An Implemented Method for Incremental Systolic Design

Chua-Huang Huang & Christian Lengauer
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188, U.S.A.

Abstract

We present a mathematically rigorous and, at the same time, convenient method for systolic design and derive alternative systolic designs for one expository matrix computation problem: matrix multiplication. Each design is synthesized from a simple program and a proposed layout of processors. The synthesis derives (1) a systolic parallel execution, (2) channel connections for the proposed processor layout, and (3) an arrangement of data streams such that the systolic execution can begin. Our choices of alternative designs are governed by formal theorems. The synthesis method is implementable and is particularly effective if implemented with graphics capability. Our implementation on the Symbolics 3600 displays the resulting designs and simulated executions graphically on the screen. The method has also been successfully applied to other matrix computation problems.

1. Introduction

The development of programs need not immediately address implementation concerns. Instead, one can proceed in stages. One can first derive a program that conforms with the problem specification, and then derive an execution (or "trace") and provide an architecture. Programs do not contain concepts of execution but, from programs, executions can be derived for a variety of computer architectures. Such an approach bridges the gap between two separate concerns: correctness and efficiency. To keep the correctness proof simple, the program which is shown to solve the given problem should be simple. After correctness has been established, the program's execution can be complicated into a more desirable execution. The complicated execution must have exactly the input-output behavior established for the program.

This division of concerns can be of great help in program development. In the best of all worlds, where there is a proven, mechanical way to obtain efficient, complicated executions from simple programs, the programmer never has to go beyond the program in his understanding of the problem solution. In fact, he or she might even get help in constructing a suitable computer architecture without delving into the intricacies of program execution. We will provide a glimpse into such a world. We cannot deal with all programming problems, but the horizons of our world are expanding. Presently, it contains sorting problems [15] and matrix computation problems [9]. Our notion of efficiency is parallelism. Programs do not address the question of sequencing but may result in complicated, i.e., parallel executions.

For exposition, we will confine ourselves here to matrix computations - in fact, to just one matrix computation problem: matrix multiplication. We will present matrix multiplication programs and, automatically, derive parallel executions for them. We will then proceed to propose architectures that can perform these parallel executions. Our architectures will be systolic [11], i.e., they will be networks of processors that are connected in simple patterns and perform simple operations under global synchronization. We will only have to propose the layout of the processors. If it is suitable for our execution, the links of communication channels between processors and the layout and direction of the data travelling through the network can be synthesized automatically. After scrutiny of the resulting design, we might want to improve it by altering either the processor layout or the program. In our example, matrix multiplication, we will make one adjustment to the processor layout and then one adjustment to the program. Our search for alternative designs is guided by a number of theorems about our design method.

2. The Design Method

2.1. Programs

Our programs are expressed in a refinement language with the following features:

- The definition of a *refinement* consists of a refinement name with an optional list of formal parameters, separated by a colon from a refinement body. The following are the only three choices of a refinement body.
- The *null statement*, skip, does nothing.
- The *basic statement* is a statement that is not refined any further. For matrix multiplication, we will use a basic statement called the *inner product step* [11]. An inner product step accesses the elements $a_{i,k}$, $b_{k,j}$ and $c_{i,j}$ of three distinct matrices A, B, and C, respectively, and performs the operation

  $$c_{i,j} := c_{i,j} + a_{i,k} * b_{k,j}$$

  If variables A, B, and C are fixed, we can express the inner product step solely in terms of the matrix subscripts i, j, and k. We will use the notation $(i{:}j{:}k)$.
- The *composition* $S0;S1$ of refinements $S0$ and $S1$ applies $S1$ to the results of $S0$. Each of $S0$ and $S1$ can be a refinement call (i.e., a refinement name, maybe, with an actual parameter list), a basic statement, or the null statement. Sequences of compositions $S0;S1;...;Sn$ are also permitted. Refinement calls may be recursive.

2.2. Traces

Following the conventional implementation of composition as sequential execution, a sequential execution is obtained from a refinement by replacing every semicolon with a right-pointing arrow. That is, program $S0;S1$ has trace $S0 \rightarrow S1$. This implementation of composition is always safe, but may be overly restrictive. We can transform it into different executions with the same effect. Such transformations can relax sequencing and incorporate parallelism into executions. In certain cases we will execute program $S0;S1$ by trace $\langle S0\ S1\rangle$ (angle brackets denote parallel execution). We call $\langle S0\ S1\rangle$ a *parallel command*, and a trace with parallel commands a *parallel trace*.

We denote the length of a trace S by $|S|$ and define it as follows:

$|\text{stat}| =_{\text{def}} 1$ for every basic statement stat

$|S0 \rightarrow S1| =_{\text{def}} |S0| + |S1|$

$|\langle S0\ S1\rangle| =_{\text{def}} \max(|S0|,|S1|)$

The length of a trace serves as an estimate of the trace's execution time. Our estimate is rather crude. For more accurate estimates, the previous definitions can be adjusted accordingly.

2.3. Trace Transformations

Our intent is to shorten the length of a trace by a sequence of transformations. Each transformation must preserve the trace's effect. Trace transformations are justified by semantic relations that program components may or may not satisfy:

(1) A program component S that is *idempotent* can be executed once or any number of times consecutively with identical effect. Thus, $S \rightarrow S$ in a trace may be transformed to S, and vice versa. The idempotence of S is declared as: idem S.

(2) A program component S that is *neutral* has no effect other than that it may take time to execute. Thus, S may be omitted from or added to a trace. Neutrality implies idempotence. The neutrality of S is declared as: ntr S.

(3) Two program components $S0$ and $S1$ that are *commutative* can be executed in any order with identical effect. Thus, $S0 \rightarrow S1$ in a trace may be transformed to $S1 \rightarrow S0$. The commutativity of $S0$ and $S1$ is declared as: $S0$ com $S1$.

(4) Two program components $S0$ and $S1$ that are *independent* can be executed in parallel and in se-

quence with identical effect. Thus $S0 \rightarrow S1$ in a trace may be transformed to $<S0\ S1>$. Independence implies commutativity. The independence of $S0$ and $S1$ is declared as: $S0$ ind $S1$.

Semantic relations are made explicit by declarations that accompany the refinement program. The format of a semantic declaration is:

enabling predicate $\Rightarrow$ *semantic relation*

The enabling predicate is a condition on the parameters of the program components that are semantically related. Just like the correctness of refinements, the correctness of semantic declarations can be proved formally [14].

We will exploit semantic declarations for different programs in one and the same way. After having obtained a sequential trace, say *l*, from the program, we transform this trace into concurrency by exploiting the declared semantic relations according to the following pattern:

transform(*l*) = *remove-all-ntr*(*ravel-trans*(*l*))

Informally, *ravel-trans*(*l*) ravels all basic statements in *l*, one by one, from right to left to a parallel trace. First, the right-most basic statement is ravelled into the empty trace to form a single-statement parallel command. Then each of the remaining basic statements in *l* is ravelled into the parallel trace produced so far. Duplicate idempotent statements are discarded if possible. The ravelling process merges the basic statement with the right-most possible parallel command as permitted by the declared semantic relations; otherwise, it commutes the basic statement to the right-most possible position and forms another single-statement parallel command. Then *remove-all-ntr* removes all neutral basic statements. This transformation strategy is the heart of our method. It has been defined formally in the Boyer-Moore computational logic [1] and mechanically proved correct [9].

2.4. Architectures

A parallel trace specifies a partial order of basic statements without reference to a particular architecture. We will develop systolic arrays that can execute the parallel trace. We specify a systolic array with the help of four functions.

The first two functions are called *step* and *place*. The domain of both functions is the set of basic statements that occur in the parallel trace. *Step* determines when basic statements are to be executed, and *place* determines where basic statements are to be executed.[1]

Step maps basic statements to the integers. The intention is to count the parallel commands of the parallel trace in their order of execution. *Step* is derived from the parallel trace. The derivation of *step* must adhere to two conditions:

(S1) basic statements of the same parallel command must be mapped to the same integer,

(S2) basic statements of adjacent parallel commands must be mapped to consecutive integers.

