A Genetic Approach to the Overlapped Scheduling of Iterative Data-Flow Graphs for Target Architectures with Communication Delays

Erwin R. Bonsma and Sabih H. Gerez
Department of Electrical Engineering, University of Twente, The Netherlands
E-mail: s.h.gerez@el.utwente.nl

Abstract— **This paper presents a method to solve the overlapped fully-static multiprocessor scheduling problem. An iterative data-flow graph (IDFG) is mapped on a target architecture that allows fine-grain parallelism. The goal is the minimization of the iteration period. The method can deal with nonzero delay times to communicate data between processors as well as with link capacities in the interconnection network. Excellent results for benchmark IDFGs have been obtained by the method that consists of three layers, each concentrating on a different aspect of the optimization problem.**

I. Introduction

An algorithm that contains computations that can be executed simultaneously, offers possibilities of exploiting the parallelism present by implementing it on appropriate hardware such as a *multiprocessor* system. The class of algorithms considered in this paper is limited to algorithms that can be represented by *homogeneous synchronous data-flow graphs* [1], also called *iterative data-flow graphs* (IDFGs) [2]. Such an algorithm consists of a core computation that is iterated an infinite number of times and that does not contain data-dependent decisions. Many algorithms in the field of *digital signal processing* (DSP) belong to this class.

A parallel implementation of an algorithm implies that each operation in the data-flow graph is mapped on a hardware unit and a time instant. Finding these mappings is called *(processor) assignment* and *scheduling* respectively. A schedule is called *nonoverlapped* if all operations belonging to the same iteration have to finish before the operations of the next iteration can start; if operations belonging to different iterations can execute simultaneously, the schedule is called *overlapped* [3]. Overlapped scheduling is sometimes also called *loop folding* [4, 5]. An overlapped schedule can be characterized by two entities: the *iteration period* T_0, the time between the invocation of the same operation in subsequent iterations, and the *latency*, the time that passes between the consumption of the first input and the production of the last output in the same iteration. Both times are expressed in multiples of the global system clock period.

Erwin Bonsma is currently following the *Artificial Intelligence Programme* at the University of Edinburgh.

In this paper only *fully-static* overlapped schedules are considered which means that all iterations have the same schedule and the same processor assignment. Two different optimization goals for scheduling are normally distinguished: in *time-constrained* scheduling, the goal is to use as little hardware as possible for a given execution speed, while in *resource-constrained* scheduling, the goal is to execute the IDFG as fast as possible on a given hardware configuration. In this paper, the resource-constrained problem is addressed.

The multiprocessor scheduling problem for iterative algorithms has received quite some attention in the last years. However, most of the publications have considered the simplified problem in which the time to communicate data from one processor to another was negligible (see the references in [6]). When communication delays are taken into account, often target architectures are considered in which setting up a communication or performing the transfer of a single data word requires tens or even hundreds of system clock cycles (see e.g. [7]). In such an architecture, it only makes sense to have parallel implementations if the basic tasks performed by a single processor, require a similar number of cycles. This type of parallelism is called *coarse grain*. In this paper, however, *fine-grain* parallelism is considered. The basic tasks are primitive operations such as additions and multiplications and the processors in the target architecture are supposed to transfer a data word in just a few cycles. An example of a suitable architecture is given in [8]. Other descriptions of realizations of this type of architectures can be found in [6]. The method presented in this paper can also deal with the classical *architectural synthesis* hardware models (see e.g. [9]) where different functional unit types can be used and where results can be transmitted to another functional unit without delay. Such models are simply a special case of target architectures with communication constraints.

The rest of this paper is structured as follows: first the problem is defined, then the different aspects of the proposed solution method are explained and finally some experimental results are presented. Not all details of the solution method can be explained here; readers are referred to [10] for a full account of the method.

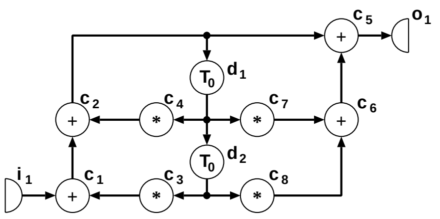

Fig. 1. IDFG of a second-order digital filter section.

II. Problem Description

The IDFG consists of a vertex set V and an edge set E. V contains computational nodes (such as additions or multiplications), I/O nodes and delay nodes (a delay node stores data received at its input during the current iteration and makes this data available at its output in the next iteration). Data in an IDFG flows along the edges of E: $(u, v) \in E$ implies that data produced by u is consumed by v and that v cannot be executed before the completion of u. One also says that the edges in E indicate *precedence constraints.* An example of an IDFG is shown in Fig. 1 that represents a second-order digital filter section. In the figure, delay elements are indicated by T_0.

