Published in: GRONICS '95, Groningen Information Technology Conference for Students, pp. 123-130,
University of Groningen, The Netherlands, February 24, 1995.

List Scheduling for Iterative Data-Flow Graphs

Martin S. Koster*and Sabih H. Gerez
University of Twente, Department of Electrical Engineering
P.O. Box 217, 7500 AE Enschede, The Netherlands
e-mail: `t-martik@microsoft.com` or `s.h.gerez@el.utwente.nl`

Abstract

This paper presents a scheduling method for the high-level synthesis of DSP algorithms described by iterative data flow graphs with an iteration period of T_0. The method is based on the concept of an operation's *scheduling range*, i.e. the range of values where the execution of the operation is allowed to start given its successors and predecessors. The goal is to find the best position for each operation such that the hardware required by the schedule is as cheap as possible for a given T_0. This goal is attained by a *time-constrained list scheduling* heuristic. It generates *overlapped* schedules, while performing retiming and loop folding implicitly. Normally, a larger T_0 means a larger search space and longer CPU times for algorithms that generate a schedule. Experiments show, however, that the CPU times of the list-scheduling heuristic remain constant for increasing T_0 when applied to well-known benchmark examples. The quality of the results generated for these benchmarks is comparable to other results reported in literature.

1 Introduction

A research area in the field of design automation of digital systems that is receiving increasing attention in the last years is *high-level synthesis* (HLS): the mapping of a computation's behavioral description at the algorithmic level to a structural description at the register-transfer level [5, 17]. In high-level synthesis, among other tasks, each operation in the behavioral description has to be *scheduled* at a specific time and *assigned* to a specific hardware unit.

In digital signal processing (DSP) many algorithms have a repetitive nature: the same computation has to be executed for a set of inputs that arrives every T_0 time units, where T_0 is called the *sampling* or *iteration period.* When scheduling such an algorithm, one can produce *nonoverlapped* schedules where each T_0 time units only computations belonging to the same iteration are performed, or *overlapped* schedules where computations belonging to different iterations are executed in parallel. For

*Author's current address: Microsoft WPGI, Sandyford Industrial Estate, Dublin 18, Republic of Ireland.

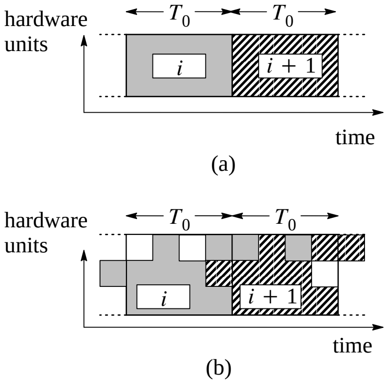

Figure 1: *A nonoverlapped (a) and an overlapped (b) schedule for an iterative algorithm.*

an overlapped schedule, apart from the parallelism in one execution of the algorithm (*intra-iteration* parallelism), the parallelism of operations belonging to different iterations (*inter-iteration* parallelism) should be taken into account during scheduling [20]. This is illustrated in Figure 1.

Intra-iteration and inter-iteration parallelism can be described by a set of inequalities the solution of which gives lower and upper bounds for the time positions where an operation can be scheduled. The difference between these bounds is called the operation's *mobility.* Mobility has been used in several high-level synthesis heuristics (see e.g. [21, 22]) for the generation of nonoverlapped schedules. This paper presents mobility-based heuristics for overlapped schedules. Overlapped scheduling of DSP algorithms based on mobility for a multiprocessor system (*homogeneous* hardware) as the target architecture has been proposed by Heemstra de Groot et al. [8, 9]. An extension of this work for high-level synthesis (with *heterogeneous* hardware) was presented by Oláh [18, 19] and later by Braaksma [1]. The work presented in this paper is a continuation of research in the last direction [13]. The main difference is that the heuristics used are based on *list scheduling* (see e.g. [5]) rather than on *force-directed scheduling* [22].

