# Constructing and Exploiting Linear Schedules with Prescribed Parallelism

Alain Darte[1], Robert Schreiber, B. Ramakrishna Rau, Frederic Vivien[2]
Compiler and Architechture Research
HP Laboratories Palo Alto
HPL-2001-238
October 3rd , 2001*

E-mail: darte@ens-lyon.fr, schreiber@hpl.hp.com, rau@hpl.hp.com, vivien@icps.u-strasbg.fr

systolic array, multicluster VLIW, linear schedule

We present two new results of importance in code generation for and synthesis of synchronously scheduled parallel processor arrays and multicluster VLIWs. The first is a new, practical method for constructing a linear schedule for the iterations of a loop nest that schedules precisely one iteration per cycle on each of a prescribed set of processors. While this problem goes back to the era in which systolic computation was in vogue, it has defied practical solution until now. We provide a closed form solution that enables the enumeration of all such schedules. The second result is a new technique that reduces the cost of code or hardware whose function is to control the flow of data, predicate operations, and generate memory addresses. The key idea is that by using the mathematical structure of any of the conflict-free schedules we construct, a very shallow recurrence can be developed to inexpensively update these qualities.

* Internal Accession Date Only Approved for External Publication
[1] LIP, ENS-Lyon, 46 allee d'Italie, 69364 Lyon Cedex 07, France
[2] ICPS/LSIIT, Pole Api, Boulevard Sebastien Brant, F-67400 Illkirch, France
To be published in the ACM Transactions on Design Automation of Electronic Systems

# Constructing and exploiting linear schedules with prescribed parallelism

Alain Darte
LIP, ENS-Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France tel: +33 4 72 72 85 49 email:: darte@ens-lyon.fr
and
Robert Schreiber
Hewlett-Packard Company, 1501 Page Mill Road, Palo Alto, CA 94304-1126, USA tel: 650 857-8156 email:: schreiber@hpl.hp.com
and
B. Ramakrishna Rau
Hewlett-Packard Company, 1501 Page Mill Road, Palo Alto, CA 94304-1126, USA tel: 650 857-8018 email:: rau@hpl.hp.com
and
Frédéric Vivien
ICPS/LSIIT, Pôle Api, Boulevard Sebastien Brant, F-67400 Illkirch, France email:: vivien@icps.u-strasbg.fr

---

We present two new results of importance in code generation for and synthesis of synchronously scheduled parallel processor arrays and multicluster VLIWs. The first is a new, practical method for constructing a linear schedule for the iterations of a loop nest that schedules precisely one iteration per cycle on each of a prescribed set of processors. While this problem goes back to the era in which systolic computation was in vogue, it has defied practical solution until now. We provide a closed form solution that enables the enumeration of all such schedules. The second result is a new technique that reduces the cost of code or hardware whose function is to control the flow of data, predicate operations, and generate memory addresses. The key idea is that by using the mathematical structure of any of the conflict-free schedules we construct, a very shallow recurrence can be developed to inexpensively update these quantities.

Categories and Subject Descriptors: B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*pipelined, special-purpose, parallel data-path design*; D.3.4 [**Programming Languages**]: Processors—*compilers*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*scheduling*

Additional Key Words and Phrases: systolic array, multicluster VLIW, linear schedule

---

A preliminary version of this paper appeared in the proceedings of the 14th International Parallel and Distributed Processing Symposium (IEEE Computer Society, 2000, pp. 815–821) under the title *A constructive solution to the juggling problem in processor array synthesis.*

## 1. INTRODUCTION

We consider the implementation of a perfect loop nest through software pipelining on statically-scheduled hardware. The target machines we consider include one VLIW processor with prescribed hardware, or a prescribed array of such VLIWs, *i.e.,* a multicluster VLIW. Alternatively, we may be asked to synthesize an array of customized, irregular VLIW processors that implements the given nest with a prescribed initiation interval (II). (The II is the number of machine cycles between two consecutive executions of a given operation in the loop body [Rau 1996].)

