

6 OVERLAPPED SCHEDULING TECHNIQUES FOR HIGH-LEVEL SYNTHESIS AND MULTIPROCESSOR REALIZATIONS OF DSP ALGORITHMS

Sabih H. Gerez
Sonia M. Heemstra de Groot
Erwin R. Bonsma
Marc J.M. Heijligers

6.1 Introduction

Algorithms that contain computations that can be executed simultaneously, offer possibilities of exploiting the parallelism present by implementing them on appropriate hardware, such as a multiprocessor system or an application-specific integrated circuit (ASIC). Many digital signal processing (DSP) algorithms contain internal parallelism and are besides meant to be repeated infinitely (or a large number of times). These algorithms, therefore, not only have *intra-iteration* parallelism (between operations belonging to the same iteration) but *inter-iteration* parallelism (between operations belonging to different iterations) as well [Par91].

The distribution in time of operations in the realization of an algorithm is called a *schedule*. A schedule in which all operations of a single iteration should terminate before any of the next iteration can be started is called *nonoverlapped*. On the other hand, schedules that exploit inter-iteration parallelism are called *overlapped* because the execution of consecutive iterations overlap in time. The two types of schedules are illustrated in Figure 6.1. In the figure, i and $i + 1$ refer to the successive iteration numbers, while T_0 is the length of the iteration period (see Section 6.2). The set of

computations belonging to the same iteration have the same shading; individual computations are not shown.

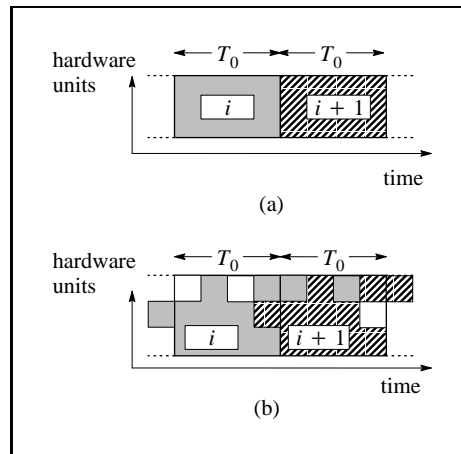


Figure 6.1. A nonoverlapped(a) and an overlapped (b) schedule

Overlapped scheduling is also called *loop folding* [Goo90] or *software pipelining* [Lam88, Jon90]. As opposed to *hardware pipelining*, that allows for overlapping the execution of subsequent operations within the same *functional unit* (FU) by an appropriate hardware design, software pipelining overlaps the execution of a set of operations in an iteration by scheduling them appropriately on the available FUs.

This chapter deals with techniques to obtain overlapped schedules for a specific class of DSP algorithms. The chapter has a tutorial character: the main ideas are explained and references to the literature are provided. On the other hand, no experimental results are presented: they can be found in the cited literature.

The chapter is organized as follows. The first sections introduce and define the problem. Then attention is paid to the theoretical lower bound on the iteration period of an overlapped schedule. Finally, different scheduling techniques are discussed.

6.2 Data-Flow Concepts

A convenient representation for parallel computations is the *data-flow graph* (DFG). A DFG consists of *nodes* and *edges*. An edge transports a stream of *tokens* each of which carries some data value. As soon as a sufficient number of tokens is present at the input edges of a node, the node consumes these tokens and produces tokens at its output edges with values corresponding to the operation that the node is supposed to perform. Edges act as first-in first-out buffers for the transported tokens.

A DFG can represent any computation [Dav82, Eij92, Lee95]. In this chapter, however, only DFGs without conditional constructs are considered. They are sufficiently powerful to model most traditional DSP algorithms (filters). DFGs that do not contain conditional constructs are called *synchronous* [Lee87]. Synchronous DFGs have the interesting property that they can be scheduled at compile time (the number of tokens consumed and produced by each node is always known) and no run-time overhead related to data-dependent scheduling is required. If one restricts each node in a synchronous DFG only to consume a single token from each input and produce a single token on each output per invocation, one gets a *homogeneous* DFG [Lee87]. Such DFGs will be called *iterative* DFGs (IDFGs) following the terminology of [Hee92]. This chapter mainly deals with IDFGs.

An IDFG is a tuple $\langle V, E \rangle$, where the vertex set V is the set of nodes and E is the set of edges of the graph. The set V can be partitioned in a set of *computational* nodes C , a set of *delay* nodes D , a set of *input* nodes I , and a set of *output* nodes O . A computational node $c \in C$ represents an *atomic* and *nonpreemptive* computation. ‘Atomic’ means that each node represents a computation of the lowest level of granularity, i.e. a computation corresponding to a single activation of an FU. ‘Non-preemptive’ means that a computation cannot be interrupted before completion to continue its execution at a later moment. It is further assumed that a single execution time $\delta(c)$ in the target architecture can be associated with each node c ; this time is expressed in integer multiples of the system clock period. Hardware pipelined FUs are not considered in this chapter. Including them would require characterization by a second entity called the *data-initiation interval*, the time necessary to execute a pipeline stage. A delay node $d \in D$ stores a token received at its input during one *iteration period* T_0 before producing it at its output. Again, T_0 is an integer multiple of the system clock period. In a more general model, a delay node can delay a token for more than one iteration period. The number of periods that a node $d \in D$ delays its input, is indicated by $\mu(d)$. Instead of using explicit delay nodes, one could as well define delay as an edge attribute. The delay multiplicity then corresponds to the number of *initial tokens* on the edge.

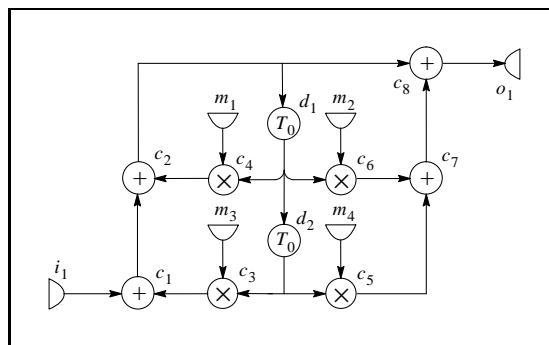


Figure 6.2. The IDFG of a second-order filter section

Although this model has some advantages from a mathematical point of view, the model with explicit delay nodes will be used here in order to be consistent with many earlier publications. An input node only produces tokens and an output node only consumes them. Figure 6.2 shows an example IDFG that represents a second-order filter section. In the figure, the node with the label T_0 represents the delay element. The nodes m_1 to m_4 that provide the constant multiplication coefficients, can be considered equivalent to inputs for the discussion in this chapter. Note that each cycle in an IDFG should at least contain a delay element. Deadlock would otherwise prevent the progress of computation in the IDFG (in terms of token flow: nodes in a cycle will never be activated if there are no initial tokens present in the cycle).

6.3. Models for the Target Architecture

Overlapped scheduling methods can be directed to different target architectures. The target architecture that is the easiest to work with from a theoretical point of view is the *ideal multiprocessor system*. It consists of a set of identical processors each of which can execute all operations in the IDFG. The time to transfer data between any pair of processors in an ideal multiprocessor system is negligible. This hardware model has been used in many studies on overlapped scheduling [Par91, Hee92, Gel93, Mad94] and will be the main model used in this chapter.

More realistic hardware models will explicitly deal with the multiprocessor network topology and the associated communication delays. Actual multiprocessor systems can roughly be divided in those that are suitable for *coarse-grain parallelism* and those suitable for *fine-grain parallelism*. In the former case, setting up a connection and transferring data will take tens of clock cycles, which means that tasks executed on the same processor should have a similar execution time in order to take advantage of the parallel hardware. Examples of scheduling approaches for DSP algorithms for this type of hardware are given in [Kon90, San96b]. Hardware that supports fine-grain parallelism, has the property that transferring a data item from one processor to the other takes just a few clock cycles. The multiprocessor system is then often integrated on a single integrated circuit [Che92, Kwe92] although systems composed of discrete commercially available digital signal processors also exist [Mad95]. A method that generates overlapped schedules for this type of architectures is discussed in Section 6.7.