A DSP algorithm without data-dependent conditionals can be described by an *iterative data-flow graph* (IDFG) consisting of computational nodes, delay nodes and i/o nodes, with edges expressing precedence relations. When only nonoverlapped schedules are considered by a synthesis method, the delay nodes can be replaced by input and output nodes and the minimal T_0 is determined by the critical path in the resulting graph. Sometimes the lower bound on T_0 can be improved by *retiming* the IDFG [16]. Retiming is also used to optimize other objective functions [23]. In high-level synthesis systems that perform retiming, retiming is a step preceding scheduling [7, 23]. The scheduling method presented here considers all *essentially equivalent networks* [4] of the IDFG simultaneously making explicit retiming unnecessary.

In [6] a method called *loop folding* for the production of overlapped schedules is mentioned. It works by explicitly "moving" a number of operations from one iteration to another and then scheduling all operations within an interval of length T_0. In the method presented here, overlapped schedules are generated directly by considering time spans longer than T_0, while computing the resource utilization after taking all time instants $\bmod T_0$.

Other HLS systems that directly produce overlapped schedules include MARS [27] and Theda.Fold [15]. The MARS system enumerates all loops, which can theoretically be quite expensive as their number grows exponentially with respect to the size of the IDFG. The results on Theda.Fold were published when the work reported here was almost terminated. It is based on starting with a valid but possibly expensive schedule. The cost of the generated hardware is reduced by applying transformations. Very good results, both for the quality of the solutions as for the CPU-time needed to find them, have been reported.

2 Problem definition

A HLS algorithm takes a data-flow graph (DFG) and library of hardware units as its input and generates a data path and a controller as its output. A *hardware model* defines the way hardware units can be interconnected. A *cost function*, that guides the search for the optimal solution, is defined in terms of this hardware model. The controller can easily be derived from the information on *scheduling* and *assignment* (mapping of an operation to a time and hardware unit, respectively). Scheduling should respect the *precedence constraints* imposed by the DFG. This section pays attention to these distinct issues such that a clear definition of the problem becomes possible.

2.1 The hardware model

The HLS method presented in this paper uses a hardware model that is similar to the one of [19] (and many other systems reported in literature) with the

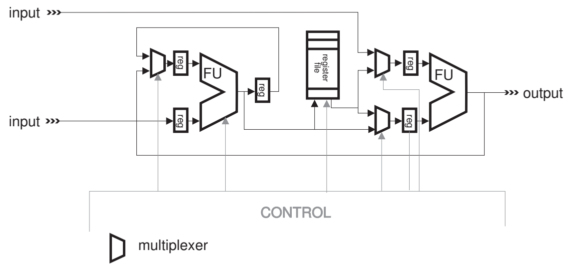

Figure 2: *An example of a hardware configuration obeying the hardware model.*

difference that no busses are used. The hardware consists of functional units (FU's), registers, multiplexers and wires. A configuration that obeys the model is shown in Figure 2. Remark that the multiplexers may be necessary to select the right input for an FU. The output of an FU can be fed to another FU, a single register or a register file.

Operations are executed on FU's to be taken from a *module library.* This library can consist of specialized hardware such as adders and multipliers and/or ALU's that can execute multiple operations. However, it is assumed that a unique mapping is provided from each operation in the IDFG to the FU on which it will be executed.

Registers are used to store results that are not immediately required by a consecutive operation. When asynchronous FU types are used, external registers can be used to make them synchronous. The registers can be placed at their inputs or outputs.

A single-phase clock cycle is used. One of the clock edges triggers master-slave updates in the registers, the other edge the start of computations and/or data transfers.

The omission of the busses is due to the fact that a bus optimization mechanism as proposed in [19] has not been implemented yet. This does not mean a bus cannot be used inside the target architecture, but the scheduler does not take into account the special characteristics of busses like bus occupation and optimization.