Traditionally, software pipelining has been done on inner loops, possibly with a preliminary loop permutation. We look at two significant generalizations. First, we consider handling one iteration on each cluster at a time, thereby using loop-level parallelism as well as instruction-level parallelism to provide enough parallelism to saturate the hardware resources. Second, we consider arbitrary linear schedules, which generalizes loop permutation. Doing so allows us to use additional loop-level parallelism to cope with recurrences that would otherwise limit the achievable II. Even though we can use an arbitrary linear schedule, we also automatically flatten the loop nest into a singly nested loop, improving efficiency because prologue and epilogue code executes only once.

Our principle motivation is ASIC synthesis. The consumer electronics industry demands low-cost, high-performance compute hardware to perform image and signal processing. Cost-effective designs often employ embedded general-purpose computers assisted by ASICs. Such systems are difficult and expensive to design. This makes automatic synthesis of application-specific hardware accelerators increasingly desirable.

We feel that the synthesis of an array of VLIWs for implementing a loop nest with prescribed throughput is a central problem in automatic ASIC synthesis. We use the techniques presented here in the HP Labs PICO[1] system [Schreiber, Aditya, Rau, Kathail, Mahlke, Abraham, and Snider 2000], which automatically synthesizes the hardware and software for such processor-and-ASIC systems. In PICO, a source program is 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.

We are concerned here only with synchronous arrays of statically-scheduled processing elements; such an array may also be viewed as a single *multicluster* machine. The term *processor* is used in this paper to mean one of the clusters. There is only one process, or thread of control, regardless of the number of such processors.

This paper addresses two important practical problems. The first is to map each iteration of the 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 that accomplishes this. We present some new theoretical insight into this problem that leads directly to an efficient solution.

The second problem is to control the cost of this sort of parallel implementation of a loop nest. Parallel realizations of sequential algorithms come at some cost: in our case additional computation – which would lead to additional hardware in an

[1] Program In, Chip Out

application-specific accelerator – needed to control and coordinate the processor. By exploiting some basic properties of our conflict-free schedules, we develop a 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.

## 2. AN EXAMPLE

*Example* 1. Consider the nest

```
for (i = 0; i < 100; i++) {
  for (j = 0; j < 10; j++) {
    x[i+1][j+1] = x[i][j+1] * x[i+1][j];
  }
}
```

Suppose the application demands an implementation that uses about 500 cycles to do all 1,000 iterations. Both the inner and outer loops are sequential, and there is a multiplication in the critical path of the recurrence, so neither the inner nor the outer loop is amenable to software pipelining with low II. Yet, there is considerable loop-level parallelism: all iterations for which $i+j=k$ can be computed in parallel, for all $k = 2, \ldots, 110$. So there are only 109 iterations on the critical path. If the latency of multiply is three cycles, then this critical path is 327 cycles, so it is not impossible to get the job done with enough hardware.

We allocate two processors, each of which is to achieve an II of one. We may map iteration $(i, j)$ to virtual processor $j$ and then to physical processor $j \div 5$. Then we schedule iteration $(i, j)$ to start at cycle $5i + 3j$. Alternatively, we may map iteration $(i, j)$ to virtual processor $i$, and then to physical processor $i \div 50$. The schedule in this case is chosen as $3i + 50j$. It is not hard to see that these maps are conflict free, causal, and that the recurrences have been covered by loop-level parallelism so that an II of one is achievable. Clearly, the first choice, which has a total schedule length of $5 \cdot 99 + 3 \cdot 9 = 522$, is preferable to the second, with a schedule length of $3 \cdot 99 + 50 \cdot 9 = 747$.

The remainer of this paper shows how such mappings and schedules can be efficiently found and effectively implemented.