Another target architecture is an *application-specific integrated circuit* (ASIC) that is the outcome of *high-level* (or *architectural*) *synthesis*. The architecture is then composed of FUs, such as adders and multipliers, for the actual calculations, registers or memories for the storage of intermediate results, and interconnection elements such as buses and multiplexers [Gaj92]. The types of architectures and clocking strategies considered normally allow the transfer of data within the same clock period as the one in which a computation terminates. This makes the scheduling problem for these architectures very similar to the one for the ideal multiprocessor system, the main difference being that FUs normally cannot execute all possible

operations (additions should e.g. be mapped on adders and multiplications on multipliers). Many overlapped scheduling methods for high-level synthesis have been proposed [Goo90, Olá92, Lee94, Kos95, Wan95, Hei96].

Yet another target architecture for overlapped scheduling is a *very-long instruction word* (VLIW) processor, a processor that contains a data path with multiple FUs that can be activated in a flexible way by appropriate parts of the instruction word. It is especially in this context that the term “software pipelining” is used [Lam88].

6.4 Definitions of the Problems

In this section, first some attention is paid to the terminology related to multiprocessor scheduling and high-level synthesis. Besides, some functions are introduced that will be used in the rest of this chapter. The optimization goals to be addressed are then defined.

6.4.1 Terminology and Definitions

Multiprocessor scheduling and high-level synthesis will map each operation in the IDFG to a time instant at which the operation should start, and to an FU on which the operation will be executed (of course, many more issues, such as the storage of intermediate values in registers, should also be settled). The mapping to a time instant is called *scheduling* (in the strict sense; “scheduling” is also often used for both mappings). The scheduling will be indicated by a function $\sigma : C \rightarrow Z$ (where Z is the set of integers). For a $c \in C$, $\sigma(c)$ represents the time instance at which c starts its execution with respect to the starting time of the iteration, which is zero by definition.

The set of FUs that will be present in the final solution, is F (a processor in a multiprocessor system will also be considered to be an FU). The set of all possible operation types (e.g. addition and multiplication) is Ω . The operation type of a node is given by the function $\gamma : C \rightarrow \Omega$ and the operations that an FU can execute by the function $\Gamma : F \rightarrow 2^\Omega$ (2^Ω is the *power set* of Ω , the set of all its subsets).

The mapping of a computational node to a specific FU is called *assignment* and is given by the function $\alpha : C \rightarrow F$. Clearly, for $c \in C$, it holds that $\gamma(c) \subset \Gamma(\alpha(c))$.

Unfortunately, no uniform terminology is used in the literature. Other terms used for “assignment” include *allocation* and *binding* (while “allocation” can also mean the reservation of hardware resources for synthesis without performing yet an actual mapping). The terminology used in this chapter follows the one of [Pot92].

Note that σ and α as defined above are independent of the iteration number. This means that the type of schedules considered in this paper are *static*. Although σ maps to values in Z , time values should be taken modulo T_0 when checking for re-

source conflicts on the same FU. This is a direct consequence of the overlapped scheduling strategy. If the values of α depend on the iteration number in a way that the FUs to which an operation is mapped, changes according to a cyclic pattern, the schedule is called *cyclostatic* [Sch85]. Figure 6.3 shows an example of a cyclostatic schedule. In the figure, the schedules of different iterations have been given different shadings. The superscripts of the operation labels refer to the iteration number. Note that the cyclostatic pattern has a period of two. Branch-and-bound methods for obtaining cyclostatic schedules, as well as for some variants, are described in [Gel93].

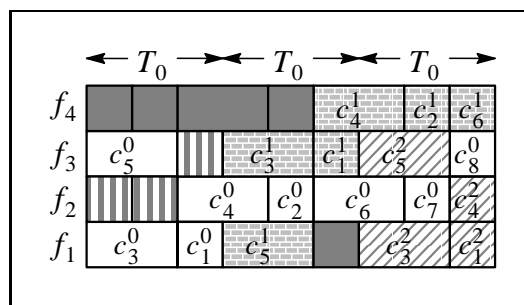


Figure 6.3. An example of a cyclostatic schedule

6.4.2 Optimization Goals

Traditionally, two optimization goals are distinguished for scheduling: one either fixes the iteration period and tries to minimize the hardware or one tries to minimize the iteration period for a given hardware configuration. The first problem is called the *time-constrained synthesis* problem, while the second is called the *resource-constrained synthesis* problem. In ASIC design, the optimization of yet a third entity, viz. *power*, is becoming more and more important. Overlapped scheduling can contribute to the reduction of power [Kim97]; this issue will, however, not be discussed further in this chapter.

In DSP, the iteration period T_0 of an algorithm is often part of the specifications and the problem to be solved is time constrained. Besides, as will become clear from the text below, T_0 should be provided in order to be able generate overlapped schedules. The resource-constrained problem can be solved by repetitively solving the time-constrained version of the problem and increasing T_0 until the resource constraints are satisfied. One can also increase T_0 during the scheduling process as is discussed in Section 6.7.

Apart from the iteration period, there is a second time entity that can be optimized. It is the *latency*, the time between the consumption of inputs and the produc-

tion of the corresponding outputs. Note, that the latency can be smaller or larger than or equal to T_0 in an overlapped schedule, while the latency is never larger than T_0 in a nonoverlapped schedule.

Apart from the just mentioned “optimization versions” of the scheduling problem, also the “decision version” may be of practical importance: finding out whether a solution exists for a given set of resources (and generate a solution if one exists). This problem could be tackled in a similar way as the resource-constrained problem.

6.5 Iteration Period Bound

This section deals with the *iteration period bound (IPB)* of an IDFG, the theoretical lower bound for T_0 derived from the IDFG topology. Below, attention will be paid to the IPB in nonoverlapped and overlapped schedules, and to an efficient method to compute the IPB.

6.5.1 Nonoverlapped Scheduling

The simplest case of an IDFG is one that does not contain any delay elements and is therefore *acyclic*. When the schedule is nonoverlapped, in such a case, the IPB is given by the longest path from any of the inputs of the IDFG to any of its outputs, where path length is the sum of the execution times $\delta(c)$ of all computational nodes c in the path. This longest path is called the *critical path* of the IDFG.

Longest paths in such *directed acyclic graphs (DAGs)* can be computed in linear time, i.e. in $O(|C| + |E|)$, using well-known shortest/longest-path algorithms from computational graph theory [Cor90].

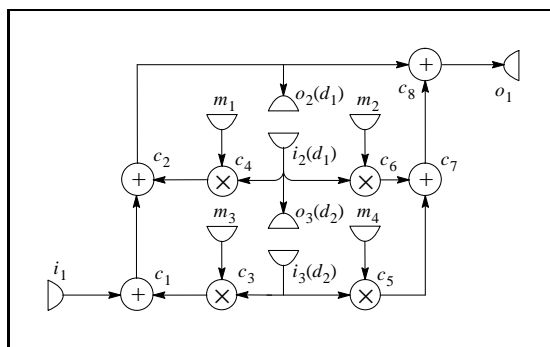


Figure 6.4. The IDFG of the second-order filter section after conversion to an acyclic graph

When the IDFG contains delay elements, these delay elements can be replaced by a pair of input and output nodes. As the scheduling is nonoverlapping anyhow, the fact that the delay element consumes a token in the current iteration and releases it in the next, can be modeled by sending the token to an output node in the current iteration and reading it from an input node in the next. Figure 6.4 shows the IDFG of Figure 6.2 after the conversion of each delay element into a pair of input and output nodes. Clearly, once that the delay nodes have been removed, the new graph is a DAG and the IPB can be computed using the linear-time longest-path algorithm mentioned above.

The fact that the IDFG contains delay elements, offers optimization possibilities using a transformation called *retiming* [Lei83, Lei91]. Retiming states that the behavior of a computation described by an IDFG does not change if delay elements connected to *all* inputs of a computational node are removed and replaced by delay elements connected to *all* outputs of the same computational node as is shown in Figure 6.5. If superscripts are again used to indicate the iteration number, one finds for Figure 6.5(a): $d^0 = a^0 + b^0$; $c^0 = d^{-1} = a^{-1} + b^{-1}$. In the case of Figure 6.5(b), one gets: $e^0 = a^{-1}$; $f^0 = b^{-1}$; $c^0 = e^0 + f^0 = a^{-1} + b^{-1}$, which shows the correctness of the transformation.