2.2 The iterative data-flow graph

The DSP algorithm to be synthesized by the system should be described as an IDFG. An IDFG is a *synchronous homogeneous data-flow graph* [14], IDFG(V, E). $V = C \cup D \cup I \cup O \cup M$ is the set of vertices (C, D, I, O, and M are pairwise disjoint), and E is the set of edges. C is the set of vertices associated with computational nodes (or operations), D is the set of delay elements, I and O are the set of inputs and set of outputs, and M is the set of (multiplication) constants. An edge $e \in E$ can connect any pair of vertices in V with the constraint that no edge is incident *to* an input or constant and no edge

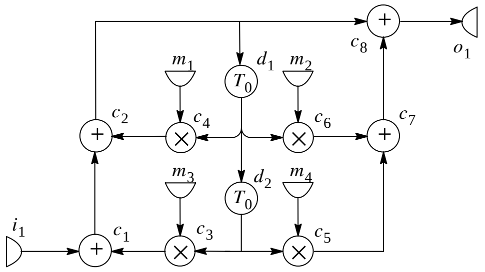

Figure 3: *IDFG of a second-order digital filter section.*

is incident *from* an output. In IDFG's token streams originate from the input nodes making the computational nodes fire repetitively and producing token streams flowing to the output nodes. The computations performed as a consequence of one set of tokens at the input nodes will be referred to as the computation of one iteration. Each token can be indexed by the iteration number in which it was produced. In this context, the iteration period T_0 is the time between two consecutive firings of a node.

A *delay* node indicates data dependencies between nodes fired by tokens of different iterations in an IDFG. It delays a token that it receives for one iteration period.

The duration of the computation performed by a vertex $v \in C$ is given by $\delta(v)$. For $v \in D \cup I \cup O \cup M$, $\delta(v) = 0$ by definition. In this paper it is assumed that all durations are expressed as integer multiples of an abstract time unit *TU*. One *TU* corresponds to one cycle (or *control step*) of the hardware system clock (not to be confused with the iteration period: also T_0 is an integer multiple of *TU*.). The computational nodes in this paper are assumed to be *atomic* and *nonpreemptive*. This respectively means that the graph is not hierarchical but flat and that operations requiring more than one *TU* cannot be interrupted.

When the IDFG is cyclic, all cycles should contain at least one delay element and T_0 should be chosen greater or equal to the *iteration period bound* [24, 25]. For acyclic IDFG's it is assumed that the minimal T_0 is the maximum of the durations of the operations in the IDFG. Figure 3 shows an example of a cyclic IDFG.

2.3 The cost function

Before the cost function can be presented, some more definitions are necessary (the notation has been inspired by the one used in [26]):

- R is the set of resource types. The cost of each $r \in R$ is indicated by ω_r. In principle the cost function should take into account any type of hardware resources that are allowed in the hardware model. However, the HLS system presented here, only deals with the cost of FU's. Besides, the system can only deal with $\omega_r = 1$ for all r (this means that the number of resources are optimized independently for each type, excluding the possibility of trade-off).

- The set of control steps is given by $S = \{s_{min}, s_{min}+1, \ldots, s_{max}-1, s_{max}\}$ and a schedule $\tau : C \rightarrow S$ assigns to each operation $c \in C$ a control step $s \in S$ in which the operation will be executed. A discussion on the possible values of s_{min} and s_{max} will be given in Section 2.5.

- The number of resources of type $r \in R$ required by a schedule τ at time $s \in S$ is given by $N_r(s, \tau)$. This number is schedule-dependent as schedules that execute more operations of the same type in a certain control step require more resources of that type.

The schedule is called *feasible* if the precedence constraints to be discussed in Section 2.4 are satisfied. A scheduling algorithm should produce feasible schedules under certain constraints and optimize a certain objective function. In *time-constrained scheduling,* constraints on the schedule length (the iteration period and also the latency, to be discussed further on) are given and the hardware costs should be minimized. In *resource-constrained scheduling,* limits on the hardware that can be used is given and the goal is to be as fast as possible.

Time-constrained scheduling is addressed in this paper. So, the cost function to be optimized is given by:

$$f(\tau) = \sum_{r \in R} \omega_r \cdot \max_{s \in S} N_r(s, \tau) \qquad (1)$$

2.4 The precedence relations

The rules for node firing in an IDFG have some consequences for scheduling, which result in so-called *precedence relations.* As these relations are an important issue in scheduling, they will be explained below.

