A Constructive Solution to the Juggling Problem in Systolic Array Synthesis

Alain Darte, Robert Schreiber,
B. Ramakrishna Rau, Frédéric Vivien
Compiler and Architecture Research
HP Laboratories Palo Alto
HPL-2000-30
February, 2000

systolic array synthesis, affine scheduling

We describe a new, practical, constructive method for solving the well-known conflict-free scheduling systolic for the locally sequential, globally parallel (LSGP) case of processor array synthesis. Previous solutions have an important practical disadvantage. Here we provide a closed form solution that enables the enumeration of all conflict-free schedules. The second part of the paper discusses reduction of the cost of hardware whose function is to control the flow of data, enable or disable functional units, and generate memory addresses. We present a new technique for controlling the complexity of these housekeeping functions in a processor array.

Internal Accession Date Only

A constructive solution to the juggling problem in systolic array synthesis

Alain Darte Robert Schreiber B. Ramakrishna Rau Frédéric Vivien

LIP, ENS-Lyon, Lyon, France Hewlett-Packard Company, Palo Alto, CA, USA
ICPS, Pole Api, Bvd S. Brand, Illkirch, France

Abstract

We describe a new, practical, constructive method for solving the well-known conflict-free scheduling problem for the locally sequential, globally parallel (LSGP) case of systolic array synthesis. Previous solutions have an important practical disadvantage. Here we provide a closed form solution that enables the enumeration of all conflict-free schedules. The second part of the paper discusses reduction of the cost of hardware whose function is to control the flow of data, enable or disable functional units, and generate memory addresses. We present a new technique for controlling the complexity of these housekeeping functions in a systolic array.

1. Introduction — hardware accelerators and systolic arrays

The rapidly growing consumer electronics industry demands low-cost electronics to perform image and signal processing at impressive computational rates. Many current designs employ embedded general-purpose computers assisted by application-specific hardware in order to meet the computational demands at low cost. Such systems are difficult and expensive to design. This makes automatic synthesis of application-specific hardware accelerators increasingly desirable. Ideally, a source program would be compiled into a system consisting of a general-purpose processor and one or more hardware accelerators (automatically designed and interfaced to the whole system) using the program as a behavioral specification.

The most obvious way to obtain significant performance from custom hardware is to exploit parallelism. Automatic parallel hardware synthesis can be accomplished by transforming a loop nest into a form that allows its implementation as a systolic-like array. This paper addresses two important practical problems in systolic synthesis. The first is to map each iteration of a loop nest to a processor and a time step in such a way that all processors are kept busy at all times, and none is overloaded. Previous theoretical solutions made it inconvenient to quickly find a mapping and schedule that accomplish this. We shall present some new theoretical insight into this problem that leads directly to an efficient solution. The second issue is the hardware cost of systolic implementations of loop nests. Parallel realizations of sequential algorithms come at some cost: in our case additional computation – which would lead to additional hardware in an application-specific accelerator – needed to control and coordinate the processor. We develop a new low-cost technique for control and coordination that is theoretically appealing, and we give some experimental evidence that it greatly reduces cost in comparison with expensive, standard approaches.

Section 2 explains the conflict-free systolic scheduling problem. Section 3 explains the known theoretical solution, developed by the first author, and discusses the problems with using it in a practical system. Section 4 presents our new theoretical approach and describes how it leads to an improved implementation. Section 5 introduces and solves the problem of the added cost of parallelization in systolic form. We offer some concluding remarks in Section 6.

2. Mapping and scheduling for systolic arrays

Classically, the iterations of a perfect n-deep loop nest are identified by the corresponding integer n-vector $\vec{j} = (j_1, \ldots, j_n)$ of loop indices. The iteration vector lies in some given polytope, $\mathcal{J}$, called the iteration space. An $n-1$-dimensional grid of systolic processors with rectangular topology is given, and each processor is identified by its coordinate vector. The systolic mapping problem is to find functions π and τ such that processor $\pi(\vec{j})$ commences computation of iteration $\vec{j}$ at clock cycle $\tau(\vec{j})$. We use the term *schedule* for the timing function τ and *mapping* for the processor assignment π. The time $\tau(\vec{j})$ is a start time for iteration $\vec{j}$, and all the detailed actions belonging to that iteration are scheduled relative to this start time. In classical systolic scheduling [7, 9, 10] the scheduling function τ is an integer-valued linear function of the iteration vector. The spatial mapping is defined by an $(n-1) \times n$ integer matrix Π that admits a unimodular extension, so that the processor array is $(n-1)$-dimensional.