The target architecture consists of a given network of processors or functional units. Each processor can execute all or a subset of the computations contained in the IDFG. The time necessary to execute a specific operation (e.g. an addition) is the same for all processors. For a computational node $v \in V$, this time is expressed in multiples of the system clock time and denoted by $d(v)$. The interconnection network can also be described as a graph with a vertex set F representing functional units (processors) and an edge set L representing *unidirectional* or *bidirectional links.* Each link is able to transfer a single data word at a time. The time to transfer a data word through a link is the same for all links and equals δ cycles of the system clock. Parallel links can be used to model the possibility of transferring multiple data words simultaneously between a pair of functional units.

The set of all operation types that can occur in an IDFG is Ω. The operation type of a computational node $v \in V$ is given by $\gamma(v)$. The set of operation types that a functional unit $f \in F$ can execute is given by $, (f)$ $(\gamma(v) \in \Omega$ and $, (f) \subset \Omega)$.

The problem addressed in this paper is to find the overlapped fully-static schedule with minimal T_0 that implements a given IDFG on a given target architecture. The solution should be fully specified in the sense that a start time and a processor is associated with each operation and sets of time instants and links with each data transfer. A secondary goal is to minimize the latency.

III. Solution Method

The investigated solution method consists of an algorithm in which three layers can be distinguished (see Fig. 2). A *genetic algorithm* at the top level takes care of the main optimization goal. Genetic algorithms (see e.g.

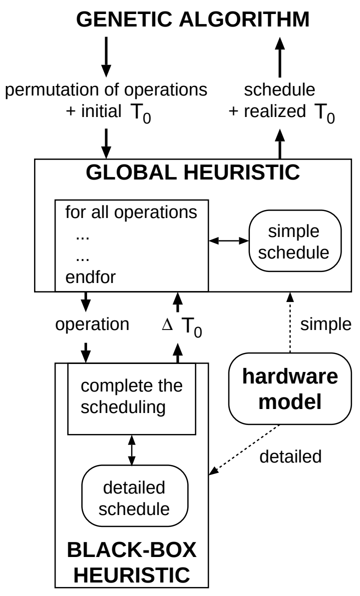

Fig. 2. The structure of the proposed solution method.

[11]) are inspired by the evolution process in nature. The evolution process is simulated in a population of feasible solutions: the better the quality of a solution, the more likely it will be that the solution will survive into the next generation or will act as a parent for new solutions added to the next generation. A key issue in genetic algorithms is that each solution is encoded as a linear string of symbols called a *chromosome.* New members of the population are created by copying parts of information of two parent chromosomes in order to create a new child chromosome. A *crossover* operator takes care of this task. The figure of merit associated to a chromosome, or rather to the feasible solution encoded by it, is called *fitness.* The goal of a genetic algorithm is to find the chromosome with optimal fitness using the techniques just described.

Experience in practice shows that a genetic algorithm is a powerful optimization method as long as the search space has a simple structure. However, trying to solve a scheduling problem with a genetic algorithm using a direct encoding of the schedule is not a good idea. It is difficult or even impossible to encode feasible solutions in such a way that a chromosome that has been obtained after the application of commonly used crossover operators, is itself the encoding of a feasible solution (it is e.g. likely that the new solution will contain violations of precedence constraints as the crossover operator does not have problem-specific knowledge).

However, an indirect encoding of the solution has recently been reported to give good results [12, 13]. The idea is to only encode a permutation of all computational operations in the IDFG in a chromosome. A greedy algorithm that uses such a permutation to generate a schedule, is part of the cost function that evaluates the fitness of a chromosome. Whenever the greedy algorithm has to make a choice between operations to schedule, the ordering in the permutation guides this choice. This greedy algorithm forms the second layer of the proposed method and is discussed in more detail in Section IV-A. It is called here the *global scheduling heuristic* because its main task is to roughly solve the scheduling and assignment problems based on detailed knowledge of the IDFG but only a simplified model of the target architecture. The main purpose of the global heuristic is to help the genetic algorithm to search the solution space fast and efficiently.

