Improved force-directed scheduling in high-throughput digital signal processing

Improved Force-Directed Scheduling in High-Throughput Digital Signal Processing

Wim F. J. Verhaegh, Paul E. R. Lippens, Emile H. L. Aarts, Jan H. M. Korst, Jef L. van Meerbergen, *Senior Member, IEEE,* and Albert van der Werf, *Member, IEEE*

***Abstract*—This paper discusses improved force-directed scheduling and its application in the design of high-throughput DSP systems, such as real-time video VLSI circuits. We present a mathematical justification of the technique of force-directed scheduling, introduced by Paulin and Knight, and we show how the algorithm can be used to find cost-effective time assignments and resource allocations, allowing trade-offs between processing units and memories. Furthermore, we present modifications that improve the effectiveness and the efficiency of the algorithm. The significance of the improvements is illustrated by an empirical performance analysis based on a number of problem instances.**

I. Introduction

HIGH-LEVEL synthesis is the translation of a behavioral description into a register transfer level description, which specifies the system's structure that implements the behavior. A behavioral description is often represented by a signal flow graph, which consists of nodes representing operations and arcs representing signal flow. An important problem in high-level synthesis is that of scheduling operations, such that a certain objective function is minimized. This objective function may reflect different criteria or combinations of them, such as area, execution time, throughput, and power consumption. The choice of an appropriate scheduling algorithm strongly depends on the application domain and on the synthesis approach that is pursued. To shed some light on these issues, we start by briefly discussing the application domain.

A. High-Throughput Digital Signal Processing

In digital signal processing, signal flow graphs must be executed repeatedly, with a fixed period. The period of repetition is determined by the sampling frequency. High-throughput applications such as real-time video, at which the design methodology PHIDEO [11] is targeted, are characterized by the fact that sampling frequencies are close to clock frequencies. Furthermore, the number of operations that has to be executed each clock cycle is large [15]. These are important characteristics which distinguish the application domain from applications with low or medium sampling frequencies, such as audio applications. Furthermore, this distinction has quite an impact on the architecture and on the synthesis approach that is pursued. High-throughput applications lead to hardwired architectures, complex processing units, large memory areas, and to pipelining.

In case of low or medium sampling frequencies, a *microcoded* architecture is often used, consisting of a limited number of multifunctional processing units, e.g., an ALU with 30 different instructions, and a limited number of central memories [2]. Many different operations of the signal flow graph can be mapped onto the same processing unit, and a microcoded controller is used to select the correct instruction at the appropriate time. In case of high sampling frequencies, however, only a few operations can share the same processing unit, and one often chooses each processing unit to perform one dedicated, complex function. Therefore, a *hardwired* architecture is used, which is characterized by a large number of processing units operating in parallel and a large number of memories.

In high-throughput applications, processing units usually perform complex functions, e.g., complete filter functions. These processing units not only contain arithmetical and logical operations and algorithmic delays, but they also contain local decision-making, which can be used to deal with local conditions in the application. In this paper we do not consider global conditions, but they can be handled in the same way as is done in [21]. In order to meet the high clock frequency, retiming, including pipelining [10], [16], [29], is used. Doing this, inputs and outputs of a processing unit may be shifted in time with respect to each other, which has to be taken into account during scheduling.

Another characteristic of the application domain is the significant role of memories. Existing chips for high-throughput video applications reveal that a large part of the area is occupied by memories. So, during scheduling not only processing units but also memories must be taken into account as an important area-consuming resource. A trade-off between the two types of resources must be made. However, in contrast to processing units whose area is known before scheduling, the area occupied by memories has to be estimated during scheduling, based on the maximum number of variables alive simultaneously and the maximum number of simultaneous accesses.

Manuscript received October 22, 1993; revised January 11, 1995. This work was supported by the EC in the ESPRIT 2260 project. This paper was recommended by Associate Editor M. McFarland.

W. F. J. Verhaegh, P. E. R. Lippens, J. H. M. Korst, J. L. van Meerbergen, and A. van der Werf are with the Philips Research Laboratories, 5656 AA Eindhoven, The Netherlands.

E. H. L. Aarts is with the Philips Research Laboratories, 5656 AA Eindhoven, The Netherlands and the Eindhoven University of Technology, 5600 MB Eindhoven, The Netherlands.

IEEE Log Number 9412155.

Another characteristic of the application domain is the need for pipelined execution of DSP algorithms [18]. We have to make a distinction between two notions of time. On the one hand we have the *algorithm period,* which is the time between two consecutive executions of the algorithm, and which is a measure of the throughput. On the other hand we have the *latency,* which is the time between inputs and corresponding outputs, i.e., the time needed to execute one iteration of the algorithm. Usually, the algorithm period is fixed in DSP applications, and there is an upper bound on the latency. If the latency is larger than the algorithm period, which is often the case in high-throughput applications, then successive executions of the signal flow graph overlap in time, and we speak of pipelined execution. In this paper we do not consider deep nestings of large loops, as is the subject of e.g., [26]. Nevertheless, small loops can be handled by flattening them. The only loop remaining is the most outer loop of infinite repetition of the algorithm, including data dependencies between different iterations.

B. High-Throughput DSP Scheduling

Informally speaking, the high-throughput DSP scheduling problem can be stated as follows. Given a signal flow graph, an algorithm period, and some timing constraints, assign the operations to clock cycles and allocate resources, such that the total area is minimized, and such that timing constraints and data dependencies are met.

Since the scheduling problem we discuss is NP-hard [4] and since practical instances are too large to be solved exhaustively, our aim is to find a good approximation algorithm. An overview of existing scheduling algorithms in high-level synthesis can be found in [14]. To position force-directed scheduling, we discuss the following two extremes.

First, we have list scheduling [6], [13], which is a fast algorithm that gives reasonable-quality solutions. Traditionally, list scheduling is an algorithm that tries to find a minimal execution time, for given constraints on the number of processing units. Additionally, list scheduling can be modified such that memory requirements are reduced, as long as the execution time is not increased. On top of this, one can then iterate on the number of processing units, in order to find a minimal solution that still meets the timing constraints. A drawback of list scheduling is that it requires a decomposition into several levels, which complicates the trade-off between processing units and memories.

Second, we have integer linear programming [5], [7], which is a general approach to combinatorial optimization. This approach can optimally solve problems in their entirety, i.e., without a decomposition. Hence, trade-offs between different objectives, e.g., processing units and memories, are made automatically. In general this goes at the expense of an exponential time complexity. Therefore, integer linear programming may be well applicable to small instances, but running times grow hazardously fast to impracticable heights as the instances get larger or as more refinements are added to the scheduling problem.

Force-directed scheduling is situated in between. Its time complexity is only slightly worse than that of list scheduling. However, the ability of simultaneously trading off processing units and memories is attained, which may result in solutions with a quality close to those obtained by integer linear programming. At the same time, this raises the question whether it is possible to further improve the effectiveness and efficiency of force-directed scheduling, in order to achieve the best of both extremes.

Force-directed scheduling was introduced by Paulin and Knight [19], [21], [22]. It tries to minimize the number of resources by smoothing the resource requirements in time, for given timing constraints. Since its introduction, the technique of force-directed scheduling has gained interest from a number of other research groups [1], [8], [17], [23], [25], [27]. Refinements of the algorithm to include memory costs, pipelined scheduling, multitime operations, multifunctional processing units, etc., can be done in a very straightforward way [21], which makes force-directed scheduling a widely applicable scheduling technique.

We have studied force-directed scheduling in order to use it in the design methodology PHIDEO. Besides a mathematical formulation of force-directed scheduling that justifies the original approach proposed by Paulin and Knight, we have been concentrating on modifications to improve the effectiveness and the efficiency of the algorithm without reducing its applicability. The effectiveness improvements are achieved by the use of global spring constants and gradual time-frame reduction. The efficiency improvements are achieved by an incremental way to calculate the changes in the distribution function.

C. Organization

The organization of this paper is as follows. In Section II, we give a mathematical model of the high-throughput DSP scheduling problem. In Section III we present the mathematical justification of a basic force-directed scheduling algorithm and discuss its time complexity. In Section IV we show how we apply force-directed scheduling to the high-throughput DSP scheduling problem. Next, we present modifications to improve the effectiveness and the efficiency of the algorithm, in Sections V and VI, respectively. Finally, Section VII illustrates the significance of the improvements by means of some experimental results.

II. MATHEMATICAL MODELING

A. Signal Flow Graphs

Input for scheduling is a signal flow graph representing a DSP algorithm, an algorithm period, and some timing constraints. The time unit we maintain throughout this paper is the clock cycle, and all time points are given by clock cycles $c \in \mathbb{Z}$. The operations represented by the nodes must be executed on dedicated processing units, i.e., we consider a one-to-one relation between operation types and processing unit types. In order to model the signal flow graph, we first define operation types and the corresponding processing unit types.

Definition II.1 (Operation types): The operation types are given by a 6-tuple $(T, \mathcal{I}, \mathcal{O}, r, s, a)$, where T is a set of types and for each $t \in T$

- $\mathcal{I}(t)$ is a set of *input ports;*
- $\mathcal{O}(t)$ is a set of *output ports;*
- $r(t,p) \in \mathbb{Z}$ is a *retiming;* for each port $p \in \mathcal{P}(t) = \mathcal{I}(t) \cup \mathcal{O}(t)$;
- $s(t) \in \mathbb{N}$ is a *restart time;* and
- $a(t) \geq 0$ is an area cost. □

The retiming of a port denotes the shift in time of that port. The restart time denotes the number of clock cycles that an execution of an operation occupies the corresponding processing unit, without interruption. If an operation of type t is scheduled in clock cycle $c \in \mathbb{Z}$, then the production or consumption of a variable at port $p \in \mathcal{P}(t)$ falls in clock cycle $c + r(t,p)$, and the processing unit on which the operation is executed is occupied in the clock cycles $c, \cdots, c + s(t) - 1$.

Input and output nodes of a signal flow graph are modeled as operations with no input ports and no output ports, respectively. Now we can define a signal flow graph as follows.

Definition II.2 (Signal flow graph): A signal flow graph G is given by a 6-tuple (V, t, I, O, E, d), where

- V is a set of operations;
- $t(v) \in T$ is the operation type, for each $v \in V$;
- $I = \{(v,i) | v \in V \wedge i \in \mathcal{I}(t(v))\}$ is a set of *operation input ports;*
- $O = \{(v,o) | v \in V \wedge o \in \mathcal{O}(t(v))\}$ is a set of *operation output ports;*
- $E \subseteq O \times I$ is a set of weighted, directed edges representing data dependencies; and
- $d(e) \in \mathbb{Z}$ is an edge weight, for each $e \in E$. □

An edge $e = (p,q) \in E$ with weight $d(e)$ denotes that for each execution of the signal flow graph, operation output port p produces a variable, which is consumed at operation input port q, $d(e)$ executions of the signal flow graph later. So, the variable produced at port p is delayed $d(e)$ executions. For each operation input port, the number of incoming edges is at most one.

B. Feasible Schedules

A signal flow graph is executed repeatedly, with an algorithm period $F \in \mathbb{N}$. Each execution is identified by an integer k. Since the executions are periodic, the kth execution takes place kF clock cycles after the zeroth execution. Therefore, we take the zeroth execution of the signal flow graph as a reference. For a broader discussion on periodic scheduling, we refer to [9].

Definition II.3 (Schedule): For a given signal flow graph G, a schedule $\sigma\colon V \to \mathbb{Z}$ assigns to each operation v a clock cycle $\sigma(v)$ in which the zeroth execution is scheduled. The set of all possible schedules is denoted by Σ. □

The timing constraints that can be imposed by the user give lower bounds and upper bounds on the clock cycles in which operations have to be scheduled, e.g., on input and output operations. For each operation $v \in V$ they are given as a closed interval $S(v) = [S_l(v), S_u(v)] \subseteq \mathbb{Z} \cup \{-\infty, +\infty\}$, which we call the *span* of operation v. Note that we use the interval notation for a discrete set of integer numbers, so $[a,b] = \{a, \cdots, b\}$. We use this notation throughout the paper.

