Period Assignment in Multidimensional Periodic Scheduling

Wim F.J. Verhaegh¹  Emile H.L. Aarts¹,²  Paul C.N. van Gorp²

¹ Philips Research Laboratories, Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands
² Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract

We discuss the problem of assigning periods in multidimensional periodic scheduling such that storage costs are minimized. This problem originates from the design of high-throughput DSP systems, where highly parallel execution of loops is of utmost importance, and thus finding an optimal order of the loops is an important task.

We formulate the period assignment problem as a linear programming (LP) problem with some additional, non-linear constraints. The non-linear constraints are handled by a branch-and-bound approach, whereas the LP-relaxation is handled by a constraint-generation technique. The effectiveness and efficiency of the approach are good, which is illustrated by means of some practical examples.

1 Introduction

Many high-throughput DSP algorithms contain nested loops and multidimensional arrays, describing repetitive executions of operations and repetitive production and consumption of data. In addition, video signal processing algorithms contain strict timing requirements, constraining the rates at which input data arrives and the rates at which output data must be produced. In order to handle these characteristics, we introduced a model of multidimensional periodic operations [6, 7], which is the mathematical backbone of Phideo [2], a design methodology aimed at the automated design of dedicated digital video signal processors.

In this model, operations are considered to be executed repeatedly, with several dimensions of repetition, each of which corresponds to one loop. A specific execution of an operation can be identified by the corresponding values of the loop iterators. The time at which such an execution takes place is explicitly given in the model by means of the operation’s period vector, whose components denote the time between two consecutive iterations in each dimension of repetition, and its start time, which denotes the time of the first execution of the operation.

In the multidimensional periodic scheduling problem [6] we have to determine the operations’ period vectors and start times, and we have to assign the operations to processing units on which they are executed. Due to the high throughput, severe timing constraints, and high memory requirements, it is of utmost importance to choose the period vectors and start times such that a highly parallel implementation is obtained, in which the original loops are executed concurrently.

Since the multidimensional periodic scheduling problem is too complex to be handled in its entirety [6], we decompose it into two stages. In the first stage, which is the subject of this paper, we assign period vectors to all operations. In the second stage, which is the subject of [7], we assign start times to the operations and assign the operations to processing units.

The objective that we have in the period assignment stage is to minimize the storage cost, i.e., the maximum number of variables simultaneously alive. Other parts of the cost function, such as the processing unit cost, are not taken into account since no start time and processing unit assignment are performed yet, and since these parts of the cost function are in practice hardly influenced by the period assignment.

The constraints that we take into account in the period assignment stage are threefold. First, we have timing constraints, which bound the operations’ periods. Secondly, we have precedence constraints, which specify that data must be produced before it is consumed. Thirdly, we have lexicographical execution constraints, due to the type of controller used in Phideo, which specify that for each operation the period vector is such that it results in a proper loop nesting. Due to the precedence constraints it is necessary to also determine preliminary start times, which are also bounded by timing constraints, and which may be altered in the second stage when the operations are assigned to processing units.

1.1 Related work

In the literature, related work is done on control flow optimizations for storage minimization in DSP applications [1, 5], where multidimensional loops containing multidimensional arrays are automatically transformed, based on a method of placements of polytopes. However, this work does not address strict periodicity and strict timing requirements, and throughput is more an objective than a constraint.

Next, work is performed on multidimensional retiming [4], where one aims to pipeline a multidimensional loop such that a minimum repetition period is obtained, subject
to resource constraints. There, however, storage costs are not taken into account and the data dependencies that are allowed are limited (in array index expressions, iterators can only have a coefficient of 1). Furthermore, only a single (multidimensional) loop is considered at a time, instead of multiple loops that might partially overlap.

1.2 Paper outline

The objective of this paper is to present the period assignment problem, in which periods and start times have to be assigned such that storage costs are minimized, and to present a solution approach to solve it. The organization is as follows. In Section 2 we formulate the period assignment problem. In Section 3 we approximate the cost function in order to express it linearly in the periods and start times. Next, in Section 4 we introduce an LP-relaxation, which is solved by means of a constraint-generation technique. Based on this, we discuss a branch-and-bound approach, that can be used to find optimal solutions as well as good approximations. Finally, Section 5 shows some experimental results for practical problem instances.