We are free to choose an appropriate integer, *fs*, for the basic statements of the first parallel command. If *step* satisfies conditions (S1) and (S2), any two basic statements in the same parallel command must have identical step values. *Step* can be derived by solving a system of equations whose formulation is guided by conditions (S1) and (S2) (see the next section).

Place maps basic statements to an integer space of some dimension *d*. We assume that every point of that space is occupied by a processor. The intention is to assign basic statements to the processors. Processors that are not assigned a statement at some step simply forward the data on their input channels to the corresponding output channels during that step. Processors that are at no step assigned a statement need not be implemented. *Place* is not derived from the parallel trace but proposed separately. *Place* has to satisfy the following condition:

[1]In general, we must distinguish multiple occurrences of identical basic statements - by some sort of counter, say. However, we omit this trivial complication here. Matrix multiplication leads to traces whose basic statements are all distinct.

(P1) basic statements of the same parallel command must be assigned distinct points.
We have a simple condition that establishes whether our proposals for *place* satisfy (P1) (see the next section).

In programs, data are represented by variables. In systolic computations, data, i.e., variables travel between processors. A variable may be accessed by one processor at one step and by another processor at a later step. We have to specify a layout and flow of variables that provides each processor with the expected inputs at the steps at which it is supposed to execute its basic statement. At present, our method is confined to systolic arrays in which processors are only connected by unidirectional channels to processors that occupy neighboring points.[2] For designs with these characteristics, we can synthesize the input pattern and flow of data from *step* and *place*. To this end, we introduce two more functions: *pattern* and *flow*. The domain of both functions is the set of program variables. *Flow* specifies the direction of data movement, and *pattern* specifies the initial data layout.

Flow maps program variables to the same d-dimensional integer space as *place*. The intention is to indicate, for every processor in the network, which of its neighbors receive its output values at the next execution step, i.e., to which of its neighbors it must be connected by an outgoing channel. *Flow* is synthesized from *step* and *place* as follows: if variable v is accessed by distinct basic statements $s0$ and $s1$,

$$flow(v) =_{\text{def}} (place(s1)-place(s0))/(step(s1)-step(s0))$$

For variables v that are accessed by only one basic statement, we must provide the definition of *flow* explicitly. *Flow* is only well-defined if its images do not depend on the particular choice of pairs $s0$ and $s1$.

Pattern maps program variables to the same space as *place*. The intention is to lay out the input data for the various processors in an initial pattern such that the systolic execution can begin. (*Flow* describes the propagation of the data towards and through the network as the execution proceeds.) With constant fs being the arbitrary step value that we choose for the first parallel command, *pattern* is synthesized from *step*, *place*, and *flow* as follows: if variable v is accessed by basic statement s,

$$pattern(v) =_{\text{def}} place(s)-(step(s)-fs)*flow(v)$$

Pattern is only well-defined if its images do not depend on the particular choice of basic statement s. With *pattern* specifying the initial data layout, we can derive the data layout for successive steps of the systolic execution: the data layout after k steps is given by $pattern(v)+k*flow(v)$.

2.5. The Graphics System

We have implemented the transformation strategy and the computation of the previous functions in a graphics system on the Symbolics 3600. Our system can display two-dimensional processor layouts and simulate sequences of execution steps on them. At any fixed step, it displays the data layout and flow and indicates the active processors. The figures in this paper are hard-copies of images produced by our system.

3. Theorems for Linear Systolic Designs

A number of researchers have analyzed systolic designs with notions of linear algebra [7, 16, 17, 19, 20]. We shall do something similar here. In this section, we investigate a specific class of systolic designs: linear systolic designs. We defer the proofs of theorems to the appendix.

A systolic design is *linear* if it is specified by linear step and place functions. Linear systolic designs are particularly interesting because their data movement proceeds at a fixed rate in straight lines. We limit our

[2]Two points $(p_0,\dots,p_{d-1})$ and $(q_0,\dots,q_{d-1})$ of the d-dimensional integer space are *neighbors* if $0 \le |p_i - q_i| \le 1$, where $0 \le i < d$.

theoretical discussion to programs with only one type of basic statement.[3] Let us denote the basic statement by $s(x_0,x_1,\ldots,x_{r-1})$. Also, we use $s[x_i'/x_i]$ to denote the substitution of x_i' for argument x_i in basic statement $s(x_0,x_1,\ldots,x_{r-1})$.

Formally, a systolic design is linear, if *step* and *place* are described by the following linear equations:

(E1) $step(s(x_0,x_1,\ldots,x_{r-1})) = \alpha_{0,0}x_0+\alpha_{0,1}x_1+\ldots+\alpha_{0,r-1}x_{r-1}+\alpha_{0,r}$

(E2) $place(s(x_0,x_1,\ldots,x_{r-1})) =$
$$(\alpha_{1,0}x_0+\alpha_{1,1}x_1+\ldots+\alpha_{1,r-1}x_{r-1}+\alpha_{1,r}, \ldots, \alpha_{d,0}x_0+\alpha_{d,1}x_1+\ldots+\alpha_{d,r-1}x_{r-1}+\alpha_{d,r})$$
where the range of *place* is the d-dimensional integer space. In a non-linear systolic design, equations (E1) and (E2) would be of a higher degree. We shall explain the derivation of *step* and discuss theorems about *place*, *flow*, and *pattern* that provide guidance in the choice of a place function.

Consider a non-empty parallel trace. The images of its individual basic statements under *step*, as defined in (E1), constitute a set of linear formulas. Take the image of the first basic statement in the parallel trace and equate it with a chosen number. Impose conditions (S1) and (S2) to derive equations for the other basic statements. The result is a set of linear equations in the variables $\alpha_{0,0}$, $\alpha_{0,1}$, ..., $\alpha_{0,r-1}$, and $\alpha_{0,r}$, whose solution determines *step*. However, the equations do not guarantee the existence of a unique solution. For example, if the parallel trace consists of only one statement, there are infinitely many solutions for *step*, all of which satisfy conditions (S1) and (S2). It is also possible that no solution exists at all.

While conditions (S1) and (S2) are, generally, sufficient to synthesize *step*, condition (P1) is not sufficient to synthesize *place*. We must propose *place* independently and test whether it satisfies (P1). The following theorem provides such a test.

Theorem 1: Let *step* be a linear step function for parallel trace *t* that satisfies (S1) and (S2). Let *place* be a linear place function for *t*. *Place* satisfies (P1) if the following equations have the zero vector as the unique solution:

$$\begin{array}{l}
\alpha_{0,0}u_0+\alpha_{0,1}u_1+\ldots+\alpha_{0,r-1}u_{r-1}=0 \\
\alpha_{1,0}u_0+\alpha_{1,1}u_1+\ldots+\alpha_{1,r-1}u_{r-1}=0 \\
\ldots \\
\alpha_{d,0}u_0+\alpha_{d,1}u_1+\ldots+\alpha_{d,r-1}u_{r-1}=0
\end{array}$$

where r is the number of arguments of basic statement s. In particular, if *place* maps to $r-1$ dimensions, i.e., $d=r-1$, *place* satisfies (P1) if the coefficient determinant of this previous system of equations is not zero.

Given a linear step function satisfying (S1) and (S2) and a linear place function satisfying (P1), we can compute *flow* and *pattern*. The computation of *flow* and *pattern* must be well-defined, that is, their result must not depend on the choice of basic statements. Matrix computation programs use subscripted variables. In our programming language, the variable subscripts appear as arguments of the program's basic statements. If the variable subscripts are determined by $r-1$ arguments of the r-argument statement, then the flow of the variable derived from *step* and *place* is well-defined. This property is stated in Theorem 2. In our programming example, matrix multiplication, matrix elements accessed by a basic statement are determined each by two of the statement's three arguments (Section 4).