A schedule is called feasible if and only if all operations are scheduled within their spans and all variables are produced before they are consumed. The first of these two constraints is equivalent to

$$\forall_{v \in V}\colon \sigma(v) \in S(v). \tag{1}$$

For the second constraint we first have to determine when variables are produced or consumed. The clock cycle in which a variable is produced at operation output port $p = (v,o) \in O$ in the zeroth execution of the signal flow graph, is $c(p) = \sigma(v) + r(t(v), o)$. Similarly, the clock cycle in which a variable is consumed at operation input port $q = (u,i) \in I$ in the zeroth execution of the signal flow graph, is $c(q) = \sigma(u) + r(t(u), i)$. Now, the constraint that variables must be produced before they are consumed, is equivalent to

$$\forall_{e=(p,q) \in E}\colon c(p) < c(q) + d(e)F. \tag{2}$$

The set of all feasible schedules is denoted by Σ'.

C. Objective Functions

Next, we determine an objective function, which reflects for a given schedule the total area cost. This area is determined by the maximum number of processing units of each type occupied simultaneously, the maximum number of variables that are simultaneously alive, and the maximum number of simultaneous memory accesses. Therefore, we define a set of *resource types* $T^* = T \cup \{t_v, t_a\}$, where t_v is the resource type corresponding to variable lifetimes, with area cost $a(t_v)$ per variable, and t_a is the resource type corresponding to memory accesses, with area cost $a(t_a)$ per access.

Definition II.4 (Requirement function): Given a signal flow graph G, a set of resource types T^*, and an algorithm period F, the requirement function $N\colon \Sigma \times T^* \times \mathbb{Z} \to \mathbb{N} \cup \{0\}$ gives for a schedule σ, resource type t, and clock cycle c the number of required resources $N(\sigma, t, c)$. This number is equal to

$$N(\sigma, t, c) = \sum_{k \in \mathbb{Z}} N_0(\sigma, t, c + kF),$$

where $N_0(\sigma, t, c)$ is the requirement function taking only execution zero of the signal flow graph into account. □

For processing units, the requirement function is given by

$$N_0(\sigma, t, c) = |\{v \in V | t(v) = t \wedge \sigma(v) \leq c < \sigma(v) + s(t)\}|.$$

For variable lifetimes, the requirement function is determined as follows. We restrict ourselves to variables produced at operation output ports p that are also consumed at some operation input ports q, since the other variables do not need to be stored. So we only consider operation output ports in $O' = \{p \in O | \exists_{q \in I}\colon (p,q) \in E\}$. Note that there is a one-to-one relation between variables and the operation output ports $p = (v,o)$ where they are produced, so we can denote a

variable by the port $p \in O'$ at which it is produced. Next, we assume that a variable is *alive* from the first clock cycle after its production up to and including the clock cycle of its last consumption. Then for execution zero of a signal flow graph, the number of variables alive in clock cycle c for a given schedule σ is given by

$$N_0(\sigma, t_v, c) = |\{p \in O' | c(p) < c \wedge \exists_{q:(p,q)\in E}: c(q) + d((p,q))F \geq c\}|.$$

For memory accesses, the requirement function is determined as follows. A variable produced in the zeroth execution of a signal flow graph at an operation output port $p \in O'$ causes an access in clock cycle c if and only if it is produced in clock cycle c, i.e., $c(p) = c$, or there exists an operation input port q with $(p,q) \in E$ for which the consumption falls in clock cycle c, i.e., $c(q) + d((p,q))F = c$. So

$$N_0(\sigma, t_a, c) = |\{p \in O' | c(p) = c \vee \exists_{q:(p,q)\in E}: c(q) + d((p,q))F = c\}|.$$

Definition II.5 (Area cost of a schedule): Given a signal flow graph G, a set of resource types T^*, and an algorithm period F, the area cost $f(\sigma)$ of a schedule is given by

$$f(\sigma) = \sum_{t\in T^*} a(t) \max_{c\in\mathbb{Z}} N(\sigma, t, c). \qquad \square$$

Note that since the requirement function N is periodic with period F, the maximum over $\mathbb{Z}$ can be replaced by a maximum over the discrete interval $[0, F-1]$.

D. Problem Formulation

Now the high-throughput DSP scheduling problem can be defined as follows.

Definition II.6 (High-Throughput DSP Scheduling (HTDS)): Given a signal flow graph G, a set of resource types T^*, an algorithm period F, and a span $S(v)$ for each operation $v \in V$, find a feasible schedule $\sigma \in \Sigma'$ such that the total area cost $f(\sigma)$ is minimal. $\square$

The decision variant of HTDS is NP-complete. For a proof, we refer to [28].

III. Basic Force-Directed Scheduling

In this section we present a mathematical model of the basic force-directed scheduling algorithm introduced by Paulin and Knight. For reasons of simplicity, the presentation is based on a less elaborate scheduling problem. In this less elaborate problem, which is used in this section only, precedence relations are explicitly given by a set of arcs between operations instead of data dependencies. Furthermore, we restrict ourselves to minimizing the area occupied by processing units, i.e., we do not take memory costs into account. In Section IV we show how the algorithm is applied to the more general HTDS problem.

A. Scheduling to Minimize Resources

We consider the following scheduling problem.

Definition III.1 (Scheduling to Minimize Resources (SMR)): Given are a set T of resource types t, each with cost $a(t) \geq 0$, and an acyclic, directed graph $G = (V, A)$, where V is a set of unit-time operations, and A is a set of arcs. Furthermore, we are given a makespan $M = [1, m] \subseteq \mathbb{Z}$, and for each operation $v \in V$ a corresponding resource type $t(v) \in T$. A schedule σ assigns to each operation $v \in V$ a clock cycle $\sigma(v) \in \mathbb{Z}$ in which v is scheduled. The set of all possible schedules is denoted by Σ. A schedule $\sigma \in \Sigma$ is called feasible if and only if

$$\forall_{v\in V}: \sigma(v) \in M \quad \text{and} \quad \forall_{(u,v)\in A}: \sigma(u) < \sigma(v).$$

Again, we denote the set of all feasible schedules by Σ'. Now the problem is to find a feasible schedule $\sigma \in \Sigma'$ that minimizes the total resource cost, given by

$$f(\sigma) = \sum_{t\in T} a(t) \max_{c\in M} N(\sigma, t, c), \tag{3}$$

where $N(\sigma, t, c) = |\{v \in V | t(v) = t \wedge \sigma(v) = c\}|$ is the requirement function, i.e., the number of operations of type t scheduled in clock cycle c in schedule σ. $\square$

The decision variant of SMR is NP-complete. A reduction from Precedence Constrained Scheduling [4] is straightforward.

To describe the force-directed scheduling algorithm we i) reformulate the problem by giving an approximate-cost function; and ii) introduce an iterative approach in order to find good solutions.

B. Towards a Solution Approach

The cost function of (3) can be rewritten as

$$f(\sigma) = \sum_{t\in T} a(t)\mu(t) + \sum_{t\in T} a(t) \max_{c\in M}(N(\sigma, t, c) - \mu(t)),$$

where $\mu(t) = (1/m)|\{v \in V | t(v) = t\}|$ is the average number of operations of type t over the makespan M. The first term of $f(\sigma)$ can be omitted since it is a constant, and with an eye to the solution method, we approximate the remaining term by

$$f'(\sigma) = \sum_{t\in T} a(t) \sum_{c\in M} (N(\sigma, t, c) - \mu(t))^2,$$

which reflects minimizing the resource cost by smoothing the distribution of resources over time, i.e., by minimizing the deviation from the average value. In the approximation algorithm we discuss below, this quadratic cost function has the advantage that it is more global than the cost function based on maxima. Rewriting $f'(\sigma)$ again, gives

$$\begin{aligned} f'(\sigma) &= \sum_{t\in T} a(t) \sum_{c\in M} (N(\sigma, t, c)^2 - 2N(\sigma, t, c)\mu(t) + \mu(t)^2) \\ &= \sum_{t\in T} a(t) \sum_{c\in M} N(\sigma, t, c)^2 - m \sum_{t\in T} a(t)\, \mu(t)^2, \end{aligned}$$

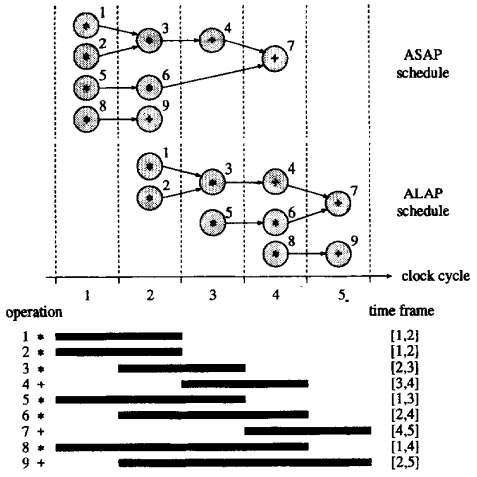

Fig. 1. An example of ASAP and ALAP schedules and the resulting initial time frames, depicted as black bars.

in which the second term is again a constant, that can be omitted. So, in the following, we use the approximate-cost function

$$f''(\sigma) = \sum_{t \in T} a(t) \sum_{c \in M} N(\sigma, t, c)^2, \tag{4}$$

and the mathematical motivation of force-directed scheduling that is given in the following sections, can be extended to any cost function of this form. Hence, the motivation is also valid for the extensions with memory related costs, as discussed in Sections IV-B-2) and IV-B-3).

C. An Iterative Approach

Next, we discuss an approximation algorithm that finds near-optimal solutions by constructing a sequence of partial solutions, where in each iteration an unscheduled operation is assigned to a clock cycle. The set of partial solutions can be defined as follows. Instead of a single clock cycle $\sigma(v)$ for each operation v, we have a *time frame,* which is a discrete interval $[\sigma_l(v), \sigma_u(v)] \subseteq \mathbb{Z}, \sigma_l(v) \leq \sigma_u(v)$, and which means that operation v has to be scheduled in a clock cycle $\sigma(v) \in [\sigma_l(v), \sigma_u(v)]$. In this way we get a *schedule frame* $\tilde{\sigma} = (\sigma_l, \sigma_u)$, and we denote the set of all possible schedule frames by $\tilde{\Sigma}$. A schedule frame $\tilde{\sigma}$ is called feasible if and only if

$$\forall_{v \in V}\colon \sigma_l(v) \in M \wedge \sigma_u(v) \in M \quad \text{and}$$
$$\forall_{(u,v) \in A}\colon \sigma_l(u) < \sigma_l(v) \wedge \sigma_u(u) < \sigma_u(v).$$

This in fact means that σ_l and σ_u are both feasible schedules. Now, the set of partial solutions is given by the set of feasible schedule frames, and is denoted by $\tilde{\Sigma}'$. Initially, the time frames are chosen as large as possible by initializing the $\sigma_l(v)$'s with the clock cycles of the ASAP (as soon as possible) schedule and the $\sigma_u(v)$'s with the clock cycles of the ALAP (as late as possible) schedule. Fig. 1 shows an example of the initial time frames of a graph with nine operations, two types of resources, and a makespan of length five.

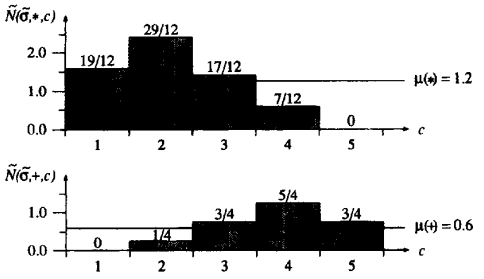

Fig. 2. An example of a resulting distribution function.

