# 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

TR-86-17 July 1986

**Abstract**

We present a 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 proved in the paper. 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.

This research was partially supported by Grant No. 26-7603-35 from the Lockheed Missiles & Space Corporation.

# 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. Admittedly, we cannot deal with all programming problems, but the horizon of our world is expanding. Presently, it contains sorting problems [9] and matrix computation problems [5]. 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 [6], i.e., they will be networks of processors that are connected in simple patterns and perform simple operations. 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. We call the end product comprising the program, parallel execution, and systolic architecture a *systolic design*. 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.

The following section is a brief introduction to the design method. First, we describe programs and traces. Second, we present a trace transformation strategy which yields parallel traces. Third, we describe systolic architectures formally with the aid of four functions. Finally, we provide a brief idea of our graphics support of the design method. In Sect. 3, we prove theorems about the four functions for a

specific class of systolic designs. Sect. 4 contains our example: matrix multiplication. Conclusions relate our work to others in systolic design.

## 2. The Design Method: Programs, Traces, and Architectures

### 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. An entry condition involving the formal parameters may be added in curly brackets. 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. It is represented by a name followed by a list of parameters, or some special notation combining a number of parameters. Each of our programs will contain a number of basic statements. They will be explained when the program is presented.

- 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→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 <*S0 S1*> (angle brackets denote parallel execution). We call <*S0 S1*> a *parallel command*, and a trace with parallel commands *parallel trace*.

### 2.3. Trace Transformations

Transformations of a trace must preserve the trace's effect but may alter its execution time. Trace transformations are justified by semantic relations that basic statements may or may not satisfy.

(1) A basic statement $S$ 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.

(2) A basic statement $S$ 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.

(3) Two basic statements *S0* and *S1* that are *commutative* can be executed in any order with identical effect. Thus, *S0→S1* in a trace may be transformed to *S1→S0*.

(4) Two basic statements *S0* and *S1* that are *independent* can be executed in parallel and in sequence with identical effect. Thus *S0→S1* in a trace may be transformed to <*S0 S1*>. Independence implies commutativity.

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

*enabling predicate* ⇒ *semantic relation*

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

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 in the following way:

$transform(l) = remove\text{-}all\text{-}ntr(ravel\text{-}trans(l))$

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 [5] in the Boyer-Moore computational logic [1] and mechanically proved correct.

### 2.4. Architectures

A parallel trace specifies a partial order of basic statements without reference to a particular architecture. We will connect the parallel execution to the systolic architecture that we have in mind. We specify a systolic architecture with the help of two functions: *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

[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. Our programming examples lead to traces whose basic statements are all distinct.

parallel trace in their order of execution. *Step* is derived from the parallel trace by solving a set of equations. 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 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.

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

(P1) basic statements of the same parallel command must be assigned distinct places.

In systolic implementations, program variables are not realized by storage cells but by input and output channels. We have to specify a layout and flow of data that provides each processor with the expected inputs at the step at which a basic statement is supposed to execute. In our systolic architectures, processors are only connected by unidirectional channels to processors that occupy neighboring coordinates. That is, data propagate through the network at a fixed rate in a fixed direction. For architectures 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 $r$-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 $v$ accessed by distinct basic statements $s0$ and $s1$,

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

For variables $v$ that accessed by only one basic statement, we have to 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 $s$,

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

*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. We can use the system to simulate graphically systolic executions on stylized architectures. Our system displays a network of processors with interconnecting channels. At any fixed step, it also displays the data layout and indicates processors active at that step.

The three central commands of the graphics system are *add-processor*, *add-design*, and *display-design*. *Add-processor* adds the specification of a processing element, which consists of the name of the basic statement the processor is supposed to apply, its number of arguments, and the identifiers of its input and output variables, i.e., channels. *Add-design* asks for a design name and the four components that are necessary to synthesize a design: a program refinement, semantic declarations, a step, and a place function. At present, we require the explicit specification of *step*, even though it could be synthesized from the parallel trace (see the following section). *Display-design* takes a design name and a refinement call. It displays the data layout for the submitted call on the submitted processor layout and simulates the systolic execution. The simulation can be controlled to advance or to back up a number of steps. At each step, the active processors are highlighted.