The schedule and mapping must satisfy several valid-

ity requirements, including local interprocessor communication, conflict-free scheduling and causality (data is not used before it is computed.)

2.1. Nonlinear mappings – tiling and clustering

Full-rank, linear mapping has the significant practical problem that one has little control over the number of processors. One would like to choose the number of processors and their topology *a priori*, based on such practical considerations as the need to map to a piece of existing hardware, or to conform to a fixed allotment of chip area, board space, or power. We therefore view Π as a mapping onto an array of *virtual* processors (VPs). The virtual processors are later associated with physical processors, but this map is many-to-one.

One approach to the problem of handling arbitrarily large data sets and iteration spaces with fixed-size systolic arrays is to decompose the whole computation into a sequence of similar computations on subsets of the data [6, 2, 11]. This approach is known as *tiling* in the compiler literature, and as the locally parallel, globally sequential method in systolic synthesis. We use tiling to control the size of the iteration space in order to limit the local storage in our synthesized processors. We do not want to constrain the tile size by requiring the number of virtual processors to match the number of physical processors. To do so would usually produce tiles too small to obtain significant benefit from reuse of intermediate results in the array.

We use an alternative approach, which we call *clustering*, and which has also been called partitioning or the locally sequential, globally parallel (LSGP) technique in the systolic synthesis literature. Clustering assigns a rectangular neighborhood in the array of virtual processors to each physical processor. This amounts to choosing a rectangular cluster shape – a small $(n-1)$-dimensional rectangle – and then covering the $(n-1)$-dimensional array of virtual processors with nonoverlapping clusters. The cluster shape is chosen so that the set of clusters forms an $n-1$-dimensional grid of the same shape as the systolic processor grid. The systolic processor will have enough throughput, and the schedule of the iterations will allow enough time, so that each systolic processor will do the work of its assigned virtual processors.

Let an $n-1$-dimensional grid of systolic processors of shape $\vec{P}$ be given: processor coordinates satisfy $0 \le p_i < P_i$. The **virtual processor array** is the image of $\mathcal{J}$ under Π. Let the smallest rectangle that covers the set of virtual processors have dimensions $\vec{V}$, so that if $\vec{v} = \Pi\vec{j}$ for some $\vec{j} \in \mathcal{J}$, then $0 \le v_i < V_i$. (We must apply a shift, in general, to make the virtual processor coordinates nonnegative.) Define the shape of the **cluster**, $\vec{C} = (C_1, \ldots, C_{n-1})$, by $C_i \equiv \lceil V_i/P_i \rceil$. The processor grid of shape $\vec{P}$, whose processors each cover a cluster of shape $\vec{C}$, covers the whole virtual processor space of shape $\vec{V}$. The number of virtual processors assigned to a systolic processor is not more than $\gamma \equiv \prod_{i=1}^{n-1} C_i$.

2.2. Schedules consistent with clustered mappings

Assume that the physical processor can perform one loop iteration per cycle. We are interested in finding a linear schedule $\tau(\vec{j}) = \vec{\tau}.\vec{j}$ for the iterations. The need to compute values before they are used leads to linear inequality constraints on $\vec{\tau}$. Let $\vec{u}$ be a smallest integer null vector of Π. Thus, $\vec{u}$ connects the iteration $\vec{j}$ to the very next iteration, $\vec{j} + \vec{u}$, that is mapped to the same virtual processor. We know that the physical processor will visit each of its γ simulated virtual processors once, in some round-robin manner, before returning to $\Pi\vec{j}$ again. Because we need to allow at least γ cycles between visits,