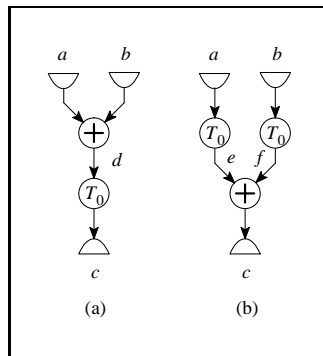


Figure 6.5. The retiming transformation

Retiming can be used to position the delay elements in the IDFG in such a way that the DAG obtained after replacing the delay elements by input-output node pairs has the smallest IPB. This can be achieved by a polynomial-time algorithm [Lei83, Lei91]. However, if resource minimization is a goal as well, the problem becomes NP-complete [Gar79] as is shown in [Pot94a].

6.5.2 Overlapped Scheduling

The value of T_0 in an IDFG without delay elements, which must be a DAG, can be arbitrarily short when overlapped schedules are allowed. New iterations can be

started at any desired time, as a computation does not depend on any of the output values of earlier iterations. This is not true if the DAG was obtained by splitting each delay node into an input and output node. Then, an iteration can only start at the moment that the output values that will be used as input for the actual iteration are available.

In general, the IPB for an overlapped realization of an algorithm is given by the following expression [Rei68, Ren81]:

$$\text{IPB} = \max_{\text{all cycles } L \text{ in the IDFG}} \frac{\sum_{c \in (L \cap C)} \delta(c)}{\sum_{d \in (L \cap D)} \mu(d)} \quad (1)$$

This equation can be understood as follows. In a cycle of the IDFG, an operation should wait until its output has propagated back to its input before it can be executed again. This is expressed in the numerator. However, this waiting time should be distributed among the total number of delay elements present in the cycle. This is expressed by the denominator. The cycle in the IDFG for which the quotient above has the highest value determines the IPB. It is called the *critical loop*. A scheduling for which T_0 equals the IPB is called *rate optimal*. Note that the expression in *Equation 1* is independent of any retiming of the IDFG (retiming does not modify the number of delay nodes in a cycle).

In Section 6.2 it was mentioned that T_0 should be an integer multiple of the system clock period, while the result of *Equation 1* may be a fraction. In case of a fraction, one should, therefore, use the smallest integer larger than the fraction given by *Equation 1*. Besides, the IPB of a static schedule cannot be smaller than the largest execution time of all operations in the IDFG:

$$\text{IPB} = \max \left(\left[\max_{\text{all cycles } L \text{ in the IDFG}} \frac{\sum_{c \in (L \cap C)} \delta(c)}{\sum_{d \in (L \cap D)} \mu(d)} \right], \max_{c \in C} \delta(c) \right) \quad (2)$$

In the rest of this chapter, the IPB as defined in *Equation 2* will be used and the term “rate optimal” will refer to this definition. It is, however, possible to realize scheduling solutions that meet a fractional bound given by *Equation 1* by *unfolding* [Par91] the IDFG, i.e. creating a new IDFG that represents multiple executions of the original IDFG [Jen94, San96a]. Cyclostatic scheduling (see Section 6.4.1) can also be used to achieve the IPB of *Equation 1*. Note that one can see a cyclostatic schedule as a special case of an unfolded one, viz. one with an unfolding factor equal to the cyclostatic period that should besides obey strict constraints on scheduling and assignment.

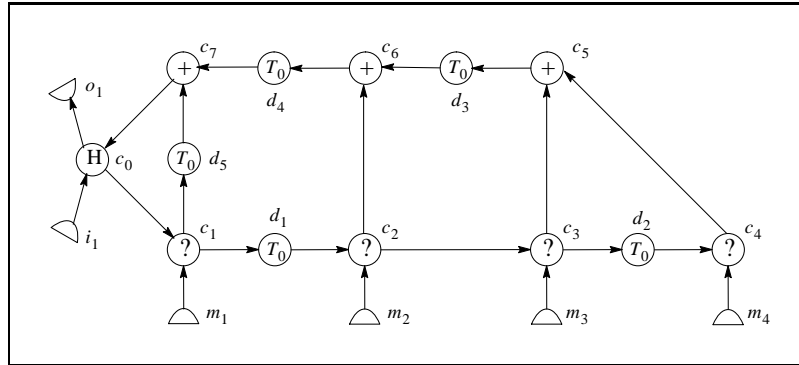


Figure 6.6. The optimally retimed correlator circuit

One might think that the IPB for a nonoverlapped schedule after optimal retiming is equal to the IPB for an overlapped schedule. This is *not* necessarily the case when the IDFG contains operations with an execution time larger than one clock period as will be illustrated by means of the example of Figure 6.6¹. The IDFG shown is the optimally retimed correlator from [Lei83, Lei91]. The IDFG consists of computational nodes for addition, comparison (indicated by the label ‘?’) and a host node that takes care of inputs and outputs (indicated by the label ‘H’; the main function of the “dummy” host node is to impose a limit on the latency). An addition takes 7 clock periods, a comparison 3, while the host node executes in 0 clock periods. It is not difficult to see that this circuit has a critical path consisting of c_2 , c_3 and c_5 , which results in an IPB of 13 if the schedule should be nonoverlapped.

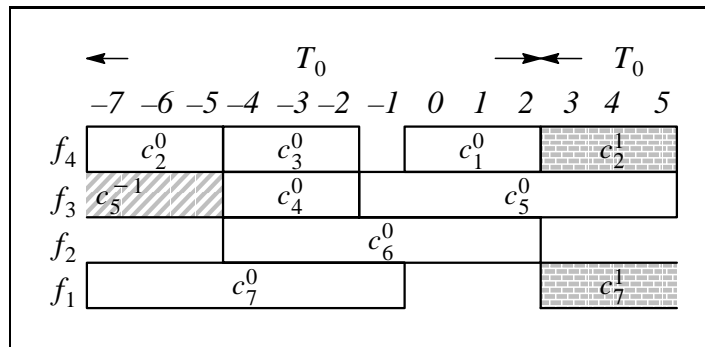


Figure 6.7. A rate-optimal overlapped schedule for the correlator

¹ The fact that the IPB of the optimally retimed correlator can be improved by choosing for an overlapped schedule, was discovered by András Oláh, currently employed by Ericsson, Hungary.

On the other hand, the circuit has an IPB of 10 for an overlapped schedule. This number follows from the three critical loops present ($c_0 - c_1 - c_7$; $c_0 - c_1 - c_2 - c_6 - c_7$; $c_0 - c_1 - c_2 - c_3 - c_5 - c_6 - c_7$). A rate-optimal schedule for an ideal multiprocessor system is shown in Figure 6.7.

It is in principle possible to realize a DSP algorithm at a speed higher than given by the IPB of *Equation 1*. This will require, however, modifications of the algorithm specifications and affect the finite word-length effects which may be quite important in DSP. Modifications involve transformations like look-ahead calculations that e.g. compute the next system state based on earlier states than the current one [Par87, Men87, Par89, Par95, Gle95].

6.5.3 Efficient Computation

The direct application of *Equation 1* (or 2) for the computation of the IPB would imply the enumeration of all cycles in the IDFG. The number of cycles can grow exponentially with respect to the number of nodes [Ger92] which implies an exponential worst-case time complexity for some graphs. Although IDFGs encountered in practice have a limited number of cycles and an enumerative approach seems to be feasible [Gel93, Wan95], it is important to notice that the problem can be solved in polynomial time. Many algorithms have been proposed for the IPB problem in IDFGs [Ger92, Kim92, Cha93, Pot94b, Ito95], some of them based on earlier solutions originally meant for other applications [Law66, Law76, Kar78]. Here, the most efficient of these methods will be explained in short.