Next, the cost of a partial solution is estimated as follows. We define a *distribution function* $\tilde{N}(\tilde{\sigma}, t, c)$ as the expected number of operations of type t in clock cycle c in schedule frame $\tilde{\sigma}$. For this, we determine the probability $P(\tilde{\sigma}, v, c)$ that operation v is eventually scheduled in clock cycle c, given schedule frame $\tilde{\sigma}$. Because of the precedence relation, these probabilities are generally not independent. Nevertheless, we estimate them by assuming a uniform probability of assigning an operation to any clock cycle in its time frame, i.e.,

$$P(\tilde{\sigma}, v, c) = \begin{cases} 1/(\sigma_u(v) - \sigma_l(v) + 1) & \text{if } c \in [\sigma_l(v), \sigma_u(v)], \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

Now for each resource type t we take the summation of probabilities of the operations for each clock cycle $c \in M$. The resulting distribution function is given by

$$\tilde{N}(\tilde{\sigma}, t, c) = \sum_{v \in V(t)} P(\tilde{\sigma}, v, c), \tag{6}$$

where $V(t) = \{v \in V | t(v) = t\}$. See Fig. 2 for the distribution function for the initial schedule frame of the example in Fig. 1.

Next we define a cost function for schedule frames, similar to the approximate-cost function for schedules, by

$$\tilde{f}(\tilde{\sigma}) = \sum_{t \in T} a(t) \sum_{c \in M} \tilde{N}(\tilde{\sigma}, t, c)^2.$$

Note that if $\sigma_l(v) = \sigma_u(v) = \sigma(v)$, for all $v \in V$, then $\tilde{N}(\tilde{\sigma}, t, c) = N(\sigma, t, c)$, and thus $\tilde{f}(\tilde{\sigma}) = f''(\sigma)$. So we want to find a schedule frame $\tilde{\sigma}$ which minimizes $\tilde{f}(\tilde{\sigma})$, under the constraint $\sigma_l(v) = \sigma_u(v)$ for all $v \in V$.

Now, the iterative approach is defined as follows. First, a schedule frame $\tilde{\sigma}$ is initialized by initializing σ_l to the ASAP schedule and σ_u to the ALAP schedule. Next, in each iteration a *neighborhood* $\mathcal{N}(\tilde{\sigma}) \subseteq \tilde{\Sigma}'$ is searched for the best neighboring solution. A neighborhood $\mathcal{N}(\tilde{\sigma})$ of a schedule frame $\tilde{\sigma}$ consists of solutions $\tilde{\tau} = (\tau_l, \tau_u)$ that can be obtained by scheduling an operation v with $\sigma_l(v) < \sigma_u(v)$ in a clock cycle $c \in [\sigma_l(v), \sigma_u(v)]$, and updating the time frames of the other operations. See Fig. 3 for an example of the effect of scheduling operation 7 in clock cycle 4 in the example of Fig. 1. The assignment of an operation v to a clock cycle c is chosen such that a neighboring schedule frame $\tilde{\tau} \in \mathcal{N}(\tilde{\sigma})$ is obtained, for which

$$\Delta\tilde{f}(\tilde{\sigma}, \tilde{\tau}) = \tilde{f}(\tilde{\tau}) - \tilde{f}(\tilde{\sigma})$$

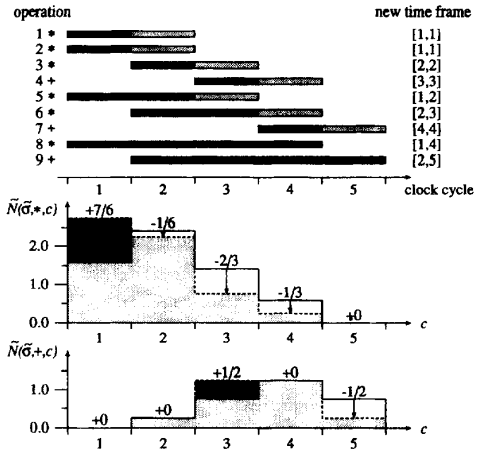

Fig. 3. The effect of scheduling operation 7 in clock cycle 4 on the time frames and on the distribution function.

is minimal. Iterations are repeated until a feasible schedule is obtained.

This iterative approach is identical to basic force-directed scheduling as defined by Paulin and Knight except that in their approach the assignment of operation v to clock cycle c is chosen for which a so-called *force* $F(v,c)$ is minimal, instead of $\Delta\tilde{f}$. The definition of $F(v,c)$ is given in Section III-D. The difference is only minor. Indeed, if we define

$$\Delta\tilde{N}(\tilde{\sigma},\tilde{\tau},t,a) = \tilde{N}(\tilde{\tau},t,a) - \tilde{N}(\tilde{\sigma},t,a),$$

for all $t \in T$ and $a \in M$, then we obtain

$$\begin{aligned}\Delta\tilde{f}(\tilde{\sigma},\tilde{\tau}) &= \sum_{t\in T} a(t) \sum_{a\in M} [(\tilde{N}(\tilde{\sigma},t,a) \\ &\quad + \Delta\tilde{N}(\tilde{\sigma},\tilde{\tau},t,a))^2 - \tilde{N}(\tilde{\sigma},t,a)^2] \\ &= \sum_{t\in T} a(t) \sum_{a\in M} (2\tilde{N}(\tilde{\sigma},t,a) \\ &\quad + \Delta\tilde{N}(\tilde{\sigma},\tilde{\tau},t,a))\Delta\tilde{N}(\tilde{\sigma},\tilde{\tau},t,a).\end{aligned}$$

Hence,

$$\begin{aligned}\tfrac{1}{2}\Delta\tilde{f}(\tilde{\sigma},\tilde{\tau}) &= \sum_{t\in T} a(t) \sum_{a\in M} \tilde{N}(\tilde{\sigma},t,a)\Delta\tilde{N}(\tilde{\sigma},\tilde{\tau},t,a) \\ &\quad + \sum_{t\in T} a(t) \sum_{a\in M} \tfrac{1}{2}(\Delta\tilde{N}(\tilde{\sigma},\tilde{\tau},t,a))^2. \qquad (7)\end{aligned}$$

As we show in the following sections, the first term in this equation is equal to $F(v,c)$ and the second term strongly resembles the look-ahead introduced by Paulin and Knight, as a modification of the first term.

D. Springs and Forces

For the selection of the assignment of an operation v to a clock cycle c, Paulin and Knight introduced a force $F(v,c)$ in the following way. Consider a schedule frame $\tilde{\sigma}$ and the schedule frame $\tilde{\tau} \in \mathcal{N}(\tilde{\sigma})$ that is obtained by scheduling operation v in clock cycle c. The changes in the probabilities $P(\tilde{\sigma},u,a)$ of all operations $u \in V$ are determined as

$$\Delta P(\tilde{\sigma},\tilde{\tau},u,a) = P(\tilde{\tau},u,a) - P(\tilde{\sigma},u,a).$$

Then a self force, a successor force, and a predecessor force are determined, which are added to yield the *total force*, given by

$$\begin{aligned}F(v,c) &= \sum_{u\in V}\sum_{a\in M} \Delta P(\tilde{\sigma},\tilde{\tau},u,a)\ \tilde{N}(\tilde{\sigma},t(u),a)\ a(t(u)) \\ &= \sum_{t\in T} a(t) \sum_{a\in M}\sum_{u\in V(t)} \Delta P(\tilde{\sigma},\tilde{\tau},u,a)\ \tilde{N}(\tilde{\sigma},t,a) \\ &= \sum_{t\in T} a(t) \sum_{a\in M} \Delta\tilde{N}(\tilde{\sigma},\tilde{\tau},t,a)\ \tilde{N}(\tilde{\sigma},t,a), \qquad (8)\end{aligned}$$

which is equal to the first term in (7). The total force can be seen as a sum of forces needed to stretch a number of springs, each with a spring constant $\tilde{N}(\tilde{\sigma},t(u),a)\ a(t(u))$, by an amount $\Delta P(\tilde{\sigma},\tilde{\tau},u,a)$.

E. Look-Ahead

To improve the effectiveness of the force-directed scheduling algorithm, Paulin and Knight [21] proposed a *look-ahead scheme*. The idea is to replace the value of $\tilde{N}(\tilde{\sigma},t(u),a)$ in (8) by a value somewhere between the current one and the value that would be obtained after the current iteration. They then propose to replace it by $\tilde{N}(\tilde{\sigma},t(u),a) + \eta\Delta P(\tilde{\sigma},\tilde{\tau},u,a)$, but according to the idea, one should replace it by $\tilde{N}(\tilde{\sigma},t(u),a) + \eta\Delta\tilde{N}(\tilde{\sigma},\tilde{\tau},t(u),a)$. This gives an additional term to the total force in (8), which is equal to

$$\begin{aligned}&\sum_{u\in V}\sum_{a\in M} \Delta P(\tilde{\sigma},\tilde{\tau},u,a)\ \eta\Delta\tilde{N}(\tilde{\sigma},\tilde{\tau},t(u),a)\ a(t(u)) \\ &\quad = \sum_{t\in T} a(t) \sum_{a\in M}\sum_{u\in V(t)} \Delta P(\tilde{\sigma},\tilde{\tau},u,a)\ \eta\Delta\tilde{N}(\tilde{\sigma},\tilde{\tau},t,a) \\ &\quad = \sum_{t\in T} a(t) \sum_{a\in M} \eta\ (\Delta\tilde{N}(\tilde{\sigma},\tilde{\tau},t,a))^2.\end{aligned}$$

If $\eta = \frac{1}{2}$, the additional term is exactly equal to the second term in (7).

F. The Algorithm

The basic force-directed scheduling algorithm can now be summarized as follows.

Basic force-directed scheduling

Step 1 Initialize time frames to ASAP and ALAP schedules.

Step 2 Calculate the distribution function.

Step 3 For each operation v that is not scheduled yet, and each clock cycle $c \in [\sigma_l(v), \sigma_u(v)]$, calculate $\Delta\tilde{f}(\tilde{\sigma},\tilde{\tau})$, where $\tilde{\tau} \in \mathcal{N}(\tilde{\sigma})$ is the feasible schedule frame obtained from $\tilde{\sigma}$ by scheduling operation v in clock cycle c.

Step 4 Schedule that operation v in that clock cycle c for which $\Delta\tilde{f}(\tilde{\sigma},\tilde{\tau})$ is minimal; i.e., assign $\sigma_l(v) = \sigma_u(v) = c$.

Step 5 Update time frames of predecessors and successors of v.

Step 6 Update the distribution function.

Step 7 If not all operations are scheduled, return to Step 3.

Here we say that an operation v is scheduled if and only if $\sigma_l(v) = \sigma_u(v)$.

G. Time Complexity

The worst-case time complexity of the basic force-directed scheduling algorithm is $\mathcal{O}(mln^3)$, when implemented in a straightforward way, where m is the length of the makespan, l is the maximum length of the initial time frame of any of the operations, and n is the number of operations. This complexity can be derived as follows.

- In each iteration at least one operation is scheduled. As a side effect of scheduling an operation, successor or predecessor operations may be scheduled too, so at most n iterations are needed.
- In each iteration, there are at most n operations that still must be scheduled.
- For each of these operations there are at most l clock cycles in which it can be scheduled.
- For each tentative scheduling of an operation in a clock cycle, the probabilities of $\mathcal{O}(n)$ operations may change and there are m clock cycles where they may change, so calculating $\Delta\tilde{f}$ takes $\mathcal{O}(mn)$ steps.

We return to the time complexity in Section VI.