## 3. Theorems for Linear Systolic Designs

In this section, we investigate a specific class of systolic designs: linear systolic designs. A systolic design is *linear* if it is specified by linear step and place functions. Linear systolic designs are particularly interesting because, usually, their data movement proceeds at a fixed rate in a fixed direction. We limit our discussion to programs with only one type of basic statement, as is the case for matrix multiplication. 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 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 may also occur 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:

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

**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 $t$,

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

$=$ {*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 $t$,

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

$=$ {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 $t$,

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

$=$ {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 $t$,

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

$\Leftarrow$ {algebraic simplification}

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

have the zero vector as the unique solution.

(End of Proof)

When *place* maps to $r-1$ dimensions, the test for (P1) reduces to the computation of a determinant.

**Corollary 1:** Given the premises of Theorem 1, if *place* maps into the $(r-1)$-dimensional integer space, it satisfies (P1) if the following determinant is not zero:

$$
\begin{vmatrix}
\alpha_{0,0} & \alpha_{0,1} & \ldots & \alpha_{0,r-1} \\
\alpha_{1,0} & \alpha_{1,1} & \ldots & \alpha_{1,r-1} \\
\ldots & & & \\
\alpha_{r-1,0} & \alpha_{r-1,1} & \ldots & \alpha_{r-1,r-1}
\end{vmatrix}
$$

In general, the reverse implication does not hold. Nevertheless, this corollary encourages the choice of $r-1$ for $d$.

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. In programs with systolic implementations, variables are usually array or matrix elements. 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 (Sect. 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 $r-1$ 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)

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 can not 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 [5].

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.

**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)

## 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

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

In the solution to this problem, we will use a basic statement called *inner product step*. An inner product step accesses the matrix elements $a_{i,k}$, $b_{k,j}$ and $c_{i,j}$, 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 $(i{:}j{:}k)$.

With inner product steps, the following program performs matrix multiplication if matrix $C$ is initially zero everywhere:

| | | |
|---|---|---|
| | *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*$(i,j,k-1)$; $(i{:}j{:}k)$ |

To accumulate the elements of result matrix $C$, we must recurse over $i$ and $j$. To compute each element of $C$, we must recurse over $k$. Refinement *matrix-matrix* consists of these three recursions.

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.[2] In the following systolic designs, we fix 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.[3] 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. We do not exploit commutativity unless it is a consequence of independence.

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

(D1) $1{<}k{-}i \vee 1{<}i{-}k \vee 1{<}j{-}k \vee 1{<}k{-}j \Rightarrow \text{ntr}\ (i{:}j{:}k)$

(D2) $(i_0{\neq}i_1 \vee j_0{\neq}j_1) \wedge (i_0{\neq}i_1 \vee k_0{\neq}k_1) \wedge (j_0{\neq}j_1 \vee k_0{\neq}k_1) \Rightarrow (i_0{:}j_0{:}k_0)\ \text{ind}\ (i_1{:}j_1{:}k_1)$

---

[2] The *distance* of a matrix element from the diagonal is the absolute value of the difference of its two subscripts.

[3] See the Independence Theorem of [8].