2 The period assignment problem

In this section we discuss the period assignment problem. We do this by means of a small, fictitious example of a video algorithm, which is given in Figure 1.

```
for f = 0 to ∞ period 16
begin
  for j = 0 to 3 period 2
  {in}
    x[f][j+4] = input()
  {for}
  for k = 0 to 3 period in {−4..., 4}
    x[f][k] = func(x[f][k+4])
  {for}
  for l = 0 to 7 period 1
  {out}
    = output(x[f][l])
end
```

Figure 1. Example of a video algorithm, describing repetitive operations and data dependencies. Between curly braces the names of the operations are given.

2.1 Signal flow graphs

A video algorithm can be represented by a signal flow graph, of which the nodes denote operations and the edges denote data dependencies. Figure 2 shows the signal flow graph corresponding to the video algorithm of Figure 1.

```
in  fu  out
```

Figure 2. An abstract picture of a signal flow graph. Each operation is repeatedly executed; details of the iterations and of the data dependencies are not shown. The black dots denote the operations’ input and output ports.

The operations we consider may have several input and output ports. For reasons of simplicity, we assume in this paper that the consumption of input data and the production of output data all take place in the same clock cycle, and that the execution of an operation takes one clock cycle. Input and output operations of a signal flow graph are modeled as operations without input ports and output ports, respectively.

Next, we give a formal definition of a signal flow graph, after which we elaborate on its attributes.

Definition 1 (signal flow graph). A signal flow graph $G$ is given by a 5-tuple $(V, J, E, A, b)$, where

- $V$ is a set of multidimensional periodic operations,
- $I(v) \in \mathbb{N}_\infty^{d(v)}$ denotes the iterator bound vector, for each $v \in V$, where $\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$,
- $E \subseteq O \times I$ is a set of directed edges representing data dependencies, where $O$ is the set of all operations’ output ports and $I$ is the set of all operations’ input ports,
- $A(p) \in \mathbb{Z}^{u(p)} \times \mathbb{Z}^{v(p)}$ denotes the index matrix, for each input or output port $p$ of each operation $v$, and
- $b(p) \in \mathbb{Z}^{u(p)}$ denotes the index offset vector, for each port $p \in O \cup I$.

The dimension $d(v)$ of an operation $v$ denotes the number of enclosing loops. Each execution of an operation $v$ can be identified by the corresponding values of its loop iterators, which are combined in an iterator vector $i$. This iterator vector is bounded between the zero vector and the operation’s iterator bound vector, i.e., $0 \leq i \leq I(v)$. The set of all executions of an operation $v$ is denoted by

$$I(v) = \{ i \in \mathbb{Z}^{d(v)} | 0 \leq i \leq I(v) \}.$$  

DSP algorithms are usually repeated infinitely often, so we assume that dimension 0 of each operation has an unbounded number of repetitions, denoted by $I_0 = \infty$; the other dimensions are finite.

The data dependencies are modeled in a signal flow graph by means of the edge set $E$ and by describing at each output and input port the relation between the indices of the array that is used there, and the iterators of the corresponding operation. More detailed, at an output or input port $p$ of an operation $v$, the relation between the array’s index vector $n$ and the operation’s iterator vector $i$ is given by

$$n(p, i) = A(p) \cdot i + b(p).$$

The length $u(p)$ of the vector $n$ denotes the number of dimensions of the array. For example, at the input of operation $fu$ of Figure 1, the indices of the 2-dimensional array $x$ are given by

$$[n_0 \ n_1] = [f \ k + 4] = [1 \ 0 \ 0 \ 1 \ f \ k] + [0 \ 4].$$

For the index expressions we assume single assignments, i.e., each element of an array can be produced at most once.
2.2 Time assignments

A signal flow graph describes what operations have to be executed and what data is used. Next, a time assignment defines when they are executed.

Definition 2 (time assignment). Given a signal flow graph $G = (V, I, E, A, b)$, a time assignment $\tau$ is given by a pair $(p, s)$, where
- $p(v) \in \mathbb{Z}^{\delta(v)}$ is a period vector, for each $v \in V$, and
- $s(v) \in \mathbb{Z}$ is a start time, for each $v \in V$.