A computational node cannot start execution before all its inputs are available, meaning that all its predecessors should have finished execution. The predecessors of a node are determined by the edges $(c_i, c_j) \in E$. From this restriction the *forward precedence relations* can be derived:

- Suppose two computational nodes $c_i, c_j \in C$ are connected by the edge $(c_i, c_j) \in E$. Then the times $\tau(c_i)$ and $\tau(c_j)$ for which operations c_i and c_j start execution in a schedule τ should satisfy:

$$\tau(c_i) + \delta(c_i) \leq \tau(c_j) \qquad (2)$$

for all $(c_i, c_j) \in E$.

The existence of delay elements results in precedence relations between operations of different iterations. These relations are called *backward precedence relations.* A backward precedence relation results from the following situation:

- Suppose a node c_i that sends its result to a delay node d_k and a node c_j that reads its input from d_k. For a correct implementation of the intended computation, the nodes c_i and c_j must be scheduled so that the result of c_i is written in d_k before the execution of c_j in the next iteration reads it. This results in:

$$\tau(c_i) + \delta(c_i) \leq \tau(c_j) + T_0 \qquad (3)$$

  for all pairs of nodes c_i and c_j connected through an intermediate delay node d_k.

The Inequalities 2 and 3 actually hold as well for input, output and constant nodes (remember that the execution time for such nodes is zero by definition).

2.5 Scheduling ranges and the scheduling range chart

The Inequalities 2 and 3 define a system of inequalities all of which can be written in the form: $\tau(c_j) - \tau(c_i) \geq w_{ij}$, with w_{ij} a constant that is positive for forward precedence constraints and negative for backward precedence constraints. These inequalities can be mapped on a so-called *inequality graph* [9]; application of a longest-path algorithm, such as *Bellman-Ford* [2], results in upper and lower bounds for the starting times in the schedule. Such an interval has been called the *scheduling range* by Heemstra de Groot [9] (Paulin uses the term *time frame* for a similar concept in nonoverlapped scheduling [22]). The scheduling range of a node v will be denoted by the interval $[\tau_-(v), \tau_+(v)]$. The length of this interval, $\tau_+(v) - \tau_-(v)$, is called the *mobility* (or *freedom* [21]) of the operation.

In the set of inequalities, there is no reference to "absolute" time and therefore no unique solution can be found without making further assumptions. Therefore, the original scheduling range concept was based on a *reference node.* This means that one node of the IDFG is fixed at control step zero. Then for this node, longest paths from and to all other nodes are determined resulting in a unique solution. A disadvantage of this method is the fact that the scheduling ranges found, and therefore the final results obtained by the heuristics are sensitive for the choice of the reference node. Also, this method may result in infinite upper and lower bounds.

In this paper, a unique solution to the scheduling range determination problem will be enforced by restricting the *latency.* Latency is defined as the number of control steps between consumption of an input sample and occurrence of the corresponding sample at the output. This parameter fits better to design specifications than the choice of a reference node. It also restricts the bounds of all nodes to finite values.

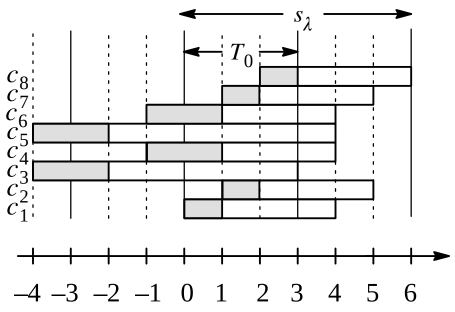

Figure 4: *The scheduling-range chart corresponding to the graph of Figure 3.*

So, the lower bounds of the scheduling ranges are determined by setting $\tau_-(v) = 0$ for all $v \in I$ (the set of input nodes) and computing the longest paths to all computational nodes in the IDFG. This results in a finite value $\tau_-(v)$ for all nodes (the value is finite because it is assumed that all "interesting" IDFG's have at least one path from the inputs to all other nodes). For the upper bounds, one sets the values of $\tau_+(v)$ for all $v \in O$ (the set of output nodes) to the value of the latency parameter s_λ, $s_\lambda \in S$ provided as part of the problem instance specification. This value should satisfy the condition $s_\lambda(v) \geq \tau_-(v), \forall v \in O$. By computing the longest-paths going backwards from the output nodes, one finds the values of $\tau_+(v)$ for all other computational nodes in the IDFG (going backwards means here considering the inequality graph in which all edge orientations have been reversed).