The fact that the global heuristic is not based on a complete and realistic multiprocessor model has a number of advantages. The first one is that the global heuristic can be used for a wide range of multiprocessor architectures because the assumed hardware model is very general. A second advantage is that the global scheduling heuristic does not have to be too complex, making it is easier to design a good heuristic.

Detailed knowledge of the target architecture is provided by the third layer of the proposed method called the *black-box scheduling heuristic* to be discussed in Section IV-B. While the global heuristic only knows the minimal time required to communicate data between pairs of processors, the black-box heuristic has knowledge of link presence and occupancy. The black-box heuristic schedules one operation at a time and uses a greedy heuristic to do so. The advantage of separating the black-box heuristic from the global heuristic is that only the black-box heuristic needs to be extended in order to support new classes of multiprocessor architectures.

IV. Explanation of the Heuristics

In this section, the main ideas of the two heuristics are presented. A step-by-step description of the two algorithms is given in Section V.

A. Global Scheduling Heuristic

Let $\sigma(v)$ denote the time at which a computational node $v \in V$ should start its execution in the solution found for the scheduling problem. Then the following inequality holds for any pair $u, v \in V$ of computational nodes for which there is a path going from u to v through n delay nodes (including the case $n = 0$) [2]:

$$\sigma(v) \geq \sigma(u) + d(u) - nT_0 \tag{1}$$

Although this inequality only holds for hardware in which communication delays are neglected, it can be used for the problem of this paper as will be explained below. All instances of Inequality 1 (for all relevant values of u and v) can be represented in an *inequality graph* [2]. The longest path between all possible pairs of vertices in this graph can be computed using e.g. the Floyd-Warshall algorithm [14] and the obtained longest-path distances can be stored in a *distance matrix* $\mathbf{D}_i^{T_0}$ [13]. Note that the contents of a distance matrix depend on the value of T_0. So, the longest-path from a node $u \in V$ to a node $v \in V$ is given by $\mathbf{D}_i^{T_0}[u, v]$ (for the simplicity of notation, symbolic rather than integer indices are used here; the subscript i refers to the inequality graph).

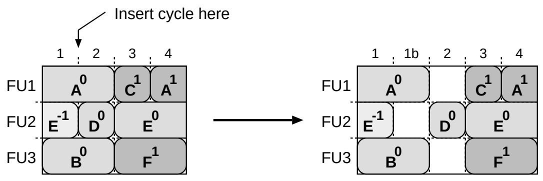

Fig. 3. Cycle insertion in a partial schedule.

The global heuristic uses a simplified model of the target hardware. Apart from the duration $d(v)$ of any operation in $v \in V$, it knows which type of operations can be handled by each of the processors in F and the shortest communication distance (in multiples of the link delay δ) between any pair of processors given by the matrix $\mathbf{D}_h$ (where the subscript h refers to "hardware"). The model, therefore, amounts to assuming an infinite number of parallel links for each actual link.

The global heuristic starts by scheduling the first operation in the list P at time 0 and assigns it to the first processor in F that is able to execute the operation. It then goes on to schedule the other operations in the order of occurrence in P with the restriction that an operation v is temporarily skipped if none of the successors or predecessors of v in the IDFG has already been scheduled.

Suppose that that the subset of operations that have been already scheduled is given by S. If the assignment to a processor is neglected, a yet unscheduled operation $v \in V$ can start its operation at any of the instants of the range $R_{nc}(v) = [R_{nc,min}(v), R_{nc,max}(v)]$, where:

$$R_{nc,min}(v) = \max_{s \in S} \left(\sigma(s) + \mathbf{D}_i^{T_0}[s, v] \right)$$
$$R_{nc,max}(v) = \min_{s \in S} \left(\sigma(s) - \mathbf{D}_i^{T_0}[v, s] \right)$$

If the assignment to a processor $f \in F$ is considered and the assignment of an operation $v \in V$ to a processor is given by $\alpha(v)$, the range of valid instants for the scheduling of a a yet unscheduled operation $v \in V$ is further constrained to $R_c(v, f) = [R_{c,min}(v, f), R_{c,max}(v, f)]$, where:

$$R_{c,min}(v, f) = \max_{s \in S} \left(\sigma(s) + \mathbf{D}_i^{T_0}[s, v] + \mathbf{D}_h[\alpha(s), f] \right)$$
$$R_{c,max}(v, f) = \min_{s \in S} \left(\sigma(s) - \mathbf{D}_i^{T_0}[v, s] - \mathbf{D}_h[f, \alpha(s)] \right)$$

For both ranges, an empty range (lower bound larger than upper bound) means that the proposed processor assignment cannot be realized without increasing T_0. Situations can occur in which the range is empty for any of the processors in the target architecture and increasing T_0 is unavoidable.