## 3. ITERATION TO PROCESSOR MAPPING

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 processors with rectangular topology is given, and each processor is identified by its coordinate vector. The mapping problem is to find functions $\pi$ and $\tau$ such that processor $\pi(\vec{j})$ commences computation of iteration $\vec{j}$ at cycle $\tau(\vec{j})$. We use the term *schedule* for the timing function $\tau$ and *mapping* for the processor assignment $\pi$. Since we are interested in using software-pipelined processors, the time $\tau(\vec{j})$ is the start time for iteration $\vec{j}$. The set of operations belonging to that iteration are scheduled relative to this start time. We require that the scheduling function $\tau$ be an integer-valued linear function of the iteration vector.

We allow spatial mappings that work, as in Example 1, by a projection $\Sigma$ of the iteration space into an $(n-1)$-dimensional array of virtual processors (VPs), followed by a partitioning of the virtual processors among the given set of physical processors. We can allow any projection $\vec{v} = \Sigma\vec{j}$, where the $(n-1) \times n$ integer matrix $\Sigma$ admits a unimodular extension.

Each physical processor is then assigned the work of a cluster of virtual processors. A cluster is a rectangular neighborhood in the array of virtual processors. 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 a grid of the same shape as the processor grid.

Note that because the clusters can be as big, in any given dimension, as the virtual processor grid, the physical processor array can have any dimensionality up to $(n-1)$. For example, we would map a $10 \times 10$ virtual processor array into a one-dimensional array of two physical processors by using either $10 \times 5$ or $5 \times 10$ clusters.

Let an $(n-1)$-dimensional grid of 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 $\Sigma$. Let the smallest rectangle that covers the set of virtual processors have dimensions $\vec{V}$, so that if $\vec{v} = \Sigma\vec{j}$ 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 VP (virtual processor) coordinates are

$$v_i = p_i C_i + c_i \,, \tag{1}$$

where $\vec{c}$ satisfies $0 \le \vec{c} \le \vec{C}$; the *cluster coordinates* $\vec{c}$ give the position of the VP within its cluster. The number of virtual processors assigned to a processor is not more than $\gamma \equiv \prod_{i=1}^{n-1} C_i$.

In Example 1, the first mapping was obtained by taking $\Sigma = (0, 1)$, which yields a one-dimensional array of 10 virtual processors. Since we target two physical processors, $\vec{C} = (\lceil 10/2 \rceil) = (5)$, and iteration $(i, j)$ maps to processor $p = j \div 5$. (We dropped the subscript because the processor array has only one dimension.) For the second mapping, $\Sigma = (1, 0)$, $\vec{V} = (100)$, and $\vec{C} = (\lceil 100/2 \rceil) = (50)$; iteration $(i, j)$ maps to processor $p = i \div 50$.

## 4. ITERATION SCHEDULING

We seek a linear schedule $\tau(\vec{j}) = \vec{\tau}.\vec{j}$. Assume that the physical processor can start one loop iteration per cycle. Given the mapping, we need to find a linear schedule that schedules, in the steady state, one iteration per clock on each processor. In Example 1, we used $\vec{\tau} = (5, 3)$ with the first projection and clustering, and $\vec{\tau} = (3, 50)$ with the second.

The schedule must be causal; this means that if there is a path in the dataflow graph of the loop from an operation to the same operation at a later iteration, then the start times of these iterations must differ by at least the sum of the latencies of the operations on the path. This causality requirement amounts to a set of linear

inequality constraints on $\vec{\tau}$. In Example 1, we required that each element of $\tau$ be three or greater because of the dependence of the multiply on itself at earlier iterations in the directions $(0,1)$ and $(1,0)$, and the latency, three, of the multiply.