It is now possible to be more precise about the lower and upper bounds of the set S introduced in Section 2.3: $s_{min} \leq 0$ and $s_{max} \geq s_\lambda - \max_{s \in S} \delta(s)$. Values outside the range $[0, s_\lambda - \max_{s \in S} \delta(s)]$ can occur because the longest-paths from input or output nodes to intermediate nodes may be longer than the longest paths from input to output and vice versa. The values of s_{min} and s_{max} are, however, finite as the lengths of all longest paths are assumed to be finite.

A useful way to visualize scheduling ranges is the usage of the *scheduling range chart* [9]. It graphically displays the duration and mobility of each computational node in the IDFG. An example of a scheduling range chart can be found in Figure 4. It displays the chart derived from the IDFG shown in Figure 3 assuming that an addition requires one TU, a multiplication 2 TU's, while $T_0 = 3$ TU's and $s_\lambda = 6$ TU's. For each operation, a gray rectangle shows its duration and a white rectangle the boundaries in which the gray rectangle could be put (the width of the white rectangle is, therefore, equal to the sum of mobility and duration). Note that $s_{min} = -4$ and $s_{max} = 5$.

3 The scheduling algorithm

The main disadvantage of the work of Oláh [18, 19] was the high computational complexity of the *force-directed* scheduling heuristic that was chosen for the scheduling task of the HLS method. In the work reported here, force-directed scheduling has been replaced by *list* scheduling, which is known for its simplicity and low computational requirements (see e.g. [5]) while still producing results of comparable or better quality [10, 12].

List scheduling uses a list of *ready operations*, operations of which all predecessors have already been scheduled, and a *priority function*, a function that decides which operation in the list should be scheduled next. This operation is scheduled at the earliest control step at which an FU of the appropriate type is available. However, list scheduling cannot be directly applied to IDFG's for the following reasons:

- List scheduling is resource constrained, while for DSP applications time constraints (T_0 and s_λ) are more appropriate.
- An IDFG contains loops. These loops result in precedence constraints that cannot be handled by an ordinary list scheduling algorithm.

Solutions for these problems are discussed in the rest of this section.

3.1 Time-constrained list scheduler

A method for time-constrained list scheduling is proposed in [10]. Basically this method consists of a list scheduler that schedules until a resource constraint keeps the scheduler from satisfying the time constraint. As soon as this situation occurs, the resource set is adjusted and the scheduler starts again. Figure 5 gives the pseudo-code of an algorithm based on this idea.

Note that unlike conventional list scheduling algorithms, this algorithm does not schedule operations in increasing order of control steps, but assigns global priorities to all operations and schedules them following these priorities regardless of the sequence in which these operations will be executed. Scheduling following the consecutive control steps makes only sense for problems in which no backward precedences have to be satisfied (like in noniterative DFG's).

The procedure `schedule` inspects the control steps in the range $[\tau_-(v), \tau_+(v)]$ of an operation v from left to right until an appropriate FU is found that is available for the duration of the operation. The

```
procedure time_constrained_schedule();
begin
   init_resources();
   while NOT all_scheduled()
   begin
      extend_resources();
         repeat
            op = getMaxPriority_node();
            schedule (op);
         until resource_violation or all_scheduled;
   end
end;
```

Figure 5: *Pseudo-code of the time-constrained list-scheduling algorithm*

resource-constraint violation mentioned above is reported when no FU of the right type is available in the whole range.

Fixing the starting time of an operation at a specific position affects the scheduling ranges of the operations still to be scheduled. They have to be recomputed in each iteration of the repeat-until loop. This can again be done by an adapted version of the Bellman-Ford algorithm [2].