Theorem 2: Let *step* be a linear step function for parallel trace *t* that satisfies (S1) and (S2). Let *place* be a linear place function for *t* that satisfies (P1). If the subscripts of variable v are determined by all but one of the r arguments of the basic statement, then *flow* is well-defined for variable v.

[3]This restriction is not as severe as it may seem. While matrix multiplication only requires single-type basic statements, we have been able to apply our theorems also to other programs that use basic statements of several types [10].

Given a parallel trace t which satisfies conditions (S1), (S2), and (P1), no two basic statements in t can be identical. If a variable's subscripts are determined by all r, not just $r-1$, arguments of a basic statement, this variable can be accessed by at most one basic statement. Therefore, we cannot derive its flow function, and have to provide that explicitly. In general, while the processor layout for a program with r-argument basic statements requires dimension $r-1$, the data layout requires dimension r. An example is matrix-vector multiplication [11].

Given a step function satisfying (S1) and (S2), a place function satisfying (P1), and a well-defined flow function, the derived pattern function is well-defined. This property is stated by Theorem 3.

Theorem 3: Let *step* be a linear step function for parallel trace t that satisfies (S1) and (S2). Let *place* be a linear place function for t that satisfies (P1). Let *flow*, derived from *step* and *place*, be well-defined. Then *pattern*, derived from *step*, *place*, and *flow*, is well-defined.

4. Systolic Designs of Matrix Multiplication

The problem is to multiply two distinct $n \times n$ matrices A and B and assign the product to a third $n \times n$ matrix C, such that

$$c_{i,j} = \sum_{k=0}^{n-1} (a_{i,k} * b_{k,j}) \qquad \text{for} \quad 0 \le i \le n-1 \text{ and } 0 \le j \le n-1$$

With inner product steps, the following program is a simple solution to matrix multiplication; it is assumed that matrix C is initially everywhere zero:

```
for $i$ from 0 to $n-1$ do
    for $j$ from 0 to $n-1$ do
        for $k$ from 0 to $n-1$ do $(i{:}j{:}k)$
```

Translated to our programming language, this program becomes:

	matrix-matrix(n):	*product*($n-1,n-1$)
	product($0,n$):	*row*($0,n,n$)
$\{i>0\}$	*product*(i,n):	*product*($i-1,n$); *row*(i,n,n)
	row($i,0,n$):	*inner-product*($i,0,n$)
$\{j>0\}$	*row*(i,j,n):	*row*($i,j-1,n$); *inner-product*(i,j,n)
	inner-product($i,j,0$):	$(i{:}j{:}0)$
$\{k>0\}$	*inner-product*(i,j,k):	*inner-product*($i,j,k-1$); $(i{:}j{:}k)$

The curly brackets on the left contain entry conditions on the formal parameters of the refinements. Our rather complex syntax has the advantage that each composition of two basic statements is represented explicitly by a semicolon. This will simplify the translation of the program into a sequential execution.

We consider matrices whose non-zero values are concentrated in a "band" around the diagonal. An inner product step $(i{:}j{:}k)$ containing off-band elements $a_{i,k}$ or $b_{k,j}$ does not change the value of $c_{i,j}$, i.e., is neutral. We exploit this neutrality. To identify off-band elements of the matrix, we must precisely describe the width of the band of non-zero elements around the diagonal. This *band width* is determined by two natural numbers: the largest distance p, of a potentially non-zero element in the upper triangle from the diagonal, and the largest distance, q, of a potentially non-zero element in the lower triangle from the diagonal. The *distance* of a matrix element from the diagonal is the absolute value of the difference of its two subscripts. In the following systolic designs, we fix the band widths of matrices A and B each to $p=1$ and $q=1$. As a result, the band width of matrix C is $p=2$ and $q=2$.

Only neutral inner product steps are idempotent. Since we exploit their neutrality, we do not exploit their idempotence.

On a parallel architecture that permits the sharing of variables, two inner product steps $(i0{:}j0{:}k0)$ and $(i1{:}j1{:}k1)$ are independent if their target variables $c_{i0,j0}$ and $c_{i1,j1}$ are distinct.[4] But we are interested in executions on particular, systolic architectures that do not permit the sharing of variables. Therefore, we must use a stronger independence criterion and require that $a_{i0,k0}$ and $a_{i1,k1}$ are distinct, $b_{k0,j0}$ and $b_{k1,j1}$ are distinct, and $c_{i0,j0}$ and $c_{i1,j1}$ are distinct. Recall that the three variables of an individual inner product step are distinct by assumption.

All inner product steps are commutative. This makes commutativity, per se, meaningless. We do not exploit commutativity in trace transformations unless it is a consequence of independence.

Therefore, we declare the following semantic relations of neutrality and independence for inner product steps:

(D1) $1<k-i \lor 1<i-k \lor 1<j-k \lor 1<k-j \Rightarrow \mathrm{ntr}\,(i{:}j{:}k)$

(D2) $(i_0 \neq i_1 \lor j_0 \neq j_1) \land (i_0 \neq i_1 \lor k_0 \neq k_1) \land (j_0 \neq j_1 \lor k_0 \neq k_1) \Rightarrow (i_0{:}j_0{:}k_0)\ \mathrm{ind}\ (i_1{:}j_1{:}k_1)$

4.1. The First Design

Substituting ";" with "→" in the program to obtain a sequential trace, and then applying *transform* to the sequential trace, we derive a parallel trace. For example, the parallel trace for the multiplication of two 4×4 matrices (*matrix-matrix*(4)) expands to:

```
  <(0:0:0)>
→ <(0:0:1) (0:1:0) (1:0:0)> → <(0:1:1) (1:0:1) (1:1:0)>
→ <(0:2:1) (1:1:1) (2:0:1)> → <(1:1:2) (1:2:1) (2:1:1)>
→ <(1:2:2) (2:1:2) (2:2:1)> → <(1:3:2) (2:2:2) (3:1:2)>
→ <(2:2:3) (2:3:2) (3:2:2)> → <(2:3:3) (3:2:3) (3:3:2)>
→ <(3:3:3)>
```

This trace has length 10. In general, the length of the parallel trace is $3n-2$ and is independent of the band width. But the band width influences the width of the trace, i.e., the degree of concurrency.

The step function is derived from the parallel trace. Let the step function be a linear function:

$$step((i{:}j{:}k)) = \alpha_0 * i + \alpha_1 * j + \alpha_2 * k + \alpha_3$$

Recall that we are allowed to choose the step value of the first parallel command. We choose the value to make the constant term, α_3, 0. In this case, the step value of the first parallel command is 0. Applying the step function to the basic statements in the first two parallel commands of the above parallel trace, we obtain the following equations:

$$\begin{aligned} step((0{:}0{:}0)) &= \alpha_3 = 0 \\ step((0{:}0{:}1)) &= \alpha_2+\alpha_3 = 1 \\ step((0{:}1{:}0)) &= \alpha_1+\alpha_3 = 1 \\ step((1{:}0{:}0)) &= \alpha_0+\alpha_3 = 1 \end{aligned}$$

The solution to these equations is $\alpha_0=\alpha_1=\alpha_2=1$ and $\alpha_3=0$. The solution is consistent for the equations obtained by applying the step function to the rest of the basic statements. Therefore, the derived step function is:

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

The place function cannot be derived from the parallel trace but must be proposed separately. It seems promising to lay the processors out in a plane, i.e., in our method, on the two-dimensional integer lattice. Our first idea is to assign each basic statement to the point whose coordinates match the indices of the statement's target variable. This decision is rather arbitrary. At this stage, we do not have any information that might guide us in the choice of a processor layout. As we shall see later, other layouts are possible. Inner product step $(i{:}j{:}k)$ has target variable $c_{i,j}$. We propose:

[4]See the Independence Theorem of [13].

$$place((i{:}j{:}k)) =_{\text{def}} (i,j)$$

The dimension of *place* is two which is one less than the number of the arguments in $(i{:}j{:}k)$. By Theorem 1, *place* satisfies condition (P1), because the determinant constructed from the coefficients of *step* and *place* is not zero:

$$\begin{vmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{vmatrix} = 1$$

where the first row, (1 1 1), is constructed from *step*, the second row, (1 0 0), from the first dimension of *place*, and the third row, (0 1 0), from the second dimension of *place*.

