Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 8512

CENTRE NATIONAL
DE LA RECHERCHE
SCIENTIFIQUE

SPI

A constructive solution to the juggling problem in systolic array synthesis

Alain Darte
Robert Schreiber
B. Ramakrishna Rau
Frédéric Vivien

Février 1999

Research Report N° RR1999-15

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

INRIA

A constructive solution to the juggling problem in systolic array synthesis

Alain Darte
Robert Schreiber
B. Ramakrishna Rau
Frédéric Vivien

Février 1999

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. Earlier attempts to solve this problem provided a solution with an important practical disadvantage, which we discuss. Here, we provide a closed form solution that enables the enumeration of all valid schedules. The second part of the paper discusses another practical issue: reducing 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. Both of these techniques have been incorporated into a software system for the automatic synthesis of hardware accelerators developed by HP Labs.

Keywords: Hardware accelerators, systolic arrays, automatic synthesis, LSGP partitioning

Résumé

Nous présentons une méthode constructive permettant de résoudre le problème bien connu de l'ordonnancement sans conflits du partitionnement localement séquentiel, globalement séquentiel (LSGP) de la synthèse de réseaux systoliques. Les techniques proposées antérieurement conduisent à des solutions comportant quelques inconvénients pratiques majeurs que nous discutons. Ici, nous donnons une expression analytique des solutions qui nous permet d'énumérer tous les ordonnancements valides. La seconde partie du rapport discute d'un autre aspect pratique: comment réduire le coût du matériel dont le rôle est de contrôler le flot des données, d'activer ou de désactiver les unités fonctionnelles et de générer les adresses mémoire. Nous montrons comment maîtriser la complexité de ces calculs de maintenance du circuit. Ces deux techniques ont été intégrées dans un logiciel de synthèse automatique d'accélérateurs matériels, développé dans les laboratoires de Hewlett-Packard.

Mots-clés: Accélérateurs matériels, réseaux systoliques, synthèse automatique, partitionnement LSGP

A constructive solution to the juggling problem in systolic array synthesis

Alain Darte * Robert Schreiber † B. Ramakrishna Rau † Frédéric Vivien ‡

February 1999

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. A loop nest and a linear mapping to virtual processors is given, as is a clustering of rectangular arrangements of virtual processors into physical processors. A solution to the scheduling problem is a linear map of iteration indices to time that satisfies linear inequality constraints determined by data dependences. The schedule is conflict-free if no two iterations are scheduled simultaneously on the same processor. We say that such a schedule *juggles*. It is *tight* if it juggles and, in the steady state, all processors are busy every cycle.

Earlier attempts to solve this problem by Darte and Delosme [1, 2] provided a solution with an important practical disadvantage, which we discuss below. Megson and Chen [3] later used Darte's analytic technique to provide a partial solution to the problem. Here, we provide a closed form solution that enables the enumeration of all tight schedules.

The second part of the paper discusses another practical issue: reducing 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.

Both of these techniques have been incorporated into a software system for the automatic synthesis of hardware accelerators developed by HP Labs.

1 Introduction — hardware accelerators and systolic arrays

The continuing rapid growth of the consumer electronics industry (for example HDTV, set-top boxes, digital photography, mobile communication, high resolution scanners, copiers, and printers, toys and games) demands low-cost and low-power digital electronics that perform image and signal processing and other data handling tasks at impressive computational rates. Many current designs employ embedded general-purpose computers that must be assisted by application-specific hardware, which may or may not be programmable, in order to meet the computational demands at low enough cost. Such systems are difficult and expensive to design. Automatic synthesis of application-specific hardware accelerators is therefore an increasingly important technology. An ideal situation would be one in which a source program can be automatically compiled onto a system consisting of a general-purpose processor and one or more hardware accelerators that have been 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. A well-understood approach to automatic parallel hardware synthesis is the automatic transformation of a loop nest into a form that enables its implementation as a systolic array [4].

This paper addresses two important practical issues in systolic array synthesis. The first concerns the problem of the combined scheduling of computation and its mapping to a prespecified array of systolic

*LIP — École normale supérieure de Lyon, Lyon, France
†Hewlett-Packard Company, Palo Alto, CA, USA
‡ICPS, Pole Api, Bvd S. Brand, Illkirch, France

processors. The goal 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 to this problem 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. All parallel realizations of sequential algorithms come at some cost; in our case, the cost is additional computation – which would lead to additional hardware in an application-specific accelerator – that is needed to control and coordinate the systolic processors. We develop a new low-cost technique for control and coordination that is theoretically appealing, and we give some experimental evidence that it significantly reduces cost in comparison with standard approaches, which can be prohibitively expensive.

The following section (Section 2) gives an overview of systolic array synthesis and presents the conflict-free scheduling problem that is the subject of the first part of this paper. Section 3 explains the existing theoretical solution to this problem, developed by the first author of this paper, 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 the issue of the added cost of parallelization in systolic form, and Section 6 gives our new approach, with experimental evidence that it is successful. We offer some concluding remarks in Section 7.

2 Mapping and scheduling in systolic computation

We assume a classical model of systolic computation, in which the iterations of a perfect n-deep loop nest, for example

```
for (j1 = 0; j1 < N; j1++) {
  ...
    for (jn = 0; jn < N; jn++) {
      a = a + 2;
    }
}
```

are identified by the corresponding integer n-vector $\vec{j} = (j_1, \ldots, j_n)$ of loop indices. (A perfect nest has no code, other than a single nested loop, in any except the innermost loop.) An m-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 define a start time $\tau(\vec{j})$ and a processor $\pi(\vec{j})$ 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. In any detailed model of systolic synthesis, and in all real implementations, the operations belonging to iteration $\vec{j}$ are not all assumed to occur at time $\tau(\vec{j})$. This time, rather, is a start time for iteration $\vec{j}$, and all the detailed actions belonging to that iteration are scheduled relative to this start time.

The scheduling and mapping must satisfy several validity requirements. Among these are the need to ensure that interprocessor communication occurs only between neighboring processors, conflict-free scheduling (no processor should be overloaded at any time), and causality (the data required by an operation must be available to the processor to which it is mapped at the time for which it is scheduled.).

In classical systolic scheduling, as introduced by Kung and Leiserson [5] and elaborated by Moldovan [6] and Quinton [7], the scheduling function τ is an integer-valued linear function of the iteration vector, defined by its integer coefficient vector. The spatial mapping is defined by an $(n-1) \times n$ integer matrix Π of full rank, so that the processor array is $(n-1)$-dimensional. (In this paper, all matrices and vectors are integer unless otherwise stated. For technical reasons, we require a bit more of Π, namely that it can be extended (by adding one more row) to a unimodular matrix. This prohibits a mapping matrix such as $\begin{pmatrix} 2 & 4 \end{pmatrix}$ that maps to processors whose coordinates are always multiples of some integer larger than one.)

2.1 Nonlinear mappings – tiling and clustering

As a practical matter, full-rank linear mapping has the significant problem that one has little control over the number of processors; in practice, 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. For this reason, the linear map Π is viewed 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 [8, 9, 10, 11, 12, 13]. This approach is known as *tiling* in the compiler literature, and as the locally parallel, globally sequential (LPGS) method in systolic synthesis. We view these techniques as being very important for reducing the size of the problem, primarily as a way to reduce the local storage in synthesized systolic processors. For this reason, 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. The idea is to tile the processor space rather than the iteration space.

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 m-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.