3.2 Priority function

A priority function should be found that is suitable for IDFG's. In the first place, mobility is important. It is quite easy to see that operations with a small mobility can better be scheduled before operations with a larger mobility: the more limited the freedom to choose a position inside the scheduling range, the likelier it is that wrong choices will be made that cut off interesting parts of the search space. Therefore, mobility will be taken into account in the priority function.

For IDFG's however, also the following point should be kept in mind: in overlapped scheduling, operations of different iterations are executed simultaneously. The schedule is fully static, meaning that it is equal for each iteration. This makes scheduling decisions complicated. As mentioned in Section 3.1, fixing an operation at a specific position, affects the scheduling ranges of the operations still to be scheduled. Some nodes have a larger impact than others. It can be argued that nodes that have a large impact should be scheduled first. If not, the likelihood that a resource conflict will occur later on increases.

Generally spoken, shifting operations with a large number of successors, might influence the scheduling range of these successors. The question whether or not these scheduling ranges will be directly influenced can be answered simply: when one or more paths exists from node c_i to c_j and the lower bound of c_i plus the sum of all execution times (including

c_i) and delays (negative duration!) of the nodes on the longest path from c_i to c_j equals the lower bound of c_j, then shifting node c_i will shift the lower bound of c_j. In this situation, c_j is a *directly shifted successor* (DSS) of c_i. The set of directly shifted successors of a node c_i will be indicated by $\Delta(c_i)$.

The two effects of "mobility" and "successor shifting" have been combined to the priority function $\pi(c)$:

$$\pi(c) = K_\mu - |C|(\tau_+(c) - \tau_-(c)) + |\Delta(c)| \qquad (4)$$

The constant K_μ determines the maximum mobility that is taken into account. It is possible, for example, to give priority only to nodes with a mobility of zero, while for nodes with a mobility larger than zero, only the DSS count is considered.

3.3 Initial resource set estimation

As mentioned in Section 3.1, the scheduler schedules until a resource violation occurs and then starts again with an adjusted resource set. To avoid unnecessary restarts, a nonempty initial resource set should be provided. This can be done by estimating a lower bound. Although more sophisticated and accurate lower bounds estimations are possible, a simple estimation method turned out to give sufficient speed-up. The method assumes that no precedence constraints should be satisfied. Then all operations can be scheduled anywhere within the T_0 control steps. If $\rho(c)$ gives the FU type of an operation, the estimation M_r of the number of FU's of type r can be computed as follows:

$$M_r = \left\lceil \frac{\sum_{c \in C : \rho(c) = r} \delta(c)}{T_0} \right\rceil \qquad (5)$$

4 Implementation

An implementation of the scheduler was made making use of the New Eindhoven Architectural synthesis Toolbox (NEAT) [11]. NEAT has been designed because HLS consists of several subtasks, each with its own peculiarities. Many research projects focus on individual subproblems, leading to a collection of software packages without standardized interfaces.

The goal of NEAT is to provide an environment in which software modules for the solution of distinct HLS subtasks can easily be implemented in such a way that the result of one module is directly available for another one. This is achieved through C++ classes that store and manipulate graphs. There are three types of graphs:

- *data-flow graphs* for the behavioral description as input for the HLS system;

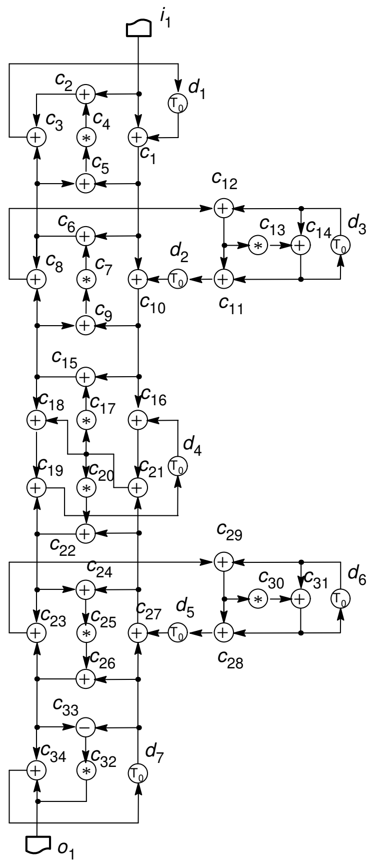

Figure 6: *The IDFG of the well-known fifth-order wave digital filter benchmark.*

- *network graphs* with which the data path produced as the output of HLS (hardware resources and their interconnections) can be described;

- *control graphs* for the specification of the finite-state machine that is the result of HLS.