Variable $a_{i,k}$ appears in basic statements $(i{:}j{:}k)$ and $(i{:}j+1{:}k)$, and these two statements are executed in consecutive steps. Therefore, we can derive the flow of $a_{i,k}$:

$$\begin{aligned} flow(a_{i,k}) &= place((i{:}j+1{:}k)) - place((i{:}j{:}k)) \\ &= (0,1) \end{aligned}$$

Similarly, we derive the flows of $b_{k,j}$ and $c_{i,j}$:

$$\begin{aligned} flow(b_{k,j}) &= place((i+1{:}j{:}k)) - place((i{:}j{:}k)) \\ &= (1,0) \end{aligned}$$

$$\begin{aligned} flow(c_{i,j}) &= place((i{:}j{:}k+1)) - place((i{:}j{:}k)) \\ &= (0,0) \end{aligned}$$

Variables $c_{i,j}$ stay stationary during the computation. By Theorem 2, *flow* is well-defined.

With functions *step*, *place*, and *flow*, we derive the initial data layout as follows:

$$\begin{aligned} pattern(a_{i,k}) &= place((i{:}j{:}k)) - step((i{:}j{:}k)) * flow(a_{i,k}) \\ &= (i, -i-k) \end{aligned}$$

$$\begin{aligned} pattern(b_{k,j}) &= place((i{:}j{:}k)) - step((i{:}j{:}k)) * flow(b_{k,j}) \\ &= (-j-k, j) \end{aligned}$$

$$\begin{aligned} pattern(c_{i,j}) &= place((i{:}j{:}k)) - step((i{:}j{:}k)) * flow(c_{i,j}) \\ &= (i,j) \end{aligned}$$

By Theorem 3, *pattern* is well-defined.

The network of processors and the initial data layout, as produced by the graphics system, is depicted in Figure 1. Each dot represents an inner product step processor. Arrows represent the propagation of data. A variable name labelling an arrow indicates the location of that variable. If the arrow points to a processor, this variable is input to that processor at the current step of the systolic execution.

The processor layout of this design mirrors the band of matrix C. The number of processors depends on the size of the input. For matrices with large size, this design may require a large number of processors. We can improve this situation by proposing a different place function.

4.2. The Second Design

Let us assume that we will keep the band widths of the input matrices constant. That is, when increasing the size of the input, we never widen the matrices' bands. Under this assumption, we can derive for the same matrix multiplication program another design whose number of processors is constant. We must simply find a place function whose coordinates depend only on the band widths of the input matrices but not on their size. The band widths of the input matrices are determined by the differences of i and k and of j and k (see the enabling condition of our neutrality declaration). We choose our coordinates from these differences:

$place((i{:}j{:}k)) =_{\text{def}} (i-k, j-k)$

Again, other choices are possible. By Theorem 1, this place function also satisfies (P1):

$$\begin{vmatrix} 1 & 1 & 1 \\ 1 & 0 & -1 \\ 0 & 1 & -1 \end{vmatrix} = 3$$

With the new proposed function, we derive the following *flow* and *pattern*:

$$\begin{aligned} flow(a_{i,k}) &= (0,1) \\ flow(b_{k,j}) &= (1,0) \\ flow(c_{i,j}) &= (-1,-1) \end{aligned}$$

$$\begin{aligned} pattern(a_{i,k}) &= (i-k, -i-2k) \\ pattern(b_{k,j}) &= (-j-2k, j-k) \\ pattern(c_{i,j}) &= (2i+j, i+2j) \end{aligned}$$

Flow and *pattern* are, again, well-defined.

The network of processors and the initial data layout is depicted in Figure 2. This design is presented in [11]. The number of processors is $(p_A+q_A+1)*(p_B+q_B+1)$. It is independent of the size of the input.

After arriving at an improved processor layout, we now modify the program to improve execution speed. We could have proceeded in the converse order.

4.3. The Third Design

Recall that any two inner product steps are commutative. In Sect. 4.1, we decided not to declare this commutativity. A search reveals that a commutation in the definition of refinement *inner-product* yields the shortest trace:

	inner-product$(i,j,0)$:	$(i{:}j{:}0)$
$\{k>0\}$	*inner-product*(i,j,k):	$(i{:}j{:}k)$; *inner-product*$(i,j,k-1)$

The parallel trace obtained for the multiplication of two 4×4 matrices (*matrix-matrix*(4)) expands to:

```
   <> → <>
→  <(0:0:1)>
→  <(0:0:0)  (0:1:1)  (1:0:1)  (1:1:2)>
→  <(0:1:0)  (0:2:1)  (1:0:0)  (1:1:1)  (1:2:2)  (2:0:1)  (2:1:2)  (2:2:3)>
→  <(1:1:0)  (1:2:1)  (1:3:2)  (2:1:1)  (2:2:2)  (2:3:3)  (3:1:2)  (3:2:3)>
→  <(2:2:1)  (2:3:2)  (3:2:2)  (3:3:3)>
→  <(3:3:2)>
→  <> → <>
```

If we do not consider band width, i.e., do not exploit neutrality, this trace has the same length as previous trace: 10 or, in general, $3n-2$. But, contrary to the previous trace, a consideration of band width can shorten this trace: the leading and trailing empty parallel commands result from the elimination of neutral basic statements. Not counting the empty parallel commands, this trace has length 6 or, in general, $n+\min(p_A,q_B)+\min(q_A,p_B)$. Hence, for constant band width and large n, we achieve a speed-up by a factor of 3. The effect of the commutation in *inner-product* is that, in the execution, k is counted down, not up. Therefore, the derived step function contains a subtraction rather than an addition of k:

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

The step value of the first (non-empty) parallel command is -1 or, in general, $-\min(p_A,q_B)$. We keep the place function of the second design:

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

Again, we derive well-defined flow and pattern functions:

$$\begin{aligned} \mathit{flow}(a_{i,k}) &= (0,1)\\ \mathit{flow}(b_{k,j}) &= (1,0)\\ \mathit{flow}(c_{i,j}) &= (1,1)\\ \mathit{pattern}(a_{i,k}) &= (i-k,-i-\min(p_A,q_B))\\ \mathit{pattern}(b_{k,j}) &= (-j-\min(p_A,q_B),j-k)\\ \mathit{pattern}(c_{i,j}) &= (-j-\min(p_A,q_B),-i-\min(p_A,q_B)) \end{aligned}$$

Note that *pattern* depends on the band width because the value of the first step does.

The network of processors and the initial data layout (at the first inner product step) is depicted in Figure 3. This design is also presented in [23].

5. Evaluation

Let us review how we develop systolic executions and designs. We provide a program (in form of a refinement) and a processor layout (in form of a place function). Given to us are properties of the programming language (in form of semantic relations) and restrictions on the architecture (implicit in the requirements on *step, place, flow,* and *pattern*). From this information, we synthesize, via a sequential execution, a parallel execution of the program and, via a step function, the data layout and movement (in form of a flow function and a pattern function). We could also exchange what we propose and derive. For example, if we proposed the data movement, we could synthesize the data layout and the processor layout.

Our work is distinguished by the combination of three factors. Embedding systolic design into a general view of programming enables us to separate distinct concerns properly. The explicit formulation of a parallel execution provides a precise link between the two components proposed by the human in a systolic design: the program and the processor layout. Our insistence on formal rigor at every stage expedites the automation of a large part of the development. Theorems aid the human in his part of the development. The systolic design at which we arrive can be informally (graphically) conveyed to the human, but it also has a precise mathematical description.