### 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, $\alpha 3$, 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:

$$\begin{aligned} \alpha 3 &= 0 \\ \alpha 2{+}\alpha 3 &= 1 \\ \alpha 1{+}\alpha 3 &= 1 \\ \alpha 0{+}\alpha 3 &= 1 \end{aligned}$$

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. The inner product step has three arguments, i.e., $r{=}3$. Without much reflection, we propose a simple $(r-1)$-dimensional, i.e., two-dimensional, place function:

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

By Corollary 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 $a_{i,k}$:

$$flow(a_{i,k}) = place((i{:}j{+}1{:}k)) - place((i{:}j{:}k)) = (0,1)$$

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

$$flow(b_{k,j}) = place((i{+}1{:}j{:}k)) - place((i{:}j{:}k)) = (1,0)$$

$$flow(c_{i,j}) = place((i{:}j{:}k{+}1)) - place((i{:}j{:}k)) = (0,0)$$

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:

$$pattern(a_{i,k}) = place((i{:}j{:}k)) - step((i{:}j{:}k)) * flow(a_{i,k}) = (i{-}k, {-}i{-}2k)$$

$$pattern(b_{k,j}) = place((i{:}j{:}k)) - step((i{:}j{:}k)) * flow(b_{k,j}) = ({-}j{-}2k, j{-}k)$$

$$pattern(c_{i,j}) = place((i{:}j{:}k)) - step((i{:}j{:}k)) * flow(c_{i,j}) = (i,j)$$

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 the current value of that variable. If the arrow points to a processor, this value 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

For the program of the first design, we can derive another architecture whose number of processors is independent of the size of the input. Recall that the neutrality of basic statements is determined by the band width. The corresponding semantic declaration, (D1), makes reference to the distance of an element from the diagonal. We define *place* in terms of that distance:

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

*Place* lays out processors in the bound of the band width, thereby, eliminating the dependence on the size of the input. What remains is a dependence only on the band width. By Corollary 1, this place function also satisfies (P1).

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*, again, are well-defined.

The network of processors and the initial data layout is depicted in Figure 2. This design is presented in [6]. The number of processors is $(p_A+q_A+1)*(p_B+q_B+1)$, where $p_A$ and $q_A$ describe the band width of matrix $A$, and $p_B$ and $q_B$ describe the band width of matrix $B$. 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. We decided not to declare this commutativity. Now we realize that a specific commutation can lead to a better systolic design. We perform this commutation in the definition of refinement *inner-product*:

| | | |
|---|---|---|
| | *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 $4\times4$ 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:

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

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 [12].

## 5. Conclusions

Our work is distinguished from others, e.g., [2, 4, 7, 10, 12], in several respects. Embedding systolic design into a general view of programming enables us to separate distinct concerns properly (Chandy and Misra [3] adopt a similar view). The explicit formulation of a parallel execution provides a precise link between the two components proposed by the human: the program (the solution of the correctness problem) and the processor layout (the solution of the implementation problem). 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.

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. The density of the data layout is determined only by the pair but not by either component alone. For example, our first and second designs 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. We conclude that one should not judge a systolic design by the density of its data layout.

We are not very satisfied with the way in which we identified the commutation in the definition of *inner-product* that lead to our third design. We also attempted commutations in the other refinements, *product* and *row*, but they lead to executions that are never faster and sometimes slower. All we can provide at this time is an implemented system that lets us conduct these searches conveniently.

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 the lengths of the soaking and draining phases from *step* and *place*. Soaking and draining influences the performance of the design.

The restrictions that we impose on systolic architectures include that data must travel in a fixed direction. However, we can also derive designs in which data are reflected or broadcast. To enable reflections we must copy variables, and to enable broadcasts we must duplicate variables in the program. For example, we have synthesized Rote's design of the Algebraic Path Problem [11] with our method.

## 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*, F. P. Preparata, Ed., JAI Press Inc., 1984, pp. 23-65.

**3.** Chandy, K. M., and Misra, J. "Systolic Algorithms as Programs". *Distributed Computing 1*, 3 (1986-87). To appear.