Given a time assignment, the time $c(v,i)$ at which an execution $i$ of operation $v$ takes place is given by

\[ c(v,i) = p^i(v) + s(v). \]

2.3 Constraints

Next, we define the three kinds of constraints that a time assignment has to satisfy. First, we have lower bounds $p_1(v) \in \mathbb{Z}_\omega^\delta(v)$ and upper bounds $p_2(v) \in \mathbb{Z}_\omega^\delta(v)$ on the period vectors of the operations $v \in V$, and lower bounds $s(v) \in \mathbb{Z}$ and upper bounds $\beta(v) \in \mathbb{Z}_\omega^\delta(v)$ on their start times, resulting in timing constraints that specify that

\[ p_1(v) \leq p(v) \leq p_2(v) \land s(v) \leq \beta(v), \]

for all $v \in V$. Here, $\mathbb{Z}_\omega = \mathbb{Z} \cup \{-\infty, +\infty\}$. For the outermost, infinite loop, we assume that the operations’ periods are fixed, positive, and all equal, i.e., $p_1(v) = p_2(v) = p_{inf} \in \mathbb{IN}$ for all operations $v \in V$.

Secondly, we have precedence constraints, which specify that data must be produced before it is consumed. So, if we have an execution $i$ of an operation $u$ and an execution $j$ of an operation $v$, and there is data going from an output $p$ of $u$ to an input $q$ of $v$, i.e., $(p, q) \in E$, and $n(p,i) = n(q,j)$, then we must have

\[ c(u,i) < c(v,j). \]

If there is such a data relation between an execution $i$ of a producing operation $u$ and an execution $j$ of a consuming operation $v$, then we denote this by

\[ (u,i) \rightarrow (v,j). \]

Thirdly, we have lexicographical execution constraints, which specify that for each operation the period vector is such that it results in a proper loop nesting. More formally, for each operation $v \in V$ this constraint, which we denote by $\text{lex}(v)$, specifies that there must be a permutation (absolute sorting) $p^'$ of the period vector $p(v)$ and a corresponding permutation $I'$ of the iterator bound vector $I(v)$ such that

\[ |p_k'| \geq \begin{cases} \frac{1}{|p_{k+1}'|}(I_{k+1}' + 1) & \text{for } k = \delta(v) - 1 \\ |p_k'| & \text{for } k = 0, \ldots, \delta(v) - 2. \end{cases} \]

For example, for an operation $v$ with

\[ p(v) = \begin{bmatrix} 90 \\ -2 \\ 10 \end{bmatrix}, \quad I(v) = \begin{bmatrix} \infty \\ 3 \\ 7 \end{bmatrix}, \]

$\text{lex}(v)$ holds, since after absolute sorting of the periods we obtain

\[ p^' = \begin{bmatrix} 90 \\ 10 \\ -2 \end{bmatrix}, \quad I' = \begin{bmatrix} \infty \\ 3 \\ 7 \end{bmatrix}, \]

and

\[ |p_1'| = 2 \geq 1, \quad |p_2'| = 10 \geq 2 \cdot 4 = |p_2'|(I_{k+1}' + 1), \quad |p_3'| = 90 \geq 10 \cdot 8 = |p_3'|\left(I_{k+1}' + 1\right). \]

The lexicographical execution constraints imply that the infinite loop has the largest period, and without loss of generality we thus may assume that $|p_k(v)|$ and $|p_{k+1}'(v)|$ are at most $p_{inf}/(I_{k}(v) + 1)$, for all $v \in V$ and $k = 1, \ldots, \delta(v) - 1$.

2.4 Storage cost

The final element of the model that we present is the cost function, which is given by the maximum number of variables simultaneously alive. To this end, we first partition the input and output ports into array clusters, each of which contains all ports that access a certain array. In other words, two ports $p, q \in O \cup I$ are in the same array cluster $A$, if and only if they access the same array. The set of all array clusters is denoted by $\mathcal{A}$.