Formally, the clustering is defined as follows. An m-dimensional grid of systolic processors is also assumed to be given, with $m \leq n-1$. Let the **iteration space**, the set of values allowed for $\vec{j}$, be $\mathcal{J}$; we assume that $\mathcal{J}$ consists of the integral points in a polytope. Let the shape of the **systolic processor array** be the m-vector $\vec{P}$, so that the systolic processor coordinates satisfy $0 \leq p_i < P_i,\ \ i = 1, \dots, m$. 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 $0 \leq v_i < V_i,\ \ i = 1, \dots, (n-1)$. (We must apply a shift, in general, to make the virtual processor coordinates nonnegative.) The idea of clustering is to first extend $\vec{P}$ into an $(n-1)$ element vector, if necessary, by adding ones in the first $n-1-m$ positions, and then to match up the elements of $\vec{P}$ with those of $\vec{V}$. There are clearly $\left(\begin{array}{c} n-1 \\ m \end{array}\right)$ possible matchings, and we will not consider the issue of how to choose a good one here; we just assume that the job has been somehow done and the elements of $\vec{P}$ have been permuted accordingly. (A later paper will consider this and other implementation questions.) We can then define the shape of the **cluster**, $\vec{C} = (C_1, \dots, C_{n-1})$, by requiring that

$$C_i = \left\lceil \frac{V_i}{P_i} \right\rceil .$$

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. (Alternatively, the processor may be software pipelined, and for some given initiation interval λ it is able to start a new iteration every λ cycles. We give more detail concerning the schedule later.) We constrain the schedule so that on each physical processor, exactly one of its assigned virtual processors is active at any given time – *i.e.* there are no conflicts for the shared physical processor. Our point of view here is that the physical processor array is given, the mapping Π is given, and a clustering (the shape of the neighborhood rectangle) is given and is compatible with the physical processor array, the mapping, and the iteration space. We will show how to construct all possible linear schedules that satisfy this no-conflict constraint. We do not address here the important questions of how these three parameters (mapping, processor topology, clustering) should be chosen — we intend to treat these and other topics in another paper. In our theory and our implementation, however, those parameters are chosen first, and the schedule is constrained by the choice. (In fact, our system tries a number of possibilities, generates a tight schedule for each of them, and evaluates the resulting systolic implementations. One reason for concern for the efficiency of the scheduling task is that it is invoked for each combination of processor grid and mapping that is explored.)

We now need to give some more detail concerning software pipelined schedules. We assume that the schedule of operations is affine. Let $O(\vec{j})$ represent the occurrence of the operation O at iteration $\vec{j}$ of the loop nest. We schedule this operation at time

$$\tau(O,\vec{j}) = \vec{\tau}.\vec{j} + \rho_O .$$

Thus, each operation has its own scalar time offset (ρ) relative to the start of the iteration. These offsets are necessary in order to use a fast processor clock, so that the cycle time is not dependent on the complexity of the loop body. For a one-dimensional loop nest, the scheduling problem is referred to as the software pipelining problem, and the 1-vector $\vec{\tau}$ is called the initiation interval. Systolic scheduling is a multidimensional generalization.

The need to compute values before they are used leads to linear inequality constraints on $\vec{\tau}$. Consider for example the loop

```
for (j = 0; j < 100; j++) {
  a[j] = a[j-1] + a[j-2];
  b[j] = a[j-1] * 3.1415926;
}
```

and assume that the latency of the add unit is 3 cycles. The two inputs to the add operation are (in most cases) the results of the add at earlier iterations. The add cannot start until these values have been produced. For this reason, we cannot start the add at iteration j, scheduled for time $\tau.j + \rho_{\mathrm{add}}$ until the latter of $\tau.(j-1) + \rho_{\mathrm{add}} + 3$ and $\tau.(j-2) + \rho_{\mathrm{add}} + 3$. (For reasons of notational cleanliness, we have dropped the subscript 1 from both j and τ in this one-dimensional example.) Thus, we have two linear inequality constraints, namely $\tau \geq 3$ and $2\tau \geq 3$. Obviously, the second is implied by the first. Note also that the shift ρ_{add} does not appear. This will always be the case – it is well known (Bellman and Ford) that the causality constraints on $\vec{\tau}$ are independent of the offsets ρ: the offsets cancel in all the relevant inequalities. The same theory shows that the multiplication in this example does not constrain $\vec{\tau}$ at all, because it is not part of a cycle in the data flow. On the contrary, once an initiation interval τ consistent with the constraint due to the cycle is found, the offsets are then constrained by $\rho_{\mathrm{mult}} \geq (-\tau) + \rho_{\mathrm{add}} + 3$, the terms involving j having canceled.

Thus, from now on, we are interested only in finding the linear schedule $\tau(\vec{j}) = \vec{\tau}.\vec{j}$ for the iterations, not the affine schedule for the operations. We will say that a virtual processor $\vec{v}$ is *active* at time t if there is an iteration $\vec{j} \in \mathcal{J}$ such that

$$t = \vec{\tau}.\vec{j} \quad \text{and} \quad \vec{v} = \Pi\vec{j} .$$

We assume that the physical processors have enough hardware resources to sustain a compute rate of one loop iteration every λ cycles. (We use λ since this time period is something like a wavelength, an inverse frequency.) This leads to different constraints on $\vec{\tau}$. Let $\vec{u}$ be a smallest integer null vector [1] 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. Therefore, we want to allow at least $\gamma\lambda$ cycles between visits. Thus

$$|\vec{\tau}.\vec{u}| \geq \gamma\lambda . \tag{1}$$

[1] Appendix A reviews some basic facts about integer matrices and lattices that we use in this paper.

But we need more. The throughput inequality (1) ensures that the physical processor is not overloaded *on average*. But we are trying to determine a schedule that is precise (all actions scheduled for a certain machine cycle) and correct. We must therefore guarantee that all scheduled iterations begin at times that are unique multiples of λ. We cannot allow two iterations to begin at the same time, nor can we allow them to begin at times that are not multiples of λ, as this will produce conflicts for hardware resources.

It is now obvious that the problem really does not depend on λ; we can just define the clock period to be λ clocks, and set λ to one, making the necessary scaling of the functional unit latencies. (Rational latencies present no particular problems.)

The problem we seek to solve here is this

CONFLICT-FREE SYSTOLIC SCHEDULING: 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*. It is simple to show that $\vec{\tau}$ cannot be a tight schedule if its elements are divisible by some nontrivial common divisor.

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 $\vec{D} \neq \vec{C}$ and $\vec{D} \geq \vec{C}$ elementwise, then this schedule is a nontight, juggling schedule for $\vec{C}$.

Example 1

We take a simple finite-impulse response filter as an example. Suppose the loop nest is

```
for (j1 = 0; j1 < 1000; j1++) {
  for (j2 = 0; j2 < 40; j2++) {
    y[j1] = y[j1] + w[j2] * x[j1+j2];
  }
}
```

Let $\vec{P} = (4)$ and $\Pi = (0, 1)$ be a given processor topology (a four-processor linear array) and linear mapping from iteration (j_1, j_2) to virtual processor $v_1 = j_2$. Clearly, there are 40 virtual processors, and a cluster of 10 adjacent virtual processors to each physical processor. We seek a schedule $\tau(\vec{j}) = \tau_1 j_1 + \tau_2 j_2$. Since the smallest integer null vector of Π is $u = (1, 0)$, the throughput constraint requires that $|\vec{\tau}.\vec{u}| = |\tau_1| \geq 10$, and for a tight schedule $\tau_1 = \pm 10$. The schedule juggles if the residues modulo 10 of the times $\{0, \tau_2, 2\tau_2, \ldots, 9\tau_2\}$ are all different. Clearly, this is so if and only if τ_2 and 10 are relatively prime. □

3 Darte's solution

The idea of Darte's (and initially Darte and Delosme's) solution [1, 2] is to produce a cluster shape $\vec{C}$ compatible with the given schedule vector $\vec{\tau}$. In fact, Darte's method can be used to enumerate all of the cluster shapes $\vec{C}$ for which the given schedule $\vec{\tau}$ is tight. The main objection to the method is not that it is incorrect (it always produces pairs $(\vec{C}, \vec{\tau})$ such that $\vec{\tau}$ is a tight schedule for $\vec{C}$) or incomplete (it can be used to find all cluster shapes for which $\vec{\tau}$ is tight). The objection is that it allows the tail to wag the dog. What is natural in many, although possibly not all, practical situations is that the physical and virtual processor arrays are known, and thus the set of possible cluster shapes is known or is quite tightly constrained. The task at hand is to find a tight schedule for the known cluster shape. Using Darte's approach, this must be done by an indirect and possibly very costly trial-and-error approach: choose $\vec{\tau}$ and then check to see if the given cluster shape is one of those for which $\vec{\tau}$ is tight. If not, try a different schedule. By way of contrast, the new theorem that we later describe leads to a simple method that directly enumerates the tight schedules.