**4.** 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.

**5.** 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, Mar., 1986.

**6.** 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.

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

**8.** 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.

**9.** 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.

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

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

**12.** 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.

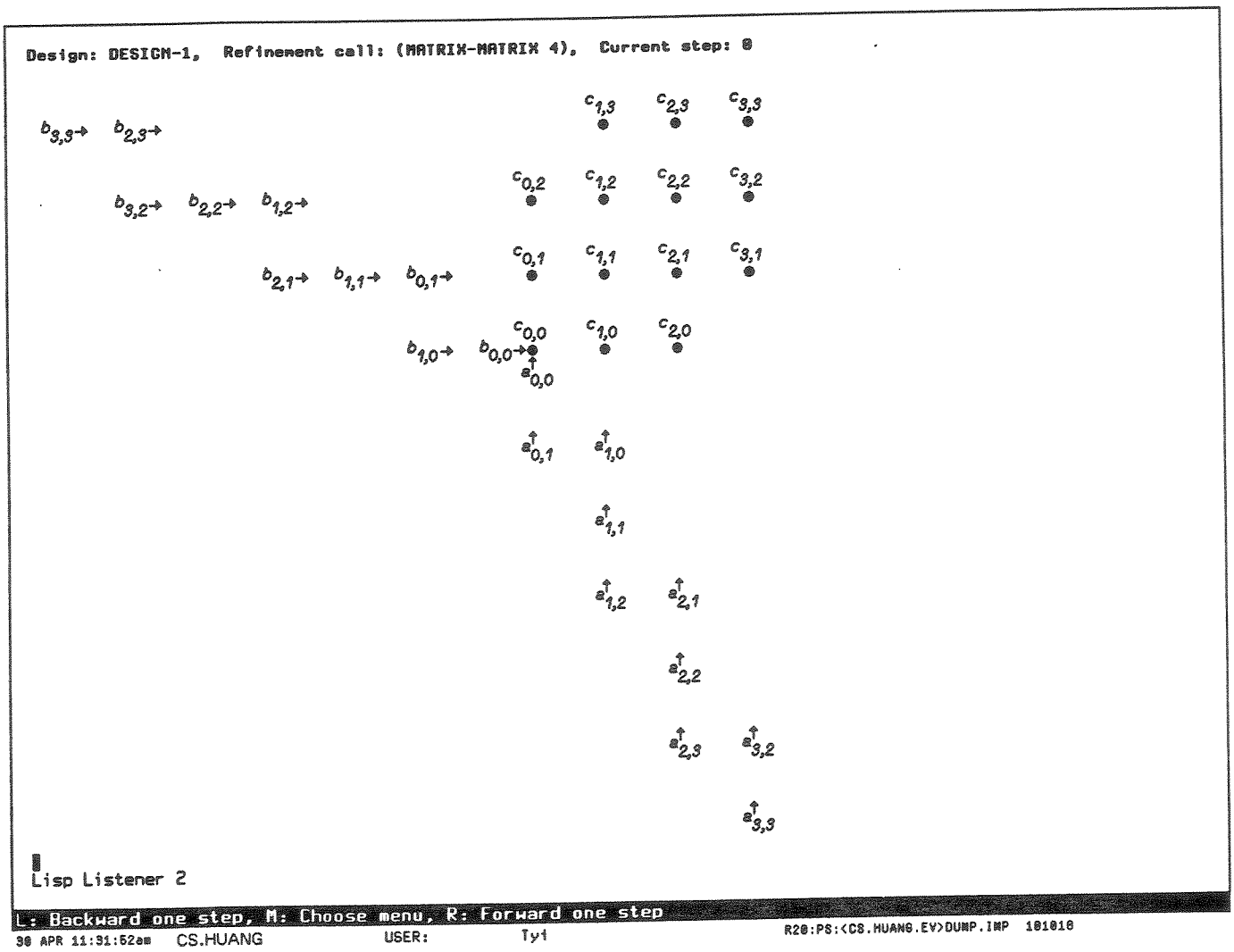


**Figure 1. The First Design**

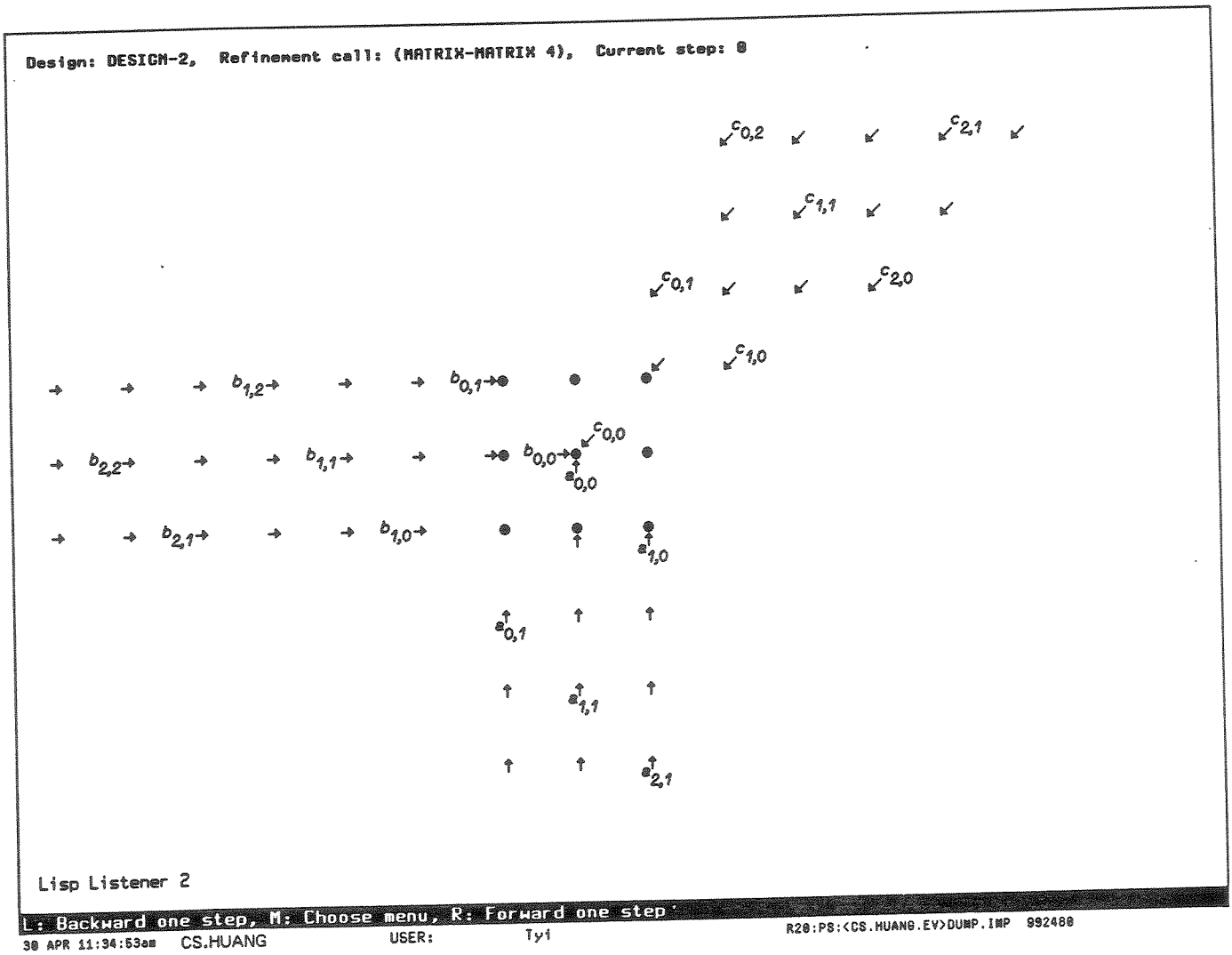


**Figure 2. The Second Design**

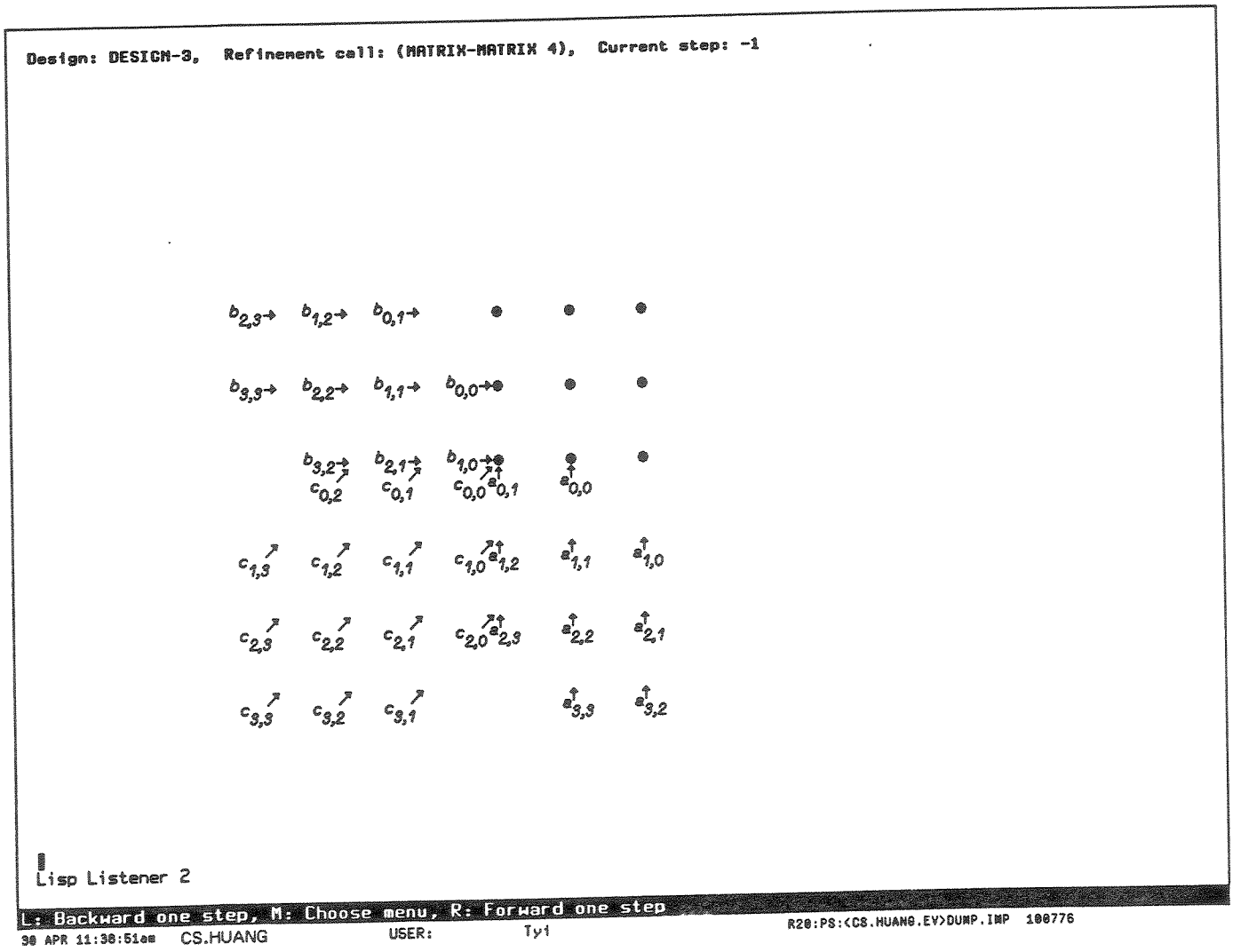