Let $\vec{u}$ be a smallest integer null vector of $\Sigma$. 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 want a schedule with the property that the physical processor visits each of its $\gamma$ simulated virtual processors once, in some round-robin manner, before returning to $\Sigma\vec{j}$ again. Because we need to allow at least $\gamma$ cycles between visits,

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

The throughput inequality (2) ensures that the physical processor is not overloaded *on average*. It remains to ensure that no two iterations start at the same time on the same processor. The problem we seek to solve here is the following **conflict-free scheduling problem**: given $\vec{C}$, the mapping $\Sigma$ 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 $\gamma$ 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 (2) with equality,

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

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

## 5. EARLIER WORK

Darte, Delosme, Megson, and Chen have provided partial solutions to this problem.

The idea of Darte's (and initially Darte and Delosme's) solution [Darte and Delosme 1990; Darte 1991] 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 $A = \Sigma Q$. Then $A$ is a square, integer 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 $A$:

$A = H_a Q_a$ with $Q_a$ unimodular [2]. 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 [Megson and Chen 1995] attempt to guarantee a tight schedule for a given cluster shape $\vec{C}$ by working with the Hermite form of $A = \Sigma Q$ directly. Relying on the fact that the Hermite form of a triangular matrix $X$ has the same diagonal as $X$, they choose $A$ to be triangular with the elements of $\vec{C}$ on the diagonal, and they assume that $\Sigma$ is known. They then look at the general solution $Q$ to the underconstrained linear system $\Sigma Q = A$ 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.

## 6. 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 $\Sigma$ consists of the first $(n-1)$ rows of the identity matrix, so that $\Sigma u = \Sigma e_n = \Sigma(0, \ldots, 0, 1)^t = 0$. We write $x \wedge z$ for the greatest common divisor of $x$ and $z$. Then, we have the following result:

THEOREM 1. *Let $\vec{C}$ be a given cluster shape. If $\Sigma$ consists of the first $(n-1)$ 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 $\vec{\tau}$,*

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

*where $k_i \wedge C_i = 1$ and $k_n = \pm 1$.*

PROOF. The **if** part is easy. For the **only if** part, we use Hajós theorem on a representation of a finite abelian group as a direct sum [Hajós 1942], also employed in [Darte 1991]. The complete proof is available in the extended version of this paper [Darte, Schreiber, Rau, and Vivien 1999]. □

The restriction on $\Sigma$ is unnecessary. Let $S$ be the inverse of a unimodular extension of $\Sigma$. The last column of $S$ is the projection vector $\vec{u}$, and $\Sigma S = I_{n-1}$, the identity matrix of order $(n-1)$. Define the linear loop transformation matrix $M$

[2] For more about Hermite forms and lattice theory, we refer to [Newman 1972] and [Schrijver 1986].

to be the matrix whose first row is $\vec{\tau}$ and whose last $(n-1)$ rows are $\Sigma$:

$$M \equiv \begin{pmatrix} \vec{\tau} \\ \Sigma \end{pmatrix}; \text{ thus } \begin{pmatrix} t \\ \vec{v} \end{pmatrix} = M\vec{j} \tag{4}$$

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

$$\begin{pmatrix} t \\ \vec{v} \end{pmatrix} = MS\vec{j'} = \begin{pmatrix} \vec{\tau}.S \\ \Sigma S \end{pmatrix} \vec{j'} = \begin{pmatrix} \vec{\tau}.S \\ I_{n-1} \quad 0 \end{pmatrix} \vec{j'}$$

Clearly, $\vec{\tau}$ is a tight schedule with cluster shape $\vec{C}$ and mapping $\Sigma$ if and only if $\vec{\tau}.S$ 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 $S^{-1}$.

*Example* 2. 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 tableau represents the $4 \times 5$ array of virtual processors assigned to one physical processor. 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.)