IV. Application in High-Throughput DSP

In this section we discuss how we can apply force-directed scheduling to the high-throughput DSP scheduling problem. First, we discuss how to handle the precedence constraints, next we discuss the distribution function, and finally we discuss the impact of these modifications on the algorithm's time complexity.

A. Precedence Constraints

In high-throughput DSP scheduling, precedence constraints are due to the fact that variables must be produced before they are consumed. In (2) this is formulated in terms of the clock cycles in which variables are produced and consumed at operation ports. In terms of the clock cycles in which operations are scheduled, these constraints are given by

$$\forall_{e=((u,o),(v,i))\in E}: \quad \sigma(u) + r(t(u), o) < \sigma(v) + r(t(v), i) + d(e)F. \quad (9)$$

On the given node set V, we now define an arc set $A \subseteq V \times V$ as

$$A = \{(u, v) | \exists_{o,i}: ((u, o), (v, i)) \in E\},$$

and we define weights $w(a)$ for arcs $a = (u, v) \in A$ as

$$w(a) = \max\{r(t(u), o) - r(t(v), i) - d(e)F + 1 | e = ((u, o), (v, i)) \in E\}.$$

Now (9) is equivalent to

$$\forall_{(u,v)\in A}: \sigma(v) - \sigma(u) \geq w((u, v)). \quad (10)$$

In this way we get a weighted, directed graph $H = (V, A)$, which we call a *precedence graph*. In general, weights may be negative, and H may contain cycles. However, if H contains a cycle for which the sum of the arc weights is positive, then no schedule σ exists for which (10) holds, and hence no feasible schedule exists.

B. Distribution Functions

Similar to the requirement function, we determine the distribution function $\tilde{N}_0(\tilde{\sigma}, t, c)$ for all $t \in T^*$ considering only execution zero of the signal flow graph, and with that function we determine the distribution function

$$\tilde{N}(\tilde{\sigma}, t, c) = \sum_{k\in\mathbb{Z}} \tilde{N}_0(\tilde{\sigma}, t, c + kF).$$

Note that similar to the requirement function, also the distribution function $\tilde{N}$ is periodic with a period F. Therefore it is sufficient to determine it for one period, $[0, F-1]$, and compute the $\Delta\tilde{f}$ criterion only over the clock cycles in $[0, F-1]$.

We next have to determine the distribution function for processing units, for variable lifetimes, and for memory accesses. Once obtained, the application of force-directed scheduling to the high-throughput DSP scheduling problem is straightforward. Before we discuss the distribution function in more detail, we mention that the first demand posed on it is that it should be equal to the requirement function if all operations are scheduled.

1) Distribution functions for processing units: The distribution function for processing unit types is similar to the one used in basic force-directed scheduling in Section III, with the exception that now an operation can occupy a processing unit for more than one clock cycle. This is one of the refinements that were also presented by Paulin and Knight [21]. The probability that operation v occupies a processing unit in clock cycle c is given by

$$P(\tilde{\sigma}, v, c) = \frac{|\{a \in [\sigma_l(v), \sigma_u(v)] | a \leq c < a + s(t(v))\}|}{\sigma_u(v) - \sigma_l(v) + 1}.$$

The distribution function for processing units is then given by

$$\tilde{N}_0(\tilde{\sigma}, t, c) = \sum_{v\in V(t)} P(\tilde{\sigma}, v, c). \quad (11)$$

2) Distribution functions for variable lifetimes: In this section we determine the probability that a variable is alive in a clock cycle. The probability function presented by Paulin and Knight [21] is very rough, and leads to a very inaccurate distribution. For instance, for an inverse discrete cosine transformation application (IDCT, see Fig. 11), the maximum number of variables at the beginning of the scheduling algorithm was estimated at 275, whereas after scheduling the eventual maximum number of variables was 19. This is too inaccurate to make good quantitative trade-offs. With the probability function described below, the initial maximum number of variables is about 20, and the eventual number is 17.

This more refined probability function is derived as follows. As already mentioned, we denote a variable by the operation output port at which it is produced. First, we determine in which clock cycles variable $p = (u, o) \in O'$ may begin to live, and in which ones it may end living.

The earliest possible clock cycles for the beginning and the end of a variable's lifetime are determined by the earliest possible schedule, σ_l, and the latest ones are determined by the schedule σ_u. The earliest possible clock cycle in which variable p can begin to live, is $b_l(p) = \sigma_l(u) + r(t(u), o) + 1$,

and the latest possible clock cycle in which it can begin to live, is $b_u(p) = \sigma_u(u) + r(t(u), o) + 1$. Note that a variable begins to live one clock cycle after it is produced, as discussed in Section II.

Given a schedule σ, then variable $p = (u, o)$ is consumed at operation input port $q = (v, i)$, with $(p, q) \in E$, in clock cycle $\sigma(v) + r(t(v), i) + d((p, q))F$. Considering all consumptions, variable p ends living in clock cycle $\max\{\sigma(v) + r(t(v), i) + d(e)F | e = ((u, o), (v, i)) \in E\}$. So, for the end of variable p's lifetime, the earliest possible clock cycle is given by

$$e_l(p) = \max\{\sigma_l(v) + r(t(v), i) + d(e)F | e = ((u, o), (v, i)) \in E\}.$$

The latest possible clock cycle is given by

$$e_u(p) = \max\{\sigma_u(v) + r(t(v), i) + d(e)F | e = ((u, o), (v, i)) \in E\}.$$

Now eventually, when all operations are scheduled, variable p begins to live in a clock cycle b and ends living in a clock cycle e for which $b_l(p) \leq b \leq b_u(p), e_l(p) \leq e \leq e_u(p)$, and $b \leq e$. So, its lifetime can be represented by a pair $(b, e) \in L(p)$, where

$$L(p) = \{(b, e) | b_l(p) \leq b \leq b_u(p) \wedge e_l(p) \leq e \leq e_u(p) \wedge b \leq e\}.$$

If we now assume that all pairs $(b, e) \in L(p)$ occur with equal probability in the eventual schedule, then the probability that variable p is alive in clock cycle c is given by

$$P_v(\tilde{\sigma}, p, c) = \frac{|\{(b, e) \in L(p) | b \leq c \leq e\}|}{|L(p)|}.$$

If $b_u(p) \leq e_l(p)$, then this function is piecewise linear. If $b_u(p) > e_l(p)$, the function is quadratic, but since a piecewise linear function has some computational advantages, we replace it by a piecewise linear approximation. For more detail, we refer to [28].

The distribution function for variable lifetimes is now given by

$$\tilde{N}_0(\tilde{\sigma}, t_v, c) = \sum_{p \in O'} P_v(\tilde{\sigma}, p, c).$$

3) Access distribution functions: For memory accesses we again have to consider only variables in O'. Let $p = (u, o) \in O'$ be such a variable. A memory access for this variable occurs in a clock cycle c if and only if a write or a read action for this variable takes place in clock cycle c; simultaneous actions for one variable result in only one access. We first determine the probabilities of these actions.

The write action for variable p takes place in a clock cycle in $[w_l(p), w_u(p)]$, where

$$w_l(p) = \sigma_l(u) + r(t(u), o) = b_l(p) - 1$$

and

$$w_u(p) = \sigma_u(u) + r(t(u), o) = b_u(p) - 1.$$

If we assume uniform probability for all clock cycles in this interval, then the probability that a write action occurs in clock cycle c is given by

$$P_w(\tilde{\sigma}, p, c) = \begin{cases} 1/(w_u(p) - w_l(p) + 1) & \\ \quad \text{if } c \in [w_l(p), w_u(p)], & \\ 0 \quad \text{otherwise.} & \end{cases}$$

Read actions only take place at operation input ports in $I' = \{q \in I | \exists_{p \in O}: (p, q) \in E\}$, i.e., at operation input ports that are connected to operation output ports. Note that each port $q \in I'$ is connected to exactly one $p \in O$. For port $q = (v, i) \in I'$, a read action of the variable p with $e = (p, q) \in E$ takes place in a clock cycle in $[r_l(q), r_u(q)]$, where $r_l(q) = \sigma_l(v) + r(t(v), i) + d(e)F$ and $r_u(q) = \sigma_u(v) + r(t(v), i) + d(e)F$. If we again assume uniform probability for all clock cycles in this interval, then the probability that a read action at operation input port q occurs in clock cycle c, is given by

$$P_r(\tilde{\sigma}, q, c) = \begin{cases} 1/(r_u(q) - r_l(q) + 1) & \text{if } c \in [r_l(q), r_u(q)], \\ 0 & \text{otherwise.} \end{cases}$$

Now the probability that a memory access for variable p occurs in clock cycle c is equal to the probability that a write or a read action occurs. If we assume that these actions are mutually independent, then the memory access probability is given by

$$P_a(\tilde{\sigma}, p, c) = 1 - (1 - P_w(\tilde{\sigma}, p, c)) \cdot \prod_{q:(p,q) \in E} (1 - P_r(\tilde{\sigma}, q, c)).$$

Given the memory access probability function, the memory access distribution function is given by

$$\tilde{N}_0(\tilde{\sigma}, t_a, c) = \sum_{p \in O'} P_a(\tilde{\sigma}, p, c).$$

C. The Algorithm for HTDS

The force-directed scheduling algorithm for high-throughput DSP scheduling can now be derived straightforwardly. The outline of the algorithm is the same as for the basic force-directed scheduling algorithm, discussed in Section III-F. However, the ASAP and ALAP schedules, and the neighborhood $\mathcal{N}(\tilde{\sigma})$ are now based on the new precedence constraints. Furthermore, the $\Delta\tilde{f}$ criterion is now given by

$$\Delta\tilde{f}(\tilde{\sigma}, \tilde{\tau}) = \sum_{t \in T^*} a(t) \sum_{c \in [0, F-1]} (2\tilde{N}(\tilde{\sigma}, t, c) + \Delta\tilde{N}(\tilde{\sigma}, \tilde{\tau}, t, c)) \Delta\tilde{N}(\tilde{\sigma}, \tilde{\tau}, t, c), \quad (12)$$

where

$$\Delta\tilde{N}(\tilde{\sigma}, \tilde{\tau}, t, c) = \tilde{N}(\tilde{\tau}, t, c) - \tilde{N}(\tilde{\sigma}, t, c), \quad (13)$$

for all resource types $t \in T^*$ and clock cycles $c \in [0, F - 1]$.

D. Time Complexity Revisited

The worst-case time complexity of the algorithm when applied to the high-throughput DSP scheduling problem, is $\mathcal{O}(Fln^3)$, compared to $\mathcal{O}(mln^3)$ for the basic algorithm; see Section III-G. Here, n is the number of operations, l is the maximum length of the initial time frame of any of the operations, and F is the algorithm period. The difference stems from the calculation of $\Delta\tilde{N}$.

In the basic algorithm, a tentative assignment of an operation to a clock cycle may change the probability functions of $\mathcal{O}(n)$ operations, and they may change in m clock cycles, so calculating $\Delta\tilde{N}$ takes $\mathcal{O}(mn)$ steps. In the algorithm for the high-throughput DSP scheduling problem, a tentative assignment of an operation to a clock cycle may change the time frames of $\mathcal{O}(n)$ operations, and thus $\mathcal{O}(n)$ probability functions for processing units, variable lifetimes, and memory accesses may change. So, $\mathcal{O}(n)$ operations and variables contribute to $\Delta\tilde{N}_0$. Next, $\Delta\tilde{N}$ has to be computed over one period $[0, F-1]$, which is given by

$$\Delta\tilde{N}(\tilde{\sigma},\tilde{\tau},t,c) = \sum_{k\in\mathbb{Z}} \Delta\tilde{N}_0(\tilde{\sigma},\tilde{\tau},t,c+kF),$$

and since the probability functions are piecewise linear, this can be done in a total of $\mathcal{O}(Fn)$ steps.