Now we have to determine for each array cluster $A \in \mathcal{A}$ and for each clock cycle $c \in \mathbb{Z}$ how many elements of $A$ are alive in clock cycle $c$. Assuming that a variable is alive from the first clock cycle after its production up to and including the clock cycle of its last consumption, we have to determine which array elements have been produced before $c$ and which elements still have to be consumed at or after $c$. The former set is given by

\[ \mathcal{P}(A,c) = \{ n = A(p) \mid p \in O(u) \cap A \land i \in I(u) \land c(u,i) < c \}, \]

where $O(u)$ is the set of output ports of operation $u$. The latter set is given by

\[ \mathcal{C}(A,c) = \{ n = A(q) \mid \exists v \in V : q \in I(v) \cap A \land j \in I(v) \land c(v,j) \geq c \}, \]

where $I(v)$ is the set of input ports of operation $v$. Combining these two sets, the set of elements of array cluster $A$ that are alive in clock cycle $c$ is given by

\[ \mathcal{L}(A,c) = \mathcal{P}(A,c) \cap \mathcal{C}(A,c). \]

Now the cost of a time assignment $\tau$, i.e., the maximum number of variables simultaneously alive, is given by

\[ g(\tau) = \max_{c \in \mathbb{Z}} \sum_{c \in \mathbb{Z}} |\mathcal{L}(A,c)|. \]

For example of the storage cost, see Figure 3. There we have one array cluster $A$, containing all input and output ports, and at e.g. time $c = 8$ we have

\[ \mathcal{P}(A,8) = \{[0],[1],[2],[4],[5],[6],[7]\}, \]

and
3.1 Average number of variables

The first step we take to obtain a linear cost function is to switch from the maximum number of variables simultaneously alive to the average number. The rationale of this is that if the average number goes down, it is likely that also the maximum goes down, in practice. After a possible run-in effect of the algorithm, everything repeats with the global period \( p_{inf} \), so then the average number of variables simultaneously alive is given by the sum of the lifetimes of the variables that are produced during one iteration of the infinite loop, divided by \( p_{inf} \).

To determine the variables’ lifetimes we have to determine when the variables are produced and when they are consumed for the last time. The production times are easy to determine since they are just the times of the executions of the producing operations. If each variable is consumed exactly once, then also the consumption times are easy to determine; they are the times of the executions of the consuming operations. For an example, see Figure 4, where the sum of the variables’ lifetimes is equal to

\[
2 + 3 + 4 + 5 = (3 - 1) + (5 - 2) + (7 - 3) + (9 - 4)
\]

\[
= (3 + 5 + 7 + 9) - (1 + 2 + 3 + 4),
\]

i.e., the sum of the times of the executions of the consuming operation minus the sum of those of the producing operation.

Figure 4. An example where all variables are consumed exactly once.

For an operation \( v \), such a sum of the times of its executions in the first iteration of the infinite loop is given by

\[
C(v) = \sum_{i \in \mathbb{Z}(v), i_0=0} (p^i(v) i + s(v)).
\]

This is equal to the average time of these executions, i.e., the time of execution \( i \) with \( i_k = \delta(v) \) for \( k = 1, \ldots, \delta(v) - 1, i_0 = 0 \), multiplied by the number of executions. So, we can rewrite the above equation into

\[
C(v) = (s(v) + \frac{1}{2} \sum_{k=1}^{\delta(v)-1} p_k(v) I_k(v)) \cdot \prod_{k=1}^{\delta(v)-1} (I_k(v) + 1),
\]

which is linear in the periods and start time of \( v \). For example, for operation \( v \) in Figure 4, which has start time 3, period 2, and iterator bound 3, this equation gives us

\[
C(v) = (3 + \frac{1}{2} \cdot 2 \cdot 3) \cdot 4 = 24 = 3 + 5 + 7 + 9.
\]
should be at or after its production time and at or after each of its consumptions, so we get extra precedence constraints between productions and consumptions on one hand and annihilations on the other hand. For an example, see Figure 5.

![Figure 5. An example where variables are consumed multiple times, so a stop operation \( s \) is introduced to denote the ends of the variables’ lifetimes. The dashed lines denote the extra precedences.](image)

After adding stop operations and corresponding precedence edges, we obtain an extended signal flow graph.

**Definition 4 (extended signal flow graph).** Given a signal flow graph \( G = (V, I, E, A, b) \), an extended signal flow graph \( G' = (V', I', E', A, b) \) is a graph with:
- \( V' = V \cup V_s \), where \( V_s \) is a set of stop operations, and
- \( E' = E \cup E_s \), where \( E_s \) is a set of stop edges, i.e., edges from input and output ports \( p \in I \cup O \) to ports of the stop operations,

such that each produced variable is annihilated exactly once by a stop operation.