These benefits are demonstrated by our graphics implementation. As a consequence of the isolation of different development stages (program, execution, architecture) in our method, we can quickly and easily change different parameters, one at a time, and obtain a clear display of the effect on the systolic design.

The pairing of a program with a processor layout makes the evaluation of a design particularly convenient: the program determines the execution speed (as the length of the parallel trace) and the processor layout determines the size of the design (as the number of processors). The density of the data layout is determined only by the pair but not by either component alone.[5] For example, our first and second designs of matrix multiplication are based on the same program but the densities of their data layouts differ. Similarly, our second and third designs have the same processor layout, but the densities of their data layouts differ.

At present, we use *transform* as a heuristic. Our initial definition of it removed neutral elements first, not last. In some cases, this version of *transform* leads to faster executions. We still abandoned it, because it also leads to more complicated step functions, and simplicity is important to us. *Transform* is just another variable in our method. So far, our specific transformation strategy has served us remarkably well [9, 10].

We are not very satisfied with the way in which we identified the commutation in the definition of *inner-product* that led to our third design for matrix multiplication. We also attempted commutations in the other refinements, *product* and *row*, but they lead to executions that are never shorter and sometimes longer. All we can

[5]In fact, it is given by the absolute value of the determinant derived from the coefficients of *step* and *place*. We have proved a theorem to that effect.

provide at this time is an implemented system that lets us conduct these searches conveniently. The fact that all statements of the matrix multiplication program are commutative is discouraging. It provides us with no information of what execution to pick.

To reach the first step of our parallel systolic execution, several steps of "soaking up" data may have to be taken. Similarly, after the last step of our execution, data remaining in the network may have to be "drained". After arriving at a particular design, we can compute the lengths of the soaking and draining phases from *step* and *place*. Soaking and draining influences the performance of the design.

We have applied our method of incremental systolic design to other problems like LU-decomposition [11] and polynomial evaluation [12]. Our method is particularly suitable for a search of different systolic designs for some fixed problem. An impressive example is our treatment of the Algebraic Path Problem. The Algebraic Path Problem subsumes many matrix computation problems, among them matrix inversion, transitive closure, and shortest paths. Its solutions are complicated systolic designs with seven different types of operations and different data items being reflected in different directions up to four times on their path through the processor array [21]. For variables whose flow is not constant over the entire execution, the well-definedness of *flow* and *pattern* is violated. However, we can cope with such cases in an incremental fashion. We can extend the parallel execution with statements that copy variables (whose direction of flow changes) to new variables (at the points of change). The flow of each of the resulting variables is then constant. We have obtained an algorithm by which the parallel execution can be successively enhanced with such reflection operations [10].

Programs lend themselves to a systolic implementation if they combine a few simple operations in a highly repetitive way. It is not easy to tell by looking at the program whether it permits a nice systolic implementation. We have not addressed this problem here. What we offer is a fast way to try. Our method works the better, the fewer types of basic operations need to be considered. Many different types of processors can cause an explosion in the number of semantic declarations. We expect our method to work best for problems in which the program does not reflect aspects of the systolic architecture. However, at least in the treatment of the Algebraic Path Problem, we were able to add operations imposed by the architecture at a later stage.

Our description of systolic designs does not explicitly address the propagation of synchronization signals as does, for instance, Snepscheut's systolic design for transitive closure [22]. We capture issues of synchronization, quite abstractly, in the parallel trace. They may be realized by synchronization signals or by some other means. For example, we think of our systolic designs as communicating an identification of the variable together with the variable's value. So far, all our examples have lead to systolic designs in which a processor can decide what operation to perform simply be inspecting the identifications of its input data.

Many researchers have investigated methods of systolic design in recent years (see the next section). All these methods require two kinds of input: one component that can be thought of as a program, and one component that gives some clue about the structure of the systolic array. In our approach both these inputs need not be cleverly chosen. Of the program, we require only that it solve the numerical problem at hand. For the place function, we can start with a simple proposition that looks promising. After evaluating the result of our inputs, we can make incremental variations. These variations may be random, or they may be carefully selected. In our matrix multiplication example, we adjusted each of the two inputs once.

6. Related Research

Chen [5, 6] chooses the inverse of our derivation. She supplies a "network", which is the analogue of our flow function, and an "abstract process structure" (a set of recurrence equations), which is the analogue of our refinement. Her informal derivation results in a "concrete structure", which is the analogue of our step and place functions. Chen does not spell out systolic executions, as we do with traces, and is, in general, less formal.

Like us, Moldovan and Fortes [19] require the input of a program, but their program must be augmented with "artificial" variables [18]. This augmentation is meant to specify parallelism and corresponds roughly to our semantic relations - except that semantic relations are properties of the programming language, not properties of individual programs. (The detection of parallelism receives more attention in another of their papers [8].) Systolic arrays are described by a space transformation which corresponds to our function *place* and a time transformation which corresponds to our function *step*. Moldovan and Fortes require the input of both transformations, while we only require the input of *place* (or even only part of *place*). Similarly to Chen, Moldovan and Fortes present an algorithm by which the space transformation can be derived from a set of proposed flow vectors. Mirankler and Winkler [17] employ the same space-time transformation as Moldovan but use a graph representation. Moldovan and Fortes propose guidelines for the derivation of some programs.

Chandy and Misra [4] propose an "invariant", which corresponds to our step function, and, with some additional assumptions, derive a systolic program from it. A program in their language, Unity [3], is a repeating multiple assignment statement. Chandy and Misra envision Unity as a tool in which programming solutions for many different architectures can be expressed with equal convenience. They equate the Unity programs that they derive with systolic executions and, indeed, with systolic architectures. An essential aspect of our synthesis method is that we distinguish the three concepts of a program, a trace, and an architecture.

Lam and Mostow [12] employ an implemented method of transformation similar to ours but, again, less precise. They require annotations to the Pascal-like program that give a clue about the processor layout ("in place" or "in parallel").

Cappello and Steiglitz [2] describe a method of systolic design by geometric transformation. They derive a first data flow scheme from a sequential program execution. The data flow scheme is expressed geometrically in space-time and is, usually, not well-suited for implementation. It is then improved by geometric transformations proposed by the human. As many other approaches in VLSI theory, this one aims at chip layout, not at programming. Our centerpiece, the parallel execution, is missing.

Systolic design spans several levels of abstraction, from a specification to a chip layout. The two ends of this spectrum are, at present, best understood. The front end is the refinement of a specification into an abstract program. Solutions to this end are offered by work in programming methodology. The back end is the refinement of an abstract systolic architecture into an optimized concrete one. Solutions to this end are offered by work in VLSI design. Our work provides a connection of both ends: it links an abstract program with an abstract systolic architecture.

Acknowledgements

This research was partially supported by Grant No. 26-7603-35 from the Lockheed Missiles & Space Corporation and by Grant No. DCR-8610427 of the National Science Foundation.

References

1. Boyer, R. S., and Moore, J S. *A Computational Logic*. ACM Monograph Series, Academic Press, 1979.

2. Cappello, P. R., and Steiglitz, K. Unifying VLSI Array Design with Linear Transformations of Space-time. In *Advances in Computing Research, Vol. 2: VLSI Theory*, F. P. Preparata, Ed., JAI Press Inc., 1984, pp. 23-65.

3. Chandy, M. Concurrent Programming for the Masses. Proc. 4th Ann. ACM Symp. on Principles of Distributed Computing, 1985, pp. 1-12.

4. Chandy, K. M., and Misra, J. "Systolic Algorithms as Programs". *Distributed Computing 1*, 3 (1986), 177-183.

5. Chen, M. C. Synthesizing Systolic Designs. YALEU/DCS/RR-374, Department of Computer Science, Yale University, Mar., 1985.