V. Effectiveness Improvements

In this section we discuss two modifications of the algorithm that on the average increase the quality of the solutions without changing its worst-case time complexity. These modifications are the use of *global spring constants* and *gradual time-frame reduction.*

A. Global Spring Constants

Consider the $\Delta\tilde{f}$ criterion of (12) again as a sum of forces needed for displacements $\Delta\tilde{N}(\tilde{\sigma},\tilde{\tau},t,c)$ of springs with spring constants $a(t)(\tilde{N}(\tilde{\sigma},t,c) + \frac{1}{2}\Delta\tilde{N}(\tilde{\sigma},\tilde{\tau},t,c))$. The factor $a(t)$ weighs the contribution of the different types of resources.

Now, the situation can occur that a neighbor $\tilde{\tau} \in \mathcal{N}(\tilde{\sigma})$ results in a decrease of $\tilde{N}(\tilde{\sigma},t_1,c_1)$ for some value of c_1 with $\tilde{N}(\tilde{\sigma},t_1,c_1) \ll \max_c \tilde{N}(\tilde{\sigma},t_1,c)$, at the expense of an increase of $\tilde{N}(\tilde{\sigma},t_2,c_2)$ for some value of c_2 with $\tilde{N}(\tilde{\sigma},t_2,c_2) \approx \max_c \tilde{N}(\tilde{\sigma},t_2,c)$, as is shown in Fig. 4. This neighbor is undesirable, since we want to minimize

$$\sum_{t\in T^*} a(t) \max_{c\in\mathbb{Z}} \tilde{N}(\tilde{\sigma},t,c),$$

but negative contributions to $\Delta\tilde{f}$ of resource type t_1 might compensate positive contributions to $\Delta\tilde{f}$ of type t_2, which might lead to an acceptance of such a neighbor. To prevent this, spring constants should be smaller if the distribution function of a resource type is far from its maximum value, and larger if it is close to it.

Furthermore, there is a second unwanted effect of using the $\Delta\tilde{f}$ criterion. If we double the area cost $a(t)$ of a certain resource type, then the contribution of that type to $\Delta\tilde{f}$ is also doubled. However, if we double the distribution function of resource type t, which also leads to twice as much area for that type of resource, then this leads to a four times larger contribution to $\Delta\tilde{f}$. This is a direct consequence of the approximate-cost function of (4). So we should change the $\Delta\tilde{f}$ criterion in such a way that it is linear in the distribution function, in order to get a better trade-off between a large number of low-cost resources and a small number of high-cost ones.

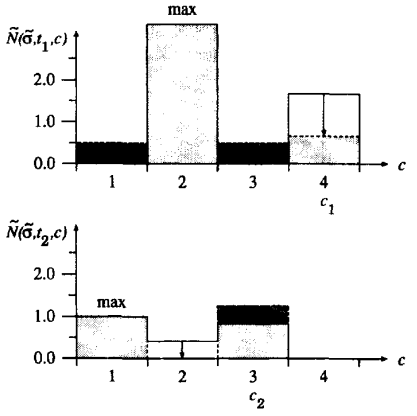

Fig. 4. The effect of an unfavorable neighbor which may be selected due to a large negative $\Delta\tilde{f}$.

To overcome these two unwanted effects we propose the following. First, new spring constants are introduced, given by

$$S(\tilde{\sigma},\tilde{\tau},t,c) = (\tilde{N}(\tilde{\sigma},t,c) + \eta\Delta\tilde{N}(\tilde{\sigma},\tilde{\tau},t,c)) \cdot \left(\frac{\tilde{N}(\tilde{\sigma},t,c) + \eta\Delta\tilde{N}(\tilde{\sigma},\tilde{\tau},t,c)}{\max_c \tilde{N}(\tilde{\sigma},t,c)}\right)^z, \quad (14)$$

where z is a positive number, and η is again a look-ahead factor. Experiments, not shown here, reveal that a good value for z is in the range $1,\cdots,5$, and that a good value for η is $\frac{1}{3}$. The latter is in accordance with observations made by Paulin and Knight [21]. Next, for each resource type $t \in T^*$, a subforce is calculated, given by

$$F_{\text{sub}}(\tilde{\sigma},\tilde{\tau},t) = \sum_{c\in[0,F-1]} \Delta\tilde{N}(\tilde{\sigma},\tilde{\tau},t,c)S(\tilde{\sigma},\tilde{\tau},t,c),$$

and then a total force is calculated, given by

$$F_{\text{gsc}}(\tilde{\sigma},\tilde{\tau}) = \sum_{t\in T^*} a(t)\,\text{sign}(F_{\text{sub}}(\tilde{\sigma},\tilde{\tau},t)) \cdot \sqrt{|F_{\text{sub}}(\tilde{\sigma},\tilde{\tau},t)|}, \quad (15)$$

where $\text{sign}(x) = 1$ if $x>0$, 0 if $x=0$, and -1 if $x<0$. In this way performing trade-offs is done more sensibly. Note that this change in spring constants and force function in fact means that we change the approximate-cost function. The extra work to be done in the algorithm is the determination of the maxima of the distribution function in each iteration. This however does not change the worst-case time complexity of the algorithm.

B. Gradual Time-Frame Reduction

The second modification we propose concerns a different iteration process for obtaining feasible schedules, which means that we define a new neighborhood structure $\mathcal{N}_g$. The aim is to reduce the time frames more gradually in order to obtain a less greedy behavior of the algorithm. Instead of reducing in each iteration the time frame $[\sigma_l(v),\sigma_u(v)]$ of an operation v to one clock cycle c, which is quite a large step, we reduce it

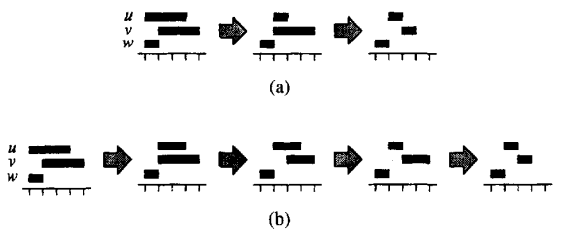

Fig. 5. Possible reductions of time frames in (a) the original iteration process and in (b) the one with gradual time-frame reduction.

by only one clock cycle at a time, i.e., to $[\sigma_l(v)+1, \sigma_u(v)]$ or to $[\sigma_l(v), \sigma_u(v)-1]$. An example is given in Fig. 5. Generally, this leads to more iterations, but in an iteration there are now at most $2n$ neighbor schedules $\tilde{\tau} \in \mathcal{N}_g(\tilde{\sigma})$ that must be investigated.

The criterion for selecting a neighbor solution is based on the forces F_{gsc} of scheduling operations in the left-most clock cycle in their time frames, and in the right-most clock cycle in their time frames. Let $F_{left}(v)$ denote the force F_{gsc} for scheduling operation v in clock cycle $\sigma_l(v)$, and let $F_{right}(v)$ denote the force F_{gsc} for scheduling v in clock cycle $\sigma_u(v)$.

Now we first determine $F_{max}(v) = \max\{F_{left}(v), F_{right}(v)\}$, which reflects the worst possible effect of scheduling operation v in the left-most or right-most clock cycle in its time frame. Next, we determine a quantity for the best possible effect of scheduling operation v somewhere in its time frame. This means that we have to determine the minimum F_{gsc} of scheduling v in a clock cycle $c \in [\sigma_l(v), \sigma_u(v)]$, but in order to save computation time, we estimate F_{gsc} for the clock cycles $c \in [\sigma_l(v)+1, \sigma_u(v)-1]$ roughly to be 0. Thus, the best possible effect of scheduling operation v somewhere in its time frame is reflected by

$$F_{min}(v) = \begin{cases} \min\{F_{left}(v), F_{right}(v)\} \\ \quad \text{if } \sigma_u(v) - \sigma_l(v) \le 1, \\ \min\{F_{left}(v), 0, F_{right}(v)\} \\ \quad \text{if } \sigma_u(v) - \sigma_l(v) > 1. \end{cases}$$

Next, we determine $F_{gain}(v) = F_{max}(v) - F_{min}(v) \ge 0$, which reflects the possible improvement by reducing the time frame of operation v.

Now we choose operation v for which $F_{gain}(v)$ is maximal and we reduce its time frame at the worst side, i.e., we reduce its time frame to $[\sigma_l(v)+1, \sigma_u(v)]$ if $F_{max}(v) = F_{left}(v)$, and to $[\sigma_l(v), \sigma_u(v)-1]$ if $F_{max}(v) = F_{right}(v)$. In case $F_{max}(v) = F_{left}(v) = F_{right}(v)$, we make an arbitrary choice. Then the time frames and distribution function are updated, and the same procedure is repeated until $\sigma_l(v) = \sigma_u(v)$ for all $v \in V$, which corresponds to a feasible schedule.

In this way, the time frames of the operations are gradually reduced, and the distribution function gradually becomes a better estimate of the final requirement function. So, the definite scheduling of operations in clock cycles is postponed until later, when the distribution function is a better estimate. This modification makes the algorithm less greedy, and we may expect that the algorithm generally obtains better solutions.

The worst-case time complexity of the algorithm remains $\mathcal{O}(Fln^3)$, which can be seen as follows. In each iteration at least one operation's time frame is reduced by at least one clock cycle, leading to a maximum of ln iterations. On the other hand, in each iteration only two instead of $\mathcal{O}(l)$ forces F_{gsc} have to be calculated for each operation.

VI. Efficiency Improvements

Since the time complexity of force-directed scheduling is rather large compared to other techniques such as list scheduling, which can be implemented to run in $\mathcal{O}(n^2)$ or $\mathcal{O}(n \log n)$ time depending on the chosen priority rules, Paulin proposes an alternative method to compute the forces of (8) in the basic force-directed scheduling algorithm [20]. Roughly speaking, this is achieved by the following. First, in each iteration, calculate for each operation $v \in V$ and each clock cycle $c \in [\sigma_l(v), \sigma_u(v)]$

- the self force of scheduling v in clock cycle c;
- the contribution to other operation's successor forces due to a reduction of v's time frame to $[c, \sigma_u(v)]$; and
- the contribution to other operation's predecessor forces due to a reduction of v's time frame to $[\sigma_l(v), c]$.

Second, the precedence graph is traversed in a forward direction, in a breadth-first way, and for each operation the corresponding predecessor forces of its direct predecessors are added to its own predecessor forces. Third, the successor forces are added similarly by traversing the graph backwards. Finally, the total forces are calculated. For more detail, see [20]. In this way the time complexity of the basic force-directed scheduling algorithm is reduced from $\mathcal{O}(mln^3)$ to $\mathcal{O}(mln^2)$.

Unfortunately, this alternative method does not yield the original forces of (8) as was claimed by Paulin; it corresponds to a different set of forces, due to the fact that successor and predecessor forces are multiply counted in the case there are reconvergent paths in the graph. Furthermore, the approach cannot be applied to the following cases.

- For cyclic precedence graphs, it is not clear how to traverse them in a forward or backward direction.
- To determine forces in the case that the distribution function for variable lifetimes and/or memory accesses is used, in order to take memory cost into account, one must consider several operations simultaneously, namely the producing operation and all consuming operations of each variable. This leads to an increase of the time complexity, instead of a decrease.
- Look-ahead schemes different from the one proposed by Paulin and Knight may require the evaluation of all changes in the probabilities to calculate the forces. This also leads to an increase of the time complexity.

The points given above have strengthened our belief that the approach falls short on several important points and therefore we developed a different way to reduce the time complexity. To this end, we first discuss some properties of the precedence constraints. Next, we present an incremental way of computing

the forces, and we show that the time complexity is reduced by a factor n.