**Figure 3. The Third Design**

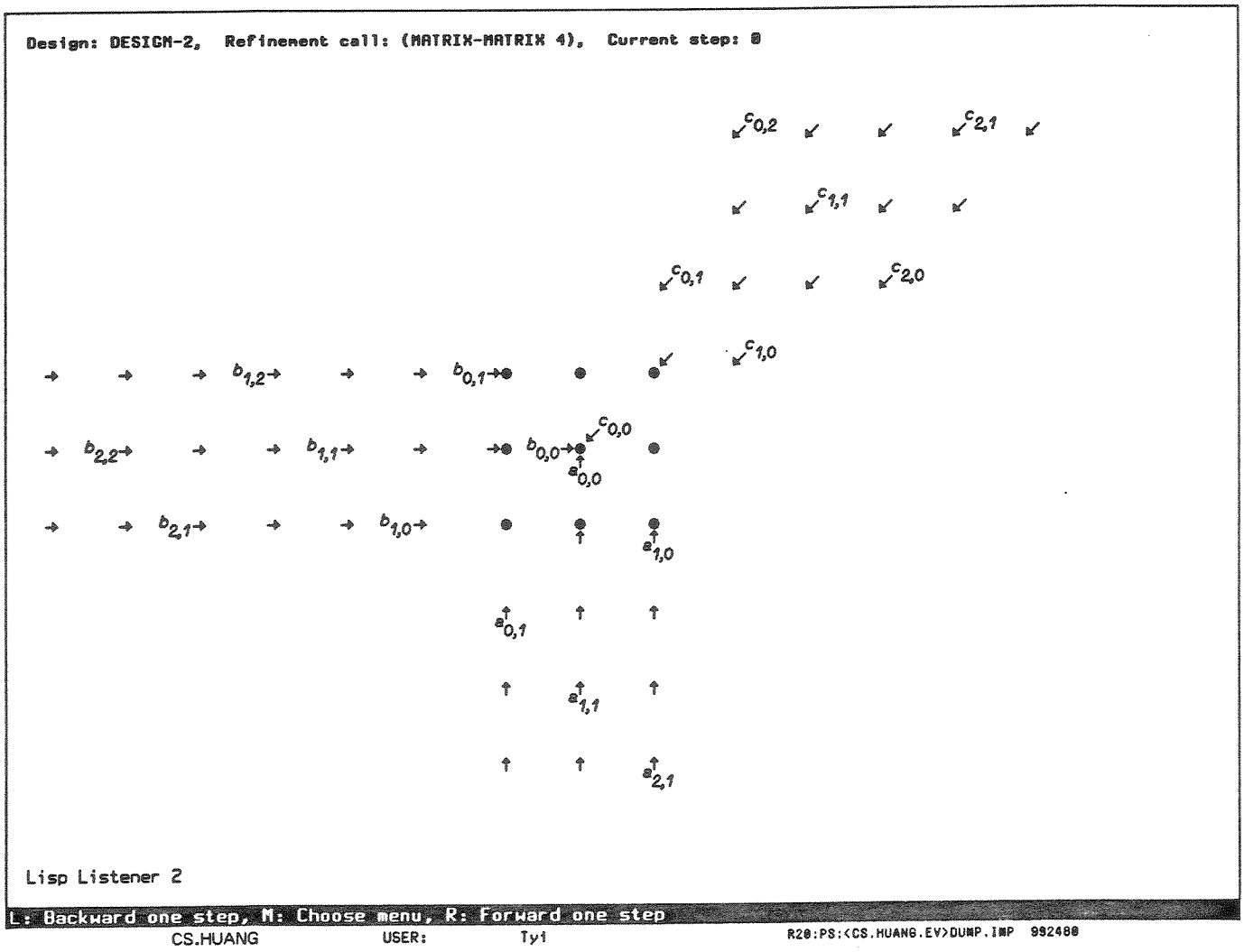


**Figure 2. The Second Design**

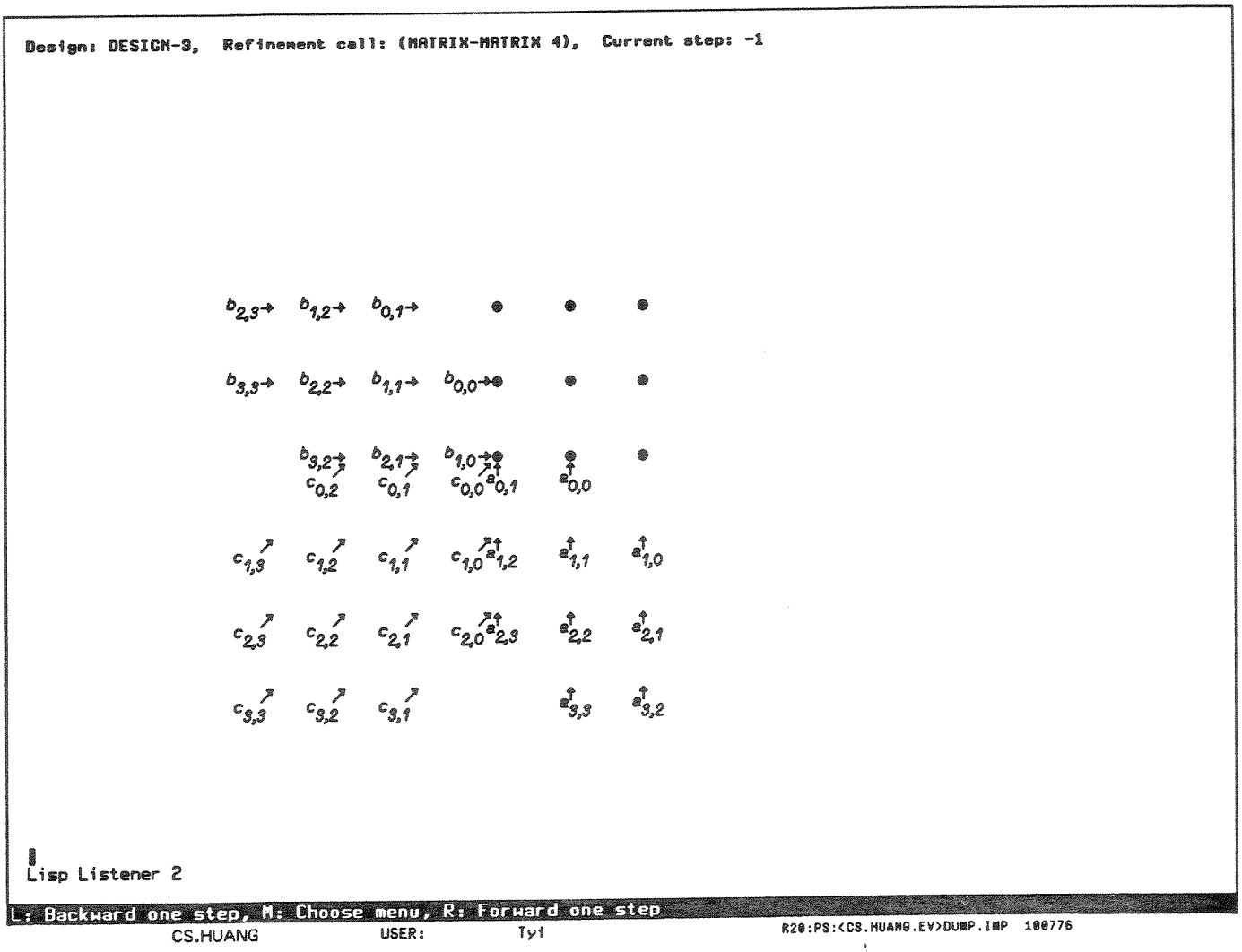


Figure 3. The Third Design