6. Chen, M. C. A Parallel Language and Its Compilation to Multiprocessor Machines or VLSI. Proc. 13th Ann. ACM Symp. on Principles of Programming Languages, 1986, pp. 131-139.

7. Delosme, J.-M., and Ipsen, I. Overview over SAGA and CONDENSE. Yale University, Jan., 1987.

8. Fortes, J. A. B., and Moldovan, D.I. "Parallelism Detection and Transformation Techniques for VLSI Algorithms". *Journal of Parallel and Distributed Computing 2*, 3 (Aug. 1985), 277-301.

9. Huang, C.-H., and Lengauer, C. The Derivation of Systolic Implementations of Programs. TR-86-10, Department of Computer Sciences, The University of Texas at Austin, Apr., 1986. Revised: Jan., 1987. To appear in *Acta Informatica*.

10. Huang, C.-H., and Lengauer, C. An Incremental, Mechanical Development of Systolic Solutions to the Algebraic Path Problem. TR-86-28, Department of Computer Sciences, The University of Texas at Austin, Dec., 1986.

11. Kung, H. T., and Leiserson, C. E. Algorithms for VLSI Processor Arrays. In *Introduction to VLSI Systems*, C. Mead and L. Conway, Eds., Addison-Wesley, 1980. Sect. 8.3.

12. Lam, M. S., and Mostow, J. "A Transformational Model of VLSI Systolic Design". *Computer 18*, 2 (Feb. 1985), 42-52.

13. Lengauer, C., and Hehner, E. C. R. "A Methodology for Programming with Concurrency: An Informal Presentation". *Science of Computer Programming 2*, 1 (Oct. 1982), 1-18.

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

15. Lengauer, C., and Huang, C.-H. A Mechanically Certified Theorem about Optimal Concurrency of Sorting Networks. Proc. 13th Ann. ACM Symp. on Principles of Programming Languages, 1986, pp. 307-317.

16. Li, G.-H., and Wah, B. W. "The Design of Optimal Systolic Arrays". *IEEE Trans. on Computers C-34*, 1 (Jan. 1985), 66-77.

17. Miranker, W. L., and Winkler, A. "Spacetime Representations of Computational Structures". *Computing 32*, 2 (1984), 93-114.

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

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

20. Quinton, P. Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations. Proc. 11th Ann. Int. Symp. on Computer Architecture, 1984, pp. 208-214.

21. Rote, G. "A Systolic Array Algorithm for the Algebraic Path Problem (Shortest Paths; Matrix Inversion)". *Computing 34*, 3 (1985), 191-219.

22. van de Snepscheut, J. L. A. A Derivation of a Distributed Implementation of Warshall's Algorithm (JAN-113a). CS 8505, Dept. of Mathematics and Computing Science, University of Groningen, 1985.

23. Weiser, U., and Davis, A. A Wavefront Notation Tool for VLSI Array Design. In *VLSI Systems and Computations*, H. T. Kung, B. Sproull, and G. Steele, Eds., Computer Science Press, 1981, pp. 226-234.

Appendix: Proofs

We restate and prove Theorems 1, 2, and 3 of Section 3.

Theorem 1: Let *step* be a linear step function for parallel trace t that satisfies (S1) and (S2). Let *place* be a lineaı place function for t. *Place* satisfies (P1) if the following equations have the zero vector as the unique solution:

$$\begin{aligned}&\alpha_{0,0}u_0+\alpha_{0,1}u_1+\ldots+\alpha_{0,r-1}u_{r-1}=0\\&\alpha_{1,0}u_0+\alpha_{1,1}u_1+\ldots+\alpha_{1,r-1}u_{r-1}=0\\&\ldots\\&\alpha_{d,0}u_0+\alpha_{d,1}u_1+\ldots+\alpha_{d,r-1}u_{r-1}=0\end{aligned}$$

where r is the number of arguments of basic statement s.

Proof:

Place satisfies (P1)

= {conditions (S1), (S2) and (P1)}

for all basic statements $s(x_0,x_1,\ldots,x_{r-1})$ and $s(y_0,y_1,\ldots,y_{r-1})$ in t,

$$\begin{aligned}&s(x_0,x_1,\ldots,x_{r-1})\neq s(y_0,y_1,\ldots,y_{r-1})\wedge step(s(x_0,x_1,\ldots,x_{r-1}))=step(s(y_0,y_1,\ldots,y_{r-1}))\\\Rightarrow\ &place(s(x_0,x_1,\ldots,x_{r-1}))\neq place(s(y_0,y_1,\ldots,y_{r-1}))\end{aligned}$$

= {*step* and *place* are linear, and equations (E1) and (E2)}

for all basic statements $s(x_0,x_1,\ldots,x_{r-1})$ and $s(y_0,y_1,\ldots,y_{r-1})$ in t,

$$\begin{aligned}&s(x_0,x_1,\ldots,x_{r-1})\neq s(y_0,y_1,\ldots,y_{r-1})\\\wedge\ &\alpha_{0,0}x_0+\alpha_{0,1}x_1+\ldots+\alpha_{0,r-1}x_{r-1}+\alpha_{0,r}=\alpha_{0,0}y_0+\alpha_{0,1}y_1+\ldots+\alpha_{0,r-1}y_{r-1}+\alpha_{0,r}\\\Rightarrow\ &(\alpha_{1,0}x_0+\alpha_{1,1}x_1+\ldots+\alpha_{1,r-1}x_{r-1}+\alpha_{1,r},\ \ldots,\ \alpha_{d,0}x_0+\alpha_{d,1}x_1+\ldots+\alpha_{d,r-1}x_{r-1}+\alpha_{d,r})\\\neq\ &(\alpha_{1,0}y_0+\alpha_{1,1}y_1+\ldots+\alpha_{1,r-1}y_{r-1}+\alpha_{1,r},\ \ldots,\ \alpha_{d,0}y_0+\alpha_{d,1}y_1+\ldots+\alpha_{d,r-1}y_{r-1}+\alpha_{d,r})\end{aligned}$$

= {algebraic simplification}

for all basic statements $s(x_0,x_1,\ldots,x_{r-1})$ and $s(y_0,y_1,\ldots,y_{r-1})$ in t,

$$\begin{aligned}&s(x_0,x_1,\ldots,x_{r-1})\neq s(y_0,y_1,\ldots,y_{r-1})\\\wedge\ &\alpha_{0,0}x_0+\alpha_{0,1}x_1+\ldots+\alpha_{0,r-1}x_{r-1}+\alpha_{0,r}=\alpha_{0,0}y_0+\alpha_{0,1}y_1+\ldots+\alpha_{0,r-1}y_{r-1}+\alpha_{0,r}\\\Rightarrow\ &\alpha_{1,0}x_0+\alpha_{1,1}x_1+\ldots+\alpha_{1,r-1}x_{r-1}+\alpha_{1,r}\neq\alpha_{1,0}y_0+\alpha_{1,1}y_1+\ldots+\alpha_{1,r-1}y_{r-1}+\alpha_{1,r}\\\vee\ &\ldots\\\vee\ &\alpha_{d,0}x_0+\alpha_{d,1}x_1+\ldots+\alpha_{d,r-1}x_{r-1}+\alpha_{d,r}\neq\alpha_{d,0}y_0+\alpha_{d,1}y_1+\ldots+\alpha_{d,r-1}y_{r-1}+\alpha_{d,r}\end{aligned}$$