$$|\vec{\tau}.\vec{u}| \ge \gamma\,. \tag{1}$$

The throughput inequality (1) ensures that the physical processor is not overloaded *on average*. But because we are trying to determine a schedule that is precise (all actions scheduled for a certain machine cycle) and correct, we cannot allow two iterations to begin at the same time. Thus, the problem we seek to solve here is the following **conflict-free systolic scheduling problem**: given $\vec{C}$, the mapping Π of rank $(n-1)$ which has $\vec{u}$ as its smallest integer null vector, and linear inequality constraints on $\vec{\tau}$, choose $\vec{\tau}$ satisfying these constraint and such that no two virtual processors assigned to a given physical processor are scheduled to be simultaneously active.

We say that a schedule that satisfies the no-conflict constraint for the given cluster "juggles"; imagine a juggling processor with its γ balls (virtual processors) in the air, and only one hand, capable of holding only one ball at any given time. If $\vec{\tau}$ juggles and satisfies (1) with equality,

$$|\vec{\tau}.\vec{u}| = \gamma\,, \tag{2}$$

then we say that the schedule is *tight*.

Our main result is a construction that produces *all* tight schedules for a given cluster $\vec{C}$. We have not obtained any results concerning nontight, juggling schedules, except for the obvious. If a schedule is tight for cluster shape $\vec{D} \ne \vec{C}$ and $\vec{D} \ge \vec{C}$ elementwise, then this schedule is a nontight, juggling schedule for $\vec{C}$.

3. Related work

The idea of Darte's (and initially Darte and Delosme's) solution [4, 3] is to produce a cluster shape $\vec{C}$ compatible with the given schedule vector $\vec{\tau}$. In many practical situations, however, the physical and virtual processor arrays and thus the set of possible cluster shapes is known. The task is to find a tight schedule for a known cluster shape. Using

Darte's approach, this must be done by an indirect and possibly costly trial-and-error approach, while the theorem that we later prove leads to a simple method that directly enumerates the tight schedules.

Darte's theorem and method work this way. The inverse of any unimodular matrix having first row equal to $\vec{\tau}$ has as its second through n-th columns an $n \times (n-1)$ matrix Q whose columns are a basis for the lattice of iterations scheduled for time zero. Let $A = \Pi Q$. Then A is a square integral matrix of order $(n-1)$ whose columns are the coordinates of a set of virtual processors active at time zero. Darte called A the "activity matrix". Let H_a be the Hermite normal form of A [1]: $A = H_a Q_a$ with Q_a unimodular. The columns of H_a generate the lattice of virtual processors active at time zero, and the diagonal elements of H_a are a cluster shape for which $\vec{\tau}$ is a tight schedule. This remains true for the Hermite normal form of any permutation of the rows of A. Furthermore, this is a necessary and sufficient condition for tight schedules (the necessity being the difficult part). Thus, given $\vec{\tau}$, Darte's method produces all cluster shapes $\vec{C}$ of size $|\vec{\tau}.\vec{u}|$ that juggle with $\vec{\tau}$. If the schedule is specified and an appropriate cluster shape is desired, then this method gives all possible choices.

Megson and Chen [8] attempt to guarantee a tight schedule for some given cluster shape $\vec{C}$ by working with the Hermite form of $A = \Pi Q$ directly. Relying on the fact that the Hermite form of a triangular matrix X has the same diagonal as X (up to sign), they choose A to be triangular with the elements of $\vec{C}$ on the diagonal, and they assume that Π is known. They then look at the general solution Q to the underconstrained linear system $\Pi Q = A$ and from the solutions they infer $\vec{\tau}$. They try to choose the unconstrained components of Q and the off-diagonal elements of A to obtain an acceptable schedule (via the inverse of a unimodular extension of Q). Megson-Chen produces tight schedules from the specified cluster shape, but does not have real advantages compared to Darte's: one will still need to search for desirable tight schedules indirectly, by manipulating parameters other than the elements of $\vec{\tau}$.