Increasing T_0 by *cycle insertion* is a key issue in the scheduling approach presented in this paper. It is illustrated in Fig. 3. Note that cycle insertion does not require rescheduling of already scheduled operations.

Given an operation v to be scheduled, the global heuristic first selects a number of time-processor pairs called *global* *base schedule instants* that look the most favorable and then evaluates the number of extra cycles to be inserted for each pair. The one requiring the least number is chosen and the heuristic continues with the next element in P. The evaluation of a global base schedule instant is first performed by the global heuristic itself in order to decide for a time and processor mapping. This may already require the insertion of additional cycles. Then the “black-box” heuristic discussed below is invoked to handle the assignment of the data transfers to the links. The information passed to the black-box heuristic is a *black-box base schedule instant.*

B. Black-Box Scheduling Heuristic

The black-box heuristic has full knowledge of the target architecture and especially the link set L. It receives the request to evaluate a black-box base schedule instant for a given operation $v \in V$ and returns the number of extra cycles required to satisfy the request (see also Fig. 2). In order to calculate this number it has to assign the input and output data transfers of v to the links connected to the proposed processor. It may be that routing decisions have to be taken in this process. Link contention will be a reason to insert cycles.

V. Step-by-Step Descriptions

This section presents the step-by-step descriptions of the two heuristics.

A. Global Scheduling Heuristic

1. Provide P, an ordered list of all the operations that have to be scheduled, and $T_{0,initial}$, the initial iteration period; set: $T_0 = T_{0,initial}$.
2. Retrieve (and remove) the first operation from the list P. Schedule this operation on the first FU that can execute this operation and set its start time to 0.
3. Choose c, the next operation to schedule, as follows: find the first operation from the list P for which at least one direct predecessor or successor operation has already been scheduled. Remove this operation from the list P.
4. Calculate the valid start times for operation c. Use the distance matrix $\mathbf{D}_c^{T_0}$, the start times of the operations that have already been scheduled, and the distance matrix $\mathbf{D}_h$ to calculate $R_{nc}(c)$ and $R_c(c,f)$ for $f \in F$.
5. Set $n_{tot,best} = \infty$. Set the preferred start time t_{pref} and the set of base schedule instants B as follows: if $R_{nc,max}(c) = \infty$, then $t_{pref} = R_{nc,min}(v)$ and $B = \{(f,t)|\gamma(c) \in , (f), R_{nc,min}(c) \leq t < R_{nc,min}(c) + T_0\}$; if $R_{nc,min}(c) = -\infty$, then $t_{pref} = R_{nc,max}(c)$ and $B = \{(f,t)|\gamma(c) \in , (f), R_{nc,max}(c) - T_0 \leq t < R_{nc,max}(c)\}$; otherwise, t_{pref} = either $R_{nc,min}(c)$ or $R_{nc,max}(c)$ (depending on the number of (un)scheduled predecessors and successors of c in the IDFG) and $B = \{(f,t)|\gamma(c) \in , (f), R_{nc,min}(c) \leq t < R_{nc,max}(c)\}$. The elements (f,t) of B will be processed in increasing order of the difference $|t_{pref} - t|$.
6. Retrieve and remove the first base schedule instant from B. Use it to construct a valid schedule instant. The start time t and the number of cycles that has to be inserted in the schedule, n_{insert}, are set such that: the communication delays with the scheduled predecessor operations are satisfied, the operation fits in the schedule, and the communication delays with the scheduled successor operations are satisfied.
7. When $n_{insert} \geq n_{tot,best}$ go to Step 9. Otherwise, use the valid heuristic schedule instant as a base black-box schedule instant to construct the corresponding valid black-box schedule instant. The black-box heuristic will return the extra number of cycles $n_{insert,bb}$ that it had to insert itself. Set $n_{tot} = n_{insert} + n_{insert,bb}$.
8. If $n_{tot} < n_{tot,best}$ then set s_{best} to the current valid schedule instant and set $n_{tot,best} = n_{tot}$.
9. If B is not empty and $n_{tot,best} > 0$ go to Step 6.
10. Schedule the operation as specified by the valid schedule instant s_{best} and set $T_0 = T_0 + n_{tot,best}$.
11. When the list P is not yet empty, go to Step 3.