Initially, one can introduce for each production an equivalent stop operation, i.e., with the same iterator bounds and index matrix and offset vector, in order to have each variable annihilated exactly once. After that, the stop operations may be split in order to be able to align them better during scheduling with the ends of the variables’ lifetimes. For more details, see [6].

**3.3 Approximate period assignment problem**

With the stop operations, the average number of variables simultaneously alive is given by:

\[
g_{\text{avg}}(\tau) = \frac{1}{p_{\text{out}}} \left( \sum_{s \in V_s} C(s) - \sum_{v \in V} C(v) |O(v)| \right),
\]

in which each producing operation is counted as many times as it has output ports. Note that this function \( g_{\text{avg}} \) is linear in the periods and start times of the operations. We now get the following approximate problem.

**Definition 5 (approximate period assignment (APA)).** Given an extended signal flow graph \( G' = (V', I', E', A, b) \) and bounds on the operations’ period vectors and start times. Find a time assignment \( \tau = (p, s) \) that minimizes \( g_{\text{avg}}(\tau) \)

subject to:

\[
\begin{align*}
p(v) & \leq p(v) \leq \overline{p}(v) & \forall v \in V \\
\underline{p}(v) & \leq s(v) \leq \overline{s}(v) & \forall v \in V \\
c(u, i) & < c(v, j) + w(v) & \forall (u, i) \rightarrow (v, j) \\
\text{lex}(v) & < \forall v \in V \\
p(v), s(v) & \text{ integer} & \forall v \in V,
\end{align*}
\]

where \( w(v) = 1 \) if \( v \in V_s \) and 0 if \( v \in V \), to distinguish between precedences to stop operations and to original operations.

Note that the first two and the last two constraints of APA only have to hold for the original operations. Without proof, we mention that APA is also NP-hard.

**4 A branch-and-bound approach**

We apply a branch-and-bound method to solve the approximate period assignment problem. To this end, we discuss in Section 4.1 an LP-relaxation of the problem, after which we elaborate on how we branch and how we bound in Sections 4.2 and 4.3, respectively.

**4.1 An LP-relaxation**

Omitting the lexicographical execution constraints and the integrality constraints of APA, we obtain the following LP-relaxation.

\[
\begin{align*}
\text{minimize} & \quad g_{\text{avg}}(\tau) \\
\text{subject to} & \quad p(v) \leq p(v) \leq \overline{p}(v) & \forall v \in V \\
& \quad \underline{p}(v) \leq s(v) \leq \overline{s}(v) & \forall v \in V \\
& \quad c(u, i) < c(v, j) + w(v) & \forall (u, i) \rightarrow (v, j)
\end{align*}
\]

Unfortunately, we cannot directly solve the above problem by means of the simplex algorithm, since there is an exponential number of constraints due to precedences \( (u, i) \rightarrow (v, j) \). Nevertheless, since most of these constraints are redundant, we apply a constraint-generation technique that iteratively adds violated precedence constraints.

So, we start using none of the precedence constraints in the LP-relaxation, and apply the simplex method to find an optimal solution. After this, we check whether there exists a precedence \( (u, i) \rightarrow (v, j) \) for which \( c(u, i) < c(v, j) + w(v) \) is violated, which is done by solving a small ILP problem for each edge \( e \in E \) [7]. As soon as we encounter such a violated precedence, then we add the corresponding constraint to the LP-relaxation, and solve it again. Since we had an optimal solution before adding the extra constraint, we can do this by means of the dual simplex method. This procedure of checking and adding precedence constraints is repeated, until an optimum is found for which no precedence constraints are violated.

**4.2 Branching**

There are two kinds of non-linear constraints in APA, being the lexicographical execution constraints and the inte-
grality constraints, so we distinguish two ways of branching.