The clear advantage of the method we propose here is that it works directly with $\vec{\tau}$. Thus, one has far more control over the resulting schedule, and may quickly determine a tight schedule that meets other requirements.

4. Construction of tight schedules

We now present a way to construct the set of all tight schedules for a given cluster $\vec{C}$. First, we assume that Π consists of the first $(n-1)$ rows of the identity matrix, so that $\Pi u = \Pi e_n = \Pi(0, \ldots, 0, 1)^t = 0$. We write $x \wedge z$ for the greatest common divisor of x and z. If $x \wedge z = 1$ we say that x and z are relatively prime. Then, we have the following result:

[1] For more about Hermite forms and lattice theory, we refer to [12].

Theorem 1 *Let $\vec{C}$ be a given cluster shape. If Π consists of $(n-1)$ rows of the identity, then $\vec{\tau}$ is a tight schedule if and only if, up to a permutation of the elements of $\vec{C}$ and the same permutation of the first $n-1$ elements of $\vec{\tau}$,.*

$$\vec{\tau} = (k_1,\ k_2 C_1,\ k_3 C_1 C_2,\ \ldots,\ k_n C_1 \cdots C_{n-1}) \qquad (3)$$

where k_i and C_i are relatively prime and $k_n = \pm 1$,

Proof. The **if** part is the easy part. For the **only if** part, we use the same same group theoretic result that was originally employed in [3]. The complete proof is available in the extended version of this paper [5]. ■

Actually, the restriction on Π is unnecessary. Let S be the inverse of a unimodular extension of Π. The last column of S is the projection vector $\vec{u}$. The remaining columns are the vectors that describe the virtual processor array. In particular, the first $(n-1)$ rows of S^{-1} are the projection matrix Π. The transformation matrix M is the matrix whose first row is $\vec{\tau}$ and whose last $(n-1)$ rows are Π:

$$M \equiv \begin{pmatrix} \vec{\tau} \\ \Pi \end{pmatrix};\ \text{thus} \begin{pmatrix} t \\ \vec{v} \end{pmatrix} = M\vec{j} \qquad (4)$$

is the mapping from iteration $\vec{j}$ to time t and virtual processor $\vec{v}$. We now change basis in the iteration space: $\vec{j'} = S^{-1}\vec{j}$ are the coordinates of the iteration with respect to the basis consisting of the columns of S. In this basis, the transformation becomes

$$\begin{pmatrix} t \\ \vec{v} \end{pmatrix} = MS\vec{j'} = \begin{pmatrix} \vec{\tau}.S \\ \Pi S \end{pmatrix} \vec{j'} = \begin{pmatrix} \vec{\tau}.S \\ I_{n-1} \quad 0 \end{pmatrix} \vec{j'} \qquad (5)$$

Clearly, $\vec{\tau}$ is a tight schedule with cluster shape $\vec{C}$ and mapping Π if and only it $\vec{\tau}.S$ is a tight schedule for $\vec{C}$ with the mapping $(I_{n-1} \quad 0)$. Hence, the generalized condition (3) applied to $\vec{\tau}.S$ is a necessary and sufficient condition for a tight schedule. The formula does not specify the components of $\vec{\tau}$ but rather the components of $\vec{\tau}.S$, and $\vec{\tau}$ is recovered through the integer matrix S^{-1}.

We have used these results to generate schedules in a practical system for synthesis of systolic arrays; the details of the implementation will appear in a later report.

Example

Let $n = 3$; let $\vec{C} = (4, 5)$. Assume that e_3 is the smallest integer null vector of the space mapping. From (3), either $\vec{\tau} = (k_1, 4k_2, \pm 20)$ or $\vec{\tau} = (5k_1, k_2, \pm 20)$ with $k_i \wedge C_i = 1$, for $i = 1, 2$. For example, $\vec{\tau} = (7, 4, 20)$ is a tight schedule (with $k_1 = 7$, $k_2 = 1$, $k_3 = 1$) that corresponds to the *activity tableau* below. The number in each box denotes the residue modulo 20 of the times at which the virtual processor that lives there is active. For a juggling schedule, these are all different. (The c_1 axis is the vertical axis in the diagram.)