Darte's theorem and method work this way. There exist unimodular matrices having a given integer vector as a row or column if and only if this vector's elements have no nontrivial common divisor. 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 Π be the mapping matrix; we require that Π has a unimodular extension. 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[2] of A: $A = H_a Q_a$ 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.

We now show that a procedure which derives the cluster shape from the schedule may give a too limited set of choices. For example, suppose that

$$\Pi = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \quad \gamma = 6, \quad \text{and} \quad \vec{\tau} = (1, 5, 6)$$

Then

$$Q = \begin{pmatrix} -5 & -6 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \text{and} \quad A = \begin{pmatrix} -5 & -6 \\ 1 & 0 \end{pmatrix}.$$

Both the Hermite normal forms of A and of the (one and only other) matrix obtained by permuting the rows of A are equal to

$$\begin{pmatrix} 1 & 0 \\ 1 & 6 \end{pmatrix}.$$

Thus, there is a single choice, the cluster of shape 1×6; in particular, a 3×2 or 2×3 cluster is forbidden. In practical applications, we have seen clusters that are extremely long and narrow produced this way, for particular values of $\vec{\tau}$, and these are often impossible to use since the virtual processor grid does not contain even one cluster of this shape.

We propose below to reverse the direction of information flow. We have argued that the cluster shape may well be determined prior to scheduling; thus, we will assume that it is given, and will show how to construct all of the tight schedules. With this capability, one may enumerate the allowable tight schedules and choose one based on other criteria, requiring that any linear inequality constraints be satisfied, and minimizing some cost or time measure, for example.

Megson and Chen attempt to guarantee a tight schedule for some given cluster shape $\vec{C}$ by working with the Hermite form of $A = \Pi Q$ more or less 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 $\Pi Q = A$ (*i.e.* a particular solution Q_0 and an arbitrary multiple of $\vec{u}$ added to each column of Q_0) 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). Thus, while Megson-Chen is a "dog wags tail" method, producing tight schedules from the specified cluster shape, it 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.

[2] See Appendix A.

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, for the sake of illustration, we assume that Π consists of the first $(n-1)$ rows of the identity matrix, so that $\Pi u = \Pi e_n = \Pi(0,\dots,0,1)^t = 0$. 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.

We shall first prove that $\vec{\tau}$ is a tight schedule for $\vec{C}$ if

$$\vec{\tau} = (k_1,\ k_2C_1,\ k_3C_1C_2,\ \dots,\ k_nC_1\cdots C_{n-1}) \tag{3}$$

where k_i and C_i are relatively prime and $k_n = \pm 1$. (Note then that $\vec{\tau}$ satisfies (2) by construction.)

Proposition 1 *Let $\vec{C}$ be given and let Π be as above. If $\vec{\tau}$ satisfies (3) then it is a tight schedule for $\vec{C}$.*

Proof We give a proof in two dimensions, as the general case presents nothing new. (The argument is rigorous, but is so simple that it doesn't warrant an overly technical presentation.)

Consider the times at which VP (virtual processor) (c_1, c_2) is active. (Here $0 \le c_i < C_i$ is the local coordinate within the cluster.) We show that the residues modulo γ of these times are some arrangement of the integers $0, 1, \dots \gamma - 1$, no two of them identical. This implies that $\vec{C}$ and $\vec{\tau}$ juggle, and since (2) must hold, it implies that $\vec{\tau}$ is a tight schedule.

First, note that if we show this for one physical processor, then it will be true for all: the activity times on some arbitrarily chosen processor $\vec{p}$ are shifted by a constant, $\vec{\tau}.(p_1C_1,\dots,p_{n-1}C_{n-1})$, compared with the times on processor zero. We therefore assume that the processor coordinates p_i are all equal to zero.

The activity times satisfy $t = \vec{\tau}.\vec{j} = k_1j_1 + k_2C_1j_2 \pm \gamma j_3$ where $j_i = p_iC_i + c_i$ for $i = 1, 2$. We set $p_i = 0$. Then the times at which VP (c_1, c_2) is active satisfy

$$t = k_1c_1 + k_2C_1c_2 \pmod{\gamma}.$$

We need to show that there are no conflicts, that is, that there does not exist another VP in the cluster that has the same activity time modulo the cluster size. Suppose that there is such a conflicting VP, and that its coordinates are $\vec{d} = (d_1, d_2)$. In other words, the VP coordinates $\vec{c}$ and $\vec{d}$ satisfy

$$k_1(c_1 - d_1) + k_2C_1(c_2 - d_2) \equiv 0 \pmod{C_1C_2}.$$

The second term is obviously congruent to zero modulo C_1. This implies that the first term must likewise be a multiple of C_1. Given that k_1 and C_1 are relatively prime and given the bounds on the virtual processor coordinates, the only possible value for the first term is zero; in other words, it must be the case that $c_1 = d_1$. We can then drop the first term, and conclude that

$$k_2(c_2 - d_2) \equiv 0 \pmod{C_2}.$$

Now the same reasoning shows that $\vec{c}$ and $\vec{d}$ are identical. ∎

Now note that we may make any preliminary permutation of the axes and apply the construction of Proposition 1.

Example 2

Suppose that $\vec{u} = (0, 0, 1)$ and that $\vec{C}$, the cluster shape, is $(2, 3)$. Let $\tau_3 = 6$. From (3), either $\vec{\tau} = (k_1, 2k_2, 6)$ or $\vec{\tau} = (3k_1, k_2, 6)$ with $k_i \wedge C_i = 1$, for $i = 1, 2$. Taking $k_1 = 1$ and $k_2 = 5$ for example, we get the *activity tableau* (which shows the residues modulo γ of the times at which each VP in the cluster is active)

1	5	3
0	4	2

in the first case (the c_1 axis is vertical) and

3	2	1
0	5	4

in the second. The number in each box denotes the residue modulo 6 of the times at which the virtual processor that lives there is active. For a juggling schedule, these are all different. □

Now consider the converse of Proposition 1. Clearly, tightness does not imply that (3) holds, because of the axis permutation that can be applied. But after some preliminary permutation of the axes, (3) does hold for all tight schedules, as we next demonstrate. We first assume that $\Pi e_n = 0$, and that $\vec{\tau}$ satisfies (2), *i.e.* that $\tau_n = \pm\gamma$.

We shall use the same group theoretic result that Darte originally employed in his theory. Let $(G, +)$ be a finite abelian group. For $g \in G$ and $1 \leq q \leq \text{order}(g)$, denote by $[g]_q$ the subset $\{0, g, \ldots, (q-1)g\}$ of G. Let $S_1, \ldots, S_n$ be subsets of G. If every $g \in G$ has a unique representation $g = g_1 + \cdots + g_n$ with $g_i \in S_i$ for all $1 \leq i \leq n$, then G is said to be the direct sum of the subsets.