B. Black-Box Scheduling Heuristic

1. Provide c, the operation that has just been scheduled by the global heuristic.
2. Set $n_{insert,bb} = 0$.
3. Find all data transfers that have to be scheduled and place them in a list, T. Every direct predecessor or successor operation of c that has already been scheduled, and that does not execute on the same FU as c does, requires that a data transfer is scheduled. A data transfer is a pair of operations: (c,s) for a transfer to a successor s and (p,c) for a transfer from a predecessor p.
4. Sort the list T so that the data transfers are ordered by increasing slack time. The slack time of a transfer (c_s, c_d), where the number of delay elements between the source c_s and the destination c_d equals n is given by:
$$t_{slack} = \sigma(c_d) - \sigma(c_s) - d(c_s) + nT_0 - \delta \times \mathbf{D}_h[\alpha(c_s), \alpha(c_d))]$$
5. If T is empty, go to Step 15. Otherwise retrieve and remove the first data transfer (c_s, c_d) from T.
6. If $c = c_d$ the data transfer will be scheduled from c_s to c_d, otherwise from c_d to c_s. Note that the direction in which the data transfer is scheduled is not necessarily the direction that the data will travel. Here, only the case from c_s to c_d is discussed for the sake of simplicity.
7. Set t_{slack} equal to the slack time of the data transfer. Set $t_{min} = \sigma(c_s) + d(c_s)$. Set the set R so that it contains all paths with minimal length that connect $\alpha(c_s)$ with $\alpha(c_d)$.
8. Set $n_{tot,best} = \infty$ and set $t_{pref} = t_{min}$. Set L so that it contains every communication link that is the first link of one or more of the paths in R. Use L to set $B = \{(l, t_s)|l \in L, t_{min} \leq t_s \leq t_{min} + t_{slack}\}$. The base schedule elements in B are ordered by an increasing distance between t and t_{pref}.

9. Retrieve and remove the first base schedule instant from B. Use it to construct a valid schedule instant for the communication task. Set the start time t and the number of cycles that has to be inserted in the schedule, n_{insert}, such that the communication task fits the in the link schedule.
10. If $n_{tot} < n_{tot,best}$ then set s_{best} to the current valid schedule instant and set $n_{tot,best} = n_{tot}$ and $t_{s,best} = t$.
11. If B is not empty and $n_{tot,best} > 0$ go to Step 6.
12. Schedule the communication task as specified by the schedule instant s_{best}. Set $n_{insert,bb} = n_{insert,bb} + n_{tot,best}$.
13. If the routing of the data transfer has been completed go to Step 5.
14. Update the slack time $t_{slack} = t_{slack} + t_{min} - t_{s,best}$. Set $t_{min} = t_{s,best} + \delta$. Update the list R by removing all paths that did not start with the communication link that was chosen. For the remaining paths, truncate them by removing the first link. Go to Step 8.
15. Return $n_{insert,bb}$, the total number of cycles that has been inserted.

VI. The Genetic Algorithm

The power of the scheduling method presented here comes from the combination of the greedy heuristics and the genetic top layer. Different experiments were performed to optimize the genetic algorithm used [10]. It turned out that best results were obtained for:

- the crossover operator called *uniform crossover for permutations* [13, 15];
- a fitness function that depends both on the iteration period as well as the latency (somewhat opposite to the expectation, taking the latency into account also gives better results for the iteration period).
- no mutation (mutation is the process of introducing random copying errors when constructing a new generation).

VII. Experimental Results

A prototype of the solution method has been implemented in CMU Common Lisp running on an HP 9000/735 server. The GECO package developed by George Williams (freely available on the Internet) was used to provide for most of the genetic algorithm code. It is implemented in CLOS (Common Lisp Object System) which makes it easily extendible to incorporate a user's requirements.

No well-documented benchmarks are known to the authors that exactly match the problem addressed in this paper (although suitable architectures have e.g. been described in [6]). Therefore, the benchmarks presented here consist of well-known IDFGs (see e.g. [2]) in combination with self-constructed target architectures. The IDFGs considered have been called here `second` (see Fig. 1), `third` (see Fig. 4), `lattice` (see Fig. 5), `jaumann` (see Fig. 6), and `elliptic` (see Fig. 7). These IDFGs have been mapped on the six multiprocessor configurations shown in Fig. 8. Of all possible combinations of IDFG-hardware pairs, 14 have been selected. They are listed in Table I. Apart from

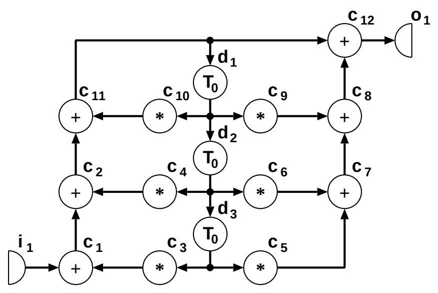

Fig. 4. The IDFG of a third-order IIR filter, called `third`.