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

We use the following method to construct a tight schedule that satisfies additional linear inequality constraints, as explained in Section 4. From the given linear inequality constraints, we derive bounds (through linear programming) for the components of $\vec{\tau}$. We construct tight schedules of the form given here, and consistent with these bounds. To do this, we simply enumerate a finite sequence of possible values for the parameters $k_i$ that satisfy the constraint of relative primality $(k_i \wedge C_i = 1)$ and such that the resulting element $\tau_i$ of $\vec{\tau}$ is within the bounds just determined. We try all of the allowed permutations of the elements of $\vec{C}$ and $\vec{\tau}$. Every choice of the parameters $k_i$ yields a tight schedule. We admit only those schedules that satisfy the full system of linear inequality constraints; finally, we choose one of the admissible schedules according to a criterion that measures total schedule length and estimates hardware cost.

For later use, we need to record an important property 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 a lower triangular matrix whose diagonal is $(1, C_1, \ldots, C_{n-1})$ (up to the permutation of $\vec{C}$ used in its construction).

## 7. REDUCING THE COST OF CONTROL

After transformation into synchronous, parallel form, the loop body serves as a specification of the special-purpose processor. The nest has a sequential outer 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 position $\vec{c}$ of the currently active VP within the cluster: $0 \leq c_k < C_k$.

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

*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 $\vec{j} = M^{-1} \begin{pmatrix} t \\ \vec{v} \end{pmatrix}$.

*Memory addresses.* When a value 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 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 $\vec{C}$) and iteration-space predicates (that test the global iteration coordinates against the limits of the iteration space).

As an example, for a two-dimensional processor array, the loop has the following form:

```
for (t = TMIN; t <= TMAX; t++) {
  for (p1 = 0; p1 < P1; p1++) in parallel {
    for (p2 = 0; p2 < P2; p2++) in parallel {
      Calculate the cluster coordinates
         c = (c1, c2)
         of the active virtual processor;
      Calculate the global VP coordinates
         v = (p1 C1 + c1, p2 C2 + c2)
         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;
      }
    }
  }
}
```

Our first experiments with processor synthesis revealed that these housekeeping computations were so costly that the resulting processor was grossly inefficient. The large number of comparisons for predicates was a big contributor. We then observed that almost all of these repeat on a given processor with period $\gamma$, and that they can therefore be obtained from a $\gamma$-bit circular buffer. We use this

technique in our current implementations with good results. A second and more important inefficiency is 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 [Ancourt and Irigoin 1991] 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 $M$ be the mapping matrix of (4), let $H_m$ be its Hermite form, and let $T$ be a unimodular matrix such that $MT = H_m$. Then $\begin{pmatrix} t \\ \vec{v} \end{pmatrix} = M\vec{j} = H_m T^{-1}\vec{j} = H_m \vec{j_T}$ where $\vec{j_T}$ is integer. Furthermore, we know (see end of Section 6) 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 $\vec{j_T}$ 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 $\vec{p}$. 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 [Darte, Schreiber, Rau, and Vivien 1999]), 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.

### 7.1 A general updating scheme

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 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 $\gamma$, 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 $\Sigma$ consists of the first $(n-1)$ 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 $\gamma$ values of any coordinate or related quantity that we wish to infer by this $\gamma$-order recurrence. When $\gamma$ 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 $\delta t$ (provided that $\delta t$ 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 $\delta t$, $\vec{C}$ 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. We now explain how we can automatically generate this decision tree.

*Back to Example 2.* By examining the activity tableau, we see that to move forward in time by one cycle, $c_1$ increases by 3 (if this is possible, *i.e.,* if we start with $c_1 = 0$) or else decreases by 1. In the former case, the move is always straight up, that is, $c_2$ remains unchanged. In the latter case, the change to $c_2$ is either 2 or $-3$. Note that the two potential changes to a coordinate always take opposite signs and have magnitudes that sum to the cluster shape, so that for any given position in the cluster, only one of them will lead to another point in the cluster.

It remains to show that these observations are true in general. Again assume for simplicity and without real loss of generality that $\Sigma$ consists of the first $(n-1)$ rows of the identity.

