation sequences are composed of the tactics shown above.

The relational program P1 shown in Section 2.4 is the one translated from the recurrence equation defining an FIR filter. In this, pp satisfies the condition set C1, which means that pp expresses a cell, while p expresses an inner-cell operation. This system is composed of one cell.

① First, apply a channel-cascading tactic to p so as to transform it to a tail-recursive predicate :

```
(WW,XXi)fir(YYo) :: (WW,XXi,[0,0,..])pp1(YYo).
(WW,[Xi|XXi],[Yi|YYi])pp1([Yo|YYo]) ::
    (WW,[Xi|XXi],Yi)p1(Yo),
    (WW,XXi,YYi)pp1(YYo).
([],XXi,Yi)p1(Yi).
([W|WW],[Xi|XXi],Yi)p1(Yo) ::
    (W,Xi,Yi)f(Ya), (WW,XXi,Ya)p1(Yo).
```

② Apply a pipeline-deriving tactic to pp1 and p1 :

```
(WW,XXi)fir(YYo) :: (WW,XXi,[0,0,..])pp2(YYo).
([],XXi,YYi)pp2(YYi).
([W|WW],[Xi|XXi],YYi)pp2(YYo) ::
    (W,[Xi|XXi],YYi)p2(YYa),
    (WW,XXi,YYa)pp2(YYo).
(W,[Xi|XXi],[Yi|YYi])p2([Yo|YYo]) ::
    (W,Xi,Yi)f(Yo), (W,XXi,YYi)p2(YYo).
```

③ Now, the innermost predicate p satisfies C1, which means that no more configuration-deriving tactics can be applied. Therefore, apply a channel-cascading tactic to pp2 so as to transform XX to a cascade channel :

```
(WW,XXi)fir(YYo) :: (WW,XXi,[0,0,..])pp3(_,YYo).
([],XXi,YYi)pp3(XXi,YYi).
([W|WW],[Xi|XXi],YYi)pp3([Xi|XXo],YYo) ::
    (W,[Xi|XXi],YYi)p3([Xi|XXa],YYa),
    (WW,XXa,YYa)pp3(XXo,YYo).
(W,[Xi|XXi],[Yi|YYi])p3([Xi|XXo],[Yo|YYo]) ::
    (W,Xi,Yi)f(Yo), (W,XXi,YYi)p3(XXo,YYo).
```

④ Lastly, apply a delay-introducing tactic to pp3 so as to make channels XX and YY have one or more beat(s) of delay (XXo and YYo are the last output channels, so "+"s along them are omitted) :

```
(WW,XXi)fir(YYo) :: (WW,XXi,[0,0,..])pp4(_,YYo).
([],XXi,YYi)pp4(XXi,YYi).
([W|WW],XXi,YYi)pp4(XXo,YYo) ::
    (W,XXi,YYi)p4(XXa,YYa),
    (WW,XXa, + YYa)pp4(XXo,YYo).
(W,[Xi|XXi],[Yi|YYi])p4([Xi|XXo],[Yo|YYo]) ::
    (W,Xi,Yi)f(Yo), (W,XXi,YYi)p4(XXo,YYo).
```

Again applying a delay-introducing tactic to pp4, we get exactly the same relational program as P2 shown in Section 2.4. It can be translated into a hardware algorithm of an FIR filter.

Figure 8 shows the transition of the FIR filter system along the transformational derivation shown above.

We show another derivation example only as a schematic diagram. Figure 9 shows the transition of a matrix-vector product system along the transformational derivation. This is a case where some transformation procedures are applicable, in which the former tree is faster than the latter pipeline, while the latter is more compact.

5. Conclusions

In this paper, we proposed a new approach to the derivation of hardware algorithms from recurrence equations, which is based on program transformation.