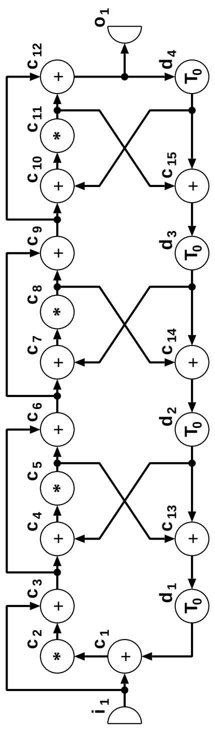

Fig. 5. The IDFG of a fourth-order all-pole lattice filter, called `lattice`.

TABLE I
The fourteen different scheduling problems.

Problem	Algorithm	Hardware			
		Structure	+	*	δ
A	`second`	`strong-ring-4`	1	2	2
B	`second`	`medium-ring-4`	1	2	2
C	`second`	`weak-chain-4`	1	2	1
D	`third`	`weak-chain-4`	1	2	1
E	`third`	`weak-chain-3`	1	2	1
F	`third`	`diamond-6`	1	2	1
G	`jaumann`	`weak-chain-3`	1	5	1
H	`jaumann`	`weak-chain-3`	1	5	2
I	`lattice`	`weak-chain-3`	1	5	1
J	`lattice`	`weak-chain-3`	1	5	2
K	`elliptic`	`weak-chain-4`	1	2	1
L	`elliptic`	`weak-chain-4`	1	2	2
M	`elliptic`	`weak-chain-3`	1	2	1
N	`elliptic`	`weak-chain-3`	1	2	2

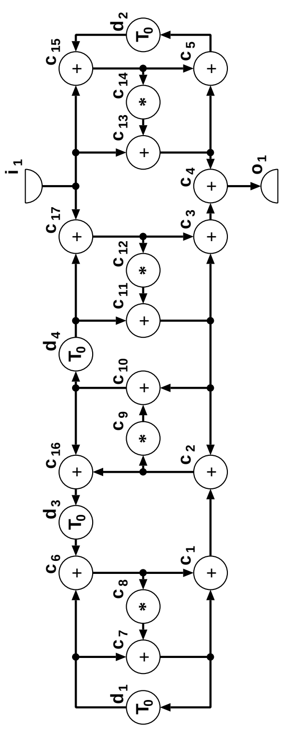

Fig. 6. The IDFG of a fourth-order Jaumann wave digital filter, called `jaumann`.

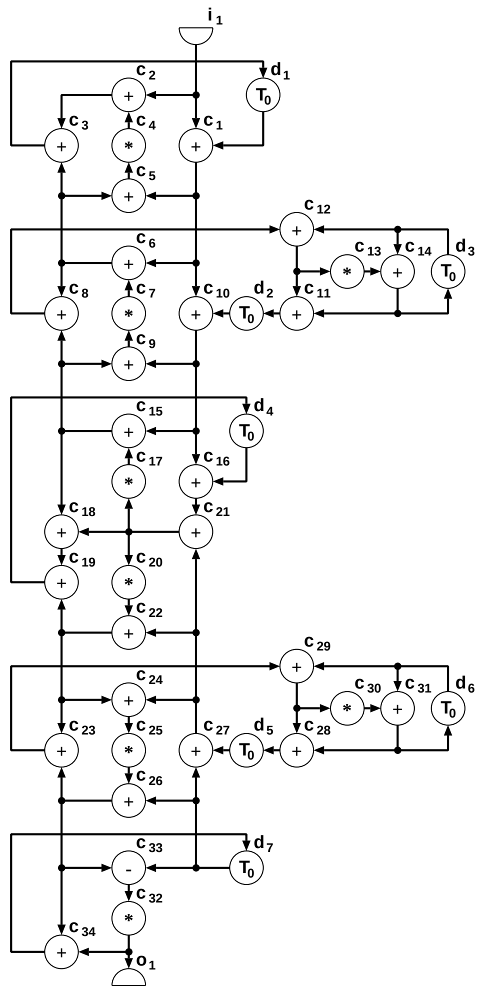

Fig. 7. The IDFG of a fifth-order wave digital elliptic filter, called `elliptic`.

showing the combinations, the table specifies the duration of an addition, multiplication and data transfer for each problem. The results obtained are summarized in Table II. For each problem, the table shows the average number of evaluations (over 50 runs), the average runtime, the best T_0, the percentage of times that the best T_0 was found, the best value for the latency for the optimal T_0 as well as the percentage of times that the best combination was found. A selection of the results obtained is given in Fig. 9.