A. Precedence Constraints

A basic step in the force-directed scheduling algorithm is the calculation of the effect of the change of an operation's time frame on the time frames of the other operations. This can be done either by traversing the precedence graph each time that an operation's time frame changes, or by means of all-pairs longest paths. The latter can be done as follows.

For each pair of operations $u, v \in V$, the maximum length $m(u,v)$ of a path in the precedence graph from u to v is determined, in terms of the sum of the arc weights on the path, with $m(u,v) = -\infty$ if no path from u to v exists. Then (10) is equivalent to

$$\forall_{u,v \in V}\colon \sigma(v) - \sigma(u) \geq m(u,v),$$

and the precedence constraint for schedule frames is equivalent to

$$\forall_{u,v \in V}\colon \sigma_{\mathrm{l}}(v) - \sigma_{\mathrm{l}}(u) \geq m(u,v) \wedge \sigma_{\mathrm{u}}(v) - \sigma_{\mathrm{u}}(u) \geq m(u,v).$$

Note that if the precedence graph contains cycles of positive length, then $m(u,v)$ is not properly defined, but in that case no feasible schedule exists. Furthermore, note that the all-pairs longest paths have to be determined only once.

If we now have a feasible schedule frame $\tilde{\sigma}$ and we change the time frame of an operation u to $[\sigma'_{\mathrm{l}}(u), \sigma'_{\mathrm{u}}(u)] \subseteq [\sigma_{\mathrm{l}}(u), \sigma_{\mathrm{u}}(u)]$, then the time frame of an operation v becomes $[\sigma'_{\mathrm{l}}(v), \sigma'_{\mathrm{u}}(v)]$, where

$$\begin{aligned} \sigma'_{\mathrm{l}}(v) &= \max\{\sigma_{\mathrm{l}}(v), \sigma'_{\mathrm{l}}(u) + m(u,v)\} \quad \text{and} \\ \sigma'_{\mathrm{u}}(v) &= \min\{\sigma_{\mathrm{u}}(v), \sigma'_{\mathrm{u}}(u) - m(v,u)\}. \end{aligned} \tag{16}$$

So, the effect on the other operations' time frames is easy to determine. Furthermore, the effect of u on v is completely determined by $m(u,v)$ and $m(v,u)$, i.e., it is independent of the time frames of other operations. This characteristic is used in the next section.

B. Distribution Function Differences

Running the force-directed scheduling algorithm with gradual time-frame reduction for a number of examples shows that the average number of operations for which the time frames have changed in an iteration is rather low, i.e., typically less than two operations; see Fig. 16. This opens the possibility for further efficiency improvements by making use of incremental calculations. The amount of work to be done in one iteration is then substantially reduced if only a few operations are affected. On the other hand, if many operations are affected in an iteration, a large step has been made towards a final solution. So, if little progress has been made, little work has to be done, and if much progress has been made, the same amount of work has to be done as in the case where no use is made of incremental calculations. This results in a lower order worst-case time complexity, as we show in Section VI-D.

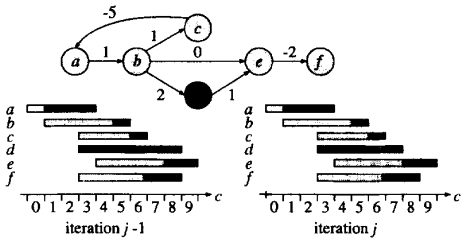

Fig. 6. A precedence graph and the operations' time frames before and after scheduling operation b in clock cycle 5, in iterations $j-1$ and j. In iteration $j-1$ only operation d's time frame is changed, from [3, 8] to [3, 7]. So, only the effect on this operation is different.

Let the schedule frame $\tilde{\sigma}(j)$ and the time frames $[\sigma_{\mathrm{l}}(v,j), \sigma_{\mathrm{u}}(v,j)]$ be functions of the iteration number j, and let for an iteration $j, C(j)$ be the subset of operations for which the time frames have changed in the previous iteration, so

$$\begin{aligned} C(j) = \{v \in V | &\sigma_{\mathrm{l}}(v,j) \neq \sigma_{\mathrm{l}}(v,j-1) \vee \\ &\sigma_{\mathrm{u}}(v,j) \neq \sigma_{\mathrm{u}}(v,j-1)\}. \end{aligned}$$

Now, for each operation $v \in V$ and for clock cycles $c = \sigma_{\mathrm{l}}(v,j)$ and $c = \sigma_{\mathrm{u}}(v,j)$, we want to determine $F_{\mathrm{gsc}}(\tilde{\sigma}(j), \tilde{\tau}(j))$ of (15), where $\tilde{\tau}(j)$ is the schedule frame obtained from $\tilde{\sigma}(j)$ by scheduling operation v in clock cycle c. For this, we distinguish between two ways of computing $\Delta\tilde{N}(\tilde{\sigma}(j), \tilde{\tau}(j), t, a)$.

If $v \in C(j)$, then the effect on other operations' time frames due to scheduling operation v in clock cycle c is most likely different from the effect in the previous iteration. Therefore, we just compute $\Delta\tilde{N}$ as given by (13), i.e.,

$$\Delta\tilde{N}(\tilde{\sigma}(j), \tilde{\tau}(j), t, a) = \tilde{N}(\tilde{\tau}(j), t, a) - \tilde{N}(\tilde{\sigma}(j), t, a). \tag{17}$$

If $v \notin C(j)$, then the time frame of v in iteration j equals the one in iteration $j-1$, and scheduling operation v in clock cycle c is already attempted in iteration $j-1$. If we now want to determine the effect on other time frames, by means of (16), then we see that for operations $u \notin C(j)$ this effect is the same as in the previous iteration. Only for operations $u \in C(j)$ the effect may be different. For an example, see Fig. 6. There we have six operations, of which only one time frame has changed in iteration $j-1$, namely of operation d, i.e., $C(j) = \{d\}$. In the figure the effect is shown of scheduling operation b in clock cycle 5, for iteration $j-1$ and j. Only the effect on operation d's time frame is different; for the other operations the effect in iteration j is the same as in iteration $j-1$.

As a consequence, only the operations in $C(j)$ have a contribution to the changes in processing unit distribution function $\Delta\tilde{N}$ that differs from the one in the previous iteration, and only for variables produced or consumed by operations in $C(j)$ new contributions have to be calculated to the changes in the distribution function for variable lifetimes and memory accesses. So, if we define for the operations

$$\Delta P(\tilde{\sigma}, \tilde{\tau}, u, a) = P(\tilde{\tau}, u, a) - P(\tilde{\sigma}, u, a),$$

then for $t \in T$ the changes in the processing unit distribution function can be written as

$$\begin{aligned}
&\Delta\tilde{N}(\tilde{\sigma}(j),\tilde{\tau}(j),t,a) \\
&\quad = \sum_{u\in V(t)}\sum_{k\in\mathbb{Z}} \Delta P(\tilde{\sigma}(j),\tilde{\tau}(j),u,a+kF) \\
&\quad = \sum_{u\in V(t)}\sum_{k\in\mathbb{Z}} \Delta P(\tilde{\sigma}(j),\tilde{\tau}(j),u,a+kF) \\
&\qquad + \Delta\tilde{N}(\tilde{\sigma}(j-1),\tilde{\tau}(j-1),t,a) \\
&\qquad - \sum_{u\in V(t)}\sum_{k\in\mathbb{Z}} \Delta P(\tilde{\sigma}(j-1),\tilde{\tau}(j-1),u,a+kF) \\
&\quad = \Delta\tilde{N}(\tilde{\sigma}(j-1),\tilde{\tau}(j-1),t,a) \\
&\qquad + \sum_{u\in V(t)\cap C(j)}\sum_{k\in\mathbb{Z}} [\Delta P(\tilde{\sigma}(j),\tilde{\tau}(j),u,a+kF) \\
&\qquad - \Delta P(\tilde{\sigma}(j-1),\tilde{\tau}(j-1),u,a+kF)]. \qquad (18)
\end{aligned}$$

Similarly, the changes in the distribution function for variable lifetimes and memory accesses can be computed incrementally [28]. Now, computing $\Delta\tilde{N}$ incrementally is restricted to computing the changes in the probability functions of the operations and variables that have been affected in the previous iteration. So, if only a few operations have been affected, the gain in computational effort is substantial.

C. Improved Force-Directed Scheduling

The improved force-directed scheduling (IFDS) algorithm can now be written as follows.

Improved force-directed scheduling

Step 1 Initialize time frames to ASAP and ALAP schedules.

Step 2 Calculate $\tilde{N}(\tilde{\sigma},t,s)$.

Step 3 For each operation v which is not scheduled yet, and for $c=\sigma_{\mathrm{l}}(v)$ and $c=\sigma_{\mathrm{u}}(v)$, calculate $\Delta\tilde{N}(\tilde{\sigma},\tilde{\tau},t,a)$, where $\tilde{\tau}$ is the schedule frame obtained from $\tilde{\sigma}$ by scheduling v in clock cycle c. Do this by
- calculating $\Delta\tilde{N}$ from scratch if v is affected in the previous iteration or if this is the first iteration, and
- otherwise, calculate $\Delta\tilde{N}$ incrementally, using the stored $\Delta\tilde{N}$ from the previous iteration.

Step 4 Store $\Delta\tilde{N}(\tilde{\sigma},\tilde{\tau},t,s)$ and compute $F_{\mathrm{gsc}}(\tilde{\sigma},\tilde{\tau})$.

Step 5 Take operation v for which $F_{\mathrm{gain}}(v)$ is maximal, and reduce its time frame to $[\sigma_{\mathrm{l}}(v)+1,\sigma_{\mathrm{u}}(v)]$ if $F_{\max}(v)=F_{\mathrm{left}}(v)$, and to $[\sigma_{\mathrm{l}}(v),\sigma_{\mathrm{u}}(v)-1]$ otherwise.

Step 6 Update time frames of the operations and detect which operations and which variables are affected.

Step 7 Update $\tilde{N}(\tilde{\sigma},t,s)$.

Step 8 If not all operations are scheduled, return to Step 3.

Note that at most $2n$ functions $\Delta\tilde{N}$ over domain $T^*\times[0,F-1]$ have to be stored, since the gradual time-frame reduction requires only two forces per operation to be calculated.

D. Time Complexity of IFDS

The worst-case time complexity of the improved algorithm is $\mathcal{O}(Fln^2)$, where n is the number of operations, l is the maximum length of an operation's initial time frame, and F is the algorithm period. This can be shown as follows.

- The amount of work for determining $\Delta\tilde{N}$ is as follows.
  a) In the first iteration $\Delta\tilde{N}$ is determined according to (17), for all $v\in V$ and $c=\sigma_{\mathrm{l}}(v)$ and $c=\sigma_{\mathrm{u}}(v)$. Calculating (17) takes $\mathcal{O}(Fn)$ steps, resulting in a total number of calculations for this step given by $\mathcal{O}(Fn^2)$.
  b) If in an iteration the total slack, i.e., the sum of the lengths of the operations' time frames, is reduced by k, then at most k operations are affected, and thus determining $\Delta\tilde{N}$ in the next iteration takes $\mathcal{O}(kFn)$ steps, since for $\mathcal{O}(k)$ operations we use (17) which takes $\mathcal{O}(Fn)$ steps, and for $\mathcal{O}(n)$ operations we use (18) which takes $\mathcal{O}(kF)$ steps.
  c) The total slack is $\mathcal{O}(ln)$, so the total number of steps to determine $\Delta\tilde{N}$ in the iterations after the first one is $\mathcal{O}(Fln^2)$.
- The amount of work for determining the forces is as follows.
  a) In each iteration two forces have to be determined for $\mathcal{O}(n)$ operations, which is a sum of $\mathcal{O}(F)$ products, so per iteration this requires $\mathcal{O}(Fn)$ steps.
  b) The total slack is $\mathcal{O}(ln)$ and in each iteration the slack is decreased, so there are $\mathcal{O}(ln)$ iterations. Therefore, the total amount of work to determine all forces is $\mathcal{O}(Fln^2)$.
- The amount of work for updating the distribution function is as follows.
  a) If in an iteration the total slack of the operations is reduced by k, then at most k operations are affected, and thus $\mathcal{O}(k)$ operations and $\mathcal{O}(k)$ variables have contributions to the $\Delta\tilde{N}$ function. These have to be determined for $\mathcal{O}(F)$ clock cycles, so updating $\tilde{N}$ takes $\mathcal{O}(kF)$ steps.
  b) The total slack is $\mathcal{O}(ln)$, so the total amount of work to update the distribution function is $\mathcal{O}(Fln)$.