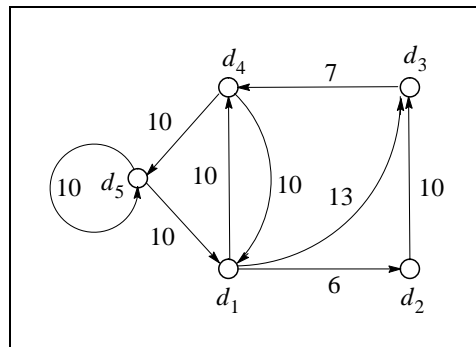


Figure 6.8. The delay graph of the optimally retimed correlator

It was originally meant to solve the *minimum cycle mean* problem [Kap78], where the “cycle mean” for a cycle in an edge-weighted graph refers to the total weight of the edges in the cycle divided by the number of edges constituting the cycle. In order to apply the minimum cycle mean method to the IPB problem for IDFGs, the original IDFG $\langle V, E \rangle$, should first be transformed into a *delay graph* $\langle D,$

E_d] [Ito95], of which the vertex set consists of the delay nodes in V . The edge set E_d of this graph consists of those (u, v_j) for which there is a directed path from delay node u to delay node v that passes through computational nodes only. The edge weight $w(u, v)$ of such an edge equals the total execution time of all nodes in the path. Figure 6.8 shows the delay graph of the optimally retimed correlator of Figure 6.6. Note that the IPB is equal to the *maximal* cycle mean of the delay graph.

The maximal cycle mean algorithm, adapted from the original minimal cycle mean algorithm, consists of the following steps:

- Choose an arbitrary node $s \in D$. Set $F_0(s) = 0$ and $F_0(v) = -\infty$, for all $v \in D, v \neq s$.
- Calculate $F_k(v)$, for $k = 1, \dots, |D|$ according to:

$$F_k(v) = \max_{(u,v) \in E_d} (F_{k-1}(u) + w(u, v))$$

- The IPB is then found from:

$$\text{IPB} = \max_{v \in D} \min_{0 \leq k \leq |D|-1} \frac{F_{|D|}(v) - F_k(v)}{|D| - k} \quad (3)$$

k	$F_k(d_1)$	$F_k(d_2)$	$F_k(d_3)$	$F_k(d_4)$	$F_k(d_5)$
0	$-\infty$	0	$-\infty$	$-\infty$	$-\infty$
1	$-\infty$	0	10	$-\infty$	$-\infty$
2	$-\infty$	0	10	17	$-\infty$
3	27	0	10	17	27
4	37	33	40	37	27
5	47	43	50	47	47

Figure 6.9. The step by step development of the maximum cycle mean algorithm

The application of this algorithm to the delay graph of Figure 6.8 is illustrated in Figure 6.9. The figure shows the values of $F_k(v)$ for all v and all k ; d_2 has been chosen as the arbitrary node s with which the algorithm starts. Equation 3 can now be applied to these values. It follows that IPB equals 10.

The construction of the delay graph from the IDFG can be achieved in $O(|D||E|)$ time using the algorithm for the construction of a *longest-path matrix* given in [Ger92]. The maximal cycle mean algorithm given above has a time complexity of $O(|D||E_d|)$. The overall time complexity of the IPB calculation method is therefore $O(|D|(|E| + |E_d|))$. Given the fact that all nodes in the IDFG have a bounded number of input edges (a multiplication or addition has e.g. two inputs), it can be stated that $|E| = O(|C|)$.

6.6 Mobility-Based Scheduling

The overlapped scheduling problem is NP-complete [Gar79, Hee92], which means that optimal solutions can only be found by algorithms that have an exponential time complexity in the worst case. Such algorithms can only be applied to small-size problems. For larger problems, one should use *heuristics* that will generate solutions that may not be optimal but can be obtained in acceptable time. One class of heuristics, the so-called *mobility-based* heuristics, are discussed in this section. First the notion of *scheduling ranges* is introduced. Then, heuristic scheduling techniques based on this notion are explained. Finally, some attention is paid to the assignment problem.

6.6.1 Scheduling Ranges

The scheduling solution as given by $\sigma(c)$ for $c \in C$ should obey in the first place the *precedence* constraints in the IDFG. For an edge $(u, v) \in E$, where $u, v \in C$, the precedence constraint implies that operation v cannot start its execution before the completion of operation u . If there is a path of n delay nodes between u and v , v cannot start before the execution of u belonging to n iterations ago is completed. In general, all precedence constraints of the IDFG for all pairs of computational nodes u and v separated by n delay nodes (n may be equal to zero) are given by:

$$\sigma(v) \geq \sigma(u) + \delta(u) - nT_0 \quad (4)$$

Note that $n = 0$ for intra-iteration precedence constraints and $n > 0$ for inter-iteration precedence constraints. All precedence constraints given by *Equation 4* can be represented in an *inequality graph* $\langle C, E_i \rangle$, using the following construction rules [Hee92]:

- The vertex set consists of the computational nodes C of the IDFG.
- Rewrite the inequalities of *Equation 4* as:

$$\sigma(v) - \sigma(u) \geq \delta(u) - nT_0 \quad (5)$$

- The edge set E_i has a directed edge (u, v) for each pair of nodes for which an inequality as given in *Equation 4* (or *5*) exists. The edge weight of the edge (u, v) is given by the right-hand side of *Equation 5*.

The inequality graph for the optimally retimed correlator of Figure 6.6 is given in Figure 6.10 when $T_0 = 10$. Note that the summation of the weights in each directed cycle is either zero (for critical loops) or negative. This is a direct consequence of the fact that T_0 should not be chosen smaller than the IPB in order for the IDFG to be computable.

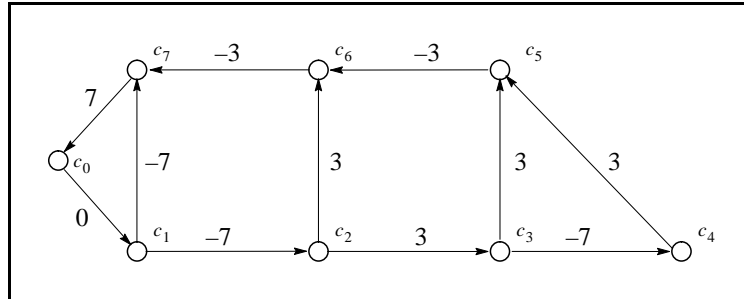


Figure 6.10. The inequality graph for the optimally retimed correlator circuit

The inequalities of Equation 4 can only be solved if the start time in the schedule of at least one node is known. Suppose that the start time $\sigma(r)$ of some reference operation $r \in C$ is set to zero. Then, the length of the *longest path* from r to some node v in the inequality graph gives the earliest possible moment at which v is allowed to start. This time is often called the *as-soon-as-possible* (ASAP) scheduling time. On the other hand, zero minus the length of the longest path from any node v to r gives the latest possible time at which v is allowed to start. This time is called the *as-late-as-possible* (ALAP) scheduling time. For each node the ASAP and ALAP times together define its *scheduling range* [Hee92] (or *mobility interval* [Pau89]) denoted by $[\sigma_-(v), \sigma_+(v)]$. The difference $\sigma_+(v) - \sigma_-(v)$ is called the *mobility* [Pau89] or *freedom* [Par86].

The computation of the scheduling ranges amounts to the *single-source longest-path problem* and can be solved by means of the *Bellman-Ford* algorithm [Cor90] in $O(|C||E_i|)$ time (for the ALAP times one should reverse the orientation of the edges [Hee92]). Because $|E_i|$ is $O(|C|)$ (see Section 6.5.3), the complexity of computing the ranges becomes $O(|C|^2)$. The Bellman-Ford algorithm is actually an algorithm for shortest paths, but can easily be modified to compute longest paths [Sch83].

The ranges for the operations c_0 through c_7 that can be derived from the inequality graph of Figure 6.10 if c_0 is chosen as the reference node, are respectively: $[0, 0]$, $[0, 0]$, $[-7, -7]$, $[-4, -4]$, $[-11, -4]$, $[-1, -1]$, $[-4, -4]$ and $[-7, -7]$. Note that c_4 is the only node with a mobility larger than zero because it is the only node that is not part of any critical loop. If T_0 is chosen equal to 11 instead of 10, the respective ranges become: $[0, 0]$, $[0, 1]$, $[-8, -6]$, $[-5, -2]$, $[-11, -4]$, $[-2, 1]$, $[-5, -3]$, $[-8, -7]$. Now, all operations except for the reference have some mobility.