Theorem 1 *If a finite abelian group is the direct sum of subsets of the form $[g]_q$, then at least one of the subsets is a (cyclic) subgroup.*

Proof Proved by Hajós [14]. ■

Now consider the residues modulo γ of the activity times in a cluster of shape $\vec{C}$. The statement that the schedule is tight is equivalent to the statement that these residues are exactly all the elements of the additive group $\mathbb{Z}_\gamma$ of the integers modulo γ, a finite abelian group. Furthermore, $\mathbb{Z}_\gamma$ is clearly the direct sum of the subsets found along the coordinate axes of the cluster: the $(n-1)$ subsets $[\tau_i]_{C_i}$, $1 \leq i < n$. Hajós's theorem implies that one of them is a subgroup. Since an arbitrary permutation of the axes is permitted, we may be allowed to label this the $(n-1)^{\text{st}}$. This implies that $\tau_{n-1}C_{n-1} \equiv 0 \bmod \gamma$, and no smaller multiple of τ_{n-1} has this property. This is equivalent to

$$\tau_{n-1} = k_{n-1}C_1 \times \cdots \times C_{n-2}$$

with $k_{n-1} \wedge C_{n-1} = 1$. Thus, the last two elements of $\vec{\tau}$ have the required form.

We now show how to proceed recursively. Define $\gamma_{n-1} \equiv C_1 \times \cdots \times C_{n-2}$. Consider the activity times in the subcluster corresponding to $c_{n-1} = 0$, taken modulo γ_{n-1}. All γ_{n-1} of these values must be different, for they repeat, each of them, C_{n-1} times as c_{n-1} varies (since τ_{n-1} is a multiple of γ_{n-1}), and each occurs C_{n-1} times in the whole cluster. So they are the elements of the finite abelian group $\mathbb{Z}_{\gamma_{n-1}}$, and again by Hajós's Theorem, the activity times along one of the remaining axes, modulo γ_{n-1}, form a cyclic subgroup. The argument above then applies, and by induction, $\vec{\tau}$ has the form (3) up to a permutation of its first $(n-1)$ elements and the corresponding permutation of $\vec{C}$.

We have therefore proved the main result of this section:

Theorem 2 *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 (3) holds, up to a permutation of the elements of $\vec{C}$ and the same permutation of those elements of $\vec{\tau}$ corresponding to nonzero columns of* Π.

Now we show that 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:

$$M \equiv \begin{pmatrix} \vec{\tau} \\ \Pi \end{pmatrix}. \tag{4}$$

Thus

$$\begin{pmatrix} t \\ \vec{v} \end{pmatrix} = M\vec{j}$$

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

$$\begin{pmatrix} t \\ \vec{v} \end{pmatrix} = M\vec{j} = MS\vec{j'} = \begin{pmatrix} \vec{\tau}.S \\ \Pi S \end{pmatrix} \vec{j'} = \begin{pmatrix} \multicolumn{2}{c}{\vec{\tau}.S} \\ I_{n-1} & 0 \end{pmatrix} \vec{j'} \tag{5}$$

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}.

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)

$$MT = H_m = \begin{pmatrix} 1 & 0 & . & . & . & 0 \\ \star & C_1 & . & & & . \\ . & . & . & . & & . \\ . & & . & . & . & . \\ . & & & . & . & 0 \\ \star & . & . & . & \star & C_{n-1} \end{pmatrix} \tag{6}$$

where the cluster shape is $\vec{C} = (C_1, \ldots, C_{n-1})$. The unimodular matrix T^{-1} is a unimodular extension of $\vec{\tau}$. We call T the "time" matrix. Its first column is a vector $\vec{w}$ such that $\vec{\tau}.\vec{w} = 1$; the vector $\vec{w}$ connects an *isochrone* (hyperplane in the iteration domain of simultaneously scheduled points, scheduled at time t) to the next isochrone (at time $t + 1$).

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.

5 Reducing the cost of control in systolic processors

The loop body of the systolic loop nest can be viewed as a behavioral specification of the systolic processor hardware. The body of the loop after transformation into LSGP form provides this specification. As an example, for a two-dimensional processor array, the loop has the following form (up to further optimization):

```
for (t = 0; t < TMAX; t++) {
  for (p1 = 0; p1 < P1; p1++) in parallel {
    for (p2 = 0; p2 < P2; p2++)  in parallel
      {
        Calculate the cluster coordinates
             c  of the active virtual processor;
        Calculate the global coordinates
             v  of the active virtual processor;
        Calculate the iteration space coordinates
             j  of the iteration mapped to VP  v  at time  t;
        if ( j is in the iteration space )
          {
            Execute the loop body of iteration  j;
          }
      }
    }
  }
```

This code is produced through a sequence of loop transformations, which we now describe.

The first transformation is a tiling. We tile the iteration space, then we synthesize a systolic array for the parallel execution of a single tile. The outer loops over tiles are executed by a host processor. The reason for this tiling is to reduce γ, the number of virtual processors simulated by each physical processor, for this will result in a reduction in the local storage requirements of the systolic processor. Tile shape is ultimately constrainted by the available bandwidth into and out of the systolic array.

A vital technique in realizing the advantages of systolic computation is to pipeline data through the systolic array instead of loading and storing data in shared memory. We model this in the code by using data arrays indexed by time and processor, a technique suggested by Collard [15]. (J. F. Collard views the idea as part of the folklore.) Left hand side occurrences of these data arrays are always indexed by t and $\vec{p}$ without offset. A right hand side occurrence with indices $t - \delta t$ and $\vec{p} - \vec{\delta p}$ refers to the value created for this data array δt clocks earlier on the processor at distance $\vec{\delta p}$ from the current referencing processor. This step requires that we do exact value-based data dependence analysis.

We also change the loop structure: the time and processor coordinates become the loop indices. The strides and loop bounds are known from the initiation interval λ, the schedule, and the given array shape.

The parallel loop body will generally contain code, which we refer to as *housekeeping* or *control* code, not present in the original loop, or implicitly present, 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: $0 \le c_k < C_k$, for $k = 1, \ldots, (n-1)$. These are the position of the currently active VP.

Virtual processor coordinates. One may also need the global virtual processor coordinate $v_k = c_k + p_k C_k$, for $k = 1, \ldots, (n-1)$.

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.)

Cluster edge predicates. When referring to data computed at a different iteration, the mapping to a neighboring virtual processor is straightforward: $\vec{\delta v}$ is just $\Pi\vec{d}$ where $\vec{d}$ is the inter-iteration distance of the dependence. But due to clustering, this virtual processor may be on the same physical processor, or on one of several neighboring processors. *Cluster edge predicates* are tests involving comparison of the cluster coordinates $\vec{c}$ and the cluster shape $\vec{C}$ that are used to find the correct source processor for the data.

Tile edge predicates. These test the global iteration coordinates against the limits of the current tile in order to find the cases in which the source of a dependence is another iteration in the same tile; if not, the data must be read from the global memory.

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.

Iteration space predicates. These test the global iteration coordinates against the limits of the iteration space, to see if there is an iteration scheduled for the given processor at the current time, so as to shut off the processor when it has no scheduled work. Iteration space predicates are present in addition to tile edge predicates when the tiling covers a superset of the iteration space.

Our first experiments with systolic processor synthesis revealed that these housekeeping computations completely overwhelmed the cost of the hardware that carries out the computations inherent in the original loop body, especially when that computation is simple. The result was an inefficient systolic processor.