1	5	9	13	17
14	18	2	6	10
7	11	15	19	3
0	4	8	12	16

For later use, we need to record some important properties of the mapping matrix and its Hermite normal form. It is fairly straightforward to show, as a consequence of Darte's theorem on schedules and the cluster shapes for which they are tight, that the Hermite normal form of the mapping matrix M (see (4) above) is (up to a row permutation if Π) a lower triangular matrix whose diagonal is $(1, C_1, \ldots, C_{n-1})$.

5. Reducing the cost of control

After transformation into LSGP form, the systolic loop body serves as a specification of the systolic processor. In its final form, the systolic nest has outer loops over tiles and an inner nest consisting of a sequential loop over time and a parallel nest over processors. The transformed parallel loop body contains generated code that we call *housekeeping* code whose cost we consider here. Housekeeping code has several forms and functions:

Cluster coordinates. For each time t on the given processor $\vec{p}$, one may need to compute the local position $\vec{c}$ within the cluster: $0 \leq c_k < C_k$. These give the position of the currently active VP.

Virtual processor coordinates. One may also need the global virtual processor coordinate $v_k = c_k + p_k C_k$.

Iteration space coordinates. Since the iteration space coordinates $\vec{j}$ may appear in the loop body, these will sometimes need to be computed. The usual technique is to use the relation $\vec{j} = M^{-1} \begin{pmatrix} t \\ \vec{v} \end{pmatrix}$. (The result is guaranteed to be integer when $\vec{v}$ is active at time t, even though M^{-1} is a matrix with rational elements.)

Memory addresses. When data is "live-in" to the loop nest, or is "live-out", it is read from or stored into global memory. The memory address, which is the location of an array element whose indices are affine functions of the iteration space coordinates $\vec{j}$, must be computed.

Predicates. In a naive approach, many comparisons are used to compute predicates. These comprise: **Cluster-edge predicates** (comparison of the cluster coordinates $\vec{c}$ and the cluster shape $\vec{C}$); **Tile-edge predicates** (tests of the global iteration coordinates against the limits of the current tile); and **Iteration-space predicates** (that test the global iteration coordinates against the limits of the iteration space).

Our first experiments with systolic processor synthesis revealed that these housekeeping computations were so costly that the resulting processor was grossly inefficient. There were two primary reasons. One was the large number of comparisons for predicates. We then observed that almost all of these repeat on a given processor with period γ, and that they can therefore be obtained from a γ-bit circular buffer. We use this technique in our current implementations with good results. The second and more important reason for this inefficiency was the method used to compute cluster coordinates. Our original approach took the rather obvious viewpoint that each processor, at each time, computes the cluster coordinates of its active virtual processor, which is a function of the processor coordinates $\vec{p}$ and the time t. We generated the code by first applying standard techniques [1] for code generation after a nonunimodular loop transformation (using Hermite form) to generate a loop nest that scans the active virtual processors for each time. We then inferred the local processor coordinates $\vec{c}$ from the lower bounds for the virtual processor loops, which are functions of $\vec{p}$ and t, by taking their residues modulo $\vec{C}$.

This technique is memory efficient, but computationally expensive. It is a form of integer triangular system solution. Let T be a unimodular matrix such that $MT = H_m$. Then

$$\begin{pmatrix} t \\ \vec{v} \end{pmatrix} = M\vec{j} = H_m T^{-1} \vec{j} = H_m \vec{i}$$

where H_m is the Hermite Normal Form of M, and $\vec{i}$ is integer. Furthermore, we know (see end of Section 4) that the $(1,1)$ element of H_m is unity and that the rest of the diagonal of H_m consists of the elements of $\vec{C}$. The requirement that the triangular system above has an integer solution $\vec{i}$ in fact completely determines the residues modulo $(1, \vec{C}) \equiv \mathrm{diag}(H_m)$ of $\vec{v}$, which are the cluster coordinates of the VP active at time t on processor $\vec{p} = \vec{v} \div \vec{C}$. This in turn determines $\vec{v}$. Solving this system, inferring the cluster coordinates in the process, has $O(n^2)$ complexity. By a slightly different use of the special form of a tight schedule (see [5]), we reduced this cost to $O(n)$. From the viewpoint of generating hardware, however, the method still has a few disadvantages since it involves a quotient and a remainder for each dimension, and it does nothing to assist with addresses, iteration space coordinates, or predicates.