So far, the scheduling ranges have been determined based solely on precedence relations from the IDFG. For heuristic scheduling methods, it is important that non-optimal solutions that are allowed by these scheduling ranges, are eliminated as much as possible, without excluding any optimal solution. This increases the probability that a heuristic finds an optimal solution. Scheduling ranges can be tightened by using *schedule constraints*. A simple example is to incorporate a latency con-

straint. This can be accomplished by adding input and output nodes to the inequality graph, fix the inputs at time zero, and the outputs at the latency time [Kos95]. This method has also the advantage that all scheduling ranges are finite (as long as each computational node has an incoming path connecting it to an input and an outgoing path connecting it to an output, which is the case in all IDFGs that make sense).

Another possibility, reported in [Tim93a], uses resource constraints to tighten the scheduling ranges. It is based on so-called *functional unit ranges* (or *module execution intervals*), which represent the range of system clock periods in which functional units should start to execute an operation. A bipartite graph G is constructed in which edges are placed between operation scheduling ranges and functional unit ranges which have clock periods in common. A feasible schedule implies that a complete bipartite matching in G exists. Edges which can never be part of such a matching can be deleted, and the operation scheduling ranges can be tightened accordingly.

In case only a resource constraint, an iteration period constraint, or a latency constraint has been given, new constraints can be generated by using lower-bound estimation techniques [Tim93b, Rim94]. These new constraints can be used to tighten scheduling ranges even more.

6.6.2 Mobility-Based Scheduling Heuristics

Below, the principles of mobility-based scheduling methods are explained without going into the details of the methods themselves. A few methods are mentioned and some attention is paid to an efficient way of “updating” scheduling ranges.

```

“determine the scheduling ranges”;
repeat
  “select an unscheduled operation  $c$  according to some rule”;
  repeat
    “select a time instance  $t$  in  $c$ ’s scheduling range
    according to some rule”;
    if “the selection satisfies all schedule constraints”
    then
      “schedule  $c$  at  $t$ ”;
    else
      “remove  $t$  from  $c$ ’s scheduling range”;
      “update the scheduling ranges of the other operations”;
    fi
  until “ $c$  has been scheduled”
until “all operations have been scheduled”;

```

Figure 6.11. The pseudo-code of a generic mobility-based scheduling heuristic

Given the fact that operations have mobility, one can say that the goal of the (time-constrained) scheduling problem is to fix each operation within its scheduling range in such a way that the resources necessary to implement the resulting schedule are minimized. The operations cannot be moved independently within their scheduling ranges: moving one operation may constrain the ranges of other operations. The pseudo-code of a generic mobility-based scheduling heuristic is given in Figure 6.11. Of course, the algorithm starts with the determination of the scheduling ranges applying the techniques mentioned in Section 6.6.1 and using as many constraints as possible. Many possibilities exist for selecting c and t in the algorithm. Examples of some algorithms are mentioned below. Two key activities in the algorithm, the check for scheduling constraint satisfaction and the updating of the ranges, determine the effectivity but also the complexity of the algorithm.

Mobility-based heuristics can both be used for nonoverlapped and overlapped scheduling and a heuristic developed for nonoverlapped scheduling [Par86] can often be easily adapted for the case of overlapped scheduling. A difference is that the computation of overlapped schedules requires that the resource requirements are computed after taking all times modulo T_0 . Besides, the updating of scheduling ranges is more complex in the case of overlapped scheduling.

The *force-directed* scheduling method [Pau89, Ver91, Ver92], is an example of an algorithm that originally was developed for nonoverlapped schedules. It can easily be adapted for overlapped scheduling [Olá92]. The method is computationally quite intensive as it investigates many different possibilities before taking a decision on fixing or constraining a single operation.

Greedy methods, on the other hand, take decisions on fixing an operation within its range without investigating many alternatives [Hee92, Kos95]. They have the advantage of a low computational complexity, but may generate solutions of lower quality.

Recently, the combination of *genetic algorithms* [Gol89, Dav91] and greedy heuristics have shown to give good results [Hei95, Hei96]. The idea is to have a greedy heuristic that can be controlled by a permutation of the operations in the IDFG. Simply stated, whenever the heuristic should select an operation to schedule among a set of candidates, it will choose the one that is first in the permutation. It is the task of the genetic algorithm to generate different permutations. For the genetic algorithm, the greedy heuristic is just an evaluation function that returns the cost of a permutation and generates a schedule as a side effect. Some more information on this method is given in Section 6.7 when discussing a generalization of this method target for architectures with communication delays.

All mobility-based heuristic have in common that scheduling ranges should be updated after fixing an operations start time or constraining its scheduling range. This can in principle be done by recomputing all ranges using the Bellman-Ford algorithm [Hee92] as mentioned in Section 6.6.1 (of course, the algorithm should be modified to respect the scheduling decisions already taken, which is straightforward). Then, each update will have a time complexity of $O(|C|^2)$ and any mobility-

based scheduling algorithm will have at least a time complexity of $O(|C|^3)$ as an update calculation is necessary after each scheduling decision.

The complexity of the range-updating problem can be reduced to $O(|C|)$ by means of a method presented in [Hei96] (similar ideas can also be found in [Lam89], where methods are used that deal with symbolic expressions in T_0). The main idea is to solve the *all-pairs longest-path* problem (see [Hee90] on how well-known shortest-path algorithms [Cor90] can be straightforwardly adapted for longest-path computations) for the inequality graph and store the results in a *inequality distance matrix* $\mathbf{D}_i[u, v]$, for all pairs $u, v \in C$. Each time that the range of some operation u is constrained by modifying either $\sigma_-(u)$ or $\sigma_+(u)$ (both are assigned the same value when the operation is fixed completely), the ranges of the remaining, yet unscheduled operations v can be updated as follows:

$$\begin{aligned}\sigma_-(v) &\leftarrow \max(\sigma_-(v), \sigma_-(u) + \mathbf{D}_i[u, v]) \\ \sigma_+(v) &\leftarrow \min(\sigma_+(v), \sigma_+(u) - \mathbf{D}_i[v, u])\end{aligned}$$

Clearly, a single update can be done in constant time and updating the ranges of all yet unscheduled operations can be done in $O(|C|)$ time. The initial effort to compute the inequality distance matrix can be limited to $O(|C|^2 \log |C|)$ using Johnson's algorithm [Cor90]. Actually, the matrix can be reused during different runs of a greedy algorithm, making this method of updating even more interesting than the one based on the Bellman-Ford algorithm.

Until now, it has been assumed that each operation in one iteration of the IDFG is executed exactly once. In case operations are enclosed by loop constructs, operations are executed multiple times during one iteration of the IDFG. In that case, *streams* [Mee93a] can be used to represent a particular sequence of executions of an operation, which are characterized by a vector representation. Scheduling these streams is defined as scheduling the first operation of such a stream inside its scheduling range, together with an extended constraint satisfaction and schedule range update, based upon specialized ILP techniques [Ver95, Ver97]. These methods are used in the *Phideo* high-level synthesis tool [Mee93a, Mee95].

6.6.3 Assignment

Scheduling algorithms try to minimize the number of resources to be used by the final solution e.g. keeping track of the number of operations that have to be executed simultaneously. In this section, some attention is paid on how to compute the actual number of resources required by mapping the operations on FUs (for the sake of simplicity, the ideal multiprocessor target architecture is assumed; if distinct FU types are present, the assignment problem should be solved separately for each type).

First the case of nonoverlapping scheduling is considered. After the completion of scheduling, it is known that all operations $c \in C$ occupy some FU during an *execution interval* $[\sigma(c), \sigma(c) + \delta(c) - 1]$. Operations whose intervals overlap must be assigned to distinct FUs. This problem can be modeled by a *conflict graph* $\langle C, E_c \rangle$ [Spr94] with as vertex set the set of computational nodes C and edges between those computational nodes whose execution intervals overlap. Because of the way it is constructed such a graph is called an *interval graph* [Gol80]. Suppose e.g. that the following intervals are given: $i_1 = [1, 4]$, $i_2 = [12, 15]$, $i_3 = [7, 13]$, $i_4 = [3, 8]$, $i_5 = [5, 10]$, $i_6 = [2, 6]$ and $i_7 = [9, 14]$. From these, the interval graph given in Figure 6.12(a) can be obtained.