**Lexicographical branching.** If at a certain point we have a solution of the relaxed problem, and there is an operation \( v \in V \) for which \( \text{lex}(v) \) does not hold, then we apply a branching rule that determines which dimension of \( v \) will have the smallest period, which one will have the next smallest period, etc. So, the first time we apply such a branch for the operation \( v \), we choose a dimension \( k \in \{1,\ldots,\delta(v)-1\} \) for which we impose \( |p_k(v)| \geq 1 \). This is done by creating for each \( k \) two sub-problems: one with the constraint \( p_k(v) \geq 1 \) (positive period), and one with the constraint \( -p_k(v) \geq 1 \) (negative period).

The next time we apply such a branch for the same operation \( v \), we choose a dimension \( k \in \{1,\ldots,\delta(v)-1\} \) that was not previously chosen to be the dimension with the next smallest period. This is done by imposing \( |p_k(v)| \geq |p_l(v)|/\delta(v) + 1 \), where \( l \) denotes the latest chosen dimension used for branching of operation \( v \). Since the sign of period \( p_l(v) \) is known from the previous branch, which we denote by \( s_l(v) \in \{ -1, +1 \} \), we can impose the above constraint by introducing again two sub-problems: one with \( p_k(v) \geq s_l(v)p_l(v)(\delta(v) + 1) \), and one with \( -p_k(v) \geq s_l(v)p_l(v)(\delta(v) + 1) \).

For an example of the above branching rule, see Figure 6.

```
 p_1 \geq 1
 p_2 \geq 1
 p_3 \geq 1
 p_4 \geq 1
 p_5 \geq 1
 p_6 \leq 1
 p_7 \leq 1
 p_8 \leq 1
 p_9 \leq 1
```

Figure 6. The branching structure to obtain a lexicographical execution for an operation with iterator bound vector \([o \ 5 \ 3]^T\). In the nodes the constraints are shown that have been added to the original relaxation.

**Integrality branching.** The second branching rule is on the integrality constraints. If at a certain point we have a solution of the relaxed problem, for which a period or start time \( t \) has a non-integer value \( x \), then we trivially create two sub-problems: one with the additional constraint \( t \leq \lfloor x \rfloor \), and one with the additional constraint \( t \geq \lceil x \rceil \). We only have to do integrality branching on the periods, since if they are integer, the simplex method will also result in integer start times [6].

The priority between lexicographical branching and integrality branching is that we prefer the former over the latter, in order to have a faster branch-and-bound approach. This is based on the fact that, in practice, if the lexicographical execution constraints are met, then the integrality constraints are also met in most cases. Next, if more than one operation can be selected for lexicographical branching, then we preferably select the operation that was used in the previous branch, if applicable.

### 4.3 Bounding

In order to be able to cut branches from the solution tree, we have to determine for each relaxation a lower bound on the cost of any of the solutions in the corresponding subtree. A straightforward lower bound is given by the cost of the LP-relaxation, determined by the simplex algorithm. For the period assignment problem, however, we observed that it is better to descend slightly deeper in the sub-tree, until the level where the operation with which we started branching, is lexicographically executed. In other words, we descend until the level where a new operation is selected for branching on the lexicographical execution constraints, or where branching on the integrality constraints is applied. By then taking the minimum of the relaxed costs of all these sub-relaxations, we obtain a significantly better lower bound on the cost of the relaxation with which we started. Figure 7 shows an example of this.

![Figure 7](image)

Now, a sub-tree can be discarded if its lower bound \( b \) is equal to or larger than the cost \( U \) of the best solution found so far, which is an upper bound on the optimal solution. In this way, the branch-and-bound algorithm finds an optimal solution. The calculated bounds are also used to determine the order in which the tree of solutions is traversed; the most promising sub-tree is traversed first. Also here we benefit from the better lower bounds obtained by descending the tree slightly deeper, since they are better indications of the achievable solutions in the sub-trees.

With a slight modification, which is rather general, we can use the branch-and-bound algorithm as an approximation algorithm, in order to reduce the run time. To this end, we discard a sub-tree if

\[
b \geq \frac{U}{1 + \epsilon},
\]
for a given positive number $e$. In this way, the algorithm will find a solution that deviates at most a fraction $e$ from the optimum [6], so $e$ can be seen as a maximum allowed relative error. In this way, we can make a trade-off between the run time and the quality of the eventual solution.