= {algebraic simplification}

for all basic statements $s(x_0,x_1,\ldots,x_{r-1})$ and $s(y_0,y_1,\ldots,y_{r-1})$ in t,

$$\begin{aligned}&s(x_0,x_1,\ldots,x_{r-1})\neq s(y_0,y_1,\ldots,y_{r-1})\\\wedge\ &\alpha_{0,0}(x_0-y_0)+\alpha_{0,1}(x_1-y_1)+\ldots+\alpha_{0,r-1}(x_{r-1}-y_{r-1})=0\\\Rightarrow\ &\alpha_{1,0}(x_0-y_0)+\alpha_{1,1}(x_1-y_1)+\ldots+\alpha_{1,r-1}(x_{r-1}-y_{r-1})\neq0\\\vee\ &\ldots\\\vee\ &\alpha_{d,0}(x_0-y_0)+\alpha_{d,1}(x_1-y_1)+\ldots+\alpha_{d,r-1}(x_{r-1}-y_{r-1})\neq0\end{aligned}$$

= {predicate calculus}

for all basic statements $s(x_0,x_1,\ldots,x_{r-1})$ and $s(y_0,y_1,\ldots,y_{r-1})$ in t,

$$\begin{aligned}&\alpha_{0,0}(x_0-y_0)+\alpha_{0,1}(x_1-y_1)+\ldots+\alpha_{0,r-1}(x_{r-1}-y_{r-1})=0\\\wedge\ &\alpha_{1,0}(x_0-y_0)+\alpha_{1,1}(x_1-y_1)+\ldots+\alpha_{1,r-1}(x_{r-1}-y_{r-1})=0\\\wedge\ &\ldots\\\wedge\ &\alpha_{d,0}(x_0-y_0)+\alpha_{d,1}(x_1-y_1)+\ldots+\alpha_{d,r-1}(x_{r-1}-y_{r-1})=0\end{aligned}$$

$\Rightarrow \quad s(x_0,x_1,\ldots,x_{r-1})=s(y_0,y_1,\ldots,y_{r-1})$

$\Leftarrow$ {algebraic simplification}

$$\begin{array}{l}\alpha_{0,0}u_0+\alpha_{0,1}u_1+\ldots+\alpha_{0,r-1}u_{r-1}=0\\ \alpha_{1,0}u_0+\alpha_{1,1}u_1+\ldots+\alpha_{1,r-1}u_{r-1}=0\\ \ldots\\ \alpha_{d,0}u_0+\alpha_{d,1}u_1+\ldots+\alpha_{d,r-1}u_{r-1}=0\end{array}$$

have the zero vector as the unique solution.

(End of Proof)

Theorem 2: Let *step* be a linear step function for parallel trace *t* that satisfies (S1) and (S2). Let *place* be a linear place function for *t* that satisfies (P1). If the subscripts of variable *v* are determined by all but one of the *r* arguments of the basic statement, then *flow* is well-defined for variable *v*.

Proof: Let $s_x=s(x_0,\ldots,x_i,\ldots,x_{r-1})$, $s_{x'}=s_x[x_i'/x_i]$, $s_y=s_x[y_i/x_i]$, and $s_{y'}=s_x[y_i'/x_i]$. Let the subscripts of variable *v* be x_0, ..., x_{i-1}, x_{i+1}, ..., and x_{r-1}, that is, the arguments of basic statement s_x, except the $(i+1)$-st one, x_i. Then, s_x, $s_{x'}$, s_y, and $s_{y'}$ all access variable $v_{x_0,\ldots,x_{i-1},x_{i+1},\ldots,x_{r-1}}$. Assuming $step(s_x)\neq step(s_{x'})$, and $step(s_y)\neq step(s_{y'})$, we can conclude:

flow is well-defined for variable $v_{x_0,\ldots,x_{i-1},x_{i+1},\ldots,x_{r-1}}$

= {well-definedness}

$(place(s_x)-place(s_{x'}))/(step(s_x)-step(s_{x'}))=(place(s_y)-place(s_{y'}))/(step(s_y)-step(s_{y'}))$

= {*step* and *place* are linear, and s_x, $s_{x'}$, s_y, and $s_{y'}$ have identical arguments in all positions but i}

$(\alpha_{1,i}(x_i-x_i'),\ldots,\alpha_{d,i}(x_i-x_i'))/\alpha_{0,i}(x_i-x_i')=(\alpha_{1,i}(y_i-y_i'),\ldots,\alpha_{d,i}(y_i-y_i'))/\alpha_{0,i}(y_i-y_i')$

= {algebraic simplification}

$(\alpha_{1,i}/\alpha_{0,i},\ldots,\alpha_{d,i}/\alpha_{0,i})=(\alpha_{1,i}/\alpha_{0,i},\ldots,\alpha_{d,i}/\alpha_{0,i})$

= {algebraic simplification}

true

(End of Proof)

Theorem 3: Let *step* be a linear step function for parallel trace *t* that satisfies (S1) and (S2). Let *place* be a linear place function for *t* that satisfies (P1). Let *flow*, derived from *step* and *place*, be well-defined. Then *pattern*, derived from *step*, *place*, and *flow*, is well-defined.

Proof: If basic statements *s0* and *s1* are distinct and access variable *v* of identical subscripts:

pattern is well-defined for variable *v*

= {well-definedness}

$place(s0)-(step(s0)-fs)*flow(v)=place(s1)-(step(s1)-fs)*flow(v)$

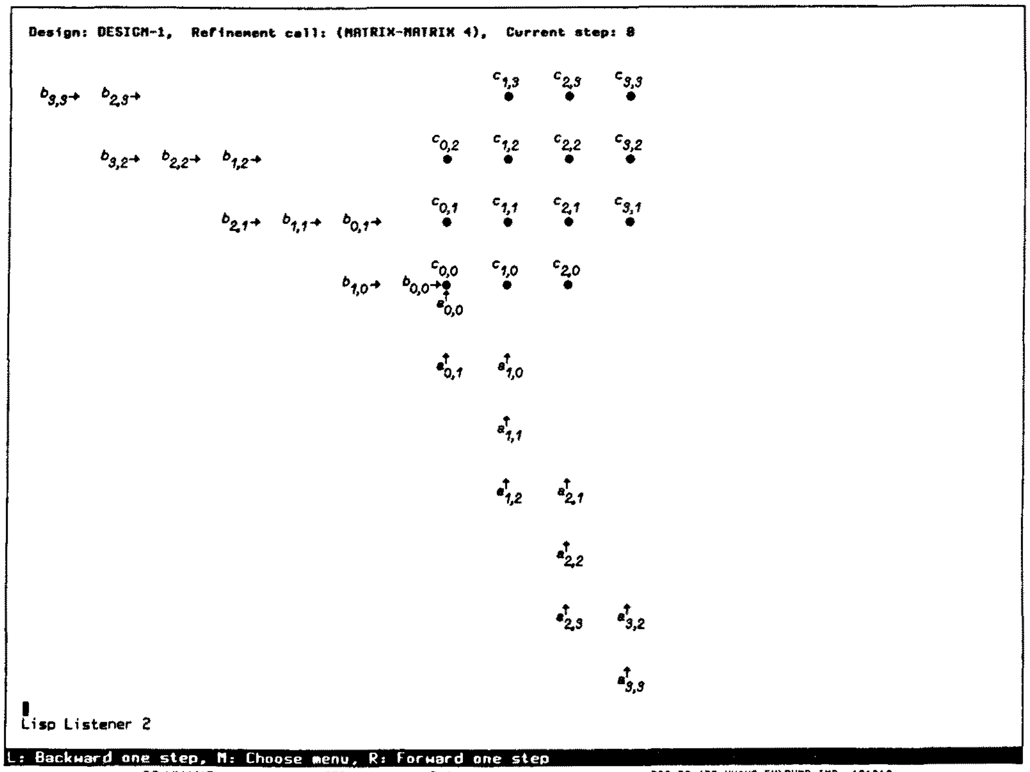
= {algebraic simplification}

$place(s0)-place(s1)=(step(s0)-step(s1))*flow(v)$

= {definition of *flow*}

true

(End of Proof)