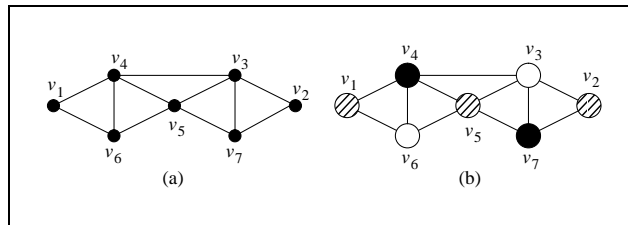


Figure 6.12. An interval graph (a) and its optimal coloring (b)

For any conflict graph, its *minimal vertex coloring* will result in optimal solution of the assignment problem. The vertex coloring problem for graphs in general is intractable [Gar79]. However, as a consequence of the way they are constructed, interval graphs form a special subset of all possible graphs that can be colored optimally in polynomial time by the *left-edge* algorithm, an algorithm that was originally published in the context of printed-circuit board routing [Has71]. The algorithm always finds a solution that uses as many FUs as the number of operations that are executed simultaneously. An optimal solution of the example just presented is shown in Figure 6.12(b).

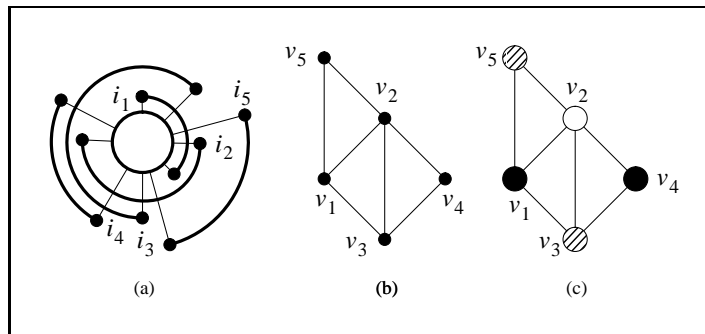


Figure 6.13. A set of circular arcs (a), its corresponding conflict graph (b) and the graph's optimal coloring (c)

In the case of overlapped scheduling, the FU occupancy cannot be modeled by linear intervals. Instead, *circular arcs* can be used to model the fact that occupancy may fold around the T_0 boundary. These arcs can be used to construct a conflict graph that is called a *circular-arc* graph. Figure 6.13 shows an example set of circular arcs, the corresponding circular-arc graph and a solution of the minimal coloring problem for this graph.

Unfortunately, the vertex coloring problem for circular-arc graphs is an NP-complete problem [Gar80]. Besides, a solution with a number of colors equal to the lower bound (the number of simultaneously executing FUs) may not exist. An algorithm that always finds the optimal solution within acceptable time for practical problems is presented in [Sto92]. A heuristic that performs very well in practice is described in [Hen93]. Both algorithms have been designed for the *register assignment* problem, the problem of mapping intermediate values that need to be stored during execution to a minimal number of registers. Note that even when nonoverlapped schedules are used for an iterative computation, the register occupancy will cross the T_0 boundary if an iteration needs results computed in previous iterations. Therefore, the register assignment problem amounts to circular-arc graph coloring even when nonoverlapped scheduling is used.

Another algorithm that also was originally developed for register assignment is described in [Mee93b]. It generates a cyclostatic assignment and in this way can deal with operations that have a longer execution time than T_0 . Besides, the algorithm has the pleasant property that it guarantees a solution with at most one more FU than the lower bound.

6.7 Target Architectures with Communication Delays

This section will present the principles of an overlapped scheduling method for a target architecture with nonnegligible communication delays suitable for fine-grain parallelism. A more detailed description of the approach can be found in [Bon97] (other approaches for the same problem are unknown to the authors).

The problem is resource constrained. The resources are given by an interconnection network graph $\langle F, L \rangle$ where F is the set of FUs and L is the set of links. Each link can transfer a single data item at a time and requires a time λ for the transport.

The solution is based on a layered approach similar to the genetic scheduling method proposed in [Hei95, Hei96] and shortly discussed in Section 6.6.2. The top-level consists of a genetic algorithm that generates permutations to control a greedy lower-level heuristic. However, the greedy heuristic itself consists of two layers: a *global heuristic* that deals with an abstraction of the interconnection network and a *black-box heuristic* that refines the decisions taken by the global heuristic by routing data across the network, taking link occupancy into account, etc. The method is illustrated in Figure 6.14. The method performs scheduling and assignment simulta-

neously. In the presence of communication delays, the scheduling needs assignment information in order to take sensible decisions.

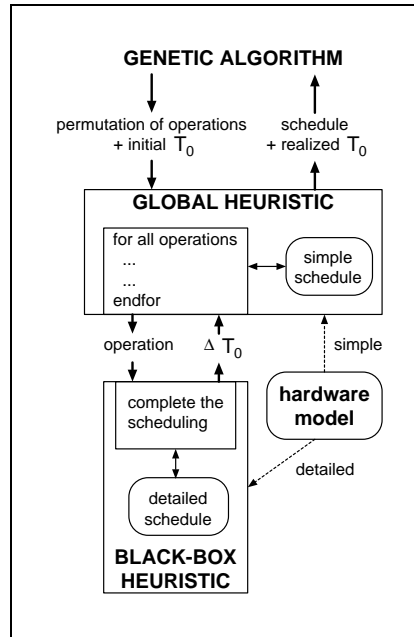


Figure 6.14. The three-layered scheduling approach

The scheduling method is based on mobility. However, the scheduling range as defined in Section 6.6.2 is not sufficient in the presence of communication delays. In order to deal with delays, the global heuristic not only uses the “inequality distance matrix” \mathbf{D}_i , but as well a *hardware distance matrix* \mathbf{D}_h . An entry $\mathbf{D}_h[f, g]$ of the matrix for two FUs $f, g \in F$ gives the shortest communication distance in system clock periods from f to g . Suppose that the computational nodes that already have been scheduled, are contained in the set S . Then the scheduling range $[\sigma_-(v), \sigma_+(v)]$ for the (tentative) assignment of a computational node v to an FU f is given by:

$$\sigma_-(v) = \max_{s \in S} (\sigma(s) + \mathbf{D}_i[s, v] + \mathbf{D}_h[\alpha(s), f])$$

$$\sigma_+(v) = \min_{s \in S} (\sigma(s) - \mathbf{D}_i[v, s] - \mathbf{D}_h[f, \alpha(s)])$$

It may now happen that a range is empty, i.e. that communication delays prevent the satisfaction of all precedence constraints. It may even happen that the ranges of all yet unscheduled operations for all possible assignments are empty. In such a case the algorithm performs *cycle insertion*, it increases T_0 with the goal to create valid

scheduling ranges when necessary. In order not to violate the greedy character of the algorithm, the already scheduled operations are not rescheduled. This illustrated in Figure 6.15.

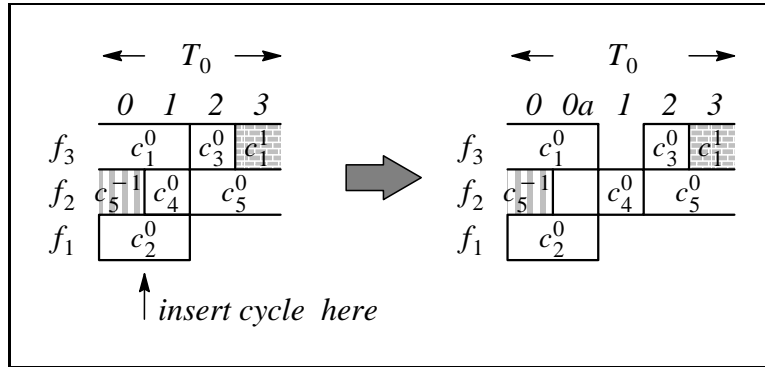


Figure 6.15. An example of cycle insertion

Both the global heuristic as well as the black-box heuristic can insert cycles when confronted with the impossibility of continuing scheduling and assignment. Clearly, it is the task of the genetic top layer to overcome the greedy nature of both heuristics and to try to find a globally optimal solution.