The C++ inheritance mechanism can be used to extend the routines provided with the NEAT system for building one's own application like e.g. the scheduler reported in this paper.

5 Experimental results

The implementation, which has received the name *ListRange*, was tested with several benchmark graphs and compared to other schedulers. Here, only the most interesting benchmark results will be presented (the interested reader is referred to [13] for many more results): the fifth-order wave digital filter due to Nouta [3]. The IDFG is given in Figure 6 in order to give an impression of its complexity. Note that the multiplication constants that provide the second input of each multiplier have not been drawn. Following the specifications of the benchmark, the duration of an addition is 1 TU and of a

T_0 [TU]	s_λ [TU]	Theda.Fold *	+	CPU [sec]	ListRange *	+	CPU [sec]
16	18	3	3	0.15	2	3	8.74
17	19	2	2	0.25	2	3	8.91
19	21	1	2	0.38	2	2	6.70

Table 1: *Scheduling results for fifth-order wave digital elliptic filter compared to the scheduler Theda.fold.*

T_0 [TU]	SYNTH *	+	CPU [sec]	ListRange *	+	CPU [sec]
16	2	3	17.2	2	3	8.74
17	2	3	34.8	2	3	8.91
18	2	3	34.3	2	2	5.95
19	2	3	50.7	2	2	6.70
20	2	3	59.8	2	2	6.97
21	2	2	75.2	2	2	6.97
22	1	2	77.1	2	2	6.95

Table 2: *Scheduling results for fifth-order wave digital elliptic filter compared to the scheduler SYNTH.*

multiplication is 2 TU.

From the schedulers reported in the literature *Theda.Fold* [15] has been selected for a comparison with ListRange as it is one of the most successful ones (the interested reader is referred to [15] for extensive comparisons of Theda.Fold with other schedulers). The comparison can be found in Table 1. The table gives the number of multipliers and adders obtained for a given T_0 and s_λ together with the CPU times necessary. The CPU-times for ListRange refer to a HP9000/735 workstation and for Theda.Fold to a SUN-4/490. It can be seen that Theda.Fold is quite faster than ListRange but that it does not always produce results of better quality.

Another interesting comparison is with SYNTH [1], which can be considered the direct predecessor of ListRange. It was as well implemented in the NEAT environment. The results of the comparison are presented in Table 2. The latency is not given in this table as it was not supported by SYNTH as an input parameter. Clearly, it can be seen from the table that SYNTH, which was based on force-directed scheduling [22], shows increasing CPU times for increasing T_0, whereas the CPU times of ListRange have the tendency to remain constant. There is no clear winner as far as the quality of the solutions is concerned.

6 Conclusions

This paper has presented the principles for an adaptation of list scheduling for the generation of overlapped schedules. As the notion of time in overlapped scheduling is cyclic (the schedule repeats itself every T_0 time units), processing operations "from left to right" (as in conventional list scheduling) does not make sense. The method proposed here assigns priorities to all unscheduled nodes and then iteratively schedules the node with highest priority. The idea of going "from left to right" is only applied within the *scheduling range* of the node itself.

The prototype implementation based on these principles shows that results of good quality are obtained while the required CPU time to find a solution does not deteriorate for increasing T_0 (contrary to expectations based on the worst-case time complexity [13]).

Future work in this direction will concentrate on dealing better with the costs of the hardware resources (an $\omega_r \neq 1$, multiplexer costs, etc.) and with a more general hardware model (buses, tri-state buffers, pipelined FU's, etc.).

References

[1] A. Braaksma. Scheduling for the high-level synthesis of iterative data-flow graphs and its implementation in the New Eindhoven Architectural synthesis Toolbox. Master's thesis, University of Twente, Department of Electrical Engineering, September 1993. EL-BSC-93N175.