The activity times on some arbitrarily chosen processor $\vec{p}$ are shifted by a constant (equal to $\vec{\tau}.(p_1C_1,\ldots,p_{n-1}C_{n-1})$) compared with the times on processor zero. This implies that for a given $\delta t$, the path through the activity tableau is the same on all processors, and the same decision tree may therefore be used. This is vital, because software pipelining and hardware synthesis are simplified when there is one loop body common to all processors. We can therefore now restrict our attention to the processors whose coordinates are equal to 0.

Let $\vec{j}$ be any iteration mapped to processor zero. Suppose it is scheduled at time $t$ and mapped to the virtual processor whose cluster coordinates are $\vec{c}$. We are given the time lag $\delta t$. Since the schedule is tight, there is a unique vector $\delta\vec{j}$ such that iteration $\vec{j}+\delta\vec{j}$ is scheduled $\delta t$ cycles after iteration $\vec{j}$ on processor zero, *i.e.,* $M(\vec{j}+\delta\vec{j}) = (t+\delta t, \vec{c}+\delta\vec{c})^t$ where $\vec{c}+\delta\vec{c}$ is "in the box:"

$$0 \le \vec{c}+\delta\vec{c} < \vec{C}\,. \tag{5}$$

It follows that the cluster coordinate change vector satisfies

$$-\vec{C} < \delta\vec{c} < \vec{C}.$$

Recall the Hermite normal form of $M$, that is, $MT = H_m$. The columns of $T$ are a unimodular basis. Since the first row of $MT$ is $(1,0,\ldots,0)$, the first column

$\vec{t_1}$ of $T$ connects an isochrone (a hyperplane of iterations scheduled for the same time) to the next isochrone, and the remaining columns $\vec{t_2}, \ldots, \vec{t_n}$ are a basis for the lattice of iterations in an isochrone.

Because a move in the iteration space in the direction $\delta\vec{j}$ moves time forward by $\delta t$ cycles, $\delta\vec{j}$ is the sum of $\delta t \times \vec{t_1}$ and a linear combination of $\vec{t_2}, \ldots, \vec{t_n}$. In the shifted lattice generated in this way, there is a unique point that maps to processor zero. We now show how it is determined.

We make a change a variables using the basis given by the columns of $T$, defining

$$\vec{j_T} = T^{-1}\vec{j}.$$

Then

$$\begin{pmatrix} \delta t \\ \delta\vec{c} \end{pmatrix} = M\delta\vec{j} = H_m\delta\vec{j_T}.$$

Now we exploit the structure of $H_m$ discussed above. It is lower triangular, having one in the (1,1) position and the elements of $\vec{C}$ along the remainder of the diagonal. The elements to the left of the diagonal have smaller absolute value than the diagonal element to their right. Clearly, the first element of $\delta\vec{j_T}$ must be $\delta t$. We proceed to solve the lower triangular system by using the bounds (5). As $\vec{c}$ varies over the whole cluster, there can be only two possible choices for the second component of $\delta\vec{j_T}$, because the $(2,2)$ element of $H_m$ is $C_2$. For each of these two choices, there can be at most two choices of the third component, and so on. This is how the decision tree is generated.

Having generated all the possible moves in the transformed coordinates $\vec{j_T}$, we may apply $T$ to get the possible moves in the iteration space coordinates $\vec{j}$, and then multiply by $\Sigma$ to get the potential moves in the virtual processor space.