6.8 Conclusions

This chapter has introduced the notion of *overlapped scheduling*. As opposed to nonoverlapped scheduling, overlapped scheduling takes advantage of the parallelism present between multiple iterations in a repetitive algorithm. Many issues relevant to overlapped scheduling have been explained and a selection of important techniques has been presented.

References

[Bon97] Bonsma, E.R. and S.H. Gerez, "A Genetic Approach to the Overlapped Scheduling of Iterative Data-Flow Graphs for Target Architectures with Communication Delays", *ProRISC Workshop on Circuits, Systems and Signal Processing*, Mierlo, The Netherlands, (November 1997).

[Cha93] Chao, D.Y. and D.T. Wang, *Iteration Bounds of Single-Rate Data Flow Graphs for Concurrent Processing*, IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, Vol. 40(9), pp. 629-634, (September 1993).

- [Che92] Chen, D.C. and J.M. Rabaey, *A Reconfigurable Multiprocessor IC for Rapid Prototyping of Algorithmic-Specific High-Speed DSP Data Paths*, IEEE Journal of Solid-State Circuits, Vol. 27(12), pp. 1895-1904, (December 1992).
- [Cor90] Cormen, T.H., C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts, (1990).
- [Dav82] Davis, A.L. and R.M. Keller, *Data Flow Program Graphs*, IEEE Computer, pp. 26-41, (February 1982).
- [Dav91] Davis, L. (Ed.), *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York, (1991).
- [Eij92] Eijndhoven, J.T.J. van and L. Stok, "A Data Flow Graph Exchange Standard", *European Conference on Design Automation, EDAC '92*, pp. 193-199, (1992).
- [Gaj92] Gajski, D.D., N.D. Dutt, A.C.H. Wu and S.Y.L. Lin, *High-Level Synthesis, Introduction to Chip and System Design*, Kluwer Academic Publishers, Boston, (1992).
- [Gar79] Garey, M.R. and D.S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, San Francisco, (1979).
- [Gar80] Garey, M.R., D.S. Johnson, G.L. Miller and C.H. Papadimitriou, *The Complexity of Coloring Circular Arcs and Chords*, SIAM Journal on Algebraic and Discrete Methods, Vol. 1(2), pp. 216-227, (June 1980).
- [Gel93] Gelabert, P.R. and T.P. Barnwell III, *Optimal Automatic Periodic Multiprocessor Scheduler for Fully Specified Flow Graphs*, IEEE Transactions on Signal Processing, Vol. 41(2), pp. 858-888, (February 1993).
- [Ger92] Gerez, S.H., S.M. Heemstra de Groot and O.E. Herrmann, *A Polynomial-Time Algorithm for the Computation of the Iteration-Period Bound in Recursive Data-Flow Graphs*, IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, Vol. 39(1), pp. 49-52, (January 1992).
- [Gle95] Glentis, G.O. and S.H. Gerez, "Very High Speed Least Squares Adaptive Multichannel Filtering", *ProRISC/IEEE Benelux Workshop on Circuits, Systems and Signal Processing*, Mierlo, The Netherlands, pp. 123-132, (March 1995).
- [Gol80] Golubic, M.C., *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, (1980).
- [Gol89] Goldberg, D.E., *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Reading, Massachusetts, (1989).
- [Goo90] Goossens, G., J. Rabaey, J. Vandewalle and H. De Man, *An Efficient Microcode Compiler for Application Specific DSP Processors*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 9(9), pp. 925-937, (September 1990).
- [Has71] Hashimoto, A. and J. Stevens, "Wire Routing by Optimizing Channel Assignment within Large Apertures", *8th Design Automation Workshop*, pp. 155-169, (1971).
- [Hee90] Heemstra de Groot, S.M., *Scheduling Techniques for Iterative Data-Flow Graphs, An Approach Based on the Range Chart*, Ph.D. Thesis, University of Twente, Department of Electrical Engineering, (December 1990).
- [Hee92] Heemstra de Groot, S.M., S.H. Gerez and O.E. Herrmann, *Range-Chart-Guided Iterative Data-Flow-Graph Scheduling*, IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, Vol. 39(5), pp. 351-364, (May 1992).

- [Hei95] Heijligers, M.J.M. and J.A.G. Jess, "High-Level Synthesis Scheduling and Allocation Using Genetic Algorithms Based on Constructive Topological Scheduling Techniques", *International Conference on Evolutionary Computation*, Perth, Australia, (1995).
- [Hei96] Heijligers, M.J.M., *The Application of Genetic Algorithms to High-Level Synthesis*, Ph.D. Thesis, Eindhoven University of Technology, Department of Electrical Engineering, (October 1996).
- [Hen93] Hendren, L.J., G.R. Gao, E.R. Altman and C. Mukerji, *A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs*, *Journal of Programming Languages*, Vol. 1(3), pp. 155-185, (1993).
- [Ito95] Ito, K. and K.K. Parhi, *Determining the Minimum Iteration Period of an Algorithm*, *Journal of VLSI Signal Processing*, Vol. 11, pp. 229-244, (1995).
- [Jen94] Jeng, L.G. and L.G. Chen, *Rate-Optimal DSP Synthesis by Pipeline and Minimum Unfolding*, *IEEE Transactions on Very Large Scale Integration Systems*, Vol. 2(1), pp. 81-88, (March 1994).
- [Jon90] Jones, R.B. and V.H. Allan, "Software Pipelining: A Comparison and Improvement", *23rd Annual Workshop on Microprogramming and Microrarchitecture, MICRO 23*, pp. 46-56, (1990).
- [Kar78] Karp, R.M., *A Characterization of the Minimum Cycle Mean in a Digraph*, *Discrete Mathematics*, Vol. 23, pp. 309-311, (1978).
- [Kim92] Kim, J.Y. and H.S. Lee, *Lower Bound of Sample Word Length in Bit/Digit Serial Architectures*, *Electronics Letters*, Vol. 28(1), pp. 60-62, (January 1992).
- [Kim97] Kim, D. and K. Choi, "Power-Conscious High-Level Synthesis Using Loop Folding", *34th Design Automation Conference*, pp. 441-445, (1997).
- [Kon90] Konstantinides, K., R.T. Kaneshiro and J.R. Tani, *Task Allocation and Scheduling Models for Multiprocessor Digital Signal Processing*, *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. 38(12), pp. 2151-2161, (December 1990).
- [Kos95] Koster, M.S. and S.H. Gerez, "List Scheduling for Iterative Data-Flow Graphs", *GRONICS '95, Groningen Information Technology Conference for Students*, pp. 123-130, (February 1995).
- [Kwe92] Kwentus, A.Y., M.J. Werter and A.N. Willson, *A Programmable Digital Filter IC Employing Multiple Processors on a Single Chip*, *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 2(2), pp. 231-244, (June 1992).
- [Lam88] Lam, M., "Software Pipelining: An Effective Scheduling Technique for VLIW Machines", *SIGLAN '88 Conference on Programming Language Design and Implementation*, pp. 318-328, (June 1988).
- [Lam89] Lam, M.S., *A Systolic Array Optimizing Compiler*, Kluwer Academic Publishers, Boston, (1989).
- [Law66] Lawler, E.L., "Optimal Cycles in Doubly Weighted Directed Linear Graphs", *International Symposium on the Theory of Graphs*, Rome, pp. 209-213, (1966).
- [Law76] Lawler, E.L., *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, (1976).
- [Lee87] Lee, E.A. and D.G. Messerschmitt, *Synchronous Data Flow*, *Proceedings of the IEEE*, Vol. 75(9), pp. 1235-1245, (September 1987).
- [Lee94] Lee, T.F., A.C.H. Wu, Y.L. Lin and D.D. Gajski, *A Transformation-Based Method for Loop Folding*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 13(4), pp. 439-450, (April 1994).