Note that the worst-case time complexity of the improved algorithm is not only determined by the calculation of the changes in the distribution function, but also by the calculation of the forces.

VII. Experimental Results

In this section we discuss some experimental results to illustrate the significance of the effectiveness and efficiency improvements. For comparison to related work on force-directed scheduling [1], [21], [22] we note that the latter approaches are comparable to the basic algorithm given in Section III, although they may contain extensions. Therefore, the improvements presented here may also be achieved when applying our ideas to those approaches.

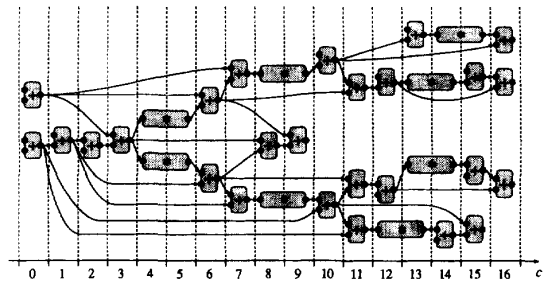

Fig. 7. An optimal schedule for the fifth-order digital elliptical wave filter for a makespan of 17 clock cycles, which uses 3 multipliers and 3 adders. All edges have a weight $d(e) = 0$.

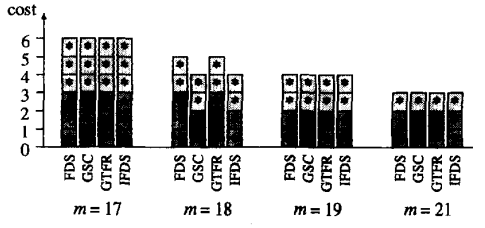

Fig. 8. Adder and multiplier allocations for the filter example for the four algorithms and for different makespans, in case of nonpipelined multipliers.

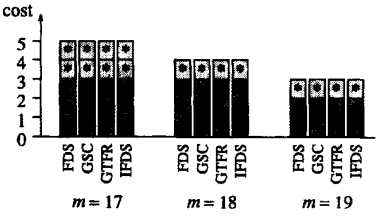

Fig. 9. Adder and multiplier allocations for the filter example with pipelined multipliers, for different makespans.

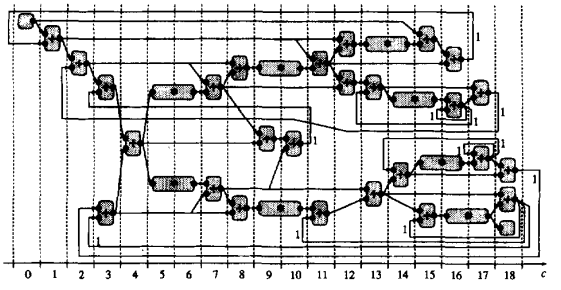

Fig. 10. The original graph of the fifth-order digital elliptical wave filter, optimally scheduled in a span [0, 32] and an algorithm period $F = 16$. The backward edges have weight 1, the others have weight 0.

A. *Effectiveness Improvements*

We illustrate the effectiveness improvements by means of four algorithms: the basic algorithm given in Section III (FDS), the algorithm with gradual time-frame reduction (GTFR), the algorithm with global spring constants (GSC), and the algorithm with both modifications (IFDS). In all algorithms we use $\eta = \frac{1}{3}$, i.e., in the $\Delta\tilde{f}$ criterion in FDS and GTFR we replace the factor $\frac{1}{2}$ for $\Delta\tilde{N}$ by $\frac{1}{3}$. Furthermore, we take $z = 3$ for the global spring constants in GSC and IFDS.

The effectiveness improvements are shown by means of three examples. The first example is the notorious fifth-order digital elliptical wave filter from Dewilde, Deprettere, and Nouta [3]. A graphical representation of this example is given in Fig. 7. The graph is scheduled without overlapping executions in a makespan of m clock cycles, where we have taken several values of m. For this example we minimize the number of functional units, by taking $a(+) = a(*) = 1.0$. The retiming of all ports equals 0, except for the output ports of the multipliers, which have retiming 1. The restart time of an addition $s(+) = 1$. The restart time of a multiplication is $s(*) = 2$ in case of a nonpipelined multiplier, and $s(*) = 1$ in case of a pipelined multiplier.

Fig. 8 shows the results for nonpipelined multipliers, and Fig. 9 shows the results in case of pipelined multipliers. Running times vary between 3 and 34 seconds on an Apollo HP 425t. All algorithms find optimal solutions in all cases, except FDS and GTFR for nonpipelined multipliers and a makespan equal to 18.

For the filter example, the graph shown in Fig. 7 is repeatedly executed. For this purpose the original graph was cut at iteration boundaries in order to get a graph without cycles.

TABLE I
SCHEDULING AND ASSIGNMENT FOR THE FIFTH-ORDER DIGITAL ELLIPTICAL WAVE FILTER, FOR AN ALGORITHM PERIOD OF 17 CLOCK CYCLES AND PIPELINED MULTIPLIERS

	IFDS	SAM	SAM	HAL
adders	2	3	3	3
multipliers	1	2	2	2
registers	12	12	14	12
mux. inputs	26	31	28	31
connections	44	50	49	-

However, since we can handle cycles as well, it is better to start with the original graph, shown in Fig. 10. For this graph it is possible to find a feasible schedule for an algorithm period equal to 16.

For comparison to other work, we have scheduled the cyclic graph with IFDS, with an algorithm period of 17 clock cycles, and with pipelined multipliers. After this, we have manually assigned operators and registers. The results are compared to the results on the acyclic graph reported by Cloutier and Thomas (SAM) [1] and Paulin and Knight (HAL) [22]; see Table I. As we can see, IFDS not only results in a solution with less adders and multipliers, but also with less interconnect costs.

The second example we use is an inverse discrete cosine transform (IDCT) [30], shown in Fig. 11. There are 30 additions/subtractions with restart time 1 and retiming 0 for all ports, and there are 16 multiplications with restart time 1, and retiming 0 for the input port and 1 for the output port. The cost of an adder/subtracter equals 2.5 and the cost of a multiplier equals 4.0. The input operations $i1, i2, \cdots, i8$ are executed in clock cycles $0, 2, \cdots, 14$, respectively, and

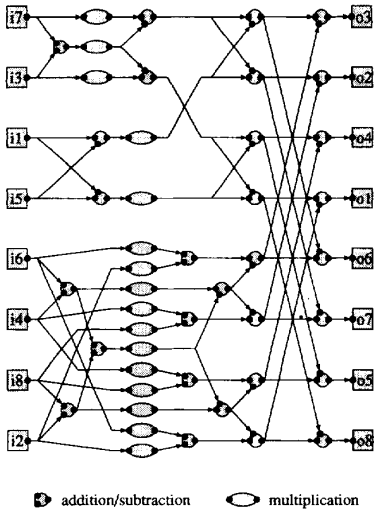

Fig. 11. Graphical representation of the IDCT algorithm. All edges have weight 0.

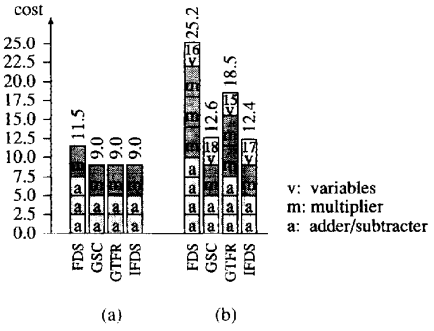

Fig. 12. Results for the IDCT example (a) without and (b) with costs for variable lifetimes.

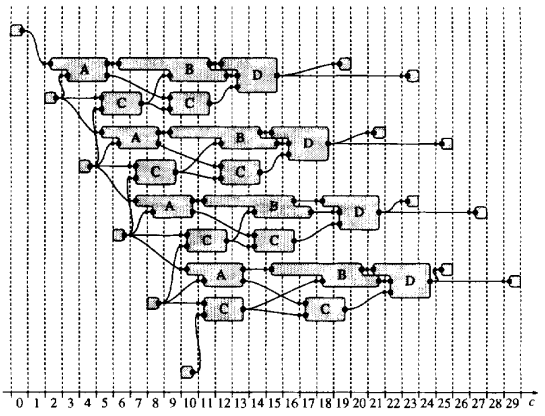

Fig. 13. Graphical representation of the more elaborate example, scheduled in 30 clock cycles. All edges have weight 0.

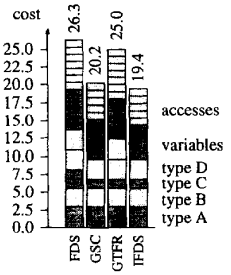

Fig. 14. Results for the more elaborate example.

Fig. 15. Graphical representation of the SEQ example. All edges have weight 0.

the output operations $o1, o2, \cdots, o8$ are executed in clock cycles $24, 26, \cdots, 38$, respectively. Furthermore, the algorithm period $F = 16$. The results of the four algorithms are shown in Fig. 12, for the case that only adders/subtracters and multipliers are taken into account (Fig. 12(a)), and for the case that also variable lifetimes are taken into account with cost $a(t_v) = 0.2$ (Fig. 12(b)). The figure clearly shows the improvements due to global spring constants, which allow a better trade-off between a large number of low-cost resources and a small number of high-cost ones. In the case of costs for variable lifetimes, FDS produces a solution that is over twice as expensive as the solution produced by IFDS.

The third example we use is more elaborate. Its graphical representation is given in Fig. 13. We have input and output nodes and four different types of processing units, denoted by A, B, C, and D. The processing units have restart times $s(A) = 2$, $s(B) = 2$, $s(C) = 1$, and $s(D) = 3$, and area costs $a(A) = 3.0$, $a(B) = 2.5$, $a(C) = 1.3$, and $a(D) = 2.8$. Input and output nodes have costs 0.0. Furthermore, costs of variables are $a(t_v) = 0.8$ and costs of memory accesses are $a(t_a) = 1.0$. Initially, the input and output nodes are fixed to the clock cycles indicated in the figure, and the other operations have spans $[-\infty, +\infty]$. The results obtained by applying the four algorithms to this example are shown in Fig. 14. For this example a total cost of 19.4 is the best result ever found. FDS produces a solution with a cost more than 35% larger than this value.

B. Efficiency Improvements

In this section we give some experimental results to illustrate the running time reductions obtained by the incremental computation of the changes in distribution function of Section VI. We do this by means of three examples: the fifth-order digital elliptical wave filter (FILTER) and the IDCT example of the previous section, and a theoretical example (SEQ). This last example consists of a sequence of 53 operations, shown in Fig. 15. The input and output ports of these operations have retiming 0, and the restart time of the operations is 1. In contrast to the IDCT, which allows parallelism in the execution of operations, this example only allows sequential execution of the operations involved.