One reason for this inefficiency was the method used to compute cluster coordinates and the various predicates. 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 [16] 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. The cluster shape appears in this loop nest as the strides for the virtual processor loops. 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 very expensive. Using the loop bounds obtained this way to generate the cluster coordinates of the active virtual processor is expensive: it involves too many maximum and minimum operations, quotient and remainder operations, additions and multiplications. The computation of the i-th cluster coordinate involves, in general, i additions and multiplications, plus a quotient operation and a remainder operation (for computing c_i). The total complexity of the computation of cluster coordinates is thus $O(n^2)$ expensive operations. Indeed, it is a form of integer triangular system solution. We know that

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

where H_m is the Hermite Normal Form of M, and $\vec{i}$ is integer. Furthermore, we know (see (6) above) 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.

This cost can be reduced to $O(n)$ by a slightly different use of the special form of a tight schedule. Let us ignore the permutation. Reconsider (5). Since $\vec{\tau}$ is tight for the given cluster shape $\vec{C}$, it follows that

$$t = (\vec{\tau}S).\vec{j}' = k_1 j'_1 + k_2 C_1 j'_2 + \ldots + k_n C_1 C_2 \cdots C_{n-1} j'_n$$

and

$$\vec{v} = (j'_1, \ldots, j'_{n-1}), \quad \vec{p} = \vec{v} \div \vec{C}, \quad \text{and} \quad \vec{c} = \vec{v} \bmod \vec{C}.$$

with the residue operation applied elementwise to the vectors. Thus,

$$(t \equiv k_1 v_1 \bmod C_1 \text{ and } c_1 = v_1 \bmod C_1) \Rightarrow c_1 = k_1^{-1} t \bmod C_1$$

where k_1^{-1} is the multiplicative inverse of $k_1 \bmod C_1$ (exists since $k_1 \wedge C_1$). In general, $c_i = v_i \bmod C_i = k_1^{-1}(\ldots) \bmod C_i$. The following Matlab function uses this idea to compute the cluster coordinates:

```
function c = time2vp(t, C, p, k)
%     mod(a,b) returns a mod b.
%     intinv(a, b) returns the multiplicative inverse of a mod b
%     when a and b are relatively prime.
for i = 1:n-1
   c(i) = mod(intinv(k(i), C(i)) * t, C(i));
   v(i) = p(i) * C(i) + c(i);
   t = (t - k(i) * v(i)) / C(i);
end
```

Obviously, the running time is $O(n)$. Some computations can be optimized further. For example the products $p_i C_i$ are constant for a given processor, the multiplicative inverses can be pre-computed since k_i is a constant. The residue $k_i^{-1} t \bmod C_i$ can be optimized into table lookup, with a table of size C_i, using $t \bmod C_i$ as the index into the table. But, from the viewpoint of generating hardware, it 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, to the point where their cost is tolerable. We shall show by three examples that once these techniques are employed the overall cost of the systolic processor is in line with what one would expect from examining the original loop body. The reduction in cost, compared with the straightforward implementation above, is typically an order of magnitude.

We shall examine two alternatives. Both of them trade space for time. They exploit the fact that the various relevant quantities may be efficiently inferred from the values they took on the same processor at an earlier time.

For the first option, we exploit the fact that in a tightly scheduled systolic design, all of the cluster coordinates 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.

In the next section, we describe a technique that 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 this recurrence map.) We may choose any time lag δt. (Provided, of course, that δt is not so small that the recurrence becomes a tight dataflow cycle inconsistent with the schedule that we have already chosen.) The technique shows how to construct the corresponding R. We make use of the properties of the tight schedules discussed above in doing this. 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. For most of the integer-valued (as opposed to boolean-valued) housekeeping parameters are linear functions of these, including the global virtual processor coordinates $\vec{v}$, the global iteration space coordinates $\vec{j}$, and the memory addresses. 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. Thus, only one addition is needed to compute each such value.

Concerning the predicates, recall that the virtual processors within a cluster are visited in a round robin schedule that repeats with period γ, since the schedule is assumed to be tight. Thus, any predicate that compares local or global virtual processor coordinates with constants must also be periodic with period γ. Depending on the relative cost of comparison and storage of a one-bit predicate, as well as the cluster size γ, we may choose not to recompute these using comparisons but rather to store them in a buffer of size γ.

We have thus reduced the problem of cost reduction to that of the update of the cluster coordinates. We consider this in the next section.

6 Updating the cluster coordinates

This section describes a temporal recurrence method for computing the cluster coordinates ($\vec{c}$) of the single active virtual processor on a given processor at some given time, without interprocessor communication. From $\vec{c}$ we recover the other relevant parameters and predicates as discussed above. There is a unique VP active at each time, precisely because the schedule is tight.

Recall the properties of the Hermite normal form of the mapping matrix and the time matrix T, whose first column is $\vec{w}$ (6). In fact, $M\vec{w} = (1,\vec{z})^t$ is the first column of H_m. Moreover, because of the properties of a Hermite form, the vector $\vec{z}$ is "in the box": $0 \le \vec{z} < \vec{C}$ (inequality elementwise). (It is therefore a legitimate cluster coordinate vector, and also a legitimate successor to the virtual processor whose cluster coordinates are zero.)

First, we explain how to infer $\vec{c}(t)$ from $\vec{c}(t-\delta t)$ with a little work and the aid of the activity tableau. We show an example before the general theory.

Example 3

Let $n = 3$; let $\vec{C} = (4,5)$. Assume that e_3 is the smallest integer null vector of the space mapping, and let $\vec{\tau} = (7,4,20)$, a tight schedule (with $k_1 = 7$, $k_2 = 1$, $k_3 = 1$). The times at which the virtual processor $\vec{c} = (c_1, c_2)$ is active are the times congruent modulo 20 to the value at position (c_1, c_2) in the activity tableau:

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

The c_1 axis is the vertical axis in this diagram.

If iteration $\vec{j}$ is scheduled at a time that is a multiple of 20, then the cluster's lower left hand virtual processor (whose cluster coordinates are $\vec{c} = 0$) computes it (this is true only for the physical processor $\vec{p} = 0$. For other processors, this property may be shifted). Iteration $\vec{j} + \vec{w}$ is the iteration computed by this physical systolic processor one cycle later, and its cluster coordinates are $\vec{c} = \vec{z}$.

In this example, the Hermite form of the mapping is

$$M = \begin{pmatrix} 7 & 4 & 20 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 3 & 4 & 0 \\ 0 & 3 & 5 \end{pmatrix} \begin{pmatrix} 7 & 4 & 20 \\ -5 & -3 & -15 \\ 3 & 2 & 9 \end{pmatrix} = H_m T^{-1} \text{ and } T = \begin{pmatrix} 3 & 4 & 0 \\ 0 & 3 & 5 \\ -1 & -2 & -1 \end{pmatrix}$$

The vector $z = \begin{pmatrix} 3 \\ 0 \end{pmatrix}$ (the first column of H_m below the diagonal entry of unity) is indeed the offset from the tableau entry at time 0 to the one at time 1. From the tableau we see that to advance one clock we would move in the direction z — up three and no change in column — unless this takes us out of the tableau, *i.e.*, unless we are in rows 2, 3, or 4. In that case, we move to the right a distance 2 modulo the tableau extent of 5, and down 1. □

How can we generalize this, to $\delta t > 1$ and to more than two dimensions? One simple idea is to construct the tableau as we have. Then, for each time $0 \leq t \leq \gamma - 1$, find t and $t + \delta t$ in the tableau. The difference of their positions in the tableau is a possible change that may need to be made. It will turn out that there are only two possible changes for the first component. For each of these, there are two possible changes to the second component, and so forth. In other words, there is a binary tree of component changes. To find the correct new cluster coordinates $\vec{c}(t + \delta t)$ from the current coordinates $\vec{c}$, first add the only one of the two possible changes to the first component that leaves it in the box. Descend from the tree root to the corresponding child; do the same for the second component, etc. The choice will always be unambiguous.