5 Experimental results

The presented algorithm has been implemented in C++, and is incorporated in the Phideo tool set [2]. In this section, we show some experimental results. The presented run times are achieved on an HP9000/869. The first example is that of Figure 1, in which only one period is free, being period $p_1(fu)$, with bounds $p_1(fu) = -4$ and $p_1(fu) = 4$. The start times of the operations are bounded between -1 and 32. The resulting time assignment is the one shown in Figure 3, with $p_1(fu) = 1$, and with an average of 34/16 variables simultaneously alive. For this example, two stop operations were added. The period assignment stage took about 0.03 seconds. The constraint-generation part added 11 precedence constraints; no branching was required.

The second example is an example where every 16 clock cycles the sum of the elements of a 4 × 4 matrix has to be calculated; see Figure 8. All periods, except for the infinite loop, are bounded between 1 and 4; all start times between 0 and 32. For this example, initially 9 stop operations were added, but they could be removed since they exactly matched the consumptions; the latter were used to determine the times of annihilation. The period assignment stage took about 0.1 second, resulting in the solution shown in Figure 9. The optimal average number of variables is 45/16; the maximum number of variables of the obtained solution is 3. Note that this is also the optimal maximum number of variables, as formulated in the original problem PA, since the optimal maximum is bounded from below by the optimal average. For the initial LP-relaxation, 9 precedence constraints were added. After that, lexicographical branching was applied on the input operation in; the maximum depth reached in the solution tree was 2.

Next, we evaluate the effect of the maximum relative error on the quality of the period assignment and the run time, for three signal flow graphs. Some characteristics of these three instances are shown in Table 1. For these three instances we applied the period assignment algorithm for different values of the maximum relative error $e$. The periods in the instances CRD and LPC were restricted to positive values; the periods in the instance BPR had no restriction on their signs. Nevertheless, they are bounded by the optimal averages.

The final example we consider is a discrete cosine transformation [3], with 35 operations. The extended signal flow graph contains 38 stop operations, resulting in 153 periods and start times to be determined. Except for the input and output operation and for the infinite loop, all periods were totally free. The run time for the period assignment was 1 minute and 18 seconds. The maximum depth that was reached in the solution tree was 19, with maximally 299 constraints added to the original relaxation.

The solution found has an average of 66 variables simultaneously alive, and maximally 73; see Figure 10. The orig-

| instance | $|V|$ | $|V_0|$ | # p & s |
|----------|------|--------|---------|
| CRD      | 6    | 8      | 32      |
| LPC      | 8    | 16     | 49      |
| BPR      | 7    | 20     | 58      |
Table 2. Results of the period assignment runs for the instances CRD, LPC, and BPR, for different values of the specified maximum relative error $e$. Shown are the average number of variables (avg), the maximum number of variables (max), the maximum number of constraints that has been added to the initial relaxation (cons), the maximum depth in the solution tree (depth), and the run time in seconds for the period assignment ($t$).

<table>
<thead>
<tr>
<th>instance</th>
<th>$e$</th>
<th>avg</th>
<th>max</th>
<th>cons</th>
<th>depth</th>
<th>$t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRD</td>
<td>¥</td>
<td>29.8</td>
<td>36</td>
<td>42</td>
<td>3</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>19.2</td>
<td>22</td>
<td>44</td>
<td>3</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>19.2</td>
<td>22</td>
<td>44</td>
<td>3</td>
<td>0.9</td>
</tr>
<tr>
<td>LPC</td>
<td>¥</td>
<td>8.7</td>
<td>21</td>
<td>69</td>
<td>5</td>
<td>1.6</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>8.7</td>
<td>21</td>
<td>69</td>
<td>5</td>
<td>2.3</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>8.7</td>
<td>21</td>
<td>69</td>
<td>5</td>
<td>2.3</td>
</tr>
<tr>
<td>BPR</td>
<td>¥</td>
<td>4577</td>
<td>4605</td>
<td>115</td>
<td>3</td>
<td>5.9</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>4577</td>
<td>4605</td>
<td>117</td>
<td>3</td>
<td>12.9</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>4574</td>
<td>4605</td>
<td>117</td>
<td>3</td>
<td>14.9</td>
</tr>
</tbody>
</table>

The algorithm has been incorporated in the design methodology Phideo [2], which is currently used by several design groups within Philips.

References