From the result just presented and many other comparable results it can be concluded that the performance of the solution method is very satisfactory. Solutions of good quality are obtained in reasonable time. When assessing the time, one should realize that the global and black-box heuristics are executed for every chromosome constructed by the genetic algorithm which shows that these heuristics operate quite fast in spite of their complexity. Another fact that confirms the confidence in the method are benchmark results for target architectures with negligible link delays. These results have the same quality as already published results, although more CPU time is required compared to dedicated algorithms.

VIII. Conclusions

This paper has presented a method for the fine-grain fully-static overlapped scheduling of iterative data-flow graphs on target architectures with nonnegligible communication delays. The method consists of three layers of knowledge. At the highest level, a genetic algorithm is allowed to view the problem as finding an optimal permutation of all operations. At the next level, the permutation

TABLE II
The scheduling results for the fourteen problems.

	Avg.	Avg.	Best Schedule			
	evals	runtime	T_0		Latency	
A	249	11s	4	0.98	6	0.44
B	265	12s	4	0.20	9	0.20
C	276	12s	3	0.56	7	0.56
D	376	24s	5	1.00	5	0.12
E	392	25s	6	1.00	4	0.08
F	385	27s	3	0.54	9	0.14
G	485	33s	16	1.00	9	0.94
H	655	43s	18	1.00	13	0.98
I	442	27s	16	0.98	30	0.62
J	508	33s	18	0.98	31	0.16
K	761	134s	18	0.84	16	0.60
L	734	129s	21	0.56	18	0.52
M	709	111s	18	0.08	20	0.02
N	701	111s	21	0.50	14	0.04

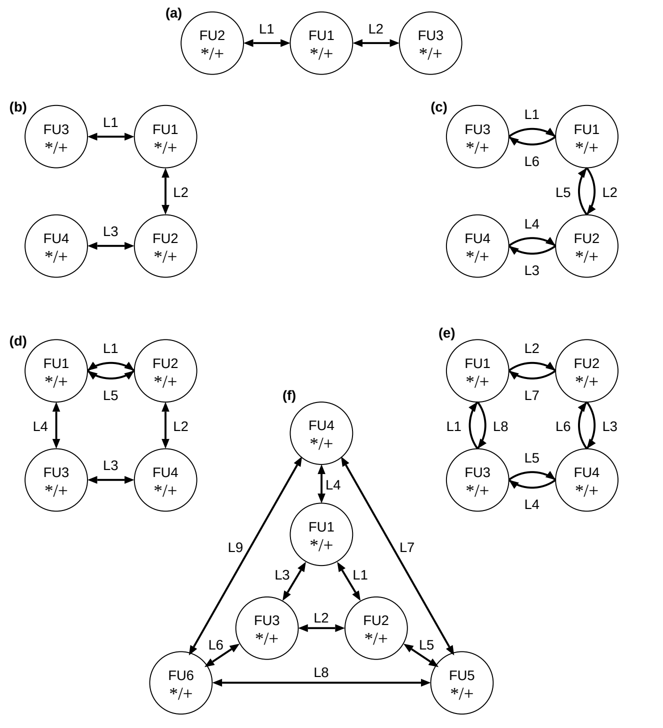

Fig. 8. The benchmark multiprocessor configurations: `weak-chain-3` (a), `weak-chain-4` (b), `strong-chain-4` (c), `medium-ring-4` (d), `strong-ring-4` (e), `diamond-6` (f).