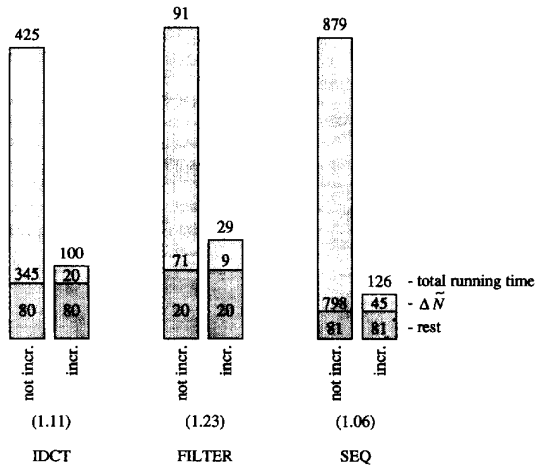

Fig. 16. Running times in seconds for the three examples. The numbers between brackets give the average number of affected operations per iteration.

We applied IFDS to these examples, with and without incremental computation of $\Delta\tilde{N}$. We only considered costs of processing units, i.e., no costs for variable lifetimes or memory accesses are taken into account. The algorithms found optimal solutions, and the running times required are shown as bars in Fig. 16. For each example, the left bar gives the running time without incremental computation, and the right bar gives the running time with incremental computation. The total running time is split into two parts. The first one is the time needed to calculate $\Delta\tilde{N}$, and the second part is the time needed to do all the remaining calculations. As we can see, the running times for determining the changes in the distribution function is drastically reduced for all examples.

VIII. Conclusion

In this paper we have discussed the technique of force-directed scheduling. We have given a mathematical justification of the basic force-directed scheduling algorithm as introduced by Paulin and Knight [19]–[21], and we proposed modifications to improve the effectiveness and the efficiency of the algorithm. The effectiveness improvements are achieved by using global spring constants, which ensures that trade-offs are done better, and gradual time-frame reduction, which makes the algorithm less greedy. The efficiency improvements are achieved by an incremental method to calculate the changes in the distribution function, which reduces the time complexity of the algorithm from cubic in the number of operations to quadratic. The effectiveness and efficiency improvements have been illustrated by an empirical performance analysis based on a number of problem instances. We have shown that the improvements obtained are substantial.

Furthermore, we have shown the application of force-directed scheduling to the high-throughput scheduling problem, which occurs in the design methodology PHIDEO. Since this problem is NP-hard and since practical instances are too large to be solved exhaustively, we decided to use an approximation algorithm. For this, we have chosen force-directed scheduling because of the following advantages. First, force-directed scheduling is a technique that is able to make a trade-off between processing units and memory. This is important if memory constitutes a significant part of a design. Second, force-directed scheduling can be used for pipelined scheduling and for cyclic signal flow graphs, which are characteristic for DSP applications. Third, force-directed scheduling finds high-quality solutions. The time complexity of force-directed scheduling is rather high, but with the presented improvements the algorithm has practical running times for signal flow graphs with a number of operations in the order of magnitude of 100.

References

[1] R. J. Cloutier and D. E. Thomas, "The combination of scheduling, allocation, and mapping in a single algorithm," Carnegie Mellon Univ., Pittsburgh, PA, Res. Rep. CMUCAD-90-17, May 1990.
[2] H. De Man, F. Catthoor, G. Goossens, J. Vanhoof, J. van Meerbergen, and J. Huisken, "Architecture-driven techniques for VLSI implementation of DSP algorithms," *Proc. IEEE*, vol. 78, no. 2, pp. 319–335, 1990.
[3] P. Dewilde, E. Deprettere, and R. Nouta, "Parallel and pipelined VLSI implementation of signal processing algorithms," *VLSI and Modern Signal Processing*, S. Y. Kung, H. J. Whitehouse, and T. Kailath, Eds. Englewood Cliffs, NJ: Prentice-Hall, 1985, pp. 258–264.
[4] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York: Freeman, 1979.
[5] C. H. Gebotys and M. I. Elmasry, "A global optimization approach for architectural synthesis," in *Proc. ICCAD*, Santa Clara, CA, Nov. 1990, pp. 258–261.
[6] R. Haupt, "A survey of priority-rule based scheduling," *OR Spektrum*, vol. 11, no. 1, pp. 3–16, 1989.
[7] C. T. Hwang, J. H. Lee, and Y. C. Hsu, "A formal approach to the scheduling problem in high level synthesis," *IEEE Trans. CAD*, vol. 10, no. 4, pp. 464–475, 1991.
[8] T. Kim, J. W. S. Liu, and C. L. Liu, "A scheduling algorithm for conditional resource sharing," in *Proc. ICCAD*, Santa Clara, CA, Nov. 1991, pp. 84–87.
[9] J. H. M. Korst, "Periodic multiprocessor scheduling," Ph.D. dissertation, Eindhoven Univ. of Technol., Eindhoven, The Netherlands, 1992.
[10] C. E. Leiserson, F. M. Rose, and J. B. Saxe, "Optimizing synchronous circuitry by retiming," in *Third Caltech Conf. VLSI*, Mar. 1983, pp. 87–116.
[11] P. E. R. Lippens, J. L. van Meerbergen, A. van der Werf, W. F. J. Verhaegh, B. T. McSweeney, J. O. Huisken, and O. P. McArdle, "PHIDEO: A silicon compiler for high speed algorithms," in *Proc. EDAC*, Amsterdam, The Netherlands, Feb. 1991, pp. 436–441.
[12] R. S. Martin and J. P. Knight, "Operations research in the high-level synthesis of integrated circuits," *Comput. Ops Res.*, vol. 20, no. 8, pp. 845–856, 1993.
[13] M. C. McFarland, "Using bottom-up design techniques in the synthesis of digital hardware from abstract behavioral descriptions," in *Proc. 23rd DAC*, Las Vegas, NV, June 1986, pp. 474–480.
[14] M. C. McFarland, A. C. Parker, and R. Camposano, "The high-level synthesis of digital systems," *Proc. IEEE*, vol. 78, no. 2, pp. 301–318, 1990.
[15] J. L. van Meerbergen, P. E. R. Lippens, B. T. McSweeney, W. F. J. Verhaegh, A. van der Werf, and A. T. van Zanten, "Architectural strategies for high-throughput applications," *J. VLSI Signal Process.*, vol. 5, no. 2/3, pp. 201–220, Apr. 1993.
[16] S. Note, J. L. van Meerbergen, F. Catthoor, and H. de Man, "Automated synthesis of a high-speed CORDIC algorithm with the CATHEDRAL-III compilation system," in *Proc. ISCAS*, Helsinki, Finland, June 1988, pp. 581–584.
[17] 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 transfer mobility," in *Proc. CompEuro*, Delft, The Netherlands, 1992, pp. 145–150.
[18] N. Park and A. C. Parker, "Sehwa: A software package for synthesis of pipelines from behavioral specification," *IEEE Trans. Computer-Aided Design*, vol. 7, no. 3, pp. 356–370, 1988.
[19] P. G. Paulin and J. P. Knight, "Force-directed scheduling in automatic data path synthesis," in *Proc. 24th DAC*, Miami Beach, FL, July 1987, pp. 195–202.

[20] P. G. Paulin, "High-level synthesis of digital circuits using global scheduling and binding algorithms," Ph.D. dissertation, Carleton University, Ottawa, Canada, 1988.
[21] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASICs," *IEEE Trans. Computer-Aided Design*, vol. 8, no. 6, pp. 661–679, 1989.
[22] ——, "Algorithms for high-level synthesis," *IEEE Design and Test of Computers*, vol. 6, pp. 18–31, Dec. 1989.
[23] L. Stok, "Architectural synthesis and optimization of digital systems," Ph.D. dissertation, Eindhoven Univ. of Technol., Eindhoven, The Netherlands, 1991.
[24] W. F. J. Verhaegh, "Scheduling problems in video signal processing," Master's thesis, Eindhoven Univ. of Technol., Eindhoven, The Netherlands, Apr. 1990.
[25] W. F. J. Verhaegh, E. H. L. Aarts, J. H. M. Korst, and P. E. R. Lippens, "Improved force-directed scheduling," in *Proc. EDAC*, Amsterdam, The Netherlands, Feb. 1991, pp. 430–435.
[26] W. F. J. Verhaegh, P. E. R. Lippens, E. H. L. Aarts, J. H. M. Korst, J. L. van Meerbergen, and A. van der Werf, "Modelling periodicity by PHIDEO streams," talk presented at the Sixth High Level Synthesis Workshop, Dana Point Resort, Nov. 1992.
[27] W. F. J. Verhaegh, 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," in *Proc. ICCAD*, Santa Clara, CA, Nov. 1992, pp. 286–291.
[28] ——, "Improved force-directed scheduling in high-throughput digital signal processing," Philips Research Laboratories, Eindhoven, The Netherlands, Nat. Lab. Rep. NL-UR-015/94, Sept. 1994.
[29] A. van der Werf, B. T. McSweeney, J. L. van Meerbergen, P. E. R. Lippens, and W. F. J. Verhaegh, "Hierarchical retiming including pipelining," in *Proc. VLSI*, Edinburgh, Aug. 1991, pp. 11.2.1–11.2.10.
[30] P. H. N. de With, "Motion-adaptive intraframe transform coding of video signals," *Philips J. Res.*, vol. 44, no. 2/3, pp. 345–364, 1989.

Wim F. J. Verhaegh received the mathematical engineering degree with honors in 1990 from the Eindhoven University of Technology, The Netherlands.

Since then, he has been with the Philips Research Laboratories in Eindhoven as a member of the group Digital VLSI. Presently, he is working on high-level synthesis of DSP systems for video applications, with the emphasis on scheduling problems and techniques.

Paul E. R. Lippens received the electrical engineering degree with honors in 1986 from the Eindhoven University of Technology, The Netherlands.

Since then, he has been with the Philips Research Laboratories in Eindhoven as a member of the group Digital VLSI. Presently, he is working on architectural level synthesis of DSP systems for video applications.

Emile H. L. Aarts received the M.Sc. degree in physics from the University of Nijmegen, The Netherlands, and the Ph.D. degree from the University of Groningen, The Netherlands.

He is with the Philips Research Laboratories in Eindhoven, The Netherlands as a Senior Scientist. He also holds an appointment as a Professor of Computer Science with the Eindhoven University of Technology, and a management consultancy position with the Research Institute for Information Systems, Maastricht, The Netherlands. His research field is combinatorial optimization in planning and design.

Jan H. M. Korst received the M.Sc. degree in mathematics from Delft University of Technology, The Netherlands, and the Ph.D. degree from the Mathematics and Computing Science Department of the Eindhoven University of Technology, The Netherlands.

Since 1985 he has been with the Philips Research Laboratories in Eindhoven, where he has been working mainly on combinatorial optimization and resource management problems in the fields of VLSI design and multimedia systems. His research interests include combinatorial optimization, complexity theory, and the design and analysis of algorithms.

Jef L. van Meerbergen (M'87–SM'92) received the electrical engineering degree and the Ph.D. degree from the Katholieke Universiteit Leuven, Belgium, in 1975 and 1980, respectively.

In 1979 he joined the Philips Research Laboratories in Eindhoven, The Netherlands. He was engaged in the design of MOS digital circuits, domain-specific processors, and general purpose digital signal processors. In 1985, he started working on application-driven high-level synthesis. Initially, this work was targeted towards audio and telecom DSP applications. The current research activities are concentrated on high-level synthesis for high-throughput video applications.

Albert van der Werf (M'91) received the M.Sc. degree with honors in electrical engineering for research in the area of telecommunications in 1987 from the University of Twente, The Netherlands.

From 1987 to 1989 he followed a postgraduate course on the design of VLSI circuits at the Graduate School Twente, The Netherlands. Since 1989 he has been with Philips Electronics Ltd. at its research laboratories in Eindhoven, The Netherlands. His present areas of interest include the development of computer-aided design methodologies for the design of integrated circuits. Currently, he is pursuing the Ph.D. degree at the Eindhoven University of Technology, The Netherlands.