[2] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, Massachusetts, 1990.

[3] P. Dewilde, E. Deprettere, and R. Nouta. Parallel and pipelined VLSI implementations of signal processing algorithms. In S.Y. Kung, H.J. Whitehouse, and T. Kailath, editors, *VLSI and Modern Signal Processing*, pages 257–276. Prentice Hall, Englewood Cliffs, New Jersey, 1985.

[4] A. Fettweis. Realizability of digital filter networks. *Archiv fuer Elektronik und Uebertragungstechnik*, 30(2):90–96, 1976.

[5] D.D. Gajski, 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.

[6] G. Goossens, 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*, 9(9):925–937, September 1990.

[7] B.S. Haroun and M.I. Elmasry. Architectural synthesis for DSP silicon compilers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(4):431–447, April 1989.

[8] S.M. Heemstra de Groot. *Scheduling Techniques for Iterative Data-Flow Graphs, An Approach Based on the Range Chart.* PhD thesis, University of Twente, Department of Electrical Engineering, December 1990.

[9] S.M. Heemstra de Groot, 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*, 39:351–364, May 1992.

[10] M.J.M. Heijligers. Time constrained scheduling for high level synthesis. Master's thesis, Eindhoven University of Technology, Department of Electrical Engineering, May 1991.

[11] M.J.M. Heijligers, H.A. Hilderink, A.H. Timmer, and J.A.G. Jess. NEAT: An object-oriented high-level synthesis interface. In *International Symposium on Circuits and Systems*, pages 1.233–1.236, 1994.

[12] R. Jain, A. Mujumdar, A. Sharma, and H. Wang. Empirical evaluation of some high-level synthesis scheduling heuristics. In *28th Design Automation Conference*, pages 686–689, 1991.

[13] M.S. Koster. List scheduling for iterative data-flow graphs. Master's thesis, University of Twente, Department of Electrical Engineering, August 1994. EL-BSC-045N94.

[14] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[15] T.F. Lee, 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*, 13(4):439–450, April 1994.

[16] C.E. Leiserson, F.M. Rose, and J.B. Saxe. Optimizing synchronous circuitry by retiming (preliminary version). In R. Bryant, editor, *Third Caltech Conference on VLSI*, pages 87–116. Springer Verlag, Berlin, 1983.

[17] M.C. McFarland, A.C. Parker, and R. Camposano. High-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, February 1990.

[18] A. Oláh. Range-chart-guided high-level synthesis. Master's thesis, University of Twente, Department of Electrical Engineering, August 1991. EL-BSC-91N134.

[19] A. Oláh, 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. In *International Conference on Computer Systems and Software Engineering, CompEuro 92*, pages 145–150, May 1992.

[20] K.K. Parhi and D.G. Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. *IEEE Transactions on Computers*, 40(2):178–195, February 1991.

[21] A.C. Parker, J.T. Pizarro, and M. Mlinar. Maha: A program for datapath synthesis. In *23rd Design Automation Conference*, pages 461–466, 1986.

[22] P.G. Paulin and J.P. Knight. Force-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(6):661–679, June 1989.

[23] J.M. Rabaey, C. Chu, P. Hoang, and M. Potkonjak. Fast-prototyping of datapath-intensive architechtures. *IEEE Design and Test of Computers*, pages 40–51, June 1991.

[24] R. Reiter. Scheduling parallel computations. *Journal of the ACM*, 15(4):590–599, October 1968.

[25] M. Renfors and Y. Neuvo. The maximum sampling rate of digital filters under hardware speed constraints. *IEEE Transactions on Circuits and Systems*, CAS-28(3):196–202, March 1981.

[26] W.F.J. Verhaegh, E.H.L. Aarts, J.H.M. Korst, and P.E.R. Lippens. Improved force-directed scheduling. In *European Design Automation Conference*, pages 430–435, 1991.

[27] C.Y. Wang and K.K. Parhi. High level DSP synthesis using the MARS design system. In *International Symposium on Circuits and Systems*, pages 164–167, 1992.