*Back to Example 2.* We take $\delta t = 1$ again. The Hermite form of the mapping ($MT = H_m$) is

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

Recall that the cluster size is $C(1) = 4$ and $C(2) = 5$. From the first column of $T$, we read that a move along move along $\vec{t_1} = (3, 0, -1)$ in the iteration space moves to the next isochrone. (This corresponds to requiring the first element of $\vec{j_T}$ to be $\delta t$.) Because of the structure of $\Sigma$, this move corresponds to the move $(3, 0)$ in the cluster. If $c_1 + 3 \geq 4$, *i.e.,* we have $c_1 > 0$, then this move would take us outside the cluster; in that case, we subtract the second column of $T$, *i.e.,* $(4, 3, -2)$; this yields the change $(-1, -3, 1)$ in the iteration space and a cluster coordinate change vector $\delta\vec{c} = (-1, -3)$. (Thus, there were two possibilities for the second element of $\vec{j_T}$, namely 0 and -1.) If $c_2 - 3 < 0$, *i.e.,* then this move takes us out of the cluster. The correct move in that case is found by adding the last column: $(3, 0, -1) - (4, 3, -2) + (0, 5, -1) = (-1, 2, 0)$.

With this technique, we get the following decision tree:

```
if (c(1) + 3 < 4) { /* out in dim. 1? */
   c(1) = c(1) + 3;
   /* c(2) is unchanged, c(2) = c(2) + 0 */
}
else {
   c(1) = c(1) - 1;
   if (c(2) + 2 < 5) { /* out in dim. 2? */
      c(2) = c(2) + 2;
   }
   else {
      c(2) = c(2) - 3;
   }
}
```

The technique works the same way for arbitrary $\delta t$. You begin with the change $\delta t \times \vec{t_1}$, and "correct" it as necessary with the remaining columns of $H_m$ in order to find the tree 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. However, the decision-tree recurrence has a certain computational latency. To be usable, we need that $\delta t$ times the desired II be at least equal to this latency; to minimize the required storage, we take the smallest usable $\delta t$.

### 7.2 Measuring the cost of housekeeping code

We show here the results of loop transformation with our efficient recurrence scheme for cluster coordinates and those parameters that depend on them linearly. We show three loop nests as test cases here. 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 [Darte, Schreiber, Rau, and Vivien 1999], we present the original and the transformed loop nests for matrix multiplication as an illustrative example. 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.

| | Photography | | | Printing | | Matrix | |
|---|---|---|---|---|---|---|---|
| Op. | 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 |
| $=$ | 1 | 6 | 35 | 18 | 18 | 1 | 3 |

It is clear from this data that the housekeeping, due to parallelization and flattening the loop nest with a general linear schedule, 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.

## 8. CONCLUSION

The first part of this paper provided a simple characterization of all tight synchronous, parallel schedules, solving a longstanding problem in processor-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. The second part proposes 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 added computational cost due to parallelization and the use of a general linear schedule can be controlled 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. The techniques of this paper provide new, powerful tools for synchronous processor-array synthesis and for software pipelining of nested loops on a single or multicluster VLIW.

## References


Ancourt, C. and Irigoin, F. 1991. Scanning polyhedra with DO loops. In *3rd ACM Symp. on Principles and Practice of Parallel Programming* (Apr. 1991). 39–50.

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

Darte, A. and Delosme, J.-M. 1990. Partitioning for array processors. Technical Report 90-23, LIP, ENS-Lyon.

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

Hajós, G. 1942. Uber einfache und mehrfache Bedeckung des $n$-dimensionalen Raumes mit einem Wurfelgitter. *Mathematische Zeitschrifte 47*, 427–467.

Megson, G. M. and Chen, X. 1995. A synthesis method of LSGP partitioning for given-shape regular arrays. In *Proc. 9th International Parallel Processing Symposium* (1995). 234–238.

Newman, M. 1972. *Integral Matrices.* Academic Press, New York.

Rau, B. R. 1996. Iterative modulo scheduling. *International Journal of Parallel Processing 24*, 3–64. Also available as HP Labs Tech. Report HPL-94-115.

Schreiber, R., Aditya, S., Rau, B. R., Kathail, V., Mahlke, S., Abraham, S., and Snider, G. 2000. High-level synthesis of nonprogrammable hardware accelerators. In *Proc. IEEE International Conference on Application-Specific Systems, Architectures, and Processors* (July 2000). 113–124.

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