Back to Example 3

In this example, we illustrate the fact that the technique works for an arbitrary δt. Let us look at a tableau in which we replace each entry with the offset to the VP that is active three cycles later:

(-3, 1)	(-3, 1)	(-3, 1)	(-3, 1)	(-3, -4)
(1, 4)	(1, -1)	(1, -1)	(1, -1)	(1, -1)
(1, 4)	(1, -1)	(1, -1)	(1, -1)	(1, -1)
(1, 4)	(1, -1)	(1, -1)	(1, -1)	(1, -1)

The decision tree that corresponds is given in Figure 1. □

In the following example, we illustrate the fact that the technique works for an arbitrary dimension. In general, the tree has depth $(n-1)$. At each leaf, we have specified the full change for the cluster coordinate vector, and by linearity we also infer the corresponding changes to all relevant virtual processor coordinates, iteration space coordinates, array indices, memory addresses, and any other linearly related and useful quantities.

Example 4

Let $\vec{C} = (4, 3, 2)$; let $\vec{\tau} = (7, 8, 12, 24)$. The three-dimensional activity tableau is

$c_3 = 0$			$c_3 = 1$		
21	5	13	9	17	1
14	22	6	2	10	18
7	15	23	19	3	11
0	8	16	12	20	4

By searching the tableau we find these six possible transitions when $\delta t = 1$:

$$\begin{pmatrix} 3 & 3 & 3 & 3 & -1 & -1 \\ 2 & 2 & -1 & -1 & -2 & 1 \\ 1 & -1 & -1 & 1 & 0 & 0 \end{pmatrix}.$$

Now consider the first coordinate, which must lie in the range $0 \le c_1 < 4$. One can add 3 if the value is 0 or subtract 1 otherwise; in other words, there is no ambiguity about which of the two possible changes to make. If the change in the first coordinate is 3, then the change in the second is either 2 or -1, etc. The decision tree is given in Figure 2. □

```
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);
      }
  }
```

Figure 1: Decision tree code for Ex. 3.

```
if (c(1) + 3 < C(1)) {
  c(1) += 3;
  if (c(2) + 2 < C(2)) {
    c(2) += 2;
    if (c(3) + 1 < C(3))
      c(3) += 1;
    else
      c(3) += (-1);
  }
  else {
    c(2) += (-1);
    if (c(3) + 1 < C(3))
      c(3) += 1;
    else
      c(3) += (-1);
  }
}
else {
  c(1) += (-1);
  if (c(2) + 1 < C(2)) {
    c(2) += 1;
    c(3) += 0;
  }
  else {
    c(2) += (-2);
    c(3) += 0;
  }
}
```

Figure 2: Decision tree code for Ex. 4.

6.1 A general updating scheme

We now show how to construct the decision tree from first principles, without resort to construction and exploration of the activity tableau. We shall also prove the assertions of unambiguity and correctness made above. Since $\vec{\tau}$ is tight, we have that (up to permutation of the indices, and with the proviso that Π consists of $(n-1)$ rows of the identity)

$$\vec{\tau} = (k_1,\ k_2C_1,\ \dots,\ k_{n-1}C_1\cdots C_{n-2},\ \pm\gamma) \tag{7}$$

where $k_i \wedge C_i = 1$ for all $1 \le i \le n-1$.

We consider the set $\mathcal{C}$ of cluster coordinates within a cluster of virtual processors, *i.e.* within one systolic processor:

$$\mathcal{C} = \{\vec{c} \in \mathbb{Z}^{n-1} \mid 0 \le c_k < C_i, \quad k = 1, \dots, (n-1)\}.$$

For every position $\vec{c} \in \mathcal{C}$, we associate a local clock at which the given virtual processor is active

$$t_c(\vec{c}) \equiv \tau_1 c_1 + \cdots \tau_{n-1}c_{n-1} \pmod{\gamma}.$$

t_c maps $\mathcal{C}$ one-to-one onto $[0..(\gamma-1)]$. Let δt be a given time lag. We wish to know the set $\mathcal{B}$ of all of the differences of positions (members of $\mathcal{C}-\mathcal{C}$, the set of differences of cluster coordinate vectors) that can occur as values of $\tau^{-1}(t+\delta t) - \tau^{-1}(t)$. We abuse notation and use $\vec{\tau}$ for its first $(n-1)$ components wherever convenient. By definition, $\mathcal{B}$ consists of the position-difference vectors $\vec{x}$ that satisfy $\vec{\tau}.\vec{x} \equiv \delta t \pmod{\gamma}$. By (7) we have

$$k_1x_1 + k_2C_1x_2 + \cdots + k_{n-1}C_1 \times \cdots \times C_{n-2}x_{n-1} \equiv \delta t \pmod{\gamma}. \tag{8}$$

Now divide δt by C_1: then $\delta t = q_1C_1 + r_1$ where $0 \le r_1 < C_1$. By (8), we have that $k_1x_1 \equiv r_1 \bmod C_1$, so that $x_1 \equiv k_1^{-1}r_1 \bmod C_1$, (recall that k_1 and C_1 are relatively prime, so that k_1 has an inverse in the additive group of the integers modulo C_1). Because $x \in \mathcal{B}$ has elements bounded in absolute value by the elements of C, it follows that there are only two possible values for x_1,

$$x_1 \in \left\{k_1^{-1}r_1 \bmod C_1\,,\ (k_1^{-1}r_1 \bmod C_1) - C_1\right\}.$$

These are the two possible differences of the first coordinate of $\vec{c}(t+\delta t)$ and $c(\vec{t})$. The choice is made on the simple basis of which leads to a new point in the box. Only one can. So

$$c_1(t+\delta t) = c_1(t) + \begin{cases} k_1^{-1}r_1 \bmod C_1 & \text{if } (c_1(t) + k_1^{-1}r_1 \bmod C_1) < C_1 \\ k_1^{-1}(r_1 \bmod C_1) - C_1 & \text{otherwise} \end{cases}$$

Pursuing this line of argument, for each choice of change in first coordinate we determine the two possible choices for the change x_2 in second coordinate. From (8) we have that $k_2C_1x_2 + \cdots + k_{n-1}C_1 \times \cdots \times C_{n-2}x_{n-1} \equiv (\delta t - k_1x_1) \pmod{\gamma}$. We already know that $\delta t - k_1x_1$ is a multiple of C_1. Thus, we have that

$$k_2x_2 \equiv ((\delta t - k_1x_1)/C_1) \pmod{C_2}.$$

Thus, as before, we conclude that

$$x_2 \in \left\{k_2^{-1}((\delta t - k_1x_1)/C_1) \bmod C_2\,,\ (k_2^{-1}((\delta t - k_1x_1)/C_1) \bmod C_2) - C_2\right\}.$$

Continuing in this way, we arrive at the tree of changes of cluster coordinates.

Matrix-oriented approach The decision tree phenomenon can also be explained from a matrix-oriented and geometric viewpoint that has the additional advantage of simplifying the computation of the different changes in coordinates.