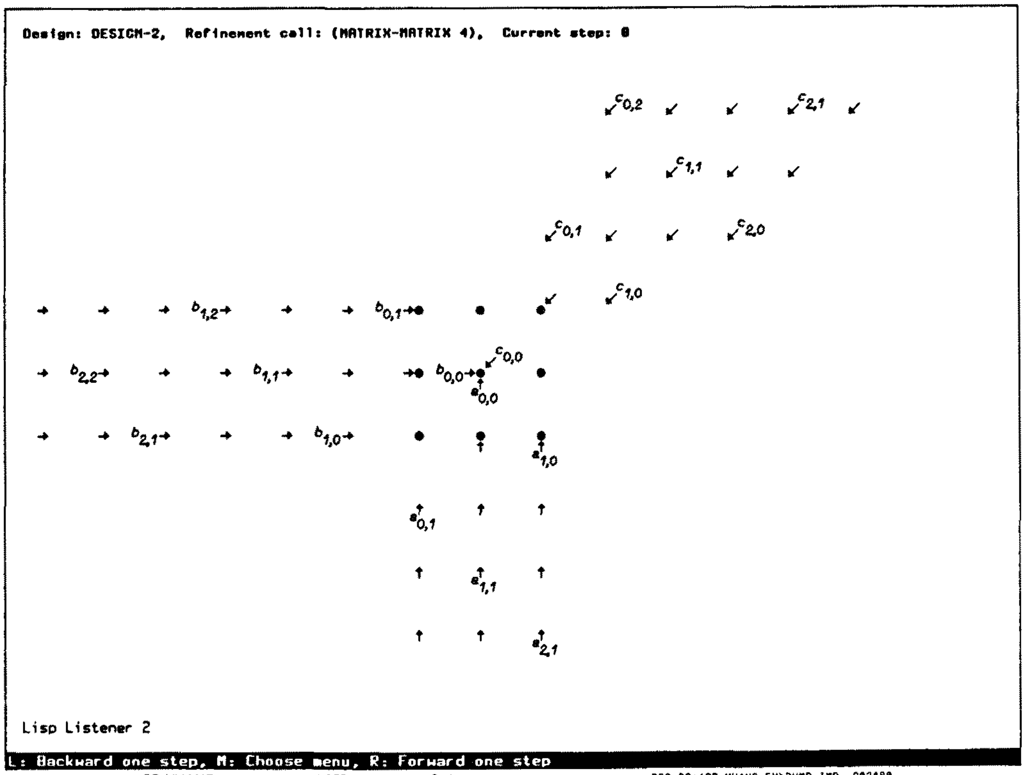


```
Design: DESIGN-1,  Refinement call: (MATRIX-MATRIX 4),  Current step: 0
```

$b_{3,3}\rightarrow$ $b_{2,3}\rightarrow$
$b_{3,2}\rightarrow$ $b_{2,2}\rightarrow$ $b_{1,2}\rightarrow$
$b_{2,1}\rightarrow$ $b_{1,1}\rightarrow$ $b_{0,1}\rightarrow$
$b_{1,0}\rightarrow$ $b_{0,0}\rightarrow$
$c_{1,3}$ $c_{2,3}$ $c_{3,3}$
$c_{0,2}$ $c_{1,2}$ $c_{2,2}$ $c_{3,2}$
$c_{0,1}$ $c_{1,1}$ $c_{2,1}$ $c_{3,1}$
$c_{0,0}$ $c_{1,0}$ $c_{2,0}$
$a_{0,0}$
$a_{0,1}$ $a_{1,0}$
$a_{1,1}$
$a_{1,2}$ $a_{2,1}$
$a_{2,2}$
$a_{2,3}$ $a_{3,2}$
$a_{3,3}$

```
Lisp Listener 2
L: Backward one step, M: Choose menu, R: Forward one step
          CS.HUANG          USER:          Ty1                    R20:PS:<CS.HUANG.EV>DUMP.IMP  101018
```

Figure 1. Matrix Multiplication -- The First Design



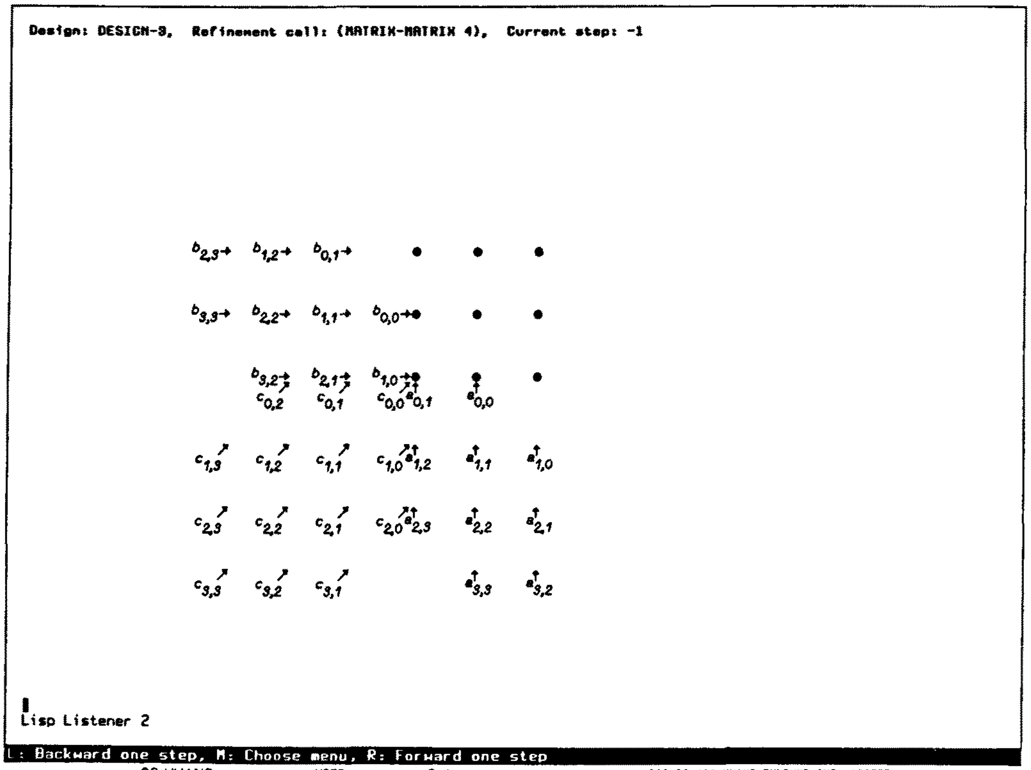
Design: DESIGN-2, Refinement cell: (MATRIX-MATRIX 4), Current step: 0

Lisp Listener 2

L: Backward one step, M: Choose menu, R: Forward one step

CS.HUANG USER: Tyi R20:PS:<CS.HUANG.EV>DUMP.IMP 892480

Figure 2. Matrix Multiplication -- The Second Design



```
Design: DESIGN-3,  Refinement call: (MATRIX-MATRIX 4),  Current step: -1
```

$b_{2,3}$ $b_{1,2}$ $b_{0,1}$
$b_{3,3}$ $b_{2,2}$ $b_{1,1}$ $b_{0,0}$
$b_{3,2}$ $b_{2,1}$ $b_{1,0}$
$c_{0,2}$ $c_{0,1}$ $c_{0,0}$ $a_{0,1}$ $a_{0,0}$
$c_{1,3}$ $c_{1,2}$ $c_{1,1}$ $c_{1,0}$ $a_{1,2}$ $a_{1,1}$ $a_{1,0}$
$c_{2,3}$ $c_{2,2}$ $c_{2,1}$ $c_{2,0}$ $a_{2,3}$ $a_{2,2}$ $a_{2,1}$
$c_{3,3}$ $c_{3,2}$ $c_{3,1}$ $a_{3,3}$ $a_{3,2}$
Lisp Listener 2
L: Backward one step, M: Choose menu, R: Forward one step

CS.HUANG USER: Ty1 R20.PS:<CS.HUANG.EV>DUMP.IMP 100776

Figure 3. Matrix Multiplication -- The Third Design