We now discuss methods for making a major reduction in the cost of housekeeping computations; tests will show that once these techniques are employed the cost of the resulting processor is close to the minimum possible. We shall examine two alternatives. Both of them trade space for time. Both use temporal recurrences to compute coordinates.

First note that with a tight schedule, the cluster and virtual processor coordinates, and all but one of the global iteration space coordinates, are periodic with period γ, as are all predicates defined by comparing these periodic functions to one another and to constants. The remaining iteration space coordinate satisfies

$$j_n(t) = j_n(t - \gamma) + 1 \,.$$

(These assertions apply when Π consist of rows of the identity; things are only slightly more complicated in general.)

Any quantity that depends linearly on j_n can be updated with a single add. Quantities (such as predicates) that depend only on the other coordinates are similarly periodic. This is the cheapest approach possible in terms of computation; its only disadvantage is in storage. We need to store the last γ values of any coordinate or related quantity that we wish to infer by this γ-order recurrence. When γ is fairly large (say more than ten or so) these costs become significant.

The alternative technique allows us to *update* the cluster coordinates $\vec{c}(t,\vec{p})$ from their values at an *arbitrary* previous cycle but on the same processor: $\vec{c}(t,\vec{p}) = R(\vec{c}(t-\delta t,\vec{p}))$ (here R stands for the recurrence map that we now explain.) We may choose any time lag δt. (Provided that δt is not so small that the recurrence becomes a tight dataflow cycle inconsistent with the schedule that we have already chosen.) The form of R is quite straightforward. Using a binary decision tree of depth $(n-1)$, we find at the leaves of the tree the increments $\vec{c}(t,\vec{p}) - \vec{c}(t-\delta t,\vec{p})$. The tests at the nodes are comparisons of scalar elements of $\vec{c}(t-\delta t,\vec{p})$ with constants that depend only on $\vec{C}$ and the schedule $\vec{\tau}$. They are thus known at compile time and can be hard coded into the processor hardware.

These cluster coordinates are the key. The global virtual processor coordinates $\vec{v}$, the global iteration space coordinates $\vec{j}$, and the memory addresses are all linear functions of them. If we know the change in $\vec{c}$ then we also know the changes in all of these derived values, and these changes appear as explicit constants in the code. Only one addition is needed to compute each such value. We have thus reduced the problem of cost reduction to that of the update of the cluster coordinates. The next section explains how we can automatically generate this decision tree.

5.1. A general updating scheme

We first present the theory, then we will illustrate the technique with an example. Note that the activity times on some arbitrarily chosen processor $\vec{p}$ are shifted by a constant, $\vec{\tau}.(p_1C_1,\ldots,p_{n-1}C_{n-1})$, compared with the times on processor zero. This implies that the same decision tree may be used on all processors. We therefore assume that the processor coordinates p_i are all equal to zero.

Recall that the columns of T are a unimodular basis and that $MT = H_m$. The first row of MT gives the time difference along each column of T, and the last rows are the coordinates of the columns of T in the virtual processor array. Since the first row of MT is $(1,0,\ldots,0)$, the first column $\vec{w}$ of T connects an isochrone (a hyperplane of iterations scheduled for the same time) to the next isochrone, and the remaining columns $\vec{t}_2,\ldots,\vec{t}_n$ lie in an isochrone. Given the iteration $\vec{j}$, what we want to find is a vector $\vec{k}$ such that $M(\vec{j}+\vec{k}) = (t+\delta t,\vec{z})^t$ where $\vec{z}$ is "in the box" $0 \le \vec{z} < \vec{C}$, which means that it corresponds to an iteration scheduled in the same physical processor. Clearly $\vec{k}$ is the sum of $\delta t \times \vec{w}$ and a linear combination of $\vec{t}_2,\ldots,\vec{t}_n$.