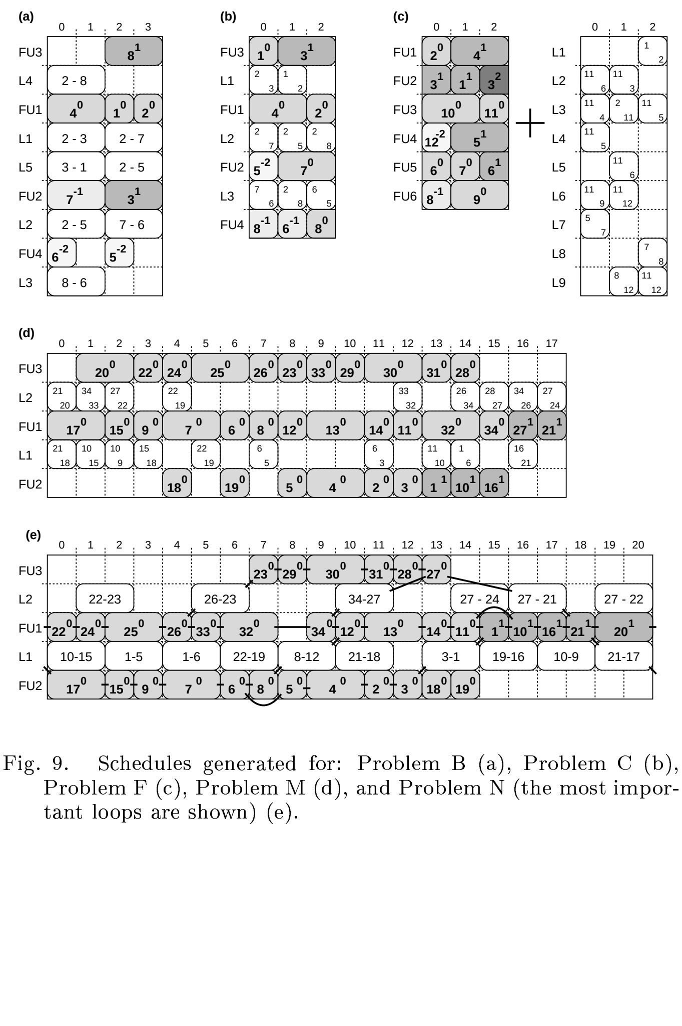

Fig. 9. Schedules generated for: Problem B (a), Problem C (b), Problem F (c), Problem M (d), and Problem N (the most important loops are shown) (e).

is used to guide a greedy heuristic with a limited knowledge of the target hardware. At the lowest level, a heuristic with detailed knowledge of the hardware takes care of the assignment of data transfers to the communication links. The goal of the method is to minimize the iteration period (and latency). A crucial aspect of the heuristics is the ability to insert cycles in a partial schedule and increase the iteration period when the scheduling algorithms get stuck. In this way, earlier scheduling decisions do not need to be reconsidered. Due to this greedy approach a genetic top-level search becomes feasible as has become clear from the results obtained.

Future research in this direction will concentrate on taking more aspects of the target architecture into account (e.g. registers) as well as experimenting with alternative genetic algorithms.

References

[1] E.A. Lee and D.G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987.

[2] 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*, vol. 39, pp. 351–364, May 1992.

[3] K.K. Parhi and D.G. Messerschmitt, "Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding," *IEEE Transactions on Computers*, vol. 40, no. 2, pp. 178–195, February 1991.

[4] 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*, vol. 9, no. 9, pp. 925–937, September 1990.

[5] 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*, vol. 13, no. 4, pp. 439–450, April 1994.

[6] V.K. Madisetti, *VLSI Digital Signal Processors, An Introduction to Rapid Prototyping and Design Synthesis*, IEEE Press and Butterworth Heinemann, Boston, 1995.

[7] J. Sanchez and H. Barral, "Multiprocessor implementation models for adaptive algorithms," *IEEE Transactions on Signal Processing*, vol. 44, no. 9, pp. 2319–2331, September 1996.

[8] D.C. Chen 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, no. 12, pp. 1895–1904, December 1992.

[9] 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.

[10] E.R. Bonsma, "Multiprocessor scheduling of fine-grain iterative data-flow graphs using genetic algorithms," M.S. thesis, University of Twente, Department of Electrical Engineering, June 1997, EL-BSC-018N97.

[11] D.E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Reading, Massachusetts, 1989.

[12] M.J.M. Heijligers and J.A.G. Jess, "High-level synthesis scheduling and allocation using genetic algorithms based on constructive topological scheduling techniques," in *International Conference on Evolutionary Computation*, Perth, Australia, 1995, pp. 56–61.

[13] M.J.M. Heijligers, *The Application of Genetic Algorithms to High-Level Synthesis*, Ph.D. thesis, Eindhoven University of Technology, Department of Electrical Engineering, October 1996.

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

[15] L. Davis, Ed., *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York, 1991.