- [Lee95] Lee, E.A. and T.M. Parks, *Dataflow Process Networks*, Proceedings of the IEEE, Vol. 83(5), pp. 773-799, (May 1995).
- [Lei83] Leiserson, C.E., F.M. Rose and J.B. Saxe, "Optimizing Synchronous Circuitry by Retiming (Preliminary Version)", In: R. Bryant (Ed.), *Third Caltech Conference on VLSI*, Springer Verlag, Berlin, pp. 87-116, (1983).
- [Lei91] Leiserson, C.E. and J.B. Saxe, *Retiming Synchronous Circuitry*, Algorithmica, Vol. 6, pp. 5-35, (1991).
- [Mad94] Madisetti, V.K. and B.A. Curtis, *A Quantitative Methodology for Rapid Prototyping and High-Level Synthesis of Signal Processing Algorithms*, IEEE Transactions on Signal Processing, Vol. 42(11), pp. 3188-3208, (November 1994).
- [Mad95] Madisetti, V.K., *VLSI Digital Signal Processors, An Introduction to Rapid Prototyping and Design Synthesis*, IEEE Press and Butterworth Heinemann, Boston, (1995).
- [Mee93a] Meerbergen, J. van, P. Lippens, B. McSweeney, W. Verhaegh, A. van der Werf and A. van Zanten, *Architectural Strategies for High-Throughput Applications*, Journal of VLSI Signal Processing, Vol. 5, pp. 201-220, (1993).
- [Mee93b] Meerbergen, J.L. van, P.E.R. Lippens, W.F.J. Verhaegh and A. van der Werf, "Relative Location Assignment for Repetitive Schedules", *European Conference on Design Automation with the European Event on ASIC Design, EDAC/EUROASIC*, pp. 403-407, (1993).
- [Mee95] Meerbergen, J.L. van, P.E.R. Lippens, W.F.J. Verhaegh and A. van der Werf, *PHIDEO: High-Level Synthesis for High Throughput Applications*, Journal of VLSI Signal Processing, Vol. 9, pp. 89-104, (1995).
- [Men87] Meng, T.H.Y. and D.G. Messerschmitt, *Arbitrarily High Sampling Rate Adaptive Filters*, IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. ASSP-35(4), pp. 455-470, (April 1987).
- [Olá92] Oláh, A., S.H. Gerez and S.M. Heemstra de Groot, "Scheduling and Allocation for the High-Level Synthesis of DSP Algorithms by Exploitation of Data Transfer Mobility", *International Conference on Computer Systems and Software Engineering, CompEuro 92*, pp. 145-150, (May 1992).
- [Par86] Parker, A.C., J.T. Pizarro and M. Mlinar, "MAHA: A Program for Datapath Synthesis", *23rd Design Automation Conference*, pp. 461-466, (1986).
- [Par87] Parhi, K.K. and D.G. Messerschmitt, *Concurrent Cellular VLSI Adaptive Filter Structures*, IEEE Transactions on Circuits and Systems, Vol. CAS-34(10), pp. 1141-1151, (October 1987).
- [Par89] Parhi, K.K., *Algorithm Transformation Techniques for Concurrent Processors*, Proceedings of the IEEE, Vol. 77(12), pp. 1879-1895, (December 1989).
- [Par91] Parhi, K.K. and D.G. Messerschmitt, *Static Rate-Optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding*, IEEE Transactions on Computers, Vol. 40(2), pp. 178-195, (February 1991).
- [Par95] Parhi, K.K., *High-Level Algorithm and Architecture Transformations for DSP Synthesis*, Journal of VLSI Signal Processing, Vol. 9, pp. 121-143, (1995).
- [Pau89] Paulin, P.G. and J.P. Knight, *Force-Directed Scheduling for the Behavioral Synthesis of ASICs*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 8(6), pp. 661-679, (June 1989).

- [Pot92] Potkonjak, M. and J.M. Rabaey, *Scheduling Algorithms for Hierarchical Data Control Flow Graphs*, International Journal of Circuit Theory and Applications, Vol. 20, pp. 217-233, (1992).
- [Pot94a] Potkonjak, M. and J. Rabaey, *Optimizing Resource Utilization Using Transformations*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 13(3), pp. 277-292, (March 1994).
- [Pot94b] Potkonjak, M. and J. Rabaey, *Optimizing Throughput and Resource Utilization Using Pipelining: Transformation Based Approach*, Journal of VLSI Signal Processing, Vol. 8, pp. 117-130, (1994).
- [Rei68] Reiter, R., *Scheduling Parallel Computations*, Journal of the ACM, Vol. 15(4), pp. 590-599, (October 1968).
- [Ren81] Renfors, M. and Y. Neuvo, *The Maximum Sampling Rate of Digital Filters Under Hardware Speed Constraints*, IEEE Transactions on Circuits and Systems, Vol. CAS-28(3), pp. 196-202, (March 1981).
- [Rim94] Rim, M. and R. Jain, *Lower-Bound Performance Estimation for the High-Level Synthesis Scheduling Problem*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 13(4), pp. 451-457, (April 1994).
- [San96a] Sanchez, F. and J. Cortadella, "Maximum-Throughput Software Pipelining", *2nd International Conference on Massively Parallel Computing Systems*, pp. 483-490, (May 1996).
- [San96b] Sanchez, J. and H. Barral, *Multiprocessor Implementation Models for Adaptive Algorithms*, IEEE Transactions on Signal Processing, Vol. 44(9), pp. 2319-2331, (September 1996).
- [Sch83] Schiele, W.L., "On a Longest Path Algorithm and Its Complexity If Applied to the Layout Compaction Problem", *European Conference on Circuit Theory and Design*, pp. 263-265, (1983).
- [Sch85] Schwartz, D.A., *Synchronous Multiprocessor Realizations of Shift-Invariant Flow Graphs*, Report no. DSPL-85-2, Ph.D. Thesis, Georgia Institute of Technology, School of Electrical Engineering, (June 1985).
- [Spr94] Springer, D.L. and D.E. Thomas, *Exploiting the Special Structure of Conflict and Compatibility Graphs in High-Level Synthesis*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 13(7), pp. 843-856, (July 1994).
- [Sto92] Stok, L. and J.A.G. Jess, *Foreground Memory Management in Data Path Synthesis*, International Journal of Circuit Theory and Applications, Vol. 20, pp. 235-255, (1992).
- [Tim93a] Timmer, A.H. and J.A.G. Jess, "Execution Interval Analysis under Resource Constraints", *International Conference on Computer-Aided Design*, pp. 454-459, (1993).
- [Tim93b] Timmer, A.H., M.J.M. Heijligers, L. Stok and J.A.G. Jess, "Module Selection and Scheduling Using Unrestricted Libraries", *European Design Automation Conference (EDAC/EUROASIC '93)*, pp. 547-551, (1993).
- [Ver91] Verhaegh, W.F.J., E.H.L. Aarts, J.H.M. Korst and P.E.R. Lippens, "Improved Force-Directed Scheduling", *European Design Automation Conference*, pp. 430-435, (1991).
- [Ver92] Verhaegh, W.F.J., P.E.R. Lippens, E.H.L. Aarts, J.H.M. Korst, A. van der Werf and J.L. van Meerbergen, "Efficiency Improvements for Force-Directed Scheduling", *International Conference on Computer-Aided Design*, pp. 286-291, (1992).

- [Ver95] Verhaegh, W.F.J., *Multidimensional Periodic Scheduling*, Ph.D. Thesis, Eindhoven University of Technology, Department of Applied Mathematics, (December 1995).
- [Ver97] Verhaegh, W.F.J., P.E.R. Lippens, E.H.L. Aarts and J.L. van Meerbergen, "Multidimensional Periodic Scheduling: A Solution Approach", *European Design and Test Conference, ED&TC '97*, pp. 468-474, (1997).
- [Wan95] Wang, C.Y. and K.K. Parhi, *High-Level DSP Synthesis Using Concurrent Transformations, Scheduling and Allocation*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 14(3), pp. 274-295, (March 1995).