We know already that $\vec{k}$ exists and is unique since the schedule is tight. This can also be seen from the fact that H_m has the C_i's on the diagonal: writing $\vec{k} = T\vec{\lambda}$, we end up with a triangular system that can be easily solved thanks to the structure of H_m. We can add a suitable linear combination of $\vec{t}_2$, ..., $\vec{t}_n$ to $\delta t \times \vec{w}$ so that the $(n-1)$ last components of $M(\delta t \times \vec{w})$ are in the box. This vector (let us denote it by $\vec{\delta}[0,\ldots,0]$) will be one of the candidates in the decision tree. Now, either the second component of $M(\vec{j}+\vec{\delta}[0,\ldots,0]$ is still strictly less than C_1 and we are in the first case (first branch of the tree), or this component is strictly less than $2C_1$ and we simply subtract $\vec{t}_2$ to go back in the box. In the latter case, $\vec{\delta}[1,0,\ldots,0] = \vec{\delta}[0,\ldots,0]-\vec{t}_2$ (plus possibly a linear combination of $\vec{t}_3$, ..., $\vec{t}_n$ so that the $(n-2)$ last components of $M\vec{\delta}[1,0,\ldots,0]$ are in the box) is one of the candidates in the decision tree. Continuing in this way, we end up with at most $2^{(n-1)}$ vectors (at most two cases for each dimension, and only one when the corresponding component of the move vector is zero).

The notation in brackets for the vectors $\vec{\delta}$ specifies if the move is nonnegative (0) or negative (1): for example, $\vec{\delta}[0,1,1]$ corresponds to the case where we move forwards in the first dimension and backwards in the two other dimensions.

Example (continued)

We take $\delta t = 1$ so that the reader can compare the result with the activity tableau on page 4. The Hermite form of the mapping ($MT = H_m$) is

$$\begin{pmatrix} 7 & 4 & 20 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}\begin{pmatrix} 3 & 4 & 0 \\ 0 & 3 & 5 \\ -1 & -2 & -1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 3 & 4 & 0 \\ 0 & 3 & 5 \end{pmatrix}$$

From the matrix H_m, we read that we may have to move along $\vec{w} = (3,0)$ in the virtual space (which corresponds, in the original space, to the first column of T, the vector $\vec{\delta}[0,0] = (3,0,-1)$). If $c_1 + 3 \ge 4$, then we subtract the second column of H_m, i.e. $(4,3)$, we find the move vector $(-1,-3)$, and we add the third column to go back in the box: $(3,0) - (4,3) + (0,5) = (-1,2)$. This corresponds in the original space to $\vec{\delta}[1,0] = (3,0,-1) - (4,3,-2) + (0,5,-1) = (-1,2,0)$. Then, for both vectors, we check the last component: in the first case, no other vector is required since the second component of $(3,0)$ is 0. In the second case, we may have to subtract $(0,5)$: the last candidate is thus $(-1,2) - (0,5) = (-1,-3)$ and $\vec{\delta}[1,1] = (-1,-3,1)$. With this technique, we get the decision tree below. □

The technique works the same way for arbitrary δt. You begin with the change $\delta t \times \vec{w}$, and "correct" it as necessary with the remaining columns of H_m in order to find the tree

```
if (c(1) + 1 < C(1)) {
   c(1) += 1;
   if (c(2) + 4 < C(2)) {
      c(2) += 4;
   } else {
      c(2) += (-1);
   }
} else {
   c(1) += (-3);
   if (c(2) + 1 < C(2)) {
      c(2) += 1;
   } else {
      c(2) += (-4);
   }
}
```

of changes. This gives the necessary tests in the decision tree directly, as well as the corresponding changes in the cluster coordinates and the original loop indices.