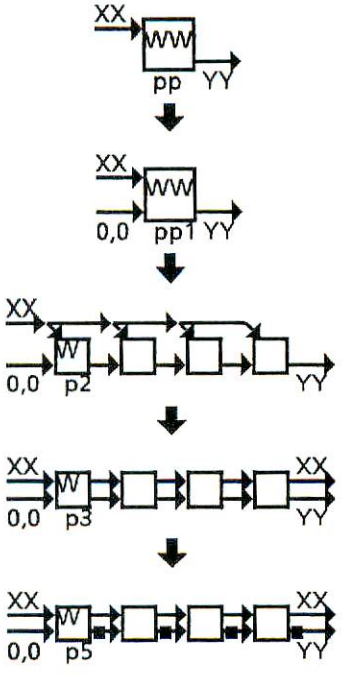


Figure 8. Derivation of an FIR Filter System.

We also introduced a new representation for hardware algorithms, which we call Relational Representation.

Relational Representation is a subset of concurrent logic language, since a set of transformation rules has only been established so far for a restricted class of concurrent logic programs. It has two eminent characteristics. One is that it can express both inner-cell operations and cell configurations in an integrated form. The other is that a program in it can be easily translated into a concurrent logic program, and we could simulate the behavior of a hardware algorithm by executing the translated one on some concurrent logic programming system.

Transformational derivation of a hardware algorithm is to transform one relational program corresponding to a given recurrence equation (namely a specification) to another relational program corresponding to a hardware algorithm (namely an implementation). In our approach, formalizing tactics promotes well-formed organization and easy augmentation of the derivation technique.

We have succeeded in deriving several implementations of hardware algorithms from their respective specifications in recurrence equations. Besides the ones shown as examples, we have derived orthogonal grids, but not hexagonal grids yet. There is one open problem. If an outcome relational 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 the transformational derivation technique.

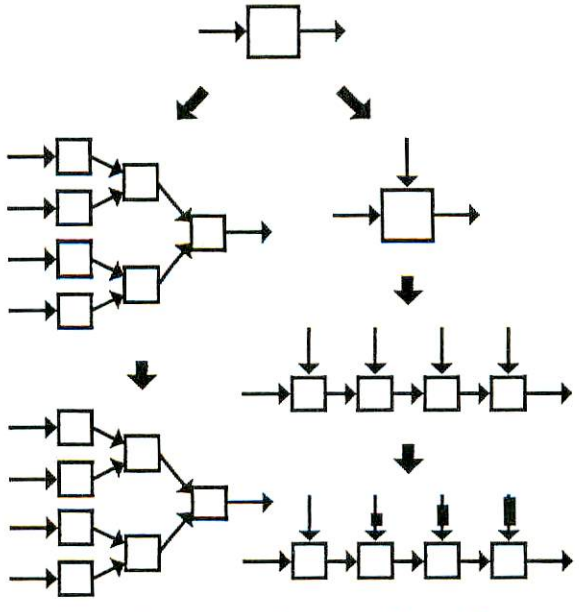


Figure 9. Derivation of a Matrix-Vector Product System.

It is worth to note that this work has a close relationship to stream programming in functional languages [18, 19, 20, 21], since both use, as their bases, stream interpretation of lists with lazy evaluation.

Hardware algorithms have a significant impact on the supercomputing of matrix computation and signal processing. We believe that this work has proved that the transformational derivation of hardware algorithms is indeed promising.

Acknowledgment

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

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

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

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

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

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

[6] Ueda,K., *Guarded Horn Clauses*, MIT Press (1988).

[7] Shapiro,E.Y., "Concurrent Prolog : A Progress Report", IEEE Comp. 19: 8 (1986) 44-58.

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

[9] Clocksin,W.F. and Mellish,C.S., *Programming in PROLOG*, Springer-Verlag (1981).

[10] Darlington,J., "Program Transformation", *Functional Programming and Its Application* (Darlington,J. et.al. eds.), Cambridge Univ. Press (1982) 193-215.

[11] Bird, R.S., "Tabulation Techniques for Recursive Programs", ACM Comp. Surv. 12:4 (1980) 403-417.

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

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

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

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

[16] 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.

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

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

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

[20] Wadler,P., "Applicative Style of Programming, Program transformation and List Operators", Proc. 1981 Conf. on Functional Programming Languages and Computer Architecture, Portsmouth (New Hampshire) (1981) 25-32.

[21] Kieburtz,R.B. and Shultis,J., "Transformation of FP Program Schemes", ibid. 41-48.