Remember the meaning of the columns of the matrix T. T is a basis of $\mathbb{Z}^n$ such that $MT = H_m$; the first row of MT gives the time difference along each column vector of T, and the last rows are the coordinates of the column vectors of T in the virtual processor array. Since the first row of MT is $(1, 0, \dots, 0)$, the first column $\vec{w}$ of T connects an isochrone 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. We know already that $\vec{k}$ exists and is unique since the schedule juggles. 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 $\vec{t_2}$, $\ldots$, $\vec{t_n}$ to $\delta t \times \vec{w}$ (the first component of $M(\delta t \times \vec{w})$ does not change) 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, then 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: $\vec{\delta}[1, 0, \ldots, 0] = \vec{\delta}[0, \ldots, 0] - \vec{t_2}$ (plus possibly a linear combination of $\vec{t_3}$, $\ldots$, $\vec{t_n}$ 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.

We illustrate this technique on the previous examples.

Back to Example 3

We take $\delta t = 1$. From the matrix H_m, we read that we may have to move along $\vec{w} = (3, 0)$ in the virtual space (which corresponds to the vector $\vec{\delta}[0, 0] = (3, 0, -1)$ in the original space). If $c_1 + 3 \geq 4$, 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 $\vec{\delta}[1, 1] = (-1, -3, 1)$.

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

Back to Example 4

In this case, $\vec{C} = (4, 3, 2)$, $\vec{\tau} = (7, 8, 12, 24)$, and we find:

$$MT = \begin{pmatrix} 7 & 8 & 12 & 24 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 3 & 4 & 0 & 0 \\ 2 & 1 & 3 & 0 \\ 1 & 1 & 0 & 2 \\ -2 & -2 & -1 & -1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 3 & 4 & 0 & 0 \\ 2 & 1 & 3 & 0 \\ 1 & 1 & 0 & 2 \end{pmatrix} = H_m$$

We thus find successively, in the virtual space, the vectors (written as rows) $(3, 2, 1)$, $(-1, 1, 0) = (3, 2, 1) - (4, 1, 1)$, $(3, -1, 1) = (3, 2, 1) - (0, 3, 0)$, $(-1, -2, 0) = (-1, 1, 0) - (0, 3, 0)$, $(3, 2, -1) = (3, 2, 1) - (0, 0, 2)$ and $(3, -1, -1) = (3, -1, 1) - (0, 0, 2)$, and, in the original space, the corresponding vectors $\vec{\delta}[0, 0, 0] = (3, 2, 1, -2)$, $\vec{\delta}[1, 0, 0] = (-1, 1, 0, 0)$, $\vec{\delta}[0, 1, 0] = (3, -1, 1, -1)$, $\vec{\delta}[1, 1, 0] = (-1, -2, 0, 1)$, $\vec{\delta}[0, 0, 1] = (3, 2, -1, -1)$, and $\vec{\delta}[0, 1, 1] = (3, -1, -1, 0)$. □

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

6.2 Measuring the cost of control 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

```
x[i1][i2][i3][i4] += y[i1][i3][i5][i6] * z[i1+i3][4*i3+i6]
```

	Photography			Printing		Matrix Mult.	
operation	original	transformed	naive	original	transformed	original	transformed
$+$	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

Table 1: Operation counts in inner loop, before and after parallelization (and with a naive scheme.)

The second, from a printing application, is a much more complicated loop body. The third is matrix multiplication.

Using the mechanisms described in this paper, we transformed the loop nests. Full details of the mechanisms used (for uniformization, loop transformation, mapping and scheduling, *etc.*) will appear in another paper. By way of illustration, we present the original and the transformed loop nests for matrix multiplication in Appendix B.

We show the number of inner-loop integer operations in the original and the fully transformed loop nests in Table 6.2. 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 source code. One add was charged in the original code for each array reference (assuming compiler strength reduction of the address calculation). One add and one compare was charged for control of the inner loop. The text was then scanned for other arithmetic operations, with duplicate expressions assumed to be caught by common subexpression elimination optimization. Division by a constant is assumed done by a multiplication.) 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 case 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 does 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 the predicates of conditionals and for array addresses. In the example given, about 36 percent of all operations in the naive code are generated in the $O(n^2)$ algorithm for cluster coordinates; about 27 percent are due to the calculation of all predicates directly from their definitions, and about 35 percent are due to array addressing; the remaining two percent are for loop control.

7 Conclusion

The first part of this paper provides a simple characterization of all tight LSGP schedules, solving a long-standing problem in systolic array synthesis. The characterization allows a synthesis system to directly enumerate all of the tight schedules in any desired region of the space of schedules, which can be very useful in generating tight schedules that have some other desirable properties.

Parallelization is known to have some cost. We have shown how to make these costs very manageable by use of a fairly sophisticated optimization strategy that takes full advantage of the form of LSGP parallel codes with tight schedules. We made essential use of the mathematical properties of tight schedules, using the characterization of them that we provided in the first part of this paper. Synthesis of specialized processors for the systolic code that we generate is likely to result in highly efficient use of chip area; indeed our experiments with a simple chip area model confirms this. 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 at which it is not overly burdensome, especially for loop nests that have more than a handful of computations in the innermost loop.

References

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

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

[3] G. M. Megson and X. Chen, "A synthesis method of LSGP partitioning for given-shape regular arrays," in *Proceedings of the Ninth International Parallel Processing Symposium*, (Santa Barbara, CA), pp. 234–238, IEEE Computer Society Press, 1995.

[4] P. Quinton and Y. Robert, *Systolic Algorithms and Architectures.* Prentice Hall, 1991.

[5] H. T. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," in *Sparse Matrix Proceedings*, (Philadelphia, PA), pp. 256–282, SIAM, 1978.

[6] D. Moldovan, "On the analysis and synthesis of VLSI systolic arrays," *IEEE Transactions on Computers*, vol. 31, pp. 1121–1126, 1982.

[7] P. Quinton, "Automatic synthesis of systolic arrays from uniform recurrent equations," in *Proceedings of the Eleventh IEEE Symposium on Computer Architecture*, (Ann Arbor, MI), pp. 208–214, IEEE Computer Society Press, 1984.

[8] R. Schreiber and J. Dongarra, "Automatic blocking of nested loops," Tech. Rep. 90-38, Research Institute for Advanced Computer Science, 1990.

[9] J. Ramanujam and P. Sadayappan, "Tiling of iteration spaces for multicomputers," in *Proceedings of the International Conference on Parallel Processing*, pp. 179–186, IEEE, 1990.

[10] J. Ramanujam, "A linear algebraic view of loop transformations and their interaction," in *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing* (J. Dongarra, K. Kennedy, P. Messina, D. Sorensen, and R. Voigt, eds.), pp. 543–548, SIAM, 1992.

[11] D. Moldovan and J. Fortes, "Partitioning and mapping algorithms into fixed-size systolic arrays," *IEEE Transactions on Computers*, vol. 35, pp. 1–12, 1986.

[12] F. Irigoin and R. Triolet, "Supernode partitioning," in *Proceedings of the Fifteenth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, (San Diego, CA), pp. 319–329, 1988.

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

[14] G. Hajós, "Uber einfache und mehrfache Bedeckung des n-dimensionalen Raumes mit einem Wurfelgitter," *Mathematische Zeitschrifte*, vol. 47, pp. 427–467, 1942.

[15] J.-F. Collard, "Code generation in automatic parallelizers," in *Proceedings of the International Conference on Applications in Parallel and Distributed Computing. IFIP W.G 10.3* (C. Girault, ed.), (Caracas, Venezuela), pp. 185–194, North Holland, 1994.

[16] C. Ancourt and F. Irigoin, "Scanning polyhedra with DO loops," in *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'91)*, (Williamsburg, VA), pp. 39–50, Apr. 1991.

[17] M. Newman, *Integral Matrices.* New York: Academic Press, 1972.