5.2. Measuring the cost of housekeeping code

We show here the results of systolic loop transformation, with our efficient recurrence scheme for cluster coordinates and the parameters that depend on them linearly. We use three loop nests as test cases. The first, from an application in digital photography, is a nest of depth six, in which the loop body contains only a simple multiply-accumulate statement. The second, from a printing application, is a much more complicated loop body. The third is matrix multiplication. We transformed the loop nests using the mechanisms described in this paper. In [5], we present the original and the transformed loop nests for matrix multiplication as an illustrative example.

	Photography			Printing		Matrix Mult.	
operation	orig.	transf.	naive	orig.	transf.	orig.	tranf.
$+$	5	7	52	22	31	5	6
$\times$	1	1	34	1	1	1	1
$\div$	0	0	4	0	0	0	0
compare	1	6	35	18	18	1	3

In the table, we show the number of inner-loop integer operations in the original and the fully transformed loop nests. In addition, for the photography application, we show the same statistics for the code as transformed by the naive methods that we have earlier described. The counts were obtained by examining the code. It is clear from this data that the housekeeping due to parallelization has added to the computational cost of the loop body. The number of operations increased by seven in the simple photography loop, nine in the more complicated printing loop, and only three in the matrix product loop. The ratio of the added operation count to the original operation count is $3:7$ for the matrix multiply loop, $1:1$ for the photography loop, and $9:41$ for the more complicated loop. The photography loop has a deeper decision tree than matrix multiply because of the deep loop nest; this accounts for the different costs. Evidently, with optimization, housekeeping costs are not trivial, but they are manageable. The naive method, however, produces intolerably costly code for calculation of coordinates, predicates, and memory addresses.

6. Conclusion

The first part of this paper provides a simple characterization of all tight LSGP schedules, solving a longstanding problem in systolic array synthesis. The characterization allows a synthesis system to directly enumerate all the tight schedules in any desired region of the space of schedules, which can be very useful in generating tight schedules in, *e.g.*, a polyhedron defined by recurrence constraints.

We also demonstrated a new technique for generating efficient parallel code that takes full advantage of our characterization of tight schedules. Our experiments have shown that the specialized processors we generate are highly efficient in their gate count and chip area. We conclude that the technique explained in the second section of this paper can control the added computational cost due to parallelization to the point where it is not overly burdensome, especially for loop nests that have more than a handful of computations in the innermost loop.

References

[1] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *3rd ACM PPoPP Symp.*, pp. 39–50, Apr. 1991.

[2] P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate tiling? *INTEGRATION, the VLSI Journal*, 17:33–51, 1994.

[3] A. Darte. Regular partitioning for synthesizing fixed-size systolic arrays. *INTEGRATION, The VLSI Jounal*, 12:293–304, 1991.

[4] A. Darte and J.-M. Delosme. Partitioning for array processors. Tech. Rep. 90-23, LIP, ENS-Lyon, 1990.

[5] A. Darte, R. Schreiber, B. R. Rau, and F. Vivien. A constructive solution to the juggling problem in systolic array synthesis. Tech. Rep. RR1999-15, LIP, ENS-Lyon, 1999.

[6] F. Irigoin and R. Triolet. Supernode partitioning. In *15th ACM POPL Symp.*, pp. 319–329, 1988.

[7] H. T. Kung and C. E. Leiserson. Systolic arrays (for VLSI). In *Sparse Matrix Proc.*, pp. 256–282. SIAM, 1978.

[8] G. M. Megson and X. Chen. A synthesis method of LSGP partitioning for given-shape regular arrays. In *9th IPPS*, pp. 234–238. IEEE, 1995.

[9] D. Moldovan. On the analysis and synthesis of VLSI systolic arrays. *IEEE Trans. on Computers*, 31:1121–1126, 1982.

[10] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *11th ISCA*, pp. 208–214. IEEE, 1984.

[11] J. Ramanujam and P. Sadayappan. Tiling of iteration spaces for multicomputers. In *ICPP*, pp. 179–186. IEEE, 1990.

[12] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, New York, 1986.