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

A Integer matrices and lattices

A few of the basic terms and facts that we use are collected here. For a thorough discussion, see [17] or [18].

A lattice in a finite-dimensional vector space is the set of integer linear combinations of a set of linearly independent vectors. If the vector space has a norm, then an equivalent definition is that a lattice is an additive subgroup (a set of vectors closed under addition and subtraction) whose elements are all separated from one another by some minimum distance.

The set of integer linear combinations of an arbitrary finite set of vectors (whether linearly independent or not) is also a lattice. (For example, the set of all integer linear combinations of 14 and 21 is the set of all integer multiples of their greatest common divisor, 7.) If the integer linear combinations of a set $S \subset L$ of vectors is all of the lattice L then we say that S generates L. If A is any matrix, the lattice generated by A is just the lattice generated by its columns. Every m-dimensional lattice (contained in an m-dimensional subspace and in no smaller subspace) has a set of m of its elements that generates it.

In this paper we consider only lattices that are subsets of $\mathbb{Z}^n$.

Let A be an integer matrix with nullity ν, its integer null vectors are a lattice, which we call the null lattice of A. Clearly, there exists a set of ν integer null vectors of A that generates its null lattice. When $\nu = 1$, we call this selected null vector the *smallest integer null vector* of A; it is unique up to sign.

A square integer matrix is called *unimodular* if the absolute value of its determinant is one.

Lemma 1 *A unimodular matrix has an integer inverse.*

Proof Cramer's rule. ∎

Lemma 2 *A unimodular matrix maps the integer vectors one-to-one and onto themselves.*

Corollary 1 *The lattice generated by the columns of A is the same as the lattice generated by AQ for any unimodular matrix Q.*

Let A be an $n \times n$ integer matrix of rank n. Then there exists a lower triangular matrix H and a unimodular matrix U for which

$$A = HU \quad \text{with} \quad h_{ii} > 0 \quad \text{and} \quad 0 \leq h_{ij} < h_{ii} \quad \text{for all} \quad 1 \leq i, j \leq n.$$

The matrix H is called the Hermite normal form of A, and it is unique.

Lemma 3 *Let Q be unimodular and let A be square and nonsingular. Then A and AQ have the same Hermite normal form.*

Proof If $H = AU$ where H is in Hermite form and U is unimodular, then $H = AU = (AQ)(Q^{-1}U)$ is also a Hermite decomposition. ∎

Lemma 4 *Let L be square and lower triangular. The diagonal of the Hermite normal form of L is the absolute value of the diagonal of L.*

Proof Adding or subtracting one column to or from another is a unimodular transformation. Multiplying a column by -1 is too. These operations can bring any lower triangular matrix into Hermite normal form. See [17] for more details. ∎

B A transformed loop nest

Here is the original loop nest for a matrix multiplication example.

```
for (i=0; i < 1400; i++)
   for (j=0; j < 1800; j++)
      for (k=0; k < 1600; k++)
          c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

The system was instructed to generate a systolic array with four processors in a 2×2 arrangement, each of which is capable of one loop iteration per cycle. The mapping direction was specified to be $u = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}$.

The inner loop was first tiled, with tiles of shape $\begin{pmatrix} 6 & 6 & 1600 \end{pmatrix}$. The tiling mechanism we use is restricted to tiling along planes parallel to the iteration space axes. The tile shape was chosen so that the memory bandwidth needed to support the systolic array's demands was two words per cycle; this available memory bandwidth is a user-prescribed parameter.

The VP array therefore had shape $\begin{pmatrix} 6 & 6 \end{pmatrix}$, giving a cluster shape $\vec{C} = \begin{pmatrix} 3 & 3 \end{pmatrix}$; there were therefore $\gamma = 9$ VPs in each cluster.

The schedule $\vec{\tau} = \begin{pmatrix} -1 & -3 & 9 \end{pmatrix}$ was selected. (According to the theory above, this schedule is tight.)

Here is the transformed inner loop. Note the presence of $3n - 2 = 7$ integer additions for calculating coordinates, 2 additions, one per addressed array, for address updates, and 8 comparisons for control purposes. This is the housekeeping burden. In order to make the schema more obvious, we have *not* optimized the loop to the greatest possible extent. All but two of the comparisons generate γ-periodic results and therefore could be removed, and replaced with one-bit wide FIFO buffers of length $\gamma = 9$ if this were deemed desirable. Moreover, some of the coordinates are not used, and their computation would be suppressed in an optimized version. The costs of an optimized systolic matrix multiply loop are given in Table 6.2.

```
/* Systolic processor code for matrix multiplication on a 2 X 2 systolic array */

    for (t = -20; t <= 14391; t++)
      {
        for (P_1 = 0; P_1 <= 1; P_1++)
          {
            for (P_2 = 0; P_2 <= 1; P_2++)
              {

                /* decision tree for update of cluster coordinates
                   and addresses using delta_t = 6 */
                delta_c1 = 0;
                if (ua_c2[t - 6][P_1][P_2] < 2)
                  {
                    delta_c2 = 1;
                    delta_k = 1;
                    addr_change_a = 8;
                    addr_change_b = 14408;
                  }
                else
                  {
                    delta_c2 = -2;
                    delta_k = 0;
                    addr_change_a = 0;
                    addr_change_b = -16;
                  }
```

```
/* updated cluster coordinates */
c1 = delta_c1 + ua_c1[t - 6][P_1][P_2];
c2 = delta_c2 + ua_c2[t - 6][P_1][P_2];
/* virtual processor coordinates.  CPi holds C(i) P(i) */
v1 = c1 + CP1[P_1][P_2];
v2 = c2 + CP2[P_1][P_2];
/* iteration coordinates */
i = i_tile + v2;
j = j_tile + v1;
k = delta_k + ua_i3[t - 6][P_1][P_2];
/* updated addresses */
addr_a = addr_change_a + ua_addr_a[t - 6][P_1][P_2];
addr_b = addr_change_b + ua_addr_b[t - 6][P_1][P_2];

/* determine if the processor is active:
   does it have a valid iteration */
if ((i < 1400) && (j < 1800) && (0 <= k) && (k < 1600))
  {
    /* get value of matrix A from its previous use... */
    if (v1 <= 26)
      {
        if (c1 < 6)
          {
            /* which may be on this processor... */
            tmp_2 = ua_a[t - 3][P_1][P_2];
          }
        else
          {
            /* or a neighboring processor */
            tmp_2 = ua_a[t - 3][P_1][P_2 + 1];
          }
        ua_a[t][P_1][P_2] = tmp_2;
      }
    else
      {
        /* or, for the first use, from memory. */
        ua_a[t][P_1][P_2] = *addr_a;
      }

    /* get value of matrix B as well */
    if (v2 <= 30)
      {
        if (c2 < 7)
          {
            tmp_3 = ua_b[t - 1][P_1][P_2];
          }
        else
          {
            tmp_3 = ua_b[t - 1][P_1 + 1][P_2];
          }
        ua_b[t][P_1][P_2] = tmp_3;
      }
    else
```

```
            {
              ua_b[t][P_1][P_2] = *addr_b;
            }

          /* perform the loop computation */
          ua_c[t][P_1][P_2] = ua_c[t - 9][P_1][P_2] +
                      ua_a[t][P_1][P_2] * ua_b[t][P_1][P_2];
        }

      /* push the coordinates and addresses into the fifos */
      ua_addr_b[t][P_1][P_2] = addr_b;
      ua_addr_a[t][P_1][P_2] = addr_a;
      ua_c2[t][P_1][P_2] = c2;
      ua_c1[t][P_1][P_2] = c1;
      ua_i3[t][P_1][P_2] = k;